Scalable Methods and Expressive Models for Planning Under Uncertainty

Andrey Kolobov

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Scalable Methods and Expressive Models for Planning Under Uncertainty

Andrey Kolobov

Co-Chairs of the Supervisory Committee:
Research Assistant Professor Mausam
Computer Science & Engineering

Professor Daniel S. Weld
Computer Science & Engineering

The ability to plan in the presence of uncertainty about the effects of one's own actions and the events of the environment is a core skill of a truly intelligent agent. This type of sequential decision-making has been modeled by Markov Decision Processes (MDPs), a framework known since at least the 1950's [45, 3]. The importance of MDPs is not merely philosophic — they have been applied to several impactful real-world scenarios, from inventory management to military operations planning [80, 1]. Nonetheless, the adoption of MDPs in practice is greatly hampered by two aspects. First, modern algorithms for solving them are still not scalable enough to handle many realistically-sized problems. Second, the MDP classes we know how to solve tend to be restrictive, often failing to model significant aspects of the planning task at hand. As a result, many probabilistic scenarios fall outside of MDPs' scope.

The research presented in this dissertation addresses both of these challenges. Its first contribution is several highly scalable approximation algorithms for existing MDP classes that combine two major planning paradigms, dimensionality reduction and deterministic relaxation. These approaches automatically extract human-understandable causal structure from an MDP and use this structure to efficiently compute a good MDP policy. Besides enabling us to handle larger planning scenarios, they bring us closer to the ideal of AI — building agents that autonomously recognize features important for solving a problem. While these techniques are applicable only to goal-oriented scenarios, this dissertation also introduces approximation algorithms for reward-oriented settings.

The second contribution of this work is new MDP classes that take into account previously ignored aspects of planning scenarios, e.g., the possibility of catastrophic failures. The thesis explores their mathematical properties and proposes algorithms for solving these problems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

As my Ph.D. adventure is nearing its end at the speed of ten double-spaced pages per day, it is striking to recall how many people it took to make it what it was — a deeply enriching meditatively relaxing occasionally frustrating immensely exhilarating one-of-a-kind experience.

As with almost any other student, many aspects of my life during graduate school years have been, directly or indirectly, influenced by my academic advisers, professors Dan Weld and Mausam. I would like to say a heartily felt "thank you" to these great researchers and, more importantly, great people. Before starting the Ph.D., I had expected an adviser to be a (possibly nonlinear) combination of a grumpy omniscient demigod and a busy micro-manager who would throw impossible problems at me and listen to my silly ideas about them once a month. None of that turned out to be true of Dan and Mausam. From the very first meeting, we simply talked, discussed scientific problems and ways of solving them. The value of the boost this gave to my confidence cannot be overstated: it made me believe that maybe, just maybe, someday my ideas would become as good as theirs. I was not an easy case, either — as time went by, I started getting opinions on different subjects that diametrically opposed theirs. Yet even then they were as patient and supportive with me as ever, and for this I would like thank to them especially.

There have been many people besides my main academic advisers with whom I have worked academically. Without any single one of them, my Ph.D. experience would have been less complete or would not have happened at all. I would like to thank all these individuals. Chronologically, the first of them are professor Stuart Russell of UC Berkeley and Brian Milch, his graduate student at the time. Ultimately, it was them who first showed me to all aspects of research, from formulating an idea to implementing it, to evaluating it, to publishing it. At the University of Washington, while Dan and Mausam were introducing me to the world of probabilistic planning, professor Dieter Fox was introducing me to the world of robotics. Although I ultimately chose research in planning, the robotics knowledge Dieter gave me serves me to this day. For my first theoretical paper about

MDPs, I was lucky to collaborate with professor Hector Geffner of Universitat Pompeu Fabra, a researcher whose depth of theoretical expertise in planning can only be matched by his inimitable ability to explain the most complicated planning concepts in a fun and accessible way. I would also like to thank Florent Teichteil-Königsbuch of ONERA and Blai Bonet of Universidad Simón Bolívar for our work on the last paper of my Ph.D. and all the lively discussions over email at three a.m. in the morning.

I wish to thank the research institutions who have hosted me at various times during my graduate career: the University of Washington, Microsoft Research center in Cambridge, UK, Microsoft Research center in Redmond, US, and the DTAI group at the Computer Science Department of Katholieke Universiteit Leuven, Belgium. At MSR Cambridge, I would like to thank Martin Szummer, under whose supervision I got a chance to study problems in information retrieval. At MSR Redmond, my gratitude goes to Ashish Kapoor, Rich Caruana, and Eric Horvitz. Eric has utterly astounded me with the breadth of his knowledge both in computer science and beyond, a renaissance man in an era of specialization. Finally, professors Luc De Raedt and Jesse Davis of KU Leuven have given me fantastic opportunities to do research with them and their extremely sharp Ph.D. student Jan in machine learning, and waded through the Belgian bureaucratic quagmires together with me to make these opportunities become reality. Besides Jesse, Jan, and Luc, I would like to thank the graduate students at the CS department of KU Leuven, especially Anton D., Wannes M., and Guy V. for all the fun moments I shared with them. My stays in Belgium would have been less productive were it not for the Belgian National Railways (NMBS), so they deserve a word of gratitude too. Their utter inability to make trains run on time, and sometimes at all, provided me with plenty of undisturbed thinking time as I was stuck at different stations and fields throughout Flanders. My last Belgium-related acknowledgment, but this time not a word of gratitude, goes to the Belgian bureaucrats, whose almost uniformly xenophobic, unjust, and insulting treatment that makes Kafka's novels feel trivial has invariably accompanied the services of the Belgian railways during my sojourns in that country.

Of course, graduate school is not just work. A huge thank you to my fellow UW graduate students, MSR interns, and simply friends who have been around during the Ph.D. journey is definitely in order, particularly to Andrei A., Marat A., Sergey B., Petra B., Scott B., Jessica C., Kamal C., Alexei C., Peng D., Max G., Michal J., Dagmara K., Karl K., Chris L., Cynthia M., Kira R., Michael R., Alexander S., Eugene S., and Veronika S. Thanks for all the ping-pong, College Inn nights, sailing, Thursty Thursdays, Settlers, long phone chats, Nepalese food, discussions of European politics, hiking, and all manner of craziness! A special thank you goes to all the wonderful people I have met and had a splendid time with at the ICAPS conferences, such as Erez K., Emil K., Nir L., Miquel R., Silvia R., and Daniel T. I hope we manage to continue to see each other often in the future!

I would like to finish this trip down the memory lane with the biggest thanks of all, to my family: my parents Nina and Vadim, my sister Darina, my nephew Vitya, my grandmother Maria, my wife Martine, and my daughter Mira. Without you, all of this would have been neither possible nor worth it. Mama and papa, everything that I have achieved so far I owe to your love, support and belief in me. Thank you for your courage in sending me across the ocean thirteen years ago. Only now that I am a parent myself I am starting to fully realize how hard of a step it must have been for you, and can only hope that I will be able to give to my children as much as you have given to me. Martine, thank you for the miracle that we have. Having come from two different continents and met on a third, we could not have made it but for your love, patience, and faith in me. Mira, you are my future. I will make up to you all the evenings spent writing this dissertation, promise.

DEDICATION

to Martine and Mira,

my present and my future

Chapter 1

INTRODUCTION

Since the early days of artificial intelligence (AI) as a field, the ability to devise a plan for achieving a goal has been considered an essential characteristic of an intelligent agent. This ability was a major factor in the success of one of the first full-fledged AI systems, implemented in Shakey the robot in the early 1970s [76]. Shakey planned with A^* [75], a deterministic algorithm that has since become classic. It assumed to be operating in a fully known static world modifiable only with Shakey's own actions, whose effects could be predicted with certainty. For Shakey and other research demonstration systems meant to work in mostly controlled environments under human supervision, these assumptions often suffice. However, their distant descendants, such as Mars rovers Spirit and Opportunity, carry out missions under the conditions where their engines may fail, their sensors may err, seemingly firm ground may turn out to be a sand pit, and failing to take into account these possibilities may mean a loss of the robot. Tackling these challenges requires a more sophisticated paradigm — *planning under uncertainty*.

A popular family of models for analyzing probabilistic planning problems is *Markov Decision Processes* (MDPs). Although MDPs have been known at least since the 1950s, i.e., before the Shakey project started, their adoption as a computational planning tool by the AI community happened only several decades later. A major reason for this delay was most contemporary computers' lack of power to solve but the tiniest MDP instances. Moreover, the first MDP models concentrated on scenarios where agents aimed to maximize reward, not to get to a particular goal state, contrasting with the dominant view of planning in AI at that time. By the early 1990s, the increasing availability of computational resources coupled with the formulation of a goal-oriented MDP class [4] and a general interest in combining decision theory and planning had finally sparked firm interest in MDPs among AI researchers. Since then, the advances in MDPs' expressiveness and solution techniques have allowed practitioners to successfully apply MDPs to a number of important problems such as military operations planning [1].

Nonetheless, the fundamental challenges in planning under uncertainty that have existed since its inception remain largely unaddressed. Most modern algorithms for solving MDPs do so by computing the behavior of an agent for each state of the world separately. Incidentally, this strategy was also used by the seminal MDP solution techniques, value iteration (VI) [3] and policy iteration (PI) [45], although its modern exponents, such as LRTDP [13] and LAO* [41], apply it much more efficiently. In spite of today's hardware making these approaches practical for problems with on the order of millions of states, at the current rate of computational power growth many larger realistic scenarios will forever remain beyond the reach of these algorithms. Thus, qualitative progress in scalability appears to call for techniques based on different principles. In addition, despite being able to model the behavior of goal-driven agents to some extent, the known MDP classes still impose many restrictions on the settings in which these agents act. For instance, well-studied goal-oriented MDP types postulate that the goal state must be reachable from any other state with probability 1. This effectively "outlaws" any catastrophic events, including those that could destroy the agent (e.g., a robot falling off an elevation) — a clearly unrealistic requirement. Removing the likes of these severe and often counterintuitive limitations would make MDPs into a much more flexible modeling tool.

This dissertation responds to the challenges facing the field of probabilistic planning with two broad contributions. The first of them consists in proposing several highly scalable approximation algorithms for existing MDP classes. For goal-oriented MDPs, they are based on the novel idea of integrating three major planning paradigms, heuristic search, dimensionality reduction and deterministic relaxation. For reward-oriented MDPs with dense transition matrices, a particularly difficult kind of planning problems, this dissertation introduces a set of altogether different techniques. They avoid considering all possible states of the world by planning online only for the state regions the agent visits, and use the strategy of reverse iterative deepening for accelerated convergence. The second contribution of this thesis are MDP classes that relax the assumptions made by the sole known goal-oriented MDP type, the stochastic shortest path (SSP) problems. In particular, unlike SSP MDPs, these new models allow describing scenarios with dead-end states, i.e., world configurations from which reaching the goal is impossible no matter how hard the agent tries. This dissertation develops a mathematical theory of such MDPs and derives algorithms for solving them.

We now survey each of the contributions in more detail, as well as examine how they interplay with each other.

1.1 Scalable Algorithms for Probabilistic Planning

In contrast to the traditional general-purpose MDP algorithms, humans routinely cope with many large scenarios that require probabilistic reasoning to achieve a goal. An example of such a task is planning a transcontinental trip, e.g., from Moscow to Seattle — a setting where plan success depends on uncertainties ranging from airport personnel strikes in Europe and lines at the border control in the US to volcano eruptions in Iceland and traffic jams on the way to the airport in Moscow. While human-generated courses of action can be significantly suboptimal in terms of utility, such as travel time or cost in the above example, they often reliably lead the agent to the goal. The "hacks" that help humans in handling large planning scenarios like this include the use of heuristics, abstractions, and problem relaxations to produce a *satisficing* solution. While automatic counterparts of these tricks — heuristic search, dimensionality reduction, and MDP determinization — are well-understood, to date they have not been integrated into a single planning framework, as they are in humans.

To realize the potential of unifying these paradigms for goal-directed probabilistic planning, this dissertation introduces algorithms that automatically extract human-understandable causal structure from an MDP via problem determinization and use this structure to compute informative planning heuristics or directly solve the given MDP. Specifically, our techniques generate two kinds of abstraction, *basis functions* and *nogoods*, each of which describes sets of states that share a relationship to the planning goal. A critical distinguishing feature of the proposed methods is their ability to construct these abstractions in a fast, fully autonomous and problem-independent way for problems whose state space lacks a natural notion of a metric. Both basis functions and nogoods are represented as logical conjunctions of an MDP's state variable values, but they encode diametrically opposite information. When a basis function holds in a state, this guarantees that a certain trajectory of action outcomes has a positive probability of reaching the goal from that state. For instance, in a scenario involving a Mars rover that needs to conduct experiments on Martian rocks, one basis function might correspond to the conjunction *(Status(RockBore) = Normal)* $\wedge$ *(Weather = Clear)* $\wedge$ *(AtExperimentLocation = True)* — from any state where these facts hold, the rover has a nonzero chance of fulfilling its objective. At the same time, even in a favorable situation like this the rover is not *guaranteed* to achieve the goal; future mechanical failures or adverse weather changes may prevent it from doing so. To account for this intuition, our algorithms associate weights with each basis function, encoding the relative "quality" of the situations the basis functions characterize. Our second type of abstraction, nogoods, gives a guarantee complementary to basis functions'. When a nogood, such as *(Status(Chassis)=Failed)* $\wedge$ *(AtExperimentLocation = False)* in the Mars rover example, holds in a state, it signifies that the state is a dead end; *no* trajectory can reach the goal from it.

Our notion of basis function is related to rules in explanation-based learning [47], and the notion of nogood — to a similarly named concept in constraint satisfaction [27], but our work applies them in a probabilistic context (e.g., learns weights for basis functions) and provides new mechanisms for their discovery. Unlike previous MDP algorithms that have used basis functions for knowledge transfer across different problems [37, 87] and have had them hand-generated by domain experts [36, 38, 39], the techniques introduced here construct basis functions automatically and immediately employ them to solve the problem at hand. Crucially, the construction procedure is very fast, as it relies on solving a deterministic relaxation of the given MDP with highly efficient classical planners. Thanks to reasoning about MDPs' high-level regularities, not individual states, our algorithms require little memory and thereby circumvent the main weakness of the traditional probabilistic planning solvers. Besides enabling us to handle larger planning scenarios, these approaches also bring us closer to the ideal of AI — building agents that autonomously recognize features important for solving a problem.

We present three algorithms that leverage the basis function and nogood abstractions to speed up goal-oriented MDP solution and reduce the amount of memory required for it:

- GOTH [56, 58] uses a full classical planner on a problem determinization to *generate a heuristic function* for an MDP solver, to be used as an initial estimate of state values. While classical planners have been known to provide an informative approximation of state value in probabilistic problems, they are too expensive to call from every newly visited state. GOTH amortizes this cost across multiple states by associating weights with basis functions and thus generalizing the heuristic computation. Empirical evaluation shows GOTH to be an informative heuristic that saves heuristic search methods, e.g., LRTDP, considerable time and memory.

- RETRASE [55, 58] is a *self-contained MDP solver* based on the same information-sharing insight as GOTH. However, unlike GOTH, which sets the weight of each basis function only once to compute an initial guess of states' values, RETRASE *learns* basis functions' weights by evaluating each function's "usefulness" in a decision-theoretic way. By aggregating the weights, RETRASE constructs a state value function approximation and, as we show empirically, produces better policies than the participants of the International Probabilistic Planning Competition (IPPC) on many domains while using little memory.

- SIXTHSENSE [57, 58] is a *method for quickly and reliably identifying dead ends*, i.e., states with no possible trajectory to the goal, in MDPs. In general, for factored MDPs this problem is intractable — one can prove that determining whether a given state has a trajectory to the goal is PSPACE-complete [35]; therefore, it is unsurprising that modern MDP solvers often waste considerable resources exploring these doomed states. SIXTHSENSE can act as a submodule of an MDP solver, helping it detect and avoid dead ends as the solver is exploring the problem's state space. SIXTHSENSE employs machine learning, using basis functions as training data, and is guaranteed never to generate false positives. The resource savings provided by SIXTHSENSE to an MDP solver are determined by the fraction of dead ends in an MDP's state space and reach 90% on some IPPC benchmark problems.

Although, as we demonstrate, these algorithms are effective at solving goal-oriented probabilistic planning problems, they do not easily extend to MDPs with no clearly-defined goal states. In the meantime, these MDPs form an important problem type. One of the first extensively studied settings in the MDP literature, inventory management [80], where the objective is to maximize reward over a finite number of steps, is naturally modeled as a problem of this kind. Moreover, in the presence of complicating factors such as a large number of possible exogenous events, *finite-horizon MDPs*, as these problems came to be called, can be even more difficult to solve than goal-oriented ones. Not only are determinization-based approaches inapplicable to them due to the lack of a goal, but the standard VI and PI grind to a halt on them too, because the main operator VI and PI are based

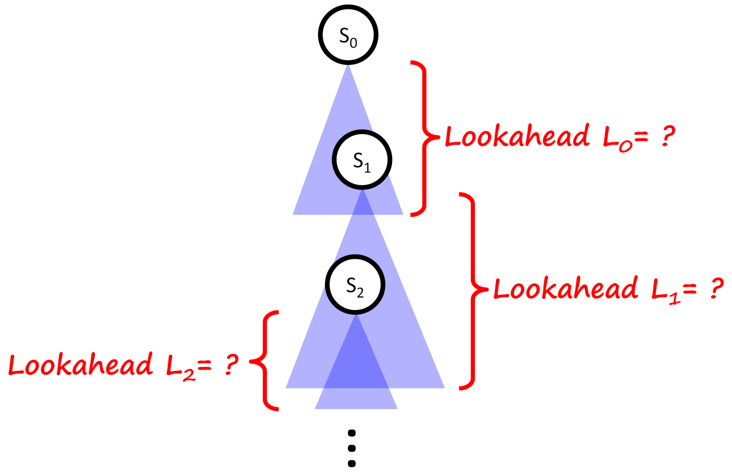


Figure 1.1: In online planning, an agent starts in some initial state s_0, evaluates the consequences of available actions for some number of steps ahead, called the *lookahead*, executes the best action according to this evaluation strategy, transitions to the next state s_1, and repeats the process. In state s_i, planning for lookahead L_i causes the agent to explore a region of the MDP's transition graph schematically depicted as a cone pivoted at s_i: the larger the lookahead, the larger the explored region. A key question in online planning is: what should the lookaheads $L_0, L_1, L_2, \ldots$ be for the agent to select good actions for the specified problem under the given constraints on planning time?

on, Bellman backup, attempts to iterate over all successors of every state under every action, which can be astronomically plentiful when exogenous events are to be reckoned with.

A promising strategy for solving complicated reward-based MDPs is to do some fraction of planning *online* (Figure 1.1), i.e., to determine an action for a state only if/when the agent ends up in that state. This approach's advantage lies in spending little resources on states that are never visited during policy execution. At the same time, computing part of the policy offline may be beneficial as well; the exact balance between online and offline planning depends on the situation.

During the online planning stage, choosing actions typically involves analyzing actions' consequences starting at the agent's current state for a certain number of steps ahead, called the *lookahead* (see Figure 1.1). Determining a good lookahead value for a given decision epoch or for a given problem in general is key to the success of an online planning method. On one hand, if the agent can get a reward only after executing at least N actions beginning at the current state (e.g., if the agent is playing a game that can end after at least N moves) but the agent only looks $L < N$ steps ahead, its action choice will be as good as random. On the other hand, picking a very large lookahead value may prevent the agent from evaluating its options properly within the specified time constraint and thus may also lead to nearly random action selection. Suitable lookahead values lie inbetween these two extremes, differing greatly from problem to problem, and even powerful planners based on Monte-Carlo Tree Search, e.g., PROST [49], suffer from the inability to determine this parameter automatically.

The contribution of this dissertation to the state of the art in solving reward-oriented MDPs with finite horizons and complex transition functions is a series of three algorithms that culminates in a top-performing, easily tunable solver for these problems:

- The algorithm that lays a theoretical foundation for the other two is LR^{2}TDP [54]. LR^{2}TDP is based on the strategy of *reverse iterative deepening*, using which it sequentially builds optimal policies for a given state for lookaheads $1, 2, \ldots, H$. Its key improvement upon the already successful iterative deepening approach, as implemented, e.g., in IDA* [62], is that the former obtains an optimal policy for lookahead L staring from a given state by augmenting the solution for lookahead $L-1$ obtained earlier, as opposed to discarding the solution for lookahead $L-1$ and building a policy for lookahead L from scratch. This gives LR^{2}TDP better speed and better anytime performance than that of its forerunner, LRTDP.

- By itself, reverse iterative deepening does not enable LR^{2}TDP to handle large branching factors caused by the presence of exogenous events. For this purpose, we introduce GLUTTON [54], a planner derived from LR^{2}TDP and our entry in IPPC-2011. GLUTTON endows LR^{2}TDP with optimizations that help achieve competitive performance on difficult problems with large branching factors: subsampling the transition function, separating out natural dynamics (a generalization of the notion of exogenous events), caching transition function samples, and using primitive cyclic policies as a fall-back solution. Thanks to these improvements, GLUTTON was IPPC-2011's second-best performer.

- Both LR^{2}TDP and GLUTTON do a lot of their planning *offline*. The last algorithm for finite-horizon MDPs that we propose, GOURMAND [59], is an *online* version of LR^{2}TDP. It incorporates many of the same optimizations as GLUTTON but plans as the agent travels through the state space, thereby saving valuable computational resources. GOURMAND's main innovation is its ability to determine good lookahead values automatically for the problem at hand, which addresses a major weakness that has plagued earlier online planning algorithms and allows GOURMAND to outperform the strongest of them, the IPPC-2011 winner PROST, on a large set of benchmarks.

1.2 Taking Disaster into Account

Many scenarios whose state and action space sizes are well within the reach of modern planning algorithms cannot be solved only because they do not fit the assumptions of any known MDP class. For example, currently there is just one extensively studied MDP type that can model goal-oriented settings, the so-called stochastic shortest path (SSP) MDPs. They come with two limitations:

- SSP MDPs must have a *proper* policy, one that can reach the goal from any state with probability 1.

- Every policy that does not lead to the goal with probability 1 from some state must incur an infinite cost from any such state.

Each of these restrictions "outlaws" realistic scenarios with very natural characteristics. The first one essentially confines SSP MDPs to problems with no catastrophic events that could prevent the agent from reaching the goal. Such catastrophic events are a possibility in many settings, e.g., robotics, and ignoring them is sometimes completely unacceptable. To make matters worse, verifying that a given problem has no dead ends can be nontrivial, further complicating the use of the SSP formalism. The requirement of policies accumulating an infinite cost if they do not reach the goal forbids the situations in which an agent is interested in the *probability* of reaching the goal, as opposed to the cost of doing so. These settings could be modeled by assigning the cost of 0 to each action and the reward of 1 for reaching the goal. However, under this reward function, policies that never lead to the goal have a cost of 0, which is unacceptable according to the SSP MDP definition. Researchers have attempted to correct SSP MDPs' shortcomings by developing planning formulations where reasoning about dead ends is possible (e.g., the aforementioned criterion of maximizing the probability of reaching the goal), but these models fail to cover many interesting cases and their mathematical properties are still relatively poorly understood.

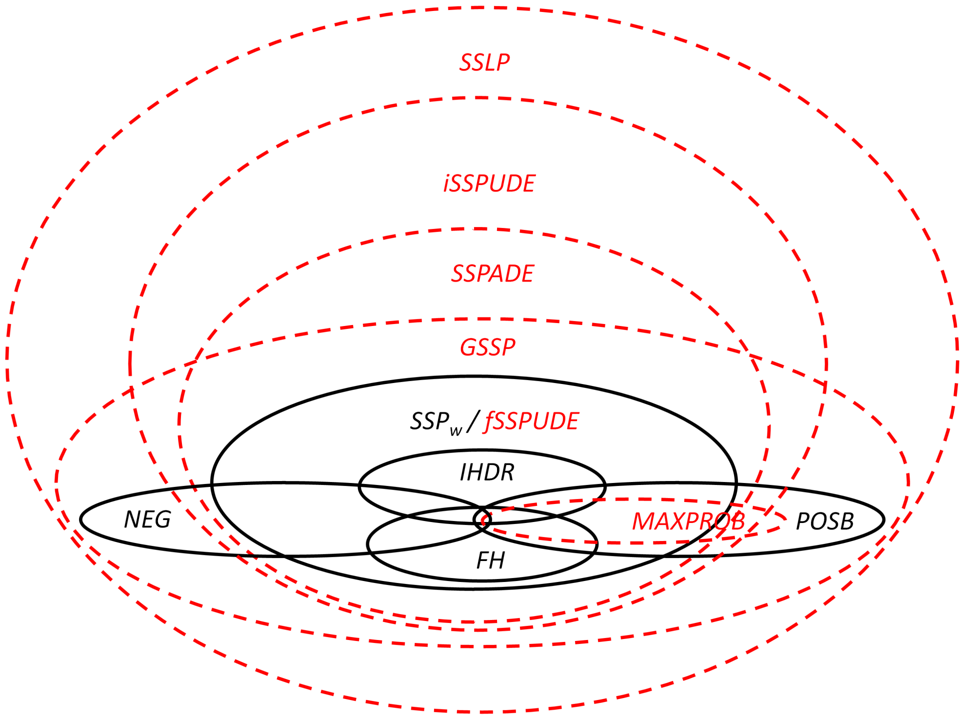


Figure 1.2: A Venn diagram showing the MDP classes introduced in this dissertation. The boundaries of the new MDP classes are marked with red dashed lines, and those of the previously known ones — with black solid lines. "*IHDR*" stands for "infinite-horizon discounted-reward MDPs", "*FH*" — for "finite-horizon MDPs", "*NEG*" — for "negative MDPs", "*POSB*" — for "positive-bounded MDPs", and "SSP_w" — for "stochastic shortest-path MDPs under the weak definition". The definitions of all these as well as of all the new MDP classes are given in the dissertation. All MDP classes in the diagram are assumed to have a known initial state (the corresponding subscript $_{s_0}$ has been dropped from their names to minimize clutter).

The final contribution of this dissertation is a set of SSP MDP extensions, shown in Figure 1.2, that gradually remove this model's restrictions, and a set algorithms for solving them optimally:

- Our exploration of SSP MDP extensions begins with **g**eneralized **SSP** MDPs ($GSSP_{s_0}$) [61], a class that allows a more general action reward model than *SSP*. We define the semantics of optimal solutions for GSSP_{s_0} problems and propose a heuristic search framework for them, called FRET (**F**ind, **R**evise, **E**liminate **T**raps). It turns out that the scenarios discussed above where an agent wants to maximize the probability of reaching the goal form a subclass of $GSSP_{s_0}$, which we call *MAXPROB* in this dissertation. Since *MAXPROB* is contained in $GSSP_{s_0}$, FRET can solve it as well and is, to our knowledge, the first efficient heuristic search framework to do so. To complete the investigation of *MAXPROB*'s mathematical properties, we derive a VI-like algorithm that can solve MAXPROB MDPs *independently of initialization* — previously, VI was known to yield optimal solutions to MAXPROB only if intialized strictly inadmissibly [80].

- Although *MAXPROB* forms the basis for our theory of goal-oriented MDPs with dead ends, by itself it evaluates policies in a rather crude manner, completely disregarding their *cost*. Our first *SSP* extension that takes costs into account as well is **SSP** MDPs with **a**voidable **d**ead **e**nds (*$SSPADE_{s_0}$*) [60]. SSPADE_{s_0} MDPs always include a known initial state and have well-defined easily computable optimal solutions if dead ends are present but avoidable from that state. Besides defining *$SSPADE_{s_0}$*, we describe the modifications required for the existing heuristic search algorithms to work correctly on these problems.

- The next two classes of MDPs with dead-end states that we introduce admit the existence of dead ends that cannot be avoided from the initial state with certainty no matter how hard the agent tries. Mathematically, there are two ways of dealing with such situations. The first is to assume that entering a dead end, while highly undesirable, has a finite "price". This is the approach we take in **SSP** MDPs with **u**navoidable **d**ead **e**nds and a **f**inite penalty (*$fSSPUDE_{s_0}$*) [60]. As with *$SSPADE_{s_0}$*, we show that existing heuristic search algorithms need only slight adjustments to work with *$fSSPUDE_{s_0}$*.

- The other way of treating dead ends is to view them as not only unavoidable but also as extorting an infinitely high cost if an agent hits one. We model such scenarios with **SSP** MDPs with **u**navoidable **d**ead **e**nds and an **i**nfinite penalty (*$iSSPUDE_{s_0}$*) [60, 96]. Mathematically, iSSPUDE_{s_0} MDPs represent the most difficult settings: since every policy in them reaches an infinite-cost state from the initial state, the expected cost of any policy at the initial state is also infinite. This makes *SSP*'s cost-minimization criterion uninformative: all policies look equally bad according to it. A previous attempt to take both policies' goal probability and cost into account assumed these criteria to be independent, and therefore constructed a Pareto set of non-dominated policies as a solution to this optimization problem [18]. Computing such a set is in the worst case intractable. Instead, we claim that a natural *primary* objective for scenarios with unavoidable infinitely costly dead ends is to maximize the probability of getting to the goal (i.e., to minimize the chance of getting into a lethal accident, a dead-end state). However, of all policies maximizing this chance we would prefer those that reach the goal in the least costly way (in expectation). This is exactly the multiobjective criterion we propose for *iSSPUDE*$_{s_0}$. Although solving *iSSPUDE*$_{s_0}$ is conceptually much more involved than handling the *SSP* extensions above, we devise an optimal tractable algorithm for it.

- Our work on *SSP* extensions culminates in the **s**tochastic **s**imple **l**ongest **p**ath MDPs (*SSLP*$_{s_0}$). The *SSLP*$_{s_0}$ definition imposes no restrictions whatsoever on action costs or the existence of proper policies. As such, *SSLP*$_{s_0}$ includes all of the aforementioned *SSP* extensions as special cases. Goal-oriented MDPs with unrestricted reward functions formalized by *SSLP*$_{s_0}$ generally have no optimal policy, but a lowest-cost *Markovian* policy for an SSLP$_{s_0}$ problem always exists. The algorithms for SSLP$_{s_0}$ MDPs proposed in this dissertation aim to find a policy of this kind. The task of discovering the best Markovian policy for an SSLP$_{s_0}$ MDP is a probabilistic counterpart of computing a simple longest path between two nodes in a graph, known to be NP-hard [89] (in fact, the latter is also a special case of the former). Thus, the most efficient algorithms for SSLP$_{s_0}$ MDPs are exponential in SSLP$_{s_0}$ problems' *flat* representation unless $P = NP$.

1.3 New Methods for New Models

Solving the newly introduced MDPs with dead-end states could be made much more efficient if we knew *which* states are dead ends. This information is usually not available in advance, so an MDP solver may have to spend considerable resources identifying them. Moreover, this computational effort is largely wasted, since, by definition, reaching the goal from a dead end is impossible, making careful action selection for such states pointless. Thus, introducing classes of problems with dead ends brings with it a new computational challenge — algorithms for these MDPs need an efficient mechanism for pinpointing states for which looking for a sophisticated policy is not worth it.

Fortunately, such a mechanism, SIXTHSENSE, exists, being enabled by the state abstractions introduced in the first part of this dissertation. In this sense, our work provides a complete set of tools for problems with dead ends: it introduces the MDP classes for modeling these problems, proposes fundamental optimal algorithms for solving them, and pioneers state-abstraction-based techniques for making these algorithms more efficient.

1.4 Dissertation Outline

The remainder of the manuscript is organized as follows. Chapter 2 surveys the background knowledge necessary for understanding the material in subsequent chapters. Chapters 3 and 4 are devoted to the new scalable techniques for solving MDPs. More concretely, Chapter 3 describes the approximation algorithms for goal-oriented MDPs, and Chapter 4 — for reward-oriented ones. Chapter 5 introduces the dissertation's second main contribution, the classes of goal-oriented MDPs with dead ends and the algorithms for solving them. Each of the Chapters 3–5 also contains Preliminaries and Related Work sections discussing area-specific state of the art not covered in the Background chapter. Finally, Chapter 6 summarizes the thesis.

1.5 List of Publications

This dissertation is based on the material from following published works:

Conference Papers:

- Andrey Kolobov, Mausam, and Daniel S. Weld. *ReTrASE: Integrating Paradigms for Approximate Probabilistic Planning*. In *Proceedings of IJCAI 2009.*

- Andrey Kolobov, Mausam, and Daniel S. Weld. *Classical Planning in MDP Heuristics: with a Little Help from Generalization.* In *Proceedings of ICAPS 2010.*

- Andrey Kolobov, Mausam, and Daniel S. Weld. *SixthSense: Fast and Reliable Recognition of Dead Ends in MDPs.* In *Proceedings of AAAI 2010.*

- Andrey Kolobov, Mausam, Daniel S. Weld, and Hector Geffner *Heuristic Search for Generalized Stochastic Shortest Path MDPs.* In *Proceedings of ICAPS 2011.*

- Andrey Kolobov, Mausam, and Daniel S. Weld. *Towards Scalable MDP Algorithms. Extended abstract in Proceedings of IJCAI 2011.*

- Andrey Kolobov, Peng Dai, Mausam, and Daniel S. Weld. *Reverse Iterative Deepening for Finite-Horizon MDPs with Large Branching Factors*. In *Proceedings of ICAPS 2012*.

- Andrey Kolobov, Mausam, and Daniel S. Weld. *LRTDP vs. UCT for Online Probabilistic Planning*. In *Proceedings of AAAI 2012*.

- Andrey Kolobov, Mausam, and Daniel S. Weld. *A Theory of Goal-Oriented MDPs with Dead Ends*. In *Proceedings of UAI 2012*.

Journal Papers:

- Andrey Kolobov, Mausam, and Daniel S. Weld. *Discovering Hidden Structure in Factored MDPs*. In *Artificial Intelligence Journal, May 2012*.

Books:

- Mausam and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. *Morgan & Claypool Publishers, 2012*.

Chapter 2
BACKGROUND

This chapter reviews the mathematical foundations of MDPs and algorithms for solving them. It is intended primarily as reference material for the rest of the thesis, with other parts of the dissertation containing pointers to specific definitions, theorems, and algorithms in it. For a significantly extended coverage of this chapter' topics please refer to [71]. Here, we present only those parts of it that are directly relevant to understanding the dissertation's contributions.

2.1 Markov Decision Processes

We begin the survey of the fundamentals with the notion of a *Markov Decision Process (MDP).* We describe the most common MDP classes studied in AI, define what it means to solve them, and discuss their computational complexity.

2.1.1 Definition

In its broadest sense, the MDP concept encompasses an extremely large variety of scenarios. However, such generality comes at a cost, since it provides too little structure for deriving efficient MDP solution techniques. Instead, practitioners in AI and other areas have come up with more specialized MDP classes by adding restrictions to the basic MDP notion. This dissertation, too, considers a specific subset of MDP models, whose traits are captured in the following definition:

Definition 2.1. ***Finite Discrete-Time Fully Observable Markov Decision Process.*** *A finite discrete-time fully observable MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{D}, \mathcal{T}, \mathcal{R} \rangle$, where:*

- *$\mathcal{S}$ is the finite nonempty set of all possible states of the system, also called the state space;*

- *$\mathcal{A}$ is the finite nonempty set of all actions an agent can take;*

- *$\mathcal{D}$ is a nonempty finite or infinite sequence of the natural numbers of the form $(1, 2, 3, \dots, H)$ or $(1, 2, 3, \dots)$ respectively, denoting the decision epochs, also called time steps, at which actions need to be taken;*

- *$\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{D} \to [0, 1]$ is a transition function, a mapping specifying the probability $\mathcal{T}(s_1, a, s_2, t)$ of going to state s_2 if action a is executed when the agent is in state s_1 at time step t; for any $s \in \mathcal{S}, a \in \mathcal{A}, t \in \mathcal{D}$, a transition function must obey $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s', t) = 1$;*

- *$\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{D} \to \mathbb{R}$ is a reward function that gives a finite numeric reward value $\mathcal{R}(s_1, a, s_2, t)$ obtained when the system goes from state s_1 to state s_2 as a result of executing action a at time step t. If for most inputs this mapping is negative, it is more appropriately viewed as a cost function.* ♣

Since we will almost exclusively discuss finite discrete-time fully observable MDPs as opposed to any others, we will refer to them simply as "MDPs" in the rest of the dissertation.

Notice that mappings $\mathcal{T}$ and $\mathcal{R}$ may depend on the number of time steps that have passed since the beginning of the process. However, neither $\mathcal{T}$ nor $\mathcal{R}$ depends on the sequence of states the system has gone through so far, instead depending only on the state the system is in currently. This is called the *first-order Markov assumption* and is reflected in the MDPs' name.

While not explicitly stated in the MDP definition, throughout the dissertation we also make another critical assumption. We assume that all parts of the MDP model describing the scenario at hand are known and available to the MDP solver. This view of MDPs is characteristic of planning under uncertainty as a research area. It is different from the stance taken in reinforcement learning [93], where the solver may not have access to certain MDP components (usually, the transition and the reward function).

2.1.2 Solutions of an MDP

Intuitively, solving an MDP means finding a way of choosing actions to control it. The characteristic of MDPs that distinguishes them from deterministic (*classical*) planning problems is that the former's actions typically have several possible outcomes. An agent cannot pick a state to which it will transition when it uses a particular action, since actions' outcomes are nondeterministic. Therefore, after executing any fixed number of actions, the agent may end up in one of many states. For instance, upon making a few moves with its manipulator, the robot may either end up holding an object it intended to pick up or, due to noise in its manipulator's motors, in a state where the object was inadvertently hit and destroyed. Thus, to be robust, our method of picking actions should enable an agent to decide on an action no matter which state the agent is in. What we need is a global *policy*, a rule for action selection that works in any state.

What information might such a rule use? The knowledge of the current state can clearly be very useful. In general, however, an agent's decisions may depend on the entire sequence of states it has been through so far, as well as the sequence of actions it has chosen up to the present time step, i.e., the entire *execution history*.

Definition 2.2. ***Execution History.*** *An execution history of an MDP up to time step $t \in \mathcal{D}$ is a sequence $h_t = ((s_1, a_1), \ldots, (s_{t-1}, a_{t-1}), s_t)$ of pairs of states the agent has visited and actions the agent has chosen in those states for all time steps t' s.t. $1 \leq t' \leq t - 1$, plus the state visited at time step t.* ♣

We denote the set of all possible execution histories up to decision epoch t as $\mathcal{H}_t$, and let $\mathcal{H} = \cup_{t \in \mathcal{D}} \mathcal{H}_t$.

Note also that a rule for selecting actions need not be deterministic. For instance, when faced with a choice of several equally good actions, it can be beneficial to pick one of them at random in order to avoid a bias. Thus, in the most general form, a solution policy for an MDP may be not only history-dependent but also randomized.

Definition 2.3. ***History-Dependent Policy.*** *A randomized history-dependent policy for an MDP is a probability distribution $\pi : \mathcal{H} \times \mathcal{A} \to [0, 1]$ that assigns to action $a \in \mathcal{A}$ a probability $\pi(h_t, a)$ of choosing it for execution at the current time step t if the execution history up to t is $h_t \in \mathcal{H}$. A deterministic history-dependent policy is a mapping $\pi : \mathcal{H} \to \mathcal{A}$ that assigns to each $h_t \in \mathcal{H}$ an action $a \in \mathcal{A}$ to be executed at the current time step t if h_t is the execution history up to t.* ♣

It is easy to see that a deterministic history-dependent policy is a randomized policy that, for every history, assigns the entire probability mass to a single action. Accordingly, the deterministic policy notation $\pi(h_t) = a$ is just a shorthand for the randomized policy notation $\pi(h_t, a) = 1$.

While very general, this definition suggests that many MDP solutions may be very hard to compute and represent. Indeed, a history-dependent policy must provide a distribution over actions for every possible history. If the number of time steps $|\mathcal{D}|$ in an MDP is infinite, its number of possible histories is infinite as well. Thus, barring special cases, solving an MDP seemingly amounts to computing a function over an infinite number of inputs. Even when $\mathcal{D}$ is a finite set, the number of histories, and hence the maximum size of an MDP solution, although finite, may grow exponentially in $|\mathcal{D}|$.

Due to the difficulties of dealing with arbitrary history-dependent policies, all algorithms presented in this dissertation aim to find more compact MDP solutions in the form of *Markovian policies*. Nonetheless, it is important to keep in mind that in general disregarding history-dependent policies may mean losing solution optimality, and wherever possible we explicitly prove that the MDP classes we are introducing have at least one optimal solution that is Markovian.

Definition 2.4. ***Markovian Policy.*** *A randomized (deterministic) history-dependent policy $\pi : \mathcal{H} \times \mathcal{A} \to [0, 1]$ ($\pi : \mathcal{H} \to \mathcal{A}$) is Markovian if for any two histories $h_{s,t}$ and $h'_{s,t}$, both of which end in the same state s at the same time step t, and for any action a, $\pi(h_{s,t}, a) = \pi(h'_{s,t}, a)$.* ♣

In other words, the choice of an action under a Markovian policy depends only on the current state and time step. To stress this fact, we will denote probabilistic Markovian policies as functions $\pi : \mathcal{S} \times \mathcal{D} \times \mathcal{A} \to [0, 1]$ and deterministic ones as $\pi : \mathcal{S} \times \mathcal{D} \to \mathcal{A}$.

Fortunately, disregarding non-Markovian policies is rarely a serious limitation. Typically, we are not just interested in finding *a* policy for an MDP. Rather, we would like to find a "good" policy, one that optimizes (or nearly optimizes) some objective function. As we will see shortly, for several important types of MDPs and objective functions, at least one optimal solution *is necessarily* Markovian. Thus, by restricting attention only to Markovian policies we are not foregoing the opportunity to solve these MDPs optimally.

Dropping non-Markovian history-dependent policies from consideration makes the task of solving an MDP much easier, as it entails deciding on a way to behave "merely" for every state and time step. In particular, if $\mathcal{D}$ is finite, the size of a policy specification is at most linear in $|\mathcal{D}|$. Otherwise, however, the policy description size may still be infinite. To address this, for MDPs with an infinite number of steps we narrow down the class of solutions even further by concentrating only on *stationary* Markovian policies.

Definition 2.5. ***Stationary Markovian Policy.*** *A randomized (deterministic) Markovian policy* $\pi : \mathcal{S} \times \mathcal{D} \times \mathcal{A} \to [0, 1]$ *(*$\pi : \mathcal{S} \times \mathcal{D} \to \mathcal{A}$*) is stationary if for any state* s*, action* a*, and two time steps* t_1 *and* t_2*,* $\pi(s, t_1, a) = \pi(s, t_2, a)$ *(*$\pi(s, t_1) = a$ *if and only if* $\pi(s, t_2) = a$*), i.e.,* π *does not depend on time.* ♣

Since the time step plays no role in dictating actions in stationary Markovian policies, we will denote probabilistic stationary Markovian policies as functions $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ and deterministic ones as $\pi : \mathcal{S} \to \mathcal{A}$. Stationary solutions look feasible to find — they require constructing an action distribution for every state and hence have finite size for any MDP with a finite state space. However, they again raise the concern of whether we are missing any important MDP solutions by tying ourselves only to the stationary ones. As with general Markovian policies, for most practically interesting MDPs with an infinite number of time steps this is not an issue, because they have at least one best solution that is stationary.

In the discussion so far, we have loosely referred to policies as being "best" and "optimal", and now turn to defining the notion of policy quality precisely. When executing a policy, i.e., applying actions recommended by it in various states, we can expect to get associated rewards. Therefore, it makes intuitive sense to prefer a policy that controls the MDP in a way that maximizes some *utility function* of collected rewards. For now, we intentionally leave this utility function unspecified and first formalize the concept of a policy's value.

Definition 2.6. ***Value Function.*** *A history-dependent value function is a mapping* $V : \mathcal{H} \to [-\infty, \infty]$*. A Markovian value function is a mapping* $V : \mathcal{S} \times \mathcal{D} \to [-\infty, \infty]$*. A stationary*

Markovian value function is a mapping $V : \mathcal{S} \to [-\infty, \infty]$. ♣

The Markovian value function notation $V : \mathcal{S} \times \mathcal{D} \to [-\infty, \infty]$ is just syntactic sugar for a history-dependent value function $V : \mathcal{H} \to [-\infty, \infty]$, that, for all pairs of histories $h_{s,t}$ and $h'_{s,t}$ that terminate at the same state at the same time, has $V(h_{s,t}) = V(h'_{s,t})$. In other words, for such a history-dependent value function, $V(s,t) = V(h_{s,t})$ for all policies $h_{s,t}$. Analogously, if a Markovian policy has $V(s,t) = V(s,t')$ for states s and for all pairs of time steps t, t', then $V(s)$ is the value denoting $V(s,t)$ for any time step t. We will use $V(s,t)$ and $V(s)$ as the shorthand notation for a value function wherever appropriate.

Definition 2.7. ***The Value Function of a Policy.*** *Let $h_{s,t}$ be a history that terminates at state s and time t. Let $R_{t'}^{\pi_{h_{s,t}}}$ be random variables for the amount of reward obtained in an MDP as a result of executing policy π starting in state s for all time steps t' s.t. $t \leq t' \leq |\mathcal{D}|$ if the MDP ended up in state s at time t via history $h_{s,t}$. The value function $V^\pi : \mathcal{H} \to [-\infty, \infty]$ of a history-dependent policy π is a utility function u of the reward sequence $R_t^{\pi_{h_{s,t}}}, R_{t+1}^{\pi_{h_{s,t}}}, \ldots$ that one can accumulate by executing π at time steps $t, t+1, \ldots$ after history $h_{s,t}$. Mathematically, $V^\pi(h_{s,t}) = u(R_t^{\pi_{h_{s,t}}}, R_{t+1}^{\pi_{h_{s,t}}}, \ldots)$.* ♣

This definition simply says that the value of a policy π is the amount of utility we can expect from executing π starting in a given situation, whatever we choose our utility to be. The notation for the policy value function is simplified for Markovian and stationary Markovian policies π. In the former case, $V^\pi(s,t) = u(R_t^{\pi_{s,t}}, R_{t+1}^{\pi_{s,t}}, \ldots)$, and in the latter case $V^\pi(s) = u(R_t^{\pi_s}, R_{t+1}^{\pi_s}, \ldots)$. As already mentioned, for most MDP classes studied in this dissertation, the optimal solution is a stationary Markovian policy, so our algorithms concentrate on this type of policy when solving MDPs, and we will be usually use the $V^\pi(s)$ notation.

The above definition of a policy's value finally allows us to describe an *optimal MDP solution*.

Definition 2.8. ***Optimal MDP Solution.*** *An optimal solution to an MDP is a policy π^* s.t. the value function of π^*, denoted as V^* and called the optimal value function, dominates the value functions* *of all other policies for all histories h_t for all time steps t. Mathematically, for all $h_t \in \mathcal{H}$, for all $t \in \mathcal{D}$ and any π, V^* must satisfy $V^*(h_t) \geq V^\pi(h_t)$.* ♣

In other words, given an optimality criterion (i.e., a measure of how good a policy is, as determined by the utility function u in Definition 2.7), an optimal MDP solution is a policy π^* that is at least as good as any other policy in every situation, according to that criterion. *An optimal policy is one that maximizes a utility of rewards.*

2.1.3 Solution Existence under Expected Linear Additive Utility

Definition 2.8 contains a caveat that concerns the possibility of comparing value functions for two different policies. Each V^π is, in effect, a vector in a Euclidean space — the dimensions correspond to histories/states with values V^π assigns to them. Viewed in this way, comparing quality of policies is equivalent to comparing vectors in $\mathbb{R}^m$ componentwise. Note, however, that vectors in $\mathbb{R}^m$ with $m > 1$, unlike points in $\mathbb{R}$, are not necessarily componentwise comparable, and for an arbitrary u the optimal value function need not exist, because no π's value function may dominate the values of all other policies everywhere.

Fortunately, there is a natural utility function that ensures the existence of an optimal policy:

Definition 2.9. ***Expected Linear Additive Utility.*** *An expected linear additive utility function is a function $u(R_t, R_{t+1}, \ldots) = \mathbb{E}[\sum_{t'=t}^{|\mathcal{D}|} \gamma^{t'-t} R_{t'}] = \mathbb{E}[\sum_{t'=0}^{|\mathcal{D}|-t} \gamma^{t'} R_{t'+t}]$ that computes the utility of a reward sequence as the expected sum of (possibly discounted) rewards in this sequence, where $\gamma \geq 0$ is the discount factor.* ♣

This definition implies that a policy is as good as the amount of discounted reward it is expected to yield. Setting $\gamma = 1$ expresses indifference of the agent to the time when a particular reward arrives. Setting it to a value $0 \leq \gamma < 1$ reflects various degrees of preference to rewards earned sooner. This is very useful, for instance, for modeling an agent's attitude to monetary rewards. The agent may value the money it gets today more than the same amount of money it could get in a month, because today's money can be invested and yield extra income in a month's time.

Expected linear additive utility is not always the most appropriate measure of a policy's quality. In particular, it assumes the agent to be *risk-neutral*, i.e., oblivious of the variance in the rewards yielded by a policy. As a concrete example of risk-neutral behavior, suppose the agent has a choice of either getting a million dollars or playing the following game. The agent should flip a fair coin, and if the coin comes up heads, the agent gets two million dollars; otherwise, the agent gets nothing. The two options can be interpreted as two policies, both yielding the expected reward of one million dollars. The expected linear additive utility model gives the agent no reason to prefer one policy over the other, since their expected payoff is the same. In reality, however, many if not most people in such circumstances would tend to be *risk-averse* and select the option with lower variance — just take one million dollars. Although expected linear additive utility does not capture these nuances, it is still a convenient indicator of policy quality in many cases.

The importance of expected linear additive utility is due to the fact that letting $V^{\pi}(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{|\mathcal{D}|-t} \gamma^{t'} R_{t'+t}^{\pi_{h_{s,t}}}]$ guarantees a very important MDP property, informally stated as follows [80]:

The Optimality Principle. *If every policy's quality can be measured by this policy's expected linear additive utility, there exists a policy that is optimal at every time step.*

In effect, it says that if the expected utility of every policy is well-defined, then there is a policy that maximizes this utility in every situation.

The statement of the Optimality Principle has two subtle points. First, its claim is valid only if "every policy's quality can be measured by the policy's expected linear additive utility." When does this premise fail to hold? Imagine, for example, an MDP with an infinite $\mathcal{D}$, $\gamma = 1$, two states, s and s', and an action a s.t. $\mathcal{T}(s, a, s', t) = 1.0$, $\mathcal{T}(s', a, s, t) = 1.0$, $\mathcal{R}(s, a, s', t) = 1$, and $\mathcal{R}(s', a, s, t) = -1$ for all $t \in \mathcal{D}$. In other words, the agent can only travel in a loop between states s and s'. In this MDP, for both states the expected sum of rewards keeps oscillating (between 1 and 0 for s and between -1 and 0 for s'), never converging to any finite value nor diverging to infinity. Second, crucially, *optimal policies are not necessarily unique*.

At the same time, as stated, the Optimality Principle may seem more informative than it actually is. Since we have not imposed any restrictions either on the γ parameter, on the sequence of the reward random variables $R_{t'+t}$, or on the number of time steps $|\mathcal{D}|$, the expectation $\mathbb{E}[\sum_{t'=0}^{|\mathcal{D}|-t} \gamma^{t'} R_{t'+t}]$

may be infinite. This may happen, for example, when the reward of any action in any state at any time step is at least $\epsilon > 0$, γ is at least 1, and the number of time steps is infinite. As a consequence, *all* policies in an MDP could be optimal under the expected linear additive utility criterion, since they all might have infinite values in all the states and hence would be indistinguishable from each other in terms of quality. In the next subsections, we discuss the three most widely studied MDP classes in AI — finite-horizon, infinite-horizon discounted-reward, and stochastic shortest-path MDPs — whose definitions can be regarded as attempts to restrict Definition 2.1 in order to enforce the finiteness of every candidate policy's value function. Besides these, there is another major type of fully observable probabilistic planning problems, the infinite-horizon average-reward MDPs, whose properties have been extensively studied in operations research [80]. However, it has not been used much in AI and is not essential to understanding the material in this dissertation, so we do not discuss it here.

2.1.4 Finite-Horizon MDPs

Perhaps the easiest way to make sure that expected linear additive utility $\mathbb{E}[\sum_{t'=0}^{|\mathcal{D}|-t} \gamma^{t'} R_{t'+t}]$ is finite for any conceivable sequence of random reward variables in a given MDP is to limit the MDP to a finite number of time steps. In this case, the summation terminates after a finite number of terms $|\mathcal{D}| = H$, called the *horizon*, and the MDP is called a *finite-horizon* MDP.

Definition 2.10. ***Finite-Horizon MDP.*** *A finite-horizon (FH) MDP is an MDP as described in Definition 2.1 with a finite number of time steps, i.e., with $|\mathcal{D}| = H < \infty$.* ♣

In most cases where finite-horizon MDPs are used, γ is set to 1, so the value of a policy becomes the expected total sum of rewards it yields.

One example [86] of a scenario appropriately modeled as a finite-horizon MDP is where the agent is trying to teach a set of skills to a student over H lessons (time steps). The agent can devote a lesson to teaching a new topic, giving a test to check the student's proficiency, or providing hints/additional explanations about a previously taught topic — this is the agent's action set. The agent gets rewarded if the student does well on the tests. The probability of a student doing well on them depends on her current proficiency level, which, in turn, depends on the amount of explanations and hints she has received from the agent. Thus, the agent's objective is to plan out the available lessons so as to teach the student well and have time to verify the student's knowledge via exams.

The Optimality Principle for finite-horizon MDPs can be restated in a precise form as follows [80]:

Theorem 2.1. ***The Optimality Principle for Finite-Horizon MDPs.*** *For a finite-horizon MDP with* $|\mathcal{D}| = H < \infty$*, define* $V^\pi(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{H-t} R_{t'+t}^{\pi_{h_{s,t}}}]$ *for all* $1 \leq t \leq H$*, and* $V^\pi(h_{s,H+1}) = 0$ *. Then the optimal value function* V^* *for this MDP exists, is Markovian, and satisfies, for all* $s \in \mathcal{S}$ *and* $1 \leq t \leq H$*,*

$$V^*(s,t) = \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s',t)[\mathcal{R}(s,a,s',t) + V^*(s',t+1)] \right]. \tag{2.1}$$

Moreover, at least one optimal policy π^* *corresponding to the optimal value function is deterministic Markovian and satisfies, for all* $s \in \mathcal{S}$ *and* $1 \leq t \leq H$*,*

$$\pi^*(s,t) = \operatorname*{argmax}_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s',t)[\mathcal{R}(s,a,s',t) + V^*(s',t+1)] \right]. \tag{2.2}$$

$\diamondsuit$

This statement of the Optimality Principle is more concrete than the previous one, as it postulates the existence of *deterministic Markovian* optimal policies for finite-horizon MDPs. Their optimal value function dominates all other value functions but satisfies the simpler Equation 2.1. Thus, finite-horizon MDPs can be solved by restricting our attention to Markovian deterministic policies without sacrificing solution quality.

2.1.5 Infinite-Horizon Discounted-Reward MDPs

Although finite-horizon MDPs have simple mathematical properties, this model is quite limited. In many scenarios, the reward is accumulated over an infinite (or virtually infinite) sequence of time steps. To handle such problems, we need a way to ensure the convergence of an infinite weighted expected reward series.

In fact, with our current definition of the reward and transition functions (Definition 2.1), infinite-horizon MDPs are hard even to write down, because the domain of both of these functions includes an infinite set of time steps $\mathcal{D}$. We circumvent this difficulty by working only with time-independent, or *stationary*, transitions and rewards.

Definition 2.11. ***Stationary Transition Function.*** *An MDP's transition function $\mathcal{T}$ is stationary if for any states $s_1, s_2 \in \mathcal{S}$ and action $a \in \mathcal{A}$, the value $\mathcal{T}(s_1, a, s_2, t)$ does not depend on t, i.e., $\mathcal{T}(s_1, a, s_2, t) = \mathcal{T}(s_1, a, s_2, t')$ for any $t, t' \in \mathcal{D}$.* ♣

Definition 2.12. ***Stationary Reward Function.*** *An MDP's reward function $\mathcal{R}$ is stationary if for any states $s_1, s_2 \in \mathcal{S}$ and action $a \in \mathcal{A}$, the value $\mathcal{R}(s_1, a, s_2, t)$ does not depend on t, i.e., $\mathcal{R}(s_1, a, s_2, t) = \mathcal{R}(s_1, a, s_2, t')$ for any $t, t' \in \mathcal{D}$.* ♣

Since stationary $\mathcal{T}$ and $\mathcal{R}$ are constant with respect to $\mathcal{D}$, we will refer to them more concisely as $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. When the transition function is stationary, we will call s' an *outcome* of a in s whenever $\mathcal{T}(s, a, s') > 0$. We call MDPs with stationary transition and reward functions *stationary MDPs.* All MDPs with infinite $\mathcal{D}$ considered in this dissertation are assumed to be stationary.

Returning to the issue of an infinite reward series expectation, an easy way to force it to converge *for a stationary MDP* is to require that $\{\gamma^i\}_{i=0}^{\infty}$ form a decreasing geometric sequence by setting γ to a value $0 \leq \gamma < 1$, as reflected in the definition of infinite-horizon discounted-reward MDPs:

Definition 2.13. ***Infinite-Horizon Discounted-Reward MDP.*** *An infinite-horizon discounted-reward (IHDR) MDP is a stationary MDP as described in Definition 2.1, in which $\mathcal{D}$ is infinite and the value* *of a policy is defined as* $V^{\pi}(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'+t}^{\pi_{h_{s,t}}}]$*, where the discount factor* γ *is a model parameter restricted to be* $0 \leq \gamma < 1$. ♣

Besides having the pleasing mathematical property of a bounded value function, discounted-reward MDPs also have a straightforward interpretation. They model problems in which the agent gravitates toward policies yielding large rewards in the near future rather than policies yielding similar rewards but in a more distant future. At the same time, any particular choice of γ is usually hard to justify.

As in the case with finite-horizon MDPs, the Optimality Principle can be specialized to infinite-horizon discounted-reward MDPs, but in an even stronger form due to the assumptions of stationarity [80]:

Theorem 2.2. ***The Optimality Principle for Infinite-Horizon MDPs.*** *For an infinite-horizon discounted-reward MDP with discount factor* γ *s.t.* $0 \leq \gamma < 1$*, define* $V^{\pi}(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{\infty} \gamma^{t'} R_{t'+t}^{\pi_{h_{s,t}}}]$*. Then the optimal value function* V^* *for this MDP exists, is stationary Markovian, and satisfies, for all* $s \in \mathcal{S}$*,*

$$V^*(s) = \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^*(s')] \right]. \tag{2.3}$$

Moreover, at least one optimal policy π^* *corresponding to the optimal value function is deterministic stationary Markovian and satisfies, for all* $s \in \mathcal{S}$*,*

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')[\mathcal{R}(s, a, s') + \gamma V^*(s')] \right]. \tag{2.4}$$

♢

2.1.6 Stochastic Shortest-Path MDPs

FH and IHDR MDPs are suitable for describing agents whose primary objective is collecting reward. This is the setting of many problems in industrial engineering, e.g., inventory management [80], but in AI, researchers have also studied a different type of scenarios. In them, the agent is trying to reach a goal state while minimizing the expected cost of doing so. Fairly innocuous at the first glance, this criterion turns out to be tricky to formalize. The intuitive reason for this is that the objectives of goal attainment and cost minimization (which is the equivalent to maximizing reward) are conflicting: policies that do not lead to a goal can sometimes be more "profitable" for the agent than those that do. Therefore, in the goal-oriented MDP class we define shortly, we will focus only on those policies that are *guaranteed* to eventually lead an agent to a goal state no matter at which state the agent starts:

Definition 2.14. ***Proper policy.*** *For a given stationary MDP, let $\mathcal{G} \subseteq \mathcal{S}$ be a set of goal states s.t. for each $s_g \in \mathcal{G}$ and all $a \in \mathcal{A}$, $\mathcal{T}(s_g, a, s_g) = 1$ and $\mathcal{R}(s_g, a, s_g) = 0$. Let h_s be an execution history that terminates at state s. For a given set $S' \subseteq \mathcal{S}$, let $P_t^\pi(h_s, S')$ be the probability that after execution history h_s, the agent transitions to some state in S' within t time steps if it follows policy π. A policy π is called proper at state s if $P_t^\pi(h_s, \mathcal{G}) = 1$ for all histories $h_s \in \mathcal{H}$ that terminate at s. If for some h_s, $\lim_{t\to\infty} P_t^\pi(h_s, \mathcal{G}) < 1$, π is called improper at state s. A policy π is called proper if it is proper at all states $s \in \mathcal{S}$. Otherwise, it is called improper.* ♣

Without loss of generality, we can assume that there is only one goal state — if there are more, we can add a special state with actions leading to it from all the goal states and thus make this new special state the de-facto unique goal. Therefore, as a shorthand, in the rest of this dissertation we will sometimes say that a policy reaches *the goal*, meaning that it reaches some state in $\mathcal{G}$. We will say that a policy reaches the goal with probability 1 about a policy that is guaranteed to eventually reach some state in $\mathcal{G}$, although possibly not the same state in every execution.

The following definition of a goal-oriented MDP class ensures that at least one proper policy not only exists but is also at least as attractive as any other in terms of expected utility:

Definition 2.15. ***Stochastic Shortest-Path MDP.*** *(Weak definition.) A stochastic shortest-path (SSP) MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G} \rangle$ where:*

- *$\mathcal{S}$ is the finite set of all possible states of the system,*
- *$\mathcal{A}$ is the finite set of all actions an agent can take,*
- *$\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a stationary transition function specifying the probability $\mathcal{T}(s_1, a, s_2)$ of going to state s_2 whenever action a is executed when the system is in state s_1,*
- *$\mathcal{C} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, \infty)$ is a stationary cost function that gives a finite strictly positive cost $\mathcal{C}(s_1, a, s_2) > 0$ incurred whenever the system goes from state s_1 to state s_2 as a result of executing action a, with the exception of transitions from a goal state, and*
- *$\mathcal{G} \subseteq \mathcal{S}$ is the set of all goal states s.t. for every $s_g \in \mathcal{G}$, for all $a \in \mathcal{A}$, and for all $s' \notin \mathcal{G}$, the transition and cost functions obey $\mathcal{T}(s_g, a, s_g) = 1$, $\mathcal{T}(s_g, a, s') = 0$, and $\mathcal{C}(s_g, a, s_g) = 0$,*

under the following condition:

- *There exists at least one proper policy.* ♣

Although superficially similar to the general MDP characterization (Definition 2.1), this definition has notable differences. Other than the standard stationarity requirements, note that instead of a reward function SSP MDPs have a cost function $\mathcal{C}$, since the notion of cost is more appropriate for scenarios modeled by SSP MDPs. Consequently, as we discuss shortly, solving an SSP MDP means finding an *expected-cost minimizing*, as opposed to a reward-maximizing, policy. Further, the set of decision epochs $\mathcal{D}$ has been omitted, since for every SSP MDP it is implicitly the set of all natural numbers. These differences are purely syntactic, as we can define $\mathcal{R} = -\mathcal{C}$ and go back to the reward-maximization-based formulation. A more fundamental distinction is the presence of a special set of (terminal) goal states, staying in which forever incurs no cost, and the requirement that SSP MDPs have at least one policy that reaches the goal with probability 1. Most importantly, according to this definition, the agent effectively has to pay a price for executing *every* action. Under such conditions, the longer an agent stays away from the goal state, the more the agent is likely to pay. In particular, if the agent uses an improper policy, it will never be able to reach the goal from some states, i.e., the total expected cost of that policy from some states will be infinite. Therefore, improper policies are always inferior to proper ones.

The Optimality Principle for SSP MDPs [5] holds in a slightly modified form that nonetheless postulates the existence of simple optimal policies, as for the infinite-horizon discounted-cost MDPs.

Theorem 2.3. ***The Optimality Principle for Stochastic Shortest-Path MDPs.*** *For an SSP MDP, define $V^{\pi}(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{\infty} R_{t'+t}^{\pi_{h_{s,t}}}]$. Then the optimal value function V^* for this MDP exists, is stationary Markovian, and satisfies, for all $s \in \mathcal{S}$,*

$$V^*(s) = \min_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')[\mathcal{C}(s, a, s') + V^*(s')] \right] \tag{2.5}$$

and, for all $s_g \in \mathcal{G}$, $V^*(s_g) = 0$. Moreover, at least one optimal policy π^* corresponding to the optimal value function is deterministic stationary Markovian and satisfies, for all $s \in \mathcal{S}$,

$$\pi^*(s) = \operatorname*{argmin}_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')[\mathcal{C}(s, a, s') + V^*(s')] \right]. \tag{2.6}$$

$\diamondsuit$

The largest differences from the previous versions of this principle are the replacement of maximizations with minimizations, the replacement of a reward function with a cost function, and the omitted γ factor, since it is fixed at 1. The following theorem, a corollary of several results from the SSP MDP theory [5], sheds more light on the semantics of this model.

Theorem 2.4. *In an SSP MDP, $V^*(s)$ is the smallest expected cost of getting from state s to the goal. Every optimal stationary deterministic Markovian policy in an SSP MDP is proper and minimizes the expected cost of getting to the goal for every state.* $\diamondsuit$

This theorem explains why the MDPs we are discussing are called stochastic shortest-path problems. It says that, under the SSP MDP definition conditions, all policies that optimize the expected undiscounted linear additive utility also optimize the expected cost of reaching the goal. Put differently, in expectation such policies are the "shortest" ways of reaching the goal from any given state.

In the AI literature (e.g., [5]), one also encounters another, broader definition of SSP MDPs:

Definition 2.16. ***Stochastic Shortest-Path MDP.*** *(Strong definition.) A stochastic shortest-path (SSP) MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G} \rangle$ where $\mathcal{S}$, $\mathcal{A}$, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$, $\mathcal{G} \subseteq \mathcal{S}$ are as in Definition 2.15, and $\mathcal{C} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a stationary real-valued cost function, under two conditions:*

- *There exists at least one proper policy,*
- *For every improper stationary deterministic Markovian policy π, for every $s \in \mathcal{S}$ where π is improper, $V^\pi(s) = \infty$.* $\clubsuit$

Theorem 2.4 holds for this SSP definition as well as for the weak one (Definition 2.15), so optimal solutions to SSP MDPs defined this way still include the stationary deterministic Markovian policies that minimize the expected cost of getting to the goal at every state.

The strong definition can be shown to be more general than the weak one. The former does not impose any local constraints on the cost function — its value can be any real number, positive or negative. However, it adds an additional constraint insisting that every improper policy has an infinite value in at least one state. The net effect of these requirements is the same; in both cases *some* proper policy for the SSP MDP is preferable to all improper ones. Thus, instead of imposing local constraints on the cost values, Definition 2.16 introduces a global one. On the positive side, this allows more choices for the cost function, and hence admits more MDPs than the weak definition. On the other hand, the weak definition provides an easier way to check whether an MDP is an SSP problem or not.

2.1.7 MDPs with an Initial State

Solutions to MDPs discussed up till now map every state in the state space to an action or a distribution over actions. We call such policies *complete*. In many settings, learning a complete optimal policy for an MDP is not needed. Instead, we may be interested in a policy that performs optimally starting from a specific state, called the *initial state* and denoted as s_0. If the initial state is known, it makes sense to calculate only a *partial policy* π_{s_0} *closed w.r.t.* s_0.

Definition 2.17. ***Partial Policy.*** *A Markovian policy* $\pi : \mathcal{S}' \times \mathcal{A} \to [0, 1]$ *over a set of states* $\mathcal{S}'$ *is* partial *if* $\mathcal{S}'$ *is a subset of but not necessarily equal to the whole state space* $\mathcal{S}$. ♣

Definition 2.18. ***Policy Closed with Respect to State*** s***.*** *A partial policy* $\pi_s : \mathcal{S}' \times \mathcal{A} \to [0, 1]$ *over a set of states* $\mathcal{S}'$ *is closed with respect to state* s *if any state* s' *reachable by* π_s *from* s *is contained in* $\mathcal{S}'$. ♣

Put differently, a policy closed w.r.t. a state s must specify an action for any state s' that can be reached via that policy from s. Such a policy can be viewed as a restriction of a complete policy π to a set of states $\mathcal{S}'$ iteratively constructed as follows:

- Let $\mathcal{S}' = \{s\}$.
- Add to $\mathcal{S}'$ all the states that π can reach from the states in $\mathcal{S}'$ in one time step.
- Repeat the above step until $\mathcal{S}'$ does not change.

Computing a closed partial policy can be easier than a complete one, because the former typically does not need to be specified over the entire state space, parts of which may not even be reachable from s_0. For the same reason, such a policy also takes less space to store. The advantages of closed partial MDP solutions have prompted research into variants of finite-horizon, infinite-horizon discounter-reward, and stochastic shortest-path problems where the initial state is assumed to be known. Below, we adapt the strong definition of SSP MDPs (Definition 2.16) to account for the knowledge of the initial state. The definitions of the other MDP classes can be modified analogously.

Definition 2.19. ***Stochastic Shortest-Path MDP with an Initial State.*** (Strong definition.) *An* SSP MDP with an initial state*, denoted as SSP*$_{s_0}$ *MDP, is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$*, where* $\mathcal{S}$*,* $\mathcal{A}$*,* $\mathcal{T}$*,* $\mathcal{C}$*, and* $\mathcal{G}$ *are as in the strong SSP MDP definition (2.16) and satisfy that definition's conditions, and* $s_0 \in \mathcal{S}$ *is the initial state where the execution of any policy for this MDP starts.* ♣

An optimal solution of an SSP$_{s_0}$ MDP is a partial policy $\pi^*_{s_0}$ closed with respect to s_0 whose value function satisfies $V^*(s_0) \leq V^{\pi'}(s_0)$ for any other policy π' closed w.r.t. s_0. The presence of the initial state does not prevent the main results for FH, IHDR, and SSP MDPs, including the Optimality Principle, from applying to FH$_{s_0}$, IHDR$_{s_0}$, and SSP$_{s_0}$ problems as well.

2.1.8 Relationships Among the Basic MDP Classes

Our discussion from Section 2.1.3 up to this point has touched upon several flavors of three fundamental MDP classes for which an optimal solution is both well-defined and guaranteed to exist — finite-horizon, infinite-horizon discounted-reward, and SSP MDPs. As we show in this subsection, SSP MDPs are, in fact, strictly more general than the other MDP types, as the following theorem states.

Theorem 2.5. *Denote the set of all finite-horizon MDPs as FH, the set of all infinite-horizon discounted-reward MDPs as IHDR, and the set of all SSP MDPs as SSP. The following statements hold [6]:*

$$FH \subset SSP$$

$$\textit{IHDR} \subset \textit{SSP}$$

Similarly, for versions of these classes with a known initial state,

$$\textit{FH}_{s_0} \subset \textit{SSP}_{s_0}$$

$$\textit{IHDR}_{s_0} \subset \textit{SSP}_{s_0}$$

$\diamondsuit$

We do not provide a formal proof of these facts, but outline its main idea. Our high-level approach is to show that each finite-horizon and infinite-horizon discounted-reward MDP can be compiled into an equivalent SSP MDP. Let us start with finite-horizon MDPs. Suppose we are given an FH MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{D}, \mathcal{T}, \mathcal{R} \rangle$. Replace the state space $\mathcal{S}$ of such an MDP with the set $\mathcal{S}' = \mathcal{S} \times \mathcal{D}$. This transformation makes the current decision epoch part of the state and is the key step in our proof. Replace $\mathcal{T}$ with a stationary transition function $\mathcal{T}'$ s.t. $\mathcal{T}'((s,t), a, (s', t+1)) = p$ whenever $\mathcal{T}(s, a, s') = p$. By similarly changing $\mathcal{R}$, construct a new reward function $\mathcal{R}'$ and replace $\mathcal{R}$ with a cost function $\mathcal{C} = -\mathcal{R}'$. Finally, assuming $|\mathcal{D}| = H < \infty$, let the set of goal states be $\mathcal{G} = \{(s, H) \mid s \in \mathcal{S}\}$. The tuple we just constructed, $\langle \mathcal{S}', \mathcal{A}, \mathcal{T}', \mathcal{C}, \mathcal{G} \rangle$, clearly satisfies the strong definition of SSP MDP.

Now suppose we are given an IHDR MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{D}, \mathcal{T}, \mathcal{R} \rangle$ with a discount factor γ. The main insight in transforming this MDP into an SSP problem is to replace $\mathcal{R}$ with $\mathcal{C} = -\mathcal{R}$ and to add a special goal state s_g to $\mathcal{S}$, where the system can transition at any time step with probability $1 - \gamma$ using any action from any state (the transition function $\mathcal{T}$ needs to be scaled so that the probabilities of all other outcomes sum to γ). Transitioning to s_g from any state using any action will incur the cost of 0. It is easy to verify that the new MDP conforms to the strong SSP MDP definition with $\mathcal{G} = \{s_g\}$, and it can be shown that every stationary deterministic Markovian policy in it is optimal if and only if it is optimal in the original IHDR MDP.

From a theoretical standpoint, Theorem 2.5 allows us to concentrate on developing algorithms for *SSP*, since *FH* and *IHDR* are merely its subclasses. In practice, however, since *FH* and *IHDR* are more specialized, some techniques that do not extend to *SSP* can handle them more efficiently than the algorithms applying to *SSP* as a whole. In particular, this is true of the approximation algorithms proposed this dissertation: those of them that target specifically FH and IHDR MDPs (Chapter 4) perform much better on these problems than the approaches designed for the SSP scenarios in general (Chapter 3).

Finally, we point out that the presence or absence of a known initial state in a stochastic shortest-path problem does not actually make a difference from the point of view of computational complexity:

Theorem 2.6. *SSP_{s_0} = SSP* $\diamondsuit$

Indeed, for every SSP$_{s_0}$ MDP, solving the same MDP but without an initial state provides a solution to the original SSP$_{s_0}$ MDP MDP. Conversely, given an SSP MDP with state space $\mathcal{S}$, we can construct its version with an initial state by adding a new state s_0 to $\mathcal{S}$ and making all actions have a possible transition from s_0 to any state in $\mathcal{S}$ with probability $1/|\mathcal{S}|$. Finding an optimal policy closed w.r.t. s_0 for the resulting problem necessarily yields a complete optimal policy for the SSP MDP we started with.

Why, then, do we need a separate class of goal-oriented MDPs with an initial state? The explanation is similar to the reason why algorithms for *SSP* do not necessarily perform well in practice on its *FH* and *IHDR* subclasses. Although *in the worst case* solving an SSP$_{s_0}$ MDP takes as much resources as solving an SSP MDP, many SSP$_{s_0}$ instances can be solved much more efficiently (from the empirical point of view) than their SSP equivalents, because the fraction of the state space reachable from s_0 (and hence even theoretically relevant to finding a solution closed w.r.t. s_0) can be much smaller than the state space as a whole. In short, although the knowledge of the initial state does not help always, in many cases it does, making it beneficial to design algorithms specifically tailored to the case when the initial state is known.

2.1.9 Factored MDPs

Although defining an MDP class takes just a few lines of text, describing an MDP instance can be much more cumbersome. The most conceptually straightforward way is to implicitly enumerate state and action spaces by specifying just the number of each and give (stationary) transition and reward functions as sets of matrices, one matrix of size $|\mathcal{S}|^2$ per action. This type of representation is called *atomic* or *flat*, and has two major drawbacks. First, it is practical only for relatively small MDPs: a flat MDP description scales quadratically in the number of states. Second, it does not explicitly convey the structure of the problem at hand, e.g., the sparseness of its transition matrices, that could otherwise let a solver perform some optimizations.

Factored Stochastic Shortest-Path MDPs

To avoid the drawbacks of the flat representation, one can instead specify each state as a combination of values of several *state variables* relevant to the problem at hand, i.e., *factor* the state space into constituent variables [16]. Doing so helps compactly describe both the state space itself and other MDP components. Below, we give a factored counterpart of the strong definition of SSP MDPs with an initial state; factored versions of FH, IHDR, and weak SSP MDPs, with and without the initial state, are defined similarly.

Definition 2.20. ***Factored SSP MDP with an initial state.*** *A factored SSP MDP with an initial state is a tuple $\langle \mathcal{X}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ obeying the conditions in Definition 2.16, where*

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ *is a set of state variables (sometimes also called features or domain variables) whose domains are sets $dom(X_1), \ldots, dom(X_n)$ respectively,*

- *The finite state space is represented as a set $\mathcal{S} = dom(X_1) \times \ldots \times dom(X_n)$,*

- *The finite set of all actions is $\mathcal{A}$,*

- *The transition function is represented as a mapping $\mathcal{T} : (dom(X_1) \times \ldots \times dom(X_n)) \times \mathcal{A} \times (dom(X_1) \times \ldots \times dom(X_n)) \rightarrow [0, 1]$ s.t. for any $s \in \mathcal{S}, a \in \mathcal{A}$, $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')$ equals either 0 or 1,*

- *The cost function is represented as a mapping $\mathcal{C} : (dom(X_1) \times \ldots \times dom(X_n)) \times \mathcal{A} \times (dom(X_1) \times \ldots \times dom(X_n)) \rightarrow \mathbb{R}$,*

- *The goal set $\mathcal{G}$ is represented as a set of states satisfying a specified logical formula over assignments of values to variables in $\mathcal{X}$, and*

- *The initial state s_0 is specified as a vector in $dom(X_1) \times \ldots \times dom(X_n)$.* ♣

Other than the state variable-oriented representation of the key MDP elements, this definition has another notable difference from the earlier definitions of MDP classes. Namely, it allows the transition function to sum to 0 for some state-action pairs. This lets us express the fact that certain actions are not *applicable* in certain states, thereby explicitly showing that some transition matrices are sparse. In a given state of a factored MDP, an agent may only choose among actions that are applicable there. In spite of this detail, all theoretical results that hold for the previously reviewed MDP types hold for their factored equivalents as well.

Definition 2.20 states what a factored MDP is at an abstract level, without saying how exactly each of its components, e.g., the transition function, should be described. In what follows we examine two approaches to characterizing factored MDPs. To make the discussion concrete, we introduce some additional terminology. We call an assignment of a value $x_i \in dom(X_i)$ to a variable X_i a *literal over X_i*. Without loss of generality, in this dissertation we assume the state variables of MDPs in question to be *binary*, i.e., have $dom(X_i) = \{\mathit{True}, \mathit{False}\}$ for all i. All finite discrete factored MDPs can be converted into this binary form. We call a literal that assigns the *True* value to its variable X_i a *positive* literal and denote it, with a slight abuse of notation, X_i, just like the variable itself. Similarly, we call a literal that assigns the *False* value a *negative* literal and denote it as $\neg X_i$.

PPDDL-style Representation

The PPDDL-style representation takes its name from the Probabilistic Planning Domain Definition Language [101] and reflects the semantics of MDP descriptions in it. Figure 3.1 in Chapter 3 shows an MDP described in PPDDL. In the PPDDL-style representation, an MDP's components have the following form:

- The set $\mathcal{X}$ is given as a collection of appropriately named variables with specified domains.

For a binary factored MDP, it may appear as follows:

$$\mathcal{X} = \{A : \{\mathit{True}, \mathit{False}\}, B : \{\mathit{True}, \mathit{False}\}, C : \{\mathit{True}, \mathit{False}\}, D : \{\mathit{True}, \mathit{False}\}\}$$

- The state space $\mathcal{S}$ is the set of all conjunctions of literals over all the state variables. Again assuming binary variables, an example of a state is $s = (A, \neg B, \neg C, D)$.
- Each action in the set $\mathcal{A}$ is a tuple of the form

$$\langle prec, \langle p_1, add_1, del_1\rangle, \cdots, \langle p_m, add_m, del_m\rangle\rangle$$

  s.t. all such tuples in $\mathcal{A}$ together specify the transition function $\mathcal{T}(s, a, s')$ as follows:

  - $prec$ is the action's *precondition* represented as a conjunction of literals. We interpret an action's precondition to say that the action may cause a transition only from those states that have all of the literals in the precondition. More formally, we say that the action is *applicable* only in states where the precondition *holds*.
  - $\langle p_1, add_1, del_1\rangle, \cdots, \langle p_m, add_m, del_m\rangle$ is the list of the action's *probabilistic outcomes* or *effects*. The description of an effect consists of the effect's probability p_i and two lists of changes the effect makes to the state. In particular, add_i, the *i-th add effect*, is the conjunction of positive literals that the i-th effect adds to the state description. Similarly, the *i-th delete effect* is the conjunction of negative literals that the i-th effect adds to the state description. Implicitly, inserting a positive literal removes the negative literal over the same variable, and vice versa. For each pair of an action's probabilistic outcomes, we assume that the effects are distinct, i.e., that either $add_i \neq add_j$ or $del_i \neq del_j$ or both.

    We interpret this specification as stating that if an action is applied in a state where its precondition holds, nature will "roll the dice" and select one of the action's effects according to the effects' probabilities. The chosen outcome will cause the agent to transition to a new state by making changes to the current state as dictated by the corresponding add and delete effect. Formally, $\mathcal{T}(s, a, s') = p_i$ whenever a is applicable in s and s' is the result of changing s with the i-th effect.

Example: Suppose we apply action $a = \langle A, \langle 0.3, B \wedge C, \bigwedge \emptyset \rangle, \langle 0.7, B, \neg A \wedge \neg D \rangle \rangle$ in state $s = (A, \neg B, \neg C, D)$, where $\bigwedge \emptyset$ stands for the empty conjunction. The action's precondition, the singleton conjunction A, holds in s, so the action is applicable in this state. Suppose that nature chooses effect $\langle 0.7, B, \neg A \wedge \neg D \rangle$. This effect causes a transition to state $s' = \neg A \wedge B \wedge \neg C \wedge \neg D$.

- The size of the cost function $\mathcal{C}(s, a, s')$ description is polynomial in $|\mathcal{X}|$, the number of state variables, and $|\mathcal{A}|$. As an example of such a specification, we could associate a *cost* attribute with each action $a \in \mathcal{A}$ and set $\mathit{cost}(a) = c_a$. The *cost* attribute's value c_a should be interpreted as the cost incurred by using a in any state s where a is applicable, independently of the state s' where the agent transitions as a result of applying a in s. $\mathcal{C}(s, a, s')$ is undefined whenever a is not applicable in s.

- The goal set $\mathcal{G}$ is implicitly given by a conjunction of literals over a subset of the variables in $\mathcal{X}$ that any goal state has to satisfy. For instance, if the goal conjunction is $A \wedge \neg C \wedge D$, then there are two goal states in our MDP, $(A, B, \neg C, D)$ and $(A, \neg B, \neg C, D)$.

- Finally, the initial state s_0 is specified by the set of all *positive* literals in that state. The literals over all other variables are assumed to be negative. For example, in an MDP with state variables A, B, C, and D, the set $\{A, C\}$ denotes the initial state $s_0 = (A, \neg B, C, \neg D)$.

A PPDDL-style description is convenient when actions change variables in a correlated manner, i.e., have *correlated effects* and just a few outcomes. For instance, under the aforementioned action $a = \langle A, \langle 0.3, B \wedge C, \bigwedge \emptyset \rangle, \langle 0.7, B, \neg A \wedge \neg D \rangle \rangle$, the sets of variables $\{B, C\}$ and $\{A, B, D\}$ always change their values together, and a has only two outcomes.

RDDL-style Representation

At the same time, for some MDPs, a PPDDL-style description is not the most convenient choice. E.g., consider the problem of managing a network of n servers, in which every running server has some probability of going down and every server that is down has some probability of restarting at every time step if the network administrator does not interfere. This setting is known as the *Sysadmin* scenario [39]. In a formal factored specification of Sysadmin, its state space could be given by a set of binary state variables $\{X_1, \ldots, X_n\}$, each variable indicating the status of the corresponding server. Thus, the state space size would be 2^n. We would have a *noop* action that applies in every state and determines what happens to each server at the next time step if the administrator does nothing. Note, however, that, since each server can go up or down independently from others (i.e., variables change values in an *uncorrelated* fashion), at each time step the system can transition from the current state to any of the 2^n states. Therefore, to write down a description of *noop*, we would need to specify 2^n outcomes — an enormous number even for small values of n. In general, such actions are common in scenarios where many objects evolve independently and simultaneously, as in Sysadmin. They also arise naturally when the MDP involves *exogenous events*, changes that the agent cannot control and that occur in parallel with those caused by the agent's own actions. Examples of exogenous events include natural cataclysms and actions of other agents.

An MDP like this is more compactly formulated as a Dynamic Bayesian Network (DBN) [26], a representation particularly well suited for describing actions with uncorrelated effects. One MDP description language that expresses an MDP as a DBN is RDDL (Relational Dynamic influence Diagram Language) [86], so we denote this representation as *RDDL-style*. Under this representation, for each action and each domain variable, an engineer needs to specify a conditional probability distribution (CPD) over the values of that variable in the next state if the action is executed. Suppose, for instance, we want to say that if the status of the i-th server is "up" at time step t and the administrator does not intervene, then with probability 0.5 it will remain "up" at the next time step; similarly, if it is currently "down," it will remain "down" with probability 0.9. Then the CPD $P(X_i^{t+1} = \text{``}up\text{''} | X_i^t = \text{``}up\text{''}, Action = noop) = 0.5$, $P(X_i^{t+1} = \text{``}down\text{''} | X_i^t = \text{``}down\text{''}, Action = noop) = 0.9$ fully characterizes the *noop*'s effect on variable X_i.

An advantage of the RDDL-style representation is that the size of each action's description, as illustrated by the above example, can be only polynomial in the number of domain variables in the cases when in the PPDDL-style representation it would be exponential. At the same time, expressing *correlated* effects with DBNs can be tedious. Thus, the PPDDL-style and RDDL-style representations are largely complementary, and the choice between them depends on the problem at hand.

2.1.10 Complexity of Solving MDPs

The computational complexity results presented in this section serve a motivation for a significant fraction of this dissertation's contributions, especially those presented in Chapter 3. When analyzing the computational complexity of a problem class, it is important to keep in mind that the complexity is defined in terms of the input size. For MDPs, the input size is the size of an MDP's description. We have seen two of ways of describing MDPs:

- By specifying them in a flat representation, i.e., by explicitly enumerating their state space, action space, etc., and
- By specifying them in a factored representation, i.e., in terms of a set of variables $X_1, \ldots, X_n$.

The key observation for the MDP complexity analysis is that for a given factored MDP, its explicitly enumerated state space is exponential in the number of state variables — if the number of state variables is $|\mathcal{X}|$, then its state space size is $2^{|\mathcal{X}|}$. Similarly, flat descriptions of the transition and reward/cost functions are exponential in the size of their factored counterparts. As we have already observed, a factored description can make MDPs much easier to write down. However, as the results below show, it makes them look difficult to solve with respect to their input length.

We begin by characterizing the complexity of solving MDPs in the flat representation.

Theorem 2.7. *Solving finite-horizon MDPs in the flat representation is P-hard. Solving infinite-horizon discounted-reward and SSP MDPs in the flat representation is P-complete [77].* $\diamondsuit$

Finite-horizon MDPs are not known to be in P, because solving them appears to require computing an action for each *augmented state* in the set $\mathcal{S} \times \mathcal{D}$, which, in turn, could be exponential in the size of S if $|\mathcal{D}| = 2^{|\mathcal{S}|}$. The above result also indicates that infinite-horizon and SSP MDPs are some of the hardest polynomially solvable problems. In particular, they likely cannot benefit significantly from parallelization [77].

Since casting MDPs into a factored representation drastically reduces their description size, factored MDPs belong to a different computational complexity class:

Theorem 2.8. *Factored finite-horizon, infinite-horizon discounted-reward, and SSP MDPs are EXPTIME-complete [66, 35].* $\diamondsuit$

EXPTIME-complete problem classes are much harder than P-complete or P-hard ones — the former are known to contain problems not solvable in polynomial time on modern computer architectures. However, it is not that factored representation makes solving MDPs very difficult; rather, it makes specifying them very easy compared to the hardness of solving them.

When the initial state is known, factored MDPs' computational complexity can be somewhat reduced by assuming that optimal policies that reach the goal from the initial state do so in a maximum number of steps polynomial in the number of state variables. Intuitively, this amounts to supposing that if we start executing an optimal policy at the initial state, we will visit only a "small" number of states before ending up at the goal. This assumption often holds true in practice; indeed, non-contrived problems requiring policies that visit a number of states exponential in the number of state variables (i.e., linear in the size of the state space) are not common. Moreover, executing such a policy would likely take a very long time, making its use impractical even if we could obtain it reasonably quickly. To get a sense for how long an execution of an exponentially-sized policy would take, observe that in an MDP with only 100 binary state variables such a policy would need to visit on the order of 2^{100} states.

For MDPs with an initial state and a polynomially sized optimal policy, the following result holds:

Theorem 2.9. *Factored SSP MDPs with an initial state in which an optimal policy reaches the goal from the initial state in a maximum number of steps polynomial in the number of state variables are*

PSPACE-complete [66, 35]. ♢

Crucially, with or without these assumptions, factored MDPs are hard not just to solve optimally but even to approximate, with no hope of discovering more efficient methods in the future unless a series of equivalences such as $P = NP$ are proven to be valid. Since factored representations are dominant in AI, the above results have pushed MDP research in the direction of methods that improve the efficiency of solving MDPs empirically but not necessarily theoretically.

2.2 Optimally Solving General MDPs: Fundamental Algorithms

We begin our review of MDP solution algorithms by concentrating on a fundamental set of optimal techniques that forms the basis for most of the advanced approaches. We describe the versions of these techniques and their theoretical results for the most general class we study, *SSP* (Definition 2.16). All of them also apply to factored SSP MDPs with and without the initial state.

The algorithms we survey here aim to compute an optimal stationary deterministic Markovian policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ for a given SSP MDP. The Optimality Principle (Theorem 2.3) guarantees the existence, although not the uniqueness, of such a policy. The policy these algorithms return is complete, i.e., prescribes an action for every state in the state space. In the rest of this section, by writing "policy" we will always mean a complete stationary deterministic Markovian one.

There are two broad groups of approaches to solving MDPs optimally. The first is based on iterative techniques that use dynamic programming, whereas the other formulates an MDP as a linear program. Iterative dynamic programming approaches are relatively more popular, thanks to being amenable to a wider variety of optimizations, and this section focuses on them. The linear programming-based methods are beyond the scope of this dissertation.

2.2.1 Policy Evaluation

The first algorithm we consider, per se, does not solve an SSP MDP as a whole. Rather, it is a building block that lets us find value functions of MDP policies.

At a high level, evaluating an SSP MDP policy amounts to solving the system of equations

$$\begin{aligned} V^{\pi}(s) &= 0 && \text{(if } s \in \mathcal{G}) \\ &= \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s') \left[\mathcal{C}(s, \pi(s), s') + V^{\pi}(s')\right] && \text{(otherwise)} \end{aligned} \tag{2.7}$$

These equations are linear, and there are a total of $|\mathcal{S}|$ variables, one for each state, so the system can be solved in $O(|\mathcal{S}|^3)$ time using Gaussian Elimination or other algorithms. However, there is another approach, iterative dynamic programming, which, as already mentioned, is more amenable to optimizations. Its pseudocode is shown in Algorithm 2.1. Its main idea is to start by initializing the values of all states with an arbitrary approximation V_0^{π} of V^{π}. Then, it computes a series of successive refinements V_n^{π} using the existing approximations V_{n-1}^{π}. Thus, the algorithm proceeds in iterations, and the n^{th} iteration applies the following operator to the values of all the non-goal states:

$$V_n^{\pi}(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s') \left[\mathcal{C}(s, \pi(s), s') + V_{n-1}^{\pi}(s')\right] \tag{2.8}$$

It can be shown that for any *proper* policy π, the sequence of value functions produced by this procedure asymptotically converges in the uniform metric to a unique fixed point. Moreover, this fixed point is the same as the solution to the system of linear equations 2.7 [6]:

Theorem 2.10. *For an SSP MDP and a proper policy* π, $\forall s \in \mathcal{S}, \lim_{n\to\infty} V_n^{\pi}(s) = V^{\pi}(s)$*, irrespective of the initialization* V_0^{π}. $\diamondsuit$

However, full convergence may take an infinite number of iterations, so for the algorithm to work in practice we need a convergence criterion. An appropriate such criterion is called *ϵ-consistency*, and we define it after first introducing an auxiliary notion of *residual*.

Definition 2.21. ***Residual (Policy Evaluation).*** *The residual at a state s at iteration n in the iterative policy evaluation algorithm, denoted as* $\mathrm{residual}_n(s)$*, is the magnitude of the change in the value of state s at iteration n in the algorithm, i.e.,* $\mathrm{residual}_n(s) = |V_n^\pi(s) - V_{n-1}^\pi(s)|$*. The residual at iteration n is the maximum residual across all states at iteration n of the algorithm, i.e.,* $\mathrm{residual}_n = \max_{s \in \mathcal{S}} \mathrm{residual}_n(s)$. ♣

Definition 2.22. ***ϵ-consistency (Policy Evaluation).*** *The value function V_n^π computed at iteration n in iterative policy evaluation is called ϵ-consistent if the residual at iteration $n+1$ is less than ϵ. The value of a state s is called ϵ-consistent at iteration n if the residual of V_n^π at s is smaller than ϵ.* ♣

Intuitively, the residual denotes the maximum change in the values of states from one iteration to the next. Our policy evaluation algorithm terminates when the value function V_n^π is ϵ-consistent, i.e., the change in values becomes less than a user-defined ϵ. Unfortunately, for general SSP MDPs, we cannot provide an easily computable bound on the number of iterations required for ϵ-consistency.

It is important to note that an ϵ-consistent V_n^π may not be ϵ-optimal (i.e., be within ϵ of the fixed-point value V^π). In fact, even for relatively small values of ϵ, an ϵ-consistent value function may be quite far from V^π. However, in practice tiny values of ϵ typically do translate to ϵ-consistent value functions that are very close to the fixed point.

2.2.2 Policy Iteration

Theoretically, using the policy evaluation algorithm from the previous section, we could find an optimal SSP MDP policy by evaluating all proper policies and choosing the lowest-cost one. Policy iteration (PI) [45] is based on this simple idea but makes it more practical. It replaces the brute-force policy enumeration by a more intelligent search, so that many suboptimal proper policies do not have to be explored. More specifically, PI evaluates a sequence of ever-better policies that eventually converges to the optimal one.

Algorithm 2.2 describes the pseudocode for PI. PI begins by evaluating an initial policy π_0 as shown previously. Next, based on the value of the current policy, it constructs a better one in

Algorithm 2.1: Iterative Policy Evaluation

```
Input: SSP MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G} \rangle$, proper policy $\pi$, $\epsilon > 0$
Output: an approximation of $\pi$'s value function $V^{\pi}$

function IterativePolicyEvaluation(SSP MDP $M$, policy $\pi$, $\epsilon > 0$)
begin
    initialize $V_0^{\pi}$ arbitrarily for each state
    $n \leftarrow 0$
    repeat
        $n \leftarrow n + 1$
        foreach $s \in \mathcal{S}$ do
            compute $V_n^{\pi}(s) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s') \left[ \mathcal{C}(s, \pi(s), s') + V_{n-1}^{\pi}(s') \right]$
            residual$_n(s) \leftarrow |V_n^{\pi}(s) - V_{n-1}^{\pi}(s)|$
        end
    until $\max_{s \in \mathcal{S}}$ residual$_n(s) < \epsilon$;
    return $V_n^{\pi}$
end
```

a policy improvement step. The algorithm keeps alternating between policy evaluation and policy improvement until it cannot improve the policy anymore. Before discussing the policy improvement step, we define a few more concepts.

Definition 2.23. ***Q-value under a Value Function***. *The* Q-value of state s and action a under a value function V, *denoted as* $Q^V(s,a)$, *is the one-step lookahead computation of the value of taking* a *in* s *under the belief that* V *is the true expected cost to reach a goal, i.e.,* $Q^V(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') \left[\mathcal{C}(s,a,s') + V(s') \right]$. ♣

Definition 2.24. ***Action Greedy w.r.t. a Value Function.*** *An action* a *is* greedy *w.r.t. a value function* V *in a state* s *if* a *has the lowest Q-value under* V *in* s *among all actions, i.e.,* $a = \operatorname{argmin}_{a' \in \mathcal{A}} Q^V(s,a')$. ♣

Any value function V induces a policy π^V that uses only actions greedy w.r.t. V:

Definition 2.25. ***Greedy Policy***. *A* greedy policy π^V *for a value function* V *is a policy that in every state uses an action greedy w.r.t* V, *i.e.,* $\pi^V(s) = \operatorname{argmin}_{a \in \mathcal{A}} Q^V(s,a)$. ♣

Algorithm 2.2: Policy Iteration

```
Input: SSP MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G} \rangle$, $\epsilon > 0$
Output: a policy, optimal if $\epsilon$ is sufficiently small

function PolicyIteration(SSP MDP $M$, $\epsilon > 0$)
begin
    initialize $\pi_0$ to be an arbitrary proper policy
    $n \leftarrow 0$
    repeat
        $n \leftarrow n + 1$
        Policy Evaluation: compute an $\epsilon$-consistent $V^{\pi_{n-1}}$
        Policy Improvement:
        foreach state $s \in \mathcal{S}$ do
            $\pi_n(s) \leftarrow \pi_{n-1}(s)$
            $\forall a \in \mathcal{A}$ compute $Q^{(V^{\pi_{n-1}})}(s, a)$
            $V_n(s) \leftarrow \min_{a \in \mathcal{A}} Q^{(V^{\pi_{n-1}})}(s, a)$
            if $Q^{(V^{\pi_{n-1}})}(s, \pi_{n-1}(s)) > V_n(s)$ then
                $\pi_n(s) \leftarrow \text{argmin}_{a \in \mathcal{A}} Q^{(V^{\pi_{n-1}})}(s, a)$
            end
        end
    until $\pi_n == \pi_{n-1}$;
    return $\pi_n$
end
```

The policy improvement step computes a greedy policy under $V^{\pi_{n-1}}$. In particular, it first computes the Q-value of each action under $V^{\pi_{n-1}}$ in a given state s. Then it assigns a greedy action in s as $\pi_n(s)$. The ties are broken arbitrarily, except if $\pi_{n-1}(s)$ still has the lowest Q-value, in which case this action is preferred.

Each policy improvement step is guaranteed to improve the policy as long as the original π_0 was proper [5]. Thus, π_n monotonically improves, guaranteeing that PI converges and yielding a policy that cannot be improved further. Moreover, it converges to an optimal policy in a finite number of iterations, since there are a finite number of distinct policies.

Theorem 2.11. *Policy iteration on an SSP MDP, if initialized with a proper policy π_0, successively improves the policy in each iteration, i.e., $\forall s \in \mathcal{S}, V^{\pi_n}(s) \leq V^{\pi_{n-1}}(s)$, and converges to an optimal policy π^* [5].* $\diamondsuit$

PI provides several opportunities for optimization. One of them consists in running the policy evaluation algorithm (Algorithm 2.1) for only one or a few iterations during each policy evaluation step. Indeed, running it to ϵ-consistency each time can be wasteful, since PI as a whole does not depend upon evaluating policies exactly. Rather, the value function needs to be improved just enough so that the next better policy can be obtained in the policy improvement step.

The algorithms that incorporate this and other changes to the basic PI method are appropriately known as modified policy iteration (MPI) [97, 81]. They converge to the optimal policy under the following general conditions:

Theorem 2.12. *Modified policy iteration on an SSP MDP, if initialized with a proper policy π_0, converges to an optimal policy π^*, as long as iterative policy evaluation is run for at least one iteration before each policy improvement step, and the value function $V_0^{\pi_0}$ that initializes the first run of iterative policy evaluation satisfies the condition $\forall s \in \mathcal{S}, \min_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{T}(s,a,s')[\mathcal{C}(s,a,s') + V_0^{\pi_0}(s')] \leq V_0^{\pi_0}(s)$.* $\diamondsuit$

2.2.3 Value Iteration

Value iteration (VI), originally proposed by Richard Bellman in 1957 [3], forms the basis of many advanced MDP algorithms. It takes a perspective complementary to PI's. PI can be visualized as searching in the policy space and computing the current value function using the current policy. VI switches the relative importance of policies and value functions. It searches directly in the value function space and whenever necessary produces a greedy policy based on the current state values. By doing this, it tries to solve the system of *Bellman equations* in the Optimality Principle for SSP MDPs (Equations 2.5), the set of identities that the optimal value function must satisfy.

VI's pseudocode is presented in Algorithm 2.3. It computes the solution to the Bellman equations via successive refinements, using an approach similar to iterative policy evaluation (Algorithm 2.1). The key idea is to approximate V^* with value functions V_n so that the sequence $\{V_n\}_{n=0}^{\infty}$ converges to V^* as n tends to infinity. The Optimality Principle tells us that once V^* is found, an optimal policy can be "read off" of it by determining a V^*-greedy action in every state.

The algorithm proceeds in iterations. It first initializes all state values with an arbitrary value function V_0. In the n-th iteration, it makes a *full sweep* of the state space, i.e. computes a new approximation $V_n(s)$ for all states with the help of the value function V_{n-1} from the previous iteration:

$$V_n(s) \leftarrow \min_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') \left[\mathcal{C}(s,a,s') + V_{n-1}(s') \right]. \tag{2.9}$$

Algorithm 2.3: Value Iteration

```
1  Input: SSP MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G} \rangle$, $\epsilon > 0$
2  Output: a policy, optimal if $\epsilon$ is sufficiently small
3
4  function ValueIteration(SSP MDP $M$, $\epsilon > 0$)
5  begin
6      initialize $V_0$ arbitrarily for each state
7      $n \leftarrow 0$
8      repeat
9          $n \leftarrow n + 1$
10         foreach $s \in \mathcal{S}$ do
11             compute $V_n(s)$ using Bellman backup at $s$ (Equation 2.9)
12             $\mathrm{residual}_n(s) \leftarrow |V_n(s) - V_{n-1}(s)|$
13         end
14     until $\max_{s \in \mathcal{S}} \mathrm{residual}_n(s) < \epsilon$;
15     return a greedy policy $\pi^{V_n}$ // see Definition 2.25
16 end
```

The operator in Equation 2.9 is known as *Bellman backup* or *Bellman update*. Thus, VI can be understood as an algorithm that searches for the fixed point of the synchronously applied Bellman backup operator (i.e., Bellman backup applied at all states simultaneously in every iteration), which also happens to be the solution of the Bellman equations. VI is also a dynamic programming algorithm — the whole layer of values V_n is stored for the next iteration. Moreover, it can be thought of as a message passing algorithm to achieve a global information flow. Via Bellman backup, it passes local messages (between states that are connected by actions) and thereby ends up computing a globally optimal value function. Last but not least, VI relates to the shortest path algorithms in graph theory. If all actions are deterministic, it reduces to the Bellman-Ford algorithm [23].

As already mentioned, the sequence of value functions generated by VI converges to the optimal in the limit. However, unlike PI, which requires an initial proper policy, VI converges without restrictions:

Theorem 2.13. *For the sequence of value functions $\{V_n\}_{n=0}^{\infty}$ produced by value iteration on an SSP MDP, $\forall s \in \mathcal{S}$, $\lim_{n\to\infty} V_n(s) = V^*(s)$, irrespective of the initializing value function V_0.* $\diamondsuit$

The termination condition of VI mimics that of the iterative policy evaluation algorithm. Below we define slightly more general notions of a residual (also called *Bellman error* in the case of VI) and ϵ-consistency, which will be used in the rest of this dissertation.

Definition 2.26. ***Residual (Bellman Backup).*** *In an SSP MDP, the residual at a state s w.r.t. a value function V, denoted as $Res^V(s)$, is the magnitude of the change in the value of state s if Bellman backup is applied to V at s once, i.e., $Res^V(s) = |V(s) - \min_{a\in\mathcal{A}}(\sum_{s'\in\mathcal{S}} \mathcal{T}(s,a,s')[C(s,a,s') + V(s')])|$. The residual w.r.t. a value function V, denoted as Res^V and called* Bellman error*, is the maximum residual w.r.t. V across all states if Bellman backup is applied to V at each state once, i.e., $Res^V = \max_{s\in\mathcal{S}} Res^V(s)$.* $\clubsuit$

Definition 2.27. ***ϵ-consistency (Bellman Backup).*** *A state s is called ϵ-consistent w.r.t. a value function V if V is ϵ-consistent at s, i.e., $Res^V(s) < \epsilon$. A value function V is called ϵ-consistent if it is ϵ-consistent at all states, i.e., if $Res^V < \epsilon$.* $\clubsuit$

As shown in Algorithm 2.3, we terminate VI when its value function is ϵ-consistent, i.e., its Bellman error is small. This typically indicates that VI is quite close to the optimal value function, and hence an optimal policy.

As a final note on VI, the Bellman backup operator that lies at the core of this algorithm satisfies a useful property called *monotonicity*. To define it, we denote the set of all value functions of an MDP as $\mathcal{V}$.

Definition 2.28. ***Operator Monotonicity***. *An operator $T : \mathcal{V} \to \mathcal{V}$, which applies to a value function to obtain a new value function, is* monotonic *if* $\forall\, V_1, V_2 \in \mathcal{V},\ V_1 \leq V_2 \implies TV_1 \leq TV_2$. ♣

In other words, if a value function V_1 is componentwise greater (or less) than another value function V_1 then the same inequality holds true between TV_1 and TV_2, i.e., the value functions that are obtained by applying this operator T on V_1 and V_2. We can prove that the synchronously applied Bellman backup operator used in VI is monotonic for value functions of SSP MDPs [40]. As a corollary, if a value function (V_k) is a lower (or upper) bound on V^* for all states, then all intermediate value functions thereafter (V_n, $n > k$) continue to remain lower (respectively, upper) bounds. This is because V^* is the fixed point of Bellman backup.

The notion of monotonicity exists for value functions as well:

Definition 2.29. ***Value Function Monotonicity***. *Let value function V' be the result of applying an operator T to a value function V. V is called* monotonic *w.r.t. T if $V \leq V^*$ and $V \leq V'$ or if $V \geq V^*$ and $V \geq V'$.* ♣

When the operator w.r.t. which a value function is monotonic is clear from context, we will refer to that value function simply as "monotonic". Bellman backup applied to a monotonic value function V of an SSP MDP gives an especially strong guarantee: the resulting value function is always "closer" to V^* than V was.

2.2.4 Asynchronous Value Iteration

One of the biggest drawbacks of VI (also called *synchronous VI*, since it uses Bellman backup in a synchronous manner) is that it requires full sweeps of the state space. The state spaces are usually large for real problems, making this strategy impractical. In the meantime, Bellman backups at some states can change the value function considerably, while at others, where the values have nearly converged, they can be a waste of time. This intuition goes a long way in suggesting optimizations for value function computation. To enable these optimizations, the *asynchronous VI* framework (Algorithm 2.4) relaxes the requirement that all states need to be backed up in each iteration.

Algorithm 2.4: Asynchronous Value Iteration

```
1  Input: SSP MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G} \rangle$, $\epsilon > 0$
2  Output: a policy, optimal if $\epsilon$ is sufficiently small
3
4  function AsynchronousValueIteration(SSP MDP $M$, $\epsilon > 0$)
5  begin
6      initialize $V$ arbitrarily for each state
7      while $Res^V \geq \epsilon$ do
8          select a state $s$
9          compute $V(s)$ using a Bellman backup at $s$ (Equation 2.9)
10         update $Res^V(s)$
11     end
12     return a greedy policy $\pi^V$
13 end
```

The convergence of asynchronous VI to the optimal value function (in the limit) requires an additional restriction that no state gets *starved*, i.e., that all states are backed up an infinite number of times. Under this constraint, for asynchronous VI we can prove that $\forall s \in \mathcal{S}, \lim_{\epsilon \to 0} V(s) = V^*(s)$ [5]. As with synchronous VI, in practice we cannot run the algorithm forever, so a termination condition similar to that of synchronous VI is employed. It checks whether the current value function is ϵ-consistent for some small nonzero ϵ.

Asynchronous VI forms the basis for several families of VI-related algorithms because it allows the flexibility to choose a backup order intelligently. Next, we review one of these families, known under the umbrella term *heuristic search*.

2.3 Optimally Solving MDPs with an Initial State: Heuristic Search

The algorithms discussed in the previous section do not use the knowledge of the initial state, even if it is available for the MDP at hand. In the meantime, as mentioned in Section 2.1.7, if the initial state is known, we are interested in *partial* MDP policies. Intuitively, since they are smaller in size than complete ones, they should also be more efficient to compute. In this section, we discuss a set of techniques that support this commonsense observation. They employ an entity called *heuristic function* to help them find an optimal partial policy closed w.r.t. the initial state s_0 without touching every state in the state space as synchronous VI and PI do. Accordingly, these algorithms are collectively known as *heuristic search* methods.

Definition 2.30. ***Heuristic Function.*** *A heuristic function V_0 is a value function that initializes state values when an MDP solution algorithm inspects them for the first time.* ♣

In fact, we have already seen an example of a heuristic function (henceforth referred to simply as *heuristic*) before — the value function V_0 that initialized VI (Algorithm 2.3). For VI, we viewed V_0 as chosen arbitrarily. In reality, both VI and heuristic search methods tend to perform better when their heuristic encodes good estimates of the optimal values of different states. Such estimates are usually derived by a domain expert for a particular problem based on some insights about the scenario in question. For example, if the agent is a robot in a setting modeled with an SSP$_{s_0}$ MDP, a reasonable heuristic would assign high values to states in which the robot is broken, indicating that these states are very costly. Compared to them, states that are close to the goal should get much lower values.

Heuristic search algorithms equipped with informative heuristics are the most efficient methods for optimally solving MDPs. For the new MDP classes presented in Chapter 5, we develop solution techniques of this type. In the remainder of this section, we formalize the notion of heuristic search by describing it as an abstract algorithmic framework. We also examine in detail two concrete approaches that implement this framework. Like in Section 2.2, we focus on stochastic shortest-path problems, those expressible as SSP$_{s_0}$ MDPs.

2.3.1 FIND-AND-REVISE— *a Schema for Heuristic Search*

All heuristic search algorithms are instances of a general schema called FIND-AND-REVISE (Algorithm 2.5) [12]. To discuss it, we need several new concepts.

Throughout our analysis, we will view the connectivity structure of an MDP M's state space as a *directed hypergraph $G_{\mathcal{S}}$*, which we call the *connectivity graph of M*. A directed hypergraph generalizes the concept of a regular graph by allowing each *hyperedge*, or *k-connector*, to have one source but several destinations. In the case of an MDP, the corresponding hypergraph $G_{\mathcal{S}}$ has $\mathcal{S}$ as the set of vertices, and for each state s and action a pair has a k-connector whose source is s and whose destinations are all states s' s.t. $\mathcal{T}(s, a, s') > 0$. In other words, it has a k-connector for linking each state via an action to the state's possible successors under that action. We now define several more hypergraph-related notions.

Definition 2.31. ***Reachability.*** *A state s_n is reachable from s_1 in $G_{\mathcal{S}}$ if there is a sequence of states and actions $s_1, a_1, s_2, \ldots, s_{n-1}, a_{n-1}, s_n$, where for each i, $1 \leq i \leq n-1$, the node for s_i is the source of the k-connector for action a_i and s_{i+1} is one of its destinations.* ♣

Definition 2.32. ***The Transition Graph of an MDP Rooted at a State.*** *The transition graph of an MDP rooted at state s is G_s, a subgraph of the MDP's connectivity graph $G_{\mathcal{S}}$. Its vertices are s and only those states s' that are reachable from s in $G_{\mathcal{S}}$. Its hyperedges are only those k-connectors that originate at s or at some state reachable from s in $G_{\mathcal{S}}$.* ♣

In this section, we will mostly refer to the transition graph G_{s_0} rooted at the initial state. This hypergraph includes only states reachable via some sequence of action outcomes from s_0. Historically, MDP transition graphs are also known as *AND-OR graphs*.

Definition 2.33. ***The Transition Graph of a Policy Rooted at a State.*** *The transition graph of a partial deterministic Markovian policy $\pi_s : \mathcal{S}' \to A$ is a subgraph of the MDP's connectivity graph $G_{\mathcal{S}}$ that contains only the states in $\mathcal{S}'$ and, for each state $s \in \mathcal{S}'$, only the k-connector for the action a s.t. $\pi(s) = a$.* ♣

Definition 2.34. ***The Greedy Graph of a Value Function Rooted at a State.*** *The greedy graph G_s^V of value function V rooted at state s is the union of transition graphs of all policies π_s^V greedy w.r.t. V and closed w.r.t. s.* ♣

That is, G_s^V contains all states that can be reached via *some* π_s^V from s. As with general transition graphs, we will mostly be interested in greedy graphs of value functions rooted at the initial state s_0.

As the final pieces of terminology before we proceed, recall that the residual $Res^V(s)$ (Definition 2.26) is the magnitude of change in the value of a state as a result of applying a Bellman backup to value function V. A state s is called ϵ-consistent w.r.t. V if $Res^V(s) < \epsilon$ (Definition 2.27) and ϵ-inconsistent otherwise.

Algorithm 2.5: FIND-AND-REVISE

```
Input: SSP$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, heuristic $V_0$, $\epsilon > 0$
Output: a policy closed w.r.t. $s_0$, optimal if $V_0$ is admissible, $\epsilon$ is sufficiently small, and FIND is
systematic

function FIND-AND-REVISE(SSP$_{s_0}$ MDP $M$, heuristic $V_0$, $\epsilon > 0$)
begin
    $V \leftarrow V_0$
    while $V$'s greedy graph $G^V_{s_0}$ contains a state $s$ with $Res^V(s) \geq \epsilon$ do
        FIND a state $s$ in $G^V_{s_0}$ with $Res^V(s) \geq \epsilon$
        REVISE $V(s)$
    end
    return a greedy policy $\pi^V_{s_0}$ closed w.r.t. $s_0$
end
```

The idea of FIND-AND-REVISE is quite simple. It iteratively searches the greedy graph of the current value function for an ϵ-inconsistent state and updates the value of that state and possibly of a few others with a Bellman backup. This typically changes the greedy graph, and the cycle repeats. Note that FIND-AND-REVISE is essentially a flavor of asynchronous VI (Section 2.2.4).

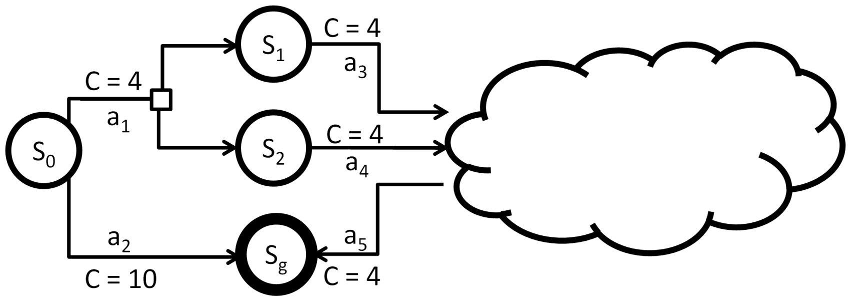


Figure 2.1: MDP showing a possible impact of a heuristic on the efficiency of policy computation via FIND-AND-REVISE. FIND-AND-REVISE guided by a good heuristic may never visit the arbitrarily large part of the state space represented by the cloud, yielding enormous time and memory savings compared to VI.

Crucially, the greedy graph that FIND-AND-REVISE starts with is induced by some heuristic function V_0. To demonstrate the difference V_0 can make on the number of states a FIND-AND-REVISE-like algorithm may have to store, we present the following example.

Example: Consider the transition graph G_{s_0} of the SSP$_{s_0}$ MDP in Figure 2.1. This MDP has many states, four of which (the initial state s_0, the goal state s_g, and two other states s_1 and s_2) are shown, while the rest are denoted by the cloud. Action costs are shown for a subset of the MDP's actions; assume that costs are nonnegative for actions in the cloud. At least one of the actions, a_1, has several probabilistic effects, whose probabilities do not matter for this example and are omitted. Note that $\pi^*_{s_0}$ for this MDP is unique and involves taking action a_2 from s_0 straight to the goal; thus, $V^*(s_0) = 10$. All other policies involve actions a_1, a_5, and a_3 or a_4. Therefore, the cost of reaching the goal from s_0 using them is at least $4 \cdot 3 = 12 > 10$, making these policies suboptimal.

Now, the transition graph G_{s_0} of the MDP in Figure 2.1 can be arbitrarily large, depending on the number of states in the cloud. This is the largest set of states an MDP solution algorithm may *have to* store while searching for $\pi^*_{s_0}$. Compare G_{s_0} to $G^{V^*}_{s_0}$, the greedy graph of the optimal value function. $G^{V^*}_{s_0}$ contains only the states visited by $\pi^*_{s_0}$, i.e., s_0 and s_g, as established above. This is the very smallest set of states we can hope to explore while looking for $\pi^*_{s_0}$. Finally, consider $G^{V_0}_{s_0}$ for a heuristic value function V_0 that assigns $V_0(s_g) = 0$ and, for instance, $V_0(s_1) = V_0(s_2) = 7$. These values would induce $Q^{V_0}(s_0, a_1) = 4 + 7 = 11 > Q^{V_0}(s_0, a_2) = 10$, making a_2 more preferable in s_0 and thus immediately helping discover the optimal policy. Thus, $G^{V_0}_{s_0}$ consists of s_0 and s_g, and starting FIND-AND-REVISE from such an V_0 allows FIND-AND-REVISE to evaluate only four states before finding $\pi^*_{s_0}$. Contrast this with an algorithm such as VI, which, even initialized with a very good V_0, will still necessarily visit the entire transition graph rooted at the initial state, G_{s_0}. As this example shows, FIND-AND-REVISE in combination with a good heuristic can make finding $\pi^*_{s_0}$ arbitrarily more efficient than via VI or PI. ■

The fact that FIND-AND-REVISE may never touch some of the states, as in the above example, might seem alarming. After all, asynchronous VI algorithms, one of which is FIND-AND-REVISE, in general fail to find an optimal policy if they starve some states, and FIND-AND-REVISE appears to be doing exactly that. As it turns out, however, if FIND-AND-REVISE's FIND procedure is *systematic* and the heuristic function FIND-AND-REVISE is using is *admissible*, FIND-AND-REVISE is guaranteed to converge to an optimal solution for a sufficiently small ϵ.

Definition 2.35. ***Systematicity.*** *Let $K_i(s)$ be the total number of* FIND-AND-REVISE *iterations* *that state s has spent in the greedy graph rooted at s_0 with $Res^V(s) > \epsilon$ between iterations i and $i + K$. The* FIND-AND-REVISE*'s FIND procedure is called* systematic *if for all $i > 1$ and for all $s \in \mathcal{S}$ the probability that FIND will choose s for the REVISE step at least once after iteration i approaches 1 as $K_i(s)$ goes to ∞.* ♣

This technical definition has an intuitive meaning — a FIND procedure is systematic (i.e., searches the greedy graph systematically) if it does not starve any of the states possibly relevant to finding the optimal solution. Such a FIND procedure will not allow an ϵ-inconsistent state to stay in the greedy graph forever without its value being revised. At the same time, it may ignore states that at some point leave the greedy graph for good.

Definition 2.36. ***Heuristic Admissibility.*** *A heuristic V_0 is* admissible *if for all states s in the transition graph G_{s_0}, $V_0(s) \leq V^*(s)$ in cost-minimization MDPs and $V_0(s) \geq V^*(s)$ in reward-maximization MDPs. Otherwise, the heuristic is called* inadmissible. ♣

Theorem 2.14. *For an SSP$_{s_0}$ MDP and an $\epsilon > 0$, if* FIND-AND-REVISE *has a systematic FIND procedure and is initialized with an admissible monotonic heuristic, it converges to a value function that is ϵ-consistent over the states in its greedy graph rooted at s_0 after a finite number of REVISE steps [11, 12].* ♢

Theorem 2.15. *For an SSP$_{s_0}$ MDP, if* FIND-AND-REVISE *has a systematic FIND procedure and is initialized with an admissible monotonic heuristic, as ϵ goes to 0 the value function and policy computed by* FIND-AND-REVISE *approaches, respectively, the optimal value function and an optimal policy over all states reachable from s_0 by at least one optimal policy [11, 12].* ♢

Theorem 2.15 contains a small caveat. As with VI, although *in the limit* a vanishingly small residual implies optimality, for finite values of ϵ FIND-AND-REVISE can return a significantly suboptimal policy. Nonetheless, in practice this rarely happens.

FIND-AND-REVISE's pseudocode intentionally leaves the FIND and REVISE procedures unspecified — it is in their implementations that various heuristic search algorithms differ from each other. Their REVISE methods tend to resemble one another, as they are all based on the Bellman backup operator. Their FIND methods, however, can be vastly distinct. An obvious approach to finding ϵ-inconsistent states is via a simple systematic search strategy such as depth-first search. Indeed, depth-first search is used in this manner by several proposed FIND-AND-REVISE algorithms, e.g., HDP [12] and LDFS [14]. Employing depth-first search for the purpose of identifying weakly explored states also echoes similar approaches in solving games, non-deterministic planning problems, and other related fields [14]. However, in tackling MDPs this strategy is usually outperformed by more sophisticated search methods. Historically, the first of them was the LAO* [41] algorithm, which eventually gave rise to several other techniques: ILAO* [41], BLAO* [7], and RLAO* [24]. However, this family is outside the scope of our work. Instead, we explore FIND-AND-REVISE approaches derived from the RTDP algorithm [2].

2.3.2 RTDP

The name "RTDP" stands for Real-Time Dynamic Programming (RTDP) [2]. Similar to LAO*, this algorithm prompted a significant number of variants — LRTDP [13], BRTDP [73], FRTDP [90], and VPI-RTDP [88] — the first of which will be discussed in the next subsection. At a high level, RTDP-based algorithms operate by simulating the current greedy policy to sample "paths", or *trajectories*, through the state space, and performing Bellman backups only on the states in those trajectories. These updates change the greedy policy and make way for further state value improvements.

The process of sampling a trajectory is called a *trial*. As shown in Algorithm 2.6, each trial consists of repeatedly selecting a greedy action a_{best} in the current state s (line 18), performing a Bellman backup on the value of s (line 19), and transitioning to a successor of s under a_{best} (line 20). A heuristic plays the same role in RTDP as in LAO* — it provides initial state values in order to guide action selection during early state space exploration.

Each sampling of a successor during a trial that does not result in a termination of the trial corresponds to a FIND operation in FIND-AND-REVISE, as it identifies a possibly ϵ-inconsistent state to update next. Each Bellman backup maps to a REVISE instance.

Algorithm 2.6: RTDP

```
Input: SSP$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, heuristic $V_0$, timeout $T$
Output: a policy closed w.r.t. $s_0$, optimal if $V_0$ is admissible and $T$ is sufficiently large

$V \leftarrow V_0$

function RTDP(SSP$_{s_0}$ MDP $M$, timeout $T$)
begin
    while time $T$ has not run out do
        TRIAL($M$, $s_0$)
    end
    return a greedy policy $\pi^V_{s_0}$ closed w.r.t. $s_0$
end


function TRIAL(SSP$_{s_0}$ MDP $M$, state $s$)
begin
    while $s \notin \mathcal{G}$ do
        $a_{best} \leftarrow \arg\min_{a \in \mathcal{A}} Q^V(s, a)$
        $V(s) \leftarrow Q^V(s, a_{best})$
        $s \leftarrow$ simulate action $a_{best}$ in $s$
    end
end
```

The original RTDP version [2] has two related weaknesses, the main one being the lack of a principled termination condition. Although RTDP is guaranteed to converge asymptotically to V^* *over the states in the domain of an optimal policy* $\pi^*_{s_0}$, it does not provide any mechanisms to detect when it gets near the optimal value function or policy. The lack of a stopping criterion, although unfortunate, is not surprising. RTDP was designed for operating under time pressure, and would almost never have the luxury of planning for long enough to arrive at an optimal policy. In these circumstances, a convergence detection condition is not necessary.

The lack of convergence detection leads to RTDP's other drawback. As RTDP runs longer, V at many states starts to converge. Visiting these states again and again becomes a waste of resources, yet this is what RTDP keeps doing because it has no way of detecting convergence. An extension of RTDP we will look at next addresses both of these problems by endowing RTDP with a method to recognize proximity to the optimal value function over relevant states.

2.3.3 LRTDP

Labeled RTDP (LRTDP) [13] works in largely the same way as RTDP, but also has a mechanism for identifying ϵ-consistent states and marking them as solved. The following theorem about the basic RTDP algorithm's convergence provides a theoretical basis for LRTDP's termination condition:

Algorithm 2.7: LRTDP

```
Input: SSP$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, heuristic $V_0$, $\epsilon > 0$, (optional) timeout $T$
Output: a policy closed w.r.t. $s_0$, optimal if $V_0$ is admissible and $\epsilon$ is sufficiently small

$V \leftarrow V_0$

function LRTDP(SSP$_{s_0}$ MDP $M$, $\epsilon > 0$, (optional) timeout $T$)
begin
    while $s_0$ is not labeled solved and time $T$ has not run out do LRTDP-TRIAL($M$, $s_0$, $\epsilon$)
    return a greedy policy $\pi^V_{s_0}$ closed w.r.t. $s_0$
end

function LRTDP-TRIAL(SSP$_{s_0}$ MDP $M$, state $s$, $\epsilon > 0$)
begin
    $visited \leftarrow$ empty stack
    while $s$ is not labeled solved do
        push $s$ onto $visited$
        if $s \in \mathcal{G}$ then break
        $a_{best} \leftarrow \arg\min_{a \in \mathcal{A}} Q^V(s, a)$
        $V(s) \leftarrow Q^V(s, a_{best})$
        $s \leftarrow$ simulate action $a_{best}$ in $s$
    end
    while $visited \neq$ empty stack do
        $s \leftarrow$ pop the top of $visited$
        if ¬CHECK-SOLVED$(M, s, \epsilon)$ then break
    end
end
```

Theorem 2.16. *For RTDP initialized with a monotonic admissible heuristic, the value of a state s is ϵ-consistent under a value function V and will remain ϵ-consistent at s for all value functions generated by RTDP from V if $Res^V(s) < \epsilon$ and $Res^V(s') < \epsilon$ for all descendants s' of s in the greedy graph G^V_s.* $\diamondsuit$

Algorithm 2.8: CHECK-SOLVED

```
function CHECK-SOLVED((SSP$_{s_0}$ MDP $M$, state $s$, $\epsilon$)
begin
    $ret\_val \leftarrow true$
    $open \leftarrow$ empty stack
    $closed \leftarrow$ empty stack
    push $s$ onto $open$
    while $open \neq$ empty stack do
        $s \leftarrow$ pop the top of $open$
        push $s$ onto $closed$
        if $Res^V(s) \geq \epsilon$ then
            $ret\_val \leftarrow false$
            continue
        end
        $a_{best} \leftarrow \arg\min_{a \in \mathcal{A}} Q^V(s, a)$
        foreach $s' \in \mathcal{S}$ s.t. $\mathcal{T}(s, a_{best}, s') > 0$ do
            if $s'$ is not labeled solved and $s' \notin open \cup closed$ then push $s'$ onto $open$
        end
    end
    if $ret\_val == true$ then
        foreach $s' \in closed$ do label $s'$ solved
    else
        while $closed \neq$ empty stack do
            $s \leftarrow$ pop the top of $closed$
            $V(s) \leftarrow \min_{a \in A} Q^V(s, a)$
        end
    end
    return $ret\_val$
end
```

In other words, s's value is guaranteed to remain ϵ-consistent forever from the point when RTDP arrives at a value function V s.t. Bellman backups cannot change either $V(s)$ or the values of any of s's descendants in G_s^V by more than ϵ. Informally, the reason for this is that the value $V(s)$ of a state is determined solely by the values of its descendants in the (implicitly maintained) greedy graph G_s^V. Therefore, hypothetically, $V(s)$ can change by more than ϵ only under two circumstances — either if G_s^V changes, or if the values of some of s's descendants in G_s^V change by more than ϵ. As it turns out, in a FIND-AND-REVISE algorithm started from a monotone admissible heuristic, the sole way to modify G_s^V is by updating a value of a state *within* G_s^V. Updating states outside G_s^V will never make them part of G_s^V because, by the monotonicity property (Definition 2.29), Bellman backups can only increase the values of states and therefore make them only less attractive and less eligible to be part of a greedy graph. This implies that there is actually just one way for RTDP to change $V(s)$ by more than ϵ — by changing a value of a descendant of s in G_s^V by more than ϵ. However, this would contradict the premise of the above theorem that all the descendants of s are already ϵ-consistent.

LRTDP (Algorithm 2.7) implements a mechanism for detecting convergence implied by the above theorem in its CHECK-SOLVED method (Algorithm 2.8). To verify that the value of a state s has stabilized, CHECK-SOLVED checks whether the residual at s or any of its descendants in G_s^V is at least ϵ (lines 10-12 of Algorithm 2.8). For this purpose, it keeps two stacks — *open*, with states still to be checked for ϵ-consistency, and *closed*, with already checked states. In every iteration it takes a state off the *open* stack (line 8), moves it onto *closed* (line 9), and sees whether the state's residual is greater than ϵ. If so, s cannot be labeled as solved yet (line 11). Otherwise, CHECK-SOLVED expands the state just examined (lines 14-16) in order to check this state's successors as well. In this way, all descendants of states s' in G_s^V with $Res^V(s') < \epsilon$ eventually end up being examined by CHECK-SOLVED. If the residuals of all these states are smaller than ϵ, all of them, including s, are labeled solved (line 20). Otherwise, those whose residuals are at least ϵ get updated with Bellman backups (lines 22-25).

LRTDP uses CHECK-SOLVED to label states it visits during the trials (lines 22-24 of Algorithm 2.7) and, most importantly, to terminate trials early once they end up at a labeled state (line 15). The labeling procedure makes LRTDP's convergence to an ϵ-consistent value function orders of magnitude faster than RTDP's [13]. Moreover, it makes LRTDP useful not only in real-time settings but for offline planning as well.

LRTDP's versatility, simplicity, and efficiency when equipped with a good heuristic make it a popular choice for many problems. In Chapter 4, we introduce its efficient derivative for finite-horizon problems, LR^{2}TDP [54].

Chapter 3
EXTRACTING LATENT STRUCTURE OF FACTORED MDPS

As discussed in the Introduction and Background chapters, one of the biggest challenges facing planning under uncertainty is the scalability of the available solution techniques. Indeed, in Sections 2.1.9 and 2.1.10, we established that the only explicit representation practical for describing large MDPs is a factored one[1], but the traditional optimal MDP algorithms such as VI scale polynomially in the size of an MDP's state space and hence exponentially in the size of its factored description. This dramatically limits MDPs' practical utility.

Meanwhile, humans perform surprisingly well at probabilistic planning. Although the policies they come up with can be far from optimal in terms of utility, usually they achieve the desired goal. There are three key ingredients that contribute to humans' relative success in this area. One of them is an extensive use of heuristics: when looking for a plan, we use plenty of intuitions about the problem that are not apparent to machines. Another helping factor is our ability to make crude but effective approximations. For instance, people rarely reason about the exact probabilities of action outcomes. In fact, sometimes they implicitly assume that actions are deterministic and have only their most likely effects, e.g., that driving to a shop will not cause a flat tire. Although a flat tire is clearly a possibility, in many circumstances ignoring it makes coming up with a decent plan much easier and faster. The third and perhaps the most important ingredient is our skill at recognizing abstractions and generalizing conclusions across different situations. As an example, after realizing that the walls of a particular Mars crater are too steep for the exploration rover to escape, a human planner would order the rover to abandon attempts to collect any of the rock samples in the crater, while a traditional MDP solver might rediscover this navigational problem over and over again as it considered collecting each rock sample in turn. Crucially, people employ these three approaches *in concert*, so that they make up for each others' weaknesses.

[1]Alternatively, one can describe MDPs implicitly by building a simulator and tackle them with reinforcement learning [93]. However, these approaches are even less scalable than techniques used in planning.

The low scalability threshold of the basic VI and PI has made researchers consider largely the same approximation tricks as those used by humans, resulting in three major families of algorithms. We have already discussed one of them, heuristic search (Section 2.3). Another one, which we call *determinization-based approximation*, is founded on the idea of assuming that when constructing a policy, the agent can choose any particular outcome of an action at will, i.e., that the agent is dealing with a much simpler, deterministic planning problem. Note how this assumption parallels ignoring undesirable action outcomes, as humans do; it also has the same advantage — speed. The last family of approximation algorithms is *dimensionality reduction*. It views an MDP as a problem in $|\mathcal{S}|$ dimensions (an agent needs to pick an action for each state) and maps it into a lower-dimensional space. This effectively ties together action choices in different states, enabling information generalization across the state space and thus mimicking our ability to carry over judgments about one situation to others.

Why, then, has automated planning under uncertainty failed to attain the human level in many applications so far? We hypothesize the lack of integration of the above techniques as the primary reason for this shortfall. Contrary to the way humans do it, AI planning systems have been employing heuristics, determinization, and dimensionality reduction *separately*, letting the drawbacks of the individual approaches manifest themselves. Indeed, both the algorithms for computing heuristics and the top determinization-based planners such as RFF [95] suffer from the same weakness as VI — they reason about each encountered state individually. Dimensionality reduction addresses this issue, but has a downside of its own. Many scenarios lack a natural metric or another obvious way to construct a dimensionality-reducing mapping automatically, which brings a human in the loop and makes these techniques non-autonomous. We will examine all three paradigms in finer detail in Section 3.7, but hope that even their cursory overview shows the potential of unifying these approaches.

In this chapter, we attempt to fulfill this vision by proposing algorithms that automatically discover and exploit state abstractions to solve large factored goal-oriented MDPs. These methods efficiently mine the latent structure of MDPs using their determinizations and turn this structure into a low-dimensional parameter space. This parameter space can then be used to compute informative heuristics for FIND-AND-REVISE-like methods or solve the MDP directly. In either case, algorithms operating on the latent problem representation are fast, have a small memory footprint, and provide previously unnoticed insights into the structure of factored MDPs.

3.1 Overview

Specifically, our algorithms generate two kinds of abstraction, *basis functions* and *nogoods*, each of which describes sets of states that share a similar relationship to the planning goal. Both basis functions and nogoods are represented as logical conjunctions of state variable values, but they encode diametrically opposite information. When a basis function holds in a state (i.e., every variable mentioned in the basis function has the same value in the state as it does in the basis function itself), this guarantees that a certain trajectory of action outcomes has a positive probability of reaching the goal. Our algorithms associate weights with each basis function, encoding the relative quality of the different trajectories. In contrast, when a nogood holds in a state, it signifies that the state is a dead end; *no* trajectory can reach the goal from this state. Continuing the Mars rover example, a conjunction that describes the presence of the rover in the steep-walled crater would be a nogood.

Our basis functions and nogoods are similar in spirit to the rules learned in logical theories in explanation-based learning [47] and constraint satisfaction [27], but our work applies them in a probabilistic context (e.g., learns weights for basis functions) and provides new mechanisms for their discovery. Previous MDP algorithms have also used basis functions [37, 87], but to perform generalization *between different problems in a domain* rather than during the course of solving a single problem. Other researchers have also used hand-generated basis functions in a manner similar to ours [36, 38, 39]; our techniques circumvent the main weakness of these approaches by generating the abstractions automatically.

3.1.1 Discovering Nogoods and Basis Functions

We generate basis functions by regressing goal descriptions along an action outcome trajectory using a determinized version of the probabilistic domain theory. Thus, the trajectory is potentially executable in all states satisfying the basis function. This justifies performing Bellman backups on basis functions, rather than states, thereby generalizing experience across similar states. Since many basis functions typically hold in a given state, the value of a state is a complex function of the applicable basis functions.

The nogoods are discovered with a novel machine learning algorithm that operates in two phases. First it generates candidate nogoods with a probabilistic sampling procedure using basis functions and previously discovered dead ends as training data. Then it tests the candidates with a planning graph [9] to ensure that no trajectories to the goal exist from states containing the nogood.

3.1.2 Exploiting Nogoods and Basis Functions

We present three algorithms that leverage the basis function and nogood abstractions to speed up MDP solution and reduce the amount of memory required for it:

- GOTH [56, 58] uses a full classical planner to *generate a heuristic function* for an MDP solver for use as an initial estimate of state values. While classical planners have been known to provide an informative approximation of state value in probabilistic problems, they are too expensive to call from every newly visited state. GOTH amortizes this cost across multiple states by associating weights to basis functions and thus generalizing the heuristic computation. Empirical evaluation shows GOTH to be an informative heuristic that saves heuristic search methods, e.g., LRTDP, considerable time and memory.

- RETRASE [55, 58] is a *self-contained MDP solver* based on the same information-sharing insight as GOTH. However, unlike GOTH, which sets the weight of each basis function only once to compute an initial guess of states' values, RETRASE *learns* the basis functions' weights by evaluating each function's "usefulness" in a decision-theoretic way. By aggregating the weights, RETRASE constructs a state value function approximation and, as we show empirically, produces better policies than the participants of the International Probabilistic Planning Competition (IPPC) on many domains while using little memory.

- SIXTHSENSE [57, 58] is a *method for quickly and reliably identifying dead ends*, i.e., states with no possible trajectory to the goal, in MDPs. In general, for factored MDPs this problem is intractable — one can prove that determining whether a given state has a trajectory to the goal is PSPACE-complete [35]; therefore, it is unsurprising that modern MDP solvers often waste considerable resources exploring these doomed states. SIXTHSENSE can act as a submodule of an MDP solver, helping it detect and avoid dead ends. SIXTHSENSE employs machine learning, using basis functions as training data, and is guaranteed never to generate false positives. The resource savings provided by SIXTHSENSE are determined by the fraction of dead ends in an MDP's state space and reach 90% on some IPPC benchmark problems.

In the rest of the chapter, some of whose content was previously published in [55], [56], [57], and [58], we describe these algorithms, discuss their theoretical properties, and evaluate them empirically. Section 3.2 reviews some additional topic-specific background material to complement what was covered in Chapter 2 and introduces relevant definitions, illustrating these with a running example. Sections 3.4, 3.5, and 3.6 present descriptions of and empirical results on GOTH, RETRASE, and SIXTHSENSE respectively. Section 3.7 discusses the related work. Section 3.8 points out potential extensions of the presented techniques.

3.2 Preliminaries

The algorithms presented in this chapter apply to a type of probabilistic planning problems that can be roughly characterized as factored SSP$_{s_0}$ MDPs (Definition 2.20) with *dead ends* — states from which the goal cannot be reached with any policy. Note that, formally, SSP$_{s_0}$ MDPs require the existence of at least one complete proper policy (Definition 2.14) and therefore cannot have dead ends. At the same time, probabilistic planning researchers have long used goal-oriented benchmark problems with dead-end states to gauge the performance of approximate MDP solvers, because many realistic scenarios do have these states and because they make for difficult test cases [65]. In this chapter, we assume that the agent is dealing with a problem that conforms to the strong *SSP*$_{s_0}$ definition (2.19) except for the existence of a proper policy, and that the agent has to pay a high user-defined penalty if it enters a dead end. In Chapter 5, we present a theoretical analysis of such problems, formalized as fSSPUDE$_{s_0}$ MDPs (Definition 5.14). This analysis will also reveal that GOTH, RETRASE, and SIXTHSENSE are indirectly geared toward solving goal-oriented MDPs so as to maximize the probability of an agent reaching the goal successfully, i.e., towards optimizing the *MAXPROB* criterion (Definition 5.12).

We assume the goal-oriented MDPs to be given in a PPDDL-style representation. As mentioned in Section 2.1.9, such representations take their name from the Probabilistic Planning Domain Description Language. Below, we introduce an example scenario in PPDDL that will help illustrate concepts related to our techniques in subsequent sections, and show how to cast it as a factored goal-oriented MDP.

3.2.1 Example

The scenario we will be referring to is called GremlinWorld. Consider a gremlin that wants to sabotage an airplane and stay alive in the process. The gremlin can pick up several tools to accomplish the task. The gremlin can either tweak the airplane with a screwdriver and a wrench, or smack it with a hammer. However, with high probability, smacking leads to accidental detonation of the airplane's fuel, which destroys the airplane but also kills the gremlin.

In Figure 3.1, GremlinWorld is formulated in PPDDL. In PPDDL, an MDP specification is split into two parts, the *domain* and the *problem*. The domain consists of a set of typed constants describing the objects in the scenario of interest, a set of typed predicates describing possible relationships among these objects/constants, and a set of parametrized action schemata describing what can be done to the objects. Grounding predicates with objects gives the MDP's state variables, while grounding action schemata yields individual actions. Thus, a domain is a characterization of an MDP's state and action spaces and of the transition function. The problem part of a PPDDL description states the set of literals that hold in the initial state and the goal.

Thus, Figure 3.1 specifies GremlinWorld as a factored goal-oriented MDP with five state variables, *gremlin-alive*, *plane-broken*, *has(Hammer)*, *has(Wrench)*, and *has(Screwdriver)*, abbreviated as G, P, H, W, and S respectively. Therefore, in the factored MDP definition, $\mathcal{X} = \{G, P, H, W, S\}$. The problem involves five actions, $\mathcal{A} = \{$*pick-up(Screwdriver)*, *pick-up(Wrench)*, *pick-up(Hammer)*, *tweak()*, *smack()*$\}$. Each action has a precondition; e.g., the *smack()* action's precondition is the single-literal conjunction $H = has(Hammer)$, so *smack()* can only be used in states where the gremlin has a hammer. Actions' preconditions, effects, and effect probabilities compactly specify the transition function $\mathcal{T}$. For simplicity, we make $\mathcal{C}$ assign the cost of 1 to all actions, which conforms to the restriction on $\mathcal{C}$ imposed by the SSP MDP definition. In general, for the simplicity of exposition, in this chapter we will assume that an action's cost is independent of the state where the action is applied, and denote it, with a slight abuse of notation, as $\mathcal{C}(a)$. We stress, however, that our algorithms do not depend on this assumption and work for arbitrary cost func-

```
(define (domain GremlinWorld)
  (:types tool)
  (:predicates (has ?t - tool)
               (gremlin-alive)
               (plane-broken))
  (:constants Wrench - tool
              Screwdriver - tool
              Hammer - tool)

  (:action pick-up
   :parameters (?t - tool)
   :precondition (and (not (has ?t)))
   :effect (and (has ?t)
                (decrease reward 1)))

  (:action tweak
   :parameters ()
   :precondition (and (has Screwdriver)
                      (has Wrench))
   :effect (and (plane-broken)
                (decrease reward 1)))

  (:action smack
   :parameters ()
   :precondition (and (has Hammer))
   :effect (and (plane-broken)
                (decrease reward 1)
                (probabilistic 0.9
                      (and (not (gremlin-alive))))))
)

(define (problem GremlinProb)
  (:domain GremlinWorld)
  (:init (gremlin-alive))
  (:goal (and (gremlin-alive) (plane-broken)))
)
```

Figure 3.1: A PPDDL-style description of the running example MDP, GremlinWorld, split into domain and problem parts.

```
(:action pick-up-0
 :parameters (?t - tool)
 :precondition (and (not (has ?t)))
 :effect (and (has ?t)))

(:action tweak-0
 :parameters ()
 :precondition (and (has Screwdriver)
                    (has Wrench))
 :effect (and (plane-broken)
              (decrease reward 1)))

(:action smack-0
 :parameters ()
 :precondition (and (has Hammer))
 :effect (and (plane-broken)
              (decrease reward 1)))

(:action smack-1
 :parameters ()
 :precondition (and (has Hammer))
 :effect (and (plane-broken)
              (not (gremlin-alive))
              (decrease reward 1)))
```

Figure 3.2: The all-outcomes determinization of the GremlinWorld domain

tions. The goal set $\mathcal{G}$ consists of all states where the gremlin is alive and the airplane is broken. Finally, we assume that the gremlin starts alive with no tools and the airplane is originally intact, i.e. $s_0 = (G, \neg P, \neg H, \neg W, \neg S)$.

3.2.2 Additional Background

In this subsection, we cover some advanced background topics not mentioned in Chapter 2 but immediately relevant to the topic of this part of the dissertation: domain determinization, inadmissible heuristics, and planning graphs.

Domain Determinization

Successes of some approximate MDP solvers starting with FFReplan [99] have demonstrated the promise of *determinizing* the domain D of the given MDP, i.e., disregarding the probabilities in the transition function, and working only with the state transition graph. The main insight of these approaches is that the determinized MDP can be solved with *classical* planning approaches, e.g., FF [44] or LAMA [83], which are much faster than the traditional optimal MDP solvers. Naturally, solutions to deterministic scenarios are merely plans, not policies; however, the speed of the classical planners allows calling them many times from different states in an MDP determinization and aggregating their plans into a partial policy for the original problem.

Our techniques use the *all-outcomes* determinization [99] D_d of the domain D at hand. Namely, in the example in Figure 3.1, let us denote the precondition of each action as $prec$, an action's outcomes as $o_1, \ldots, o_n$, and their probabilities as $p_1, \ldots, p_n$. The *all-outcomes determinization* D_d , shown for the GremlinWorld domain in Figure 3.2 with predicates and constants omitted for simplicity, contains, for every action a in the original domain, the set of deterministic actions $a_1, \ldots, a_n$, each with a's precondition $prec$ and effect o_i. D_d, coupled with a description of the initial state and the goal, can be viewed as a deterministic MDP in which a plan from a given state to the goal exists if and only if a corresponding trajectory has a positive probability in the original probabilistic MDP with domain D. Importantly, the state of the art in classical planning makes solving a deterministic problem much faster than solving a probabilistic problem of a comparable size. Our abstraction framework exploits these facts to efficiently extract the structure of the given MDP by finding plans in D_d and processing them as shown in Section 3.3.

Inadmissible Heuristics

Recall from Definition 2.30 that a heuristic is a value function that initializes the state values for an MDP algorithm. In heuristic search algorithms such as LRTDP, heuristics help avoid visiting irrelevant states but must be admissible (Definition 2.36) for these methods to converge to an optimal policy. At the same time, *inadmissible* heuristics tend to be more *informative* in practice, approximating V^* better on average. Informativeness often translates into a smaller number of explored states (and the associated memory savings) with reasonable sacrifices in optimality. The concept of heuristic informativeness has no universally accepted definition but, as is commonly done, we adopt the number of states visited by a planner under the guidance of a heuristic as a measure of it. Later in this chapter, we show how basis functions let us derive a highly informative heuristic, GOTH, at the cost of admissibility.

A successful class of MDP heuristics is based on the all-outcomes determinization of the probabilistic domain D at hand [10]. To obtain a value for state s in D, determinization heuristics try to approximate the cost of a plan from s to a goal in an MDP determinization D_d (finding a plan itself even in this relaxed version of an MDP is generally NP-hard). For instance, the FF heuristic [44], denoted h_{FF}, ignores the negative literals (the *delete effects*) in the outcomes of actions in D_d and attempts to find the cost of the cheapest solution to this new relaxed problem. It is inadmissible but, in our experience, is the most informative general MDP heuristic. We use h_{FF} as the baseline to evaluate the performance of GOTH.

Planning Graph

Our work makes use of the planning graph data structure [9], a directed graph alternating between proposition and action "levels". The 0-th level (a proposition level) contains a vertex for each literal present in the initial state s of a given planning problem. Action levels (odd levels) contain vertices for all actions, including a special no-op action, whose preconditions are present (and pairwise "nonmutex") in the previous proposition level. Subsequent proposition levels (even levels) contain all literals from the effects of the actions in the previous action level. Two literals in a proposition level are *mutex* if all actions achieving them are pairwise mutex in the previous action level. Two actions in an action level are mutex if their effects are inconsistent, one's precondition is inconsistent with the other's effect, or their preconditions are mutex in the previous proposition level. As levels increase, additional actions and literals appear (and mutexes disappear) until a fixed point is reached. The deterministic planner Graphplan [9] uses the planning graph as a polynomial-time reachability test for the goal, and we use this structure in a procedure to discover nogoods in Section 3.6.

3.3 Generating State Abstractions

One way to understand the meaning of the state abstractions at the core of the techniques presented in subsequent sections is to examine the process by which these abstractions can be generated. We start by explaining this process and then discuss the properties of the abstractions it produces.

Let an *execution trace* $e = s, a_1, s_1, \ldots, a_n, s_n$, be a sequence where s is the trace's starting state, a_1 is a probabilistic action applied in s that yielded state s_1, and so on. An example of an execution trace from GremlinWorld is $e' = (G, \neg P, \neg H, \neg W, \neg S)$, *pick-up(Hammer)*, $(G, \neg P, H, \neg W, \neg S)$, *smack()*, $(G, P, H, \neg W, \neg S)$.

We define a *trajectory* of an execution trace e to be a sequence

$$t(e) = s, out(a_1, 1, e), \ldots, out(a_n, n, e)$$

where s is e's starting state, and $out(a_k, k, e)$ is a conjunction of literals representing the particular outcome of action a_k that was sampled at the k-th step of e's execution. E.g., $t(e') = (G, \neg P, \neg H, \neg W, \neg S), H, P$ is a trajectory of the example execution trace e'.

We say that $t(e)$ is a *goal trajectory* if the last state s_n of e is a goal state; $t(e')$ just shown is a goal trajectory. A *suffix* of $t(e)$ is a sequence

$$t_i(e) = out(a_i, i, e), \ldots, out(a_n, n, e)$$

for some $1 \leq i \leq n$.

Suppose we are given an MDP and a goal trajectory $t(e)$ of some execution trace of length n in this MDP. Let $prec(a)$ denote the precondition of an action a (a literal conjunction) and $lit(c)$ stand for the set of literals forming conjunction c. Moreover, let a'_i denote the deterministic action whose precondition equals the precondition of the i-th action in $t(e)$, i.e., $prec(a_i)$, and whose effect is denoted as $\textit{effect}(a'_i)$ and equals $out(a_i, i, e)$. In terms of this notation, a trajectory can be viewed as an ordinary deterministic plan.

Now, imagine using the given trajectory $t(e)$ to generate the following sequence of literal conjunctions:

$$
\begin{aligned}
b_n &= \mathcal{G} \\
b_i &= \bigwedge \left[[lit(b_{i+1}) \setminus lit(\mathit{effect}(a'_i))] \cup lit(prec(a'_i)) \right] \text{ for } n-1 \geq i \geq 0
\end{aligned}
$$

This can be done with a simple multistep procedure. We start with $b_n = \mathcal{G}$, the MDP's goal conjunction. Afterwards, at step $i \leq n-1$, we first remove from b_{i+1} the literals of action a'_i's outcome. Then, we conjoin the result to the literals of a_i's precondition, obtaining conjunction b_i. We call this procedure *regression of the goal through trajectory t(e)*, or *regression* for short [78].

As an example, consider regressing trajectory $t(e')$ from GremlinWorld. In this case, $b_2 = \mathcal{G} = G \wedge P$. First we remove from b_2 literal P, the outcome of the *last* action, *smack()*, of e'. The result is G. Then, we add to it the precondition of *smack()*, literal H, producing $G \wedge H$. Thus, $b_1 = G \wedge H$. Similarly, we remove from b_1 the outcome of *pick-up(Hammer)* and add the precondition of this action, which is empty, to the result, obtaining $b_0 = G$. At this point regression terminates.

A *basis function* is defined to be a literal conjunction b produced at some step of regressing the goal through some trajectory. Whenever all literals of a basis function (or of a conjunction of literals in general) are present in state s we say that the conjunction *holds in* or *represents* s. For instance, $b_1 = G \wedge H$ from the above example holds in state $(G, \neg P, H, \neg W, S)$. An alternative view of a basis function b is a mathematical function $f_b : \mathcal{S} \to \{1, \infty\}$ having the value of 1 in all states in which conjunction b holds and ∞ in all others. Due to this equivalence, we will use the term "basis function" to refer to both a conjunction of literals and the corresponding mathematical function.

Basis functions are a central concept behind the algorithms in this chapter, so it is important to understand the intuition behind them. Any goal trajectory is potentially a causally important sequence of actions. Regressing it gives us preconditions for the trajectory's suffixes. Basis functions are exactly these trajectory suffix preconditions. Thus, regression of the trajectories can be thought of as unearthing the relevant causal structure necessary for the planning task at hand. Moreover, our basis functions underlie that causal structure.

There are often many trajectories whose preconditions are consistent with (i.e., are a subconjunction of) a given basis function. We say that a basis function b *enables* a set of goal trajectories T if the goal can be reached from any state represented by b by following any of the trajectories in

T assuming that nature chooses the “right” outcome for each action of the trajectory.

Since each basis function is essentially a precondition (the weakest precondition for a trajectory), it typically holds in many states of the MDP at hand. Therefore, obtaining a goal trajectory $t(e)$ from some state lets us *generalize* this qualitative reachability information to many other states via basis functions yielded by regressing the goal through this trajectory. Moreover, $t(e)$ may have interesting numeric characterizations, e.g. cost, probability of successful execution, etc. To generalize these quantitative descriptions across many states as well, we associate a $weight$ with each basis function. The semantics of basis function weight depends on the algorithm, but in general it reflects the quality of the set of trajectories enabled by the basis function.

Now, consider the value of an MDP’s state. As preconditions, basis functions tell us which goal trajectories are possible from that state. Basis function weights tell us how “good” these trajectories are. Since the quality of the set of goal trajectories possible in a state is a strong indicator of the state’s value, knowing basis functions with their weights allows for approximating the state value function.

As we just showed, a problem’s causal structure can be efficiently derived from its goal trajectories via regression. Thus, a relatively cheap source of trajectories would give us a way to readily extract the structure of the problem. Fortunately, at least two such methods exist. The first one is based on the insight that whenever a trial in an MDP solver such as RTDP or LRTDP (Sections 2.3.2 and 2.3.3) reaches the goal, we get a trajectory “for free”, as a byproduct of the solver’s usual computation. The caveat with using this technique as the primary strategy of getting trajectories is the time it takes an MDP solver’s trials to start attaining the goal. Indeed, the majority of trials at the beginning of planning terminate in states with no path to the goal, and it is at this stage that knowing the problem’s structure would be most helpful for improving the situation. Therefore, our algorithms mostly rely on a different trajectory generation approach. Note that any trajectory in an MDP is a plan in the all-outcomes determinization D_d of that MDP and vice versa. Since classical planners are very fast, we can use them to quickly find goal trajectories in D_d from several states of our choice.

By definition, basis functions represent only the states from which reaching the goal is possible. However, the MDPs we would like to solve also contain another type of states, dead ends, that fall outside of the basis function framework as presented so far. Such states, in turn, can be classified into two kinds; *explicit dead ends*, in which no actions are applicable, and *implicit* ones, which do have applicable actions but no sequence of them leads to the goal with a positive probability. In GremlinWorld, there are no explicit dead ends but every state with literal $\neg G$ is an implicit dead end.

To extend information generalization to dead ends as well, we consider another kind of literal conjunctions that we call *nogoods*. Nogoods' defining property is that any state in which a nogood holds is a dead end. Notice the duality between nogoods and basis functions: both have exactly the same form but give opposite guarantees about a state. Whereas a state represented by a basis function provably cannot be a dead end, a state represented by a nogood certainly is one. Despite the representational similarity, identifying nogoods is significantly more involved than discovering basis functions. Fortunately, the duality between the two allows using the latter to derive the former and collect the corresponding benefits, as one of the algorithms we are about to present, SIXTHSENSE, demonstrates.

3.4 **GOTH** ***Heuristic***

Our presentation of the abstraction framework begins with an example of its use in a heuristic function. As already mentioned, heuristics reduce FIND-AND-REVISE MDP solvers' resource consumption by helping them avoid many of the states (and memoizing corresponding state-value pairs) that are not visited by the final partial policy. The most informative MDP heuristics, e.g., h_{FF}, are based on the all-outcomes determinization of the domain. However, although efficiently computable, such heuristics add an extra level of relaxation of the original MDP, besides determinizing it. For instance, h_{FF} is liable to highly underestimate states' true expected cost of getting to the goal because in addition to discarding the domain's probabilities it ignores actions' delete effects (i.e., negative literals, such as $\neg G$, in actions' outcomes) in the determinized version.

On the other hand, a lot of promise has been shown by several probabilistic planners that solve *full* (non-relaxed) determinizations, e.g., FFReplan, HMDPP [50], and others. It is natural to wonder, then: do the improved heuristic estimates of using a full classical planner on non-relaxed determinized domains provide enough gains to compensate for the potentially increased cost of heuristic computation?

As we show in this section, the answer is "No and Yes". We propose a new heuristic called GOTH (**G**eneralization **O**f **T**rajectories **H**euristic) [56], which *efficiently* produces heuristic state values using deterministic planning. The most straightforward implementation of this idea, in which a classical planner is called every time a state is visited for the first time, does produce better heuristic estimates and reduces search but the cost of so many calls to the classical planner vastly outweighs any benefits. The crucial observation we make is that basis functions provide a way to amortize these expensive planner calls by generalizing the resulting heuristic values to give guidance on similar states. By performing this generalization in a careful manner, one may dramatically reduce the amount of classical planning needed, while still providing more informative heuristic values than heuristics with more levels of relaxation.

3.4.1 GOTH *Description*

In our explanations, we will be referring to GOTH's pseudocode in Algorithm 3.1. Given a factored goal-oriented MDP split into a domain $D = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$ and problem $P = \langle \mathcal{G}, s_0 \rangle$, a heuristic search solver using GOTH starts with GOTH's initialization. During initialization, GOTH determinizes D into its classic counterpart, D_d (line 5). This operation needs to be done only once. Our implementation performs the all-outcomes determinization (Section 3.2), because it is likely to give much better value estimates than the single-outcome one [99]. However, more involved flavors of determinization described in the Related Work section may yield even better estimation accuracy.

Calling a Deterministic Planner

Once D_d has been computed, the probabilistic planner starts exploring the state space. For every state s that requires heuristic initialization, GOTH first checks if it is an explicit dead end (lines 10-10). This check is in place for efficiency, since GOTH should not try to use more expensive methods of analysis on such states.

For a state s that is not an explicit dead end, GOTH tries to obtain a deterministic plan. To do this, GOTH constructs a problem P_s with the original problem's goal and s as the initial state (line 30), feeds P_s along with D_d to a classical planner (line 31), denoted as $\underline{DetPlan}$ in the pseudocode of Algorithm 3.1, and sets a timeout. If s is an implicit dead end, $\underline{DetPlan}$ either proves this or unsuccessfully searches for a plan until the timeout. In either case, it returns without a plan (line 33), at which point s is presumed to be a dead end and assigned a very high penalty value (lines 15-15). If s is not a dead end, $\underline{DetPlan}$ usually returns a plan from s to the goal. The cost of this plan is taken as the heuristic value of s (line 22). Sometimes $\underline{DetPlan}$ may fail to find a plan before the timeout, leading the MDP solver to falsely assume s to be a dead end. In practice, we have not seen this hurt GOTH's performance.

Regression-Based Generalization

By using a full-fledged classical planner, GOTH produces more informative state estimates than h_{FF}, as evidenced by our experiments. However, invoking the classical planner for every newly encountered state is costly; if GOTH did that, it would be prohibitively slow. To ensure speed, we use our insight about the generalization power of basis functions. Whenever GOTH computes a deterministic plan, it regresses it, as described in Section 3.2 and in lines 27-45 of Algorithm 3.1. In the process, it notes down the resulting basis functions with associated weights set to the costs of the regressed plan suffixes (line 39). Then it memoizes these basis function-weight pairs and, if appropriate, updates the already known functions' weights (lines 18, 19). When GOTH encounters a new state s, it minimizes over the weights of all basis functions stored so far that hold in s (lines 12-12). In doing so, GOTH, to a first approximation, sets the heuristic value of s to be the cost of the cheapest currently known trajectory that originates at s (the exact meaning of this value is explained in the next paragraph). Thus, the weight of one basis function can become generalized as the heuristic value of many states. This way of computing a state's value is very fast, and GOTH employs it *before* invoking a classical planner. However, s's heuristic value may be needed even before GOTH has any basis function that holds in s. In this case, GOTH uses the classical planner as described above (line 31), computing a value for s and augmenting its basis function set. Evaluating a state first by generalization (lines 12-12) and then, if generalization fails, by classical planning (lines 13-24) greatly amortizes the cost of each classical solver invocation and drastically reduces the computation time compared to using a deterministic planner alone.

Algorithm 3.1: GOTH Heuristic

```
Input: factored goal-oriented MDP consisting of domain $D = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$ and
problem $P = \langle \mathcal{G}, s_0 \rangle$, timeout $T$, state $s$
Output: a heuristic value of $s$

$D_d \leftarrow \underline{Det}(D)$ // $\underline{Det}$ is a determinization routine, omitted from the pseudocode
$Map \leftarrow$ empty map from basis functions to weights

function GOTH(state $s$, domain $D$, problem $P$, timeout $T$)
begin
    if no action $a \in \mathcal{A}$ is applicable in $s$ then return a large penalty
    else if a nogood holds in $s$ then return a large penalty
    else if some b.f. $f'$ from $Map$ holds in $s$ then return $\min_{\text{b.f.s } f \text{ that hold in } s}\{Map[f]\}$
    else
        $\langle cost\_of\_plan, Map' \rangle \leftarrow$ GetBasisFuncsForS($s, P, T$)
        if $cost\_of\_plan == \infty$ then return a large penalty
        else
            foreach $\langle f, weight \rangle$ in $Map'$ do
                if $f$ is not in $Map$ then insert $\langle f, weight \rangle$ into $Map$
                else update $Map[f]$ by incorporating $weight$ into $Map[f]$'s running average
            end
            if SchedulerSaysYes then learn nogoods from discovered dead ends
            return $cost\_of\_plan$
        end
    end
end

function GetBasisFuncsForS(state $s$, problem $P$, timeout $T$)
begin
    $Map' \leftarrow$ empty map from basis functions to weights
    $P_s \leftarrow \langle \mathcal{G}, s \rangle$
    $plan \leftarrow \underline{DetPlan}(D_d, P_s, T)$ // $\underline{DetPlan}$ is a classical planning routine, omitted
    if $plan == null$ then
        return $\langle \infty, null \rangle$
    else
        $f \leftarrow \mathcal{G}$
        $weight \leftarrow 0$
        foreach $i = length(plan)$ through $1$ do
            $a \leftarrow i$-th action in $plan$
            $weight \leftarrow weight + \mathcal{C}(a)$
            $f \leftarrow \bigwedge[[lit(f) \setminus lit(effect(a))] \cup lit(prec(a))]$
            insert $\langle f, weight \rangle$ into $Map'$
        end
        return $\langle weight, Map' \rangle$
    end
end
```

Weight Updates

Different invocations of the deterministic planner occasionally yield the same basis function more than once, each time potentially with a new weight. Which of these weights should we use? The different weights are caused by a variety of factors, not the least of which are nondeterministic choices made within the classical planner[2]. Thus, the basis function weight from any given invocation may be far from the average cost of the plans for which this basis function is a precondition. For this reason, it is generally beneficial to assign to a basis function the average of the weights computed for it by classical planner invocations so far (line 19). Note that to compute the average we need to keep the number of times the function has been re-discovered (this detail is omitted from the pseudocode for the simplicity of exposition).

Dealing with Implicit Dead Ends

The discussion so far has ignored an important detail. When a classical planner is called on an implicit dead end, by definition, no trajectory is discovered, and hence no basis functions. Thus, this invocation is seemingly wasted from the point of view of generalization: it does not contribute to reducing the average cost of heuristic computation as described thus far.

As it turns out, we can, in fact, amortize the cost of discovery of implicit dead ends in a way similar to reducing the average time of other states' evaluation. To do so, we use the known dead ends along with stored basis functions to derive nogoods — basis functions' duals in our abstraction framework. We remind the reader that nogoods generalize dead ends in precisely the same way as basis functions do with non-dead ends. Thus, nogoods help recognize many dead ends without resorting to classical planning. Our nogood learning mechanism is called SIXTHSENSE and is described in Section 5. It needs to be invoked at several points throughout GOTH's running time as prescribed by a scheduler that is also described in that section. For now, we abstract away the operation of SIXTHSENSE in line 21 of GOTH's pseudocode. With nogoods available, positively deciding whether a state is a dead end is as simple as checking whether any of the known nogoods hold in it (lines 11-11).

[2]For instance, LPG [34], which relies on a stochastic local search strategy for action selection, may produce distinct paths to the goal when invoked twice from the same state, with concomitant differences in basis functions and/or their weights.

Speed and Memory Performance

To facilitate empirical analysis of GOTH, it is helpful to look at the extra speed and memory cost an MDP solver incurs while using it.

Concerning GOTH's memory utilization, we emphasize that, similar to h_{FF} and many other heuristics, GOTH *does not* store any of the states it is given for heuristic evaluation. It merely returns heuristic values of these states to the MDP solver, which can then choose to store the resulting state-value pairs or discard them. However, to compute the values, GOTH needs to memoize the basis functions and nogoods it has extracted so far. As our experiments demonstrate, the size of the set of basis functions and nogoods discovered by GOTH throughout an MDP solver's running time is rather small. It is more than compensated for by the reduction in the number of states explored by the MDP solver thanks to GOTH's informativeness, when compared to h_{FF}.

Timewise, GOTH's performance is largely dictated by the speed of the employed deterministic planner and the number of times the planner is invoked. Another factor that may influence GOTH's speed is determining the "cheapest" basis function that holds in a state (line 12 of GOTH's pseudocode), since it requires iterating, on average, over a constant fraction of the known basis functions. Although fast solutions for this pattern-matching problem are possible, all that we are aware of (e.g., [31]) pay for an increase in speed with degraded memory performance.

Theoretical Properties

Two especially important theoretical properties of GOTH are the informativeness of its estimates and its inadmissibility. The former ensures that, compared to h_{FF}, GOTH causes MDP solvers to explore fewer states. At the same time, like h_{FF}, GOTH is inadmissible. One source of inadmissibility comes from the general lack of optimality of deterministic planners. Even if they were optimal, however, employing timeouts to terminate the classical planner occasionally causes GOTH to falsely assume states to be dead ends. Finally, the basis function generalization mechanism also contributes to inadmissibility. The set of discovered basis functions is almost never complete, and hence even the smallest basis function weight known so far may be an overestimate of a state's true value, as there may exist an even cheaper goal trajectory from this state that GOTH is unaware of. In spite of theoretical inadmissibility, in practice using GOTH usually yields very good policies whose quality is often better than of those found under the guidance of h_{FF}.

3.4.2 Experimental Results

Our experiments compare the performance of a heuristic search-based MDP solver using GOTH to that of the same solver under the guidance of h_{FF} across a wide range of domains. In our experience, h_{FF}, included as part of the miniGPT suite [10], outperforms all other well-known MDP heuristics on most International Probabilistic Planning Competition (IPPC) domains, e.g., the min-min and atom-min heuristics supplied in the same package.

Implementation Details

Our implementation of GOTH is done C++ and uses a portfolio of two classical planners, FF and LPG, to solve the domain determinization. To evaluate a state, it launches both planners as in line 31 of Algorithm 3.1 in parallel and takes the heuristic value from the one that returns sooner. The timeout for each deterministic planner for finding a plan from a given state to a goal was 25 seconds.

Experimental Setup

We tested GOTH and h_{FF} by letting them guide the LRTDP planner (Section 2.3.3) available in miniGPT. Our benchmarks were six probabilistic domains, five of which come from the two most recent IPPCs with goal-oriented problems, IPPC-2006 and IPPC-2008: Machine Shop [70], Triangle Tireworld (IPPC-2008), Exploding Blocksworld (IPPC-2008 version), Blocksworld (IPPC-2006 version), Elevators (IPPC-2006), and Drive (IPPC-2006). All of the remaining domains from IPPC-2006 and IPPC-2008 are either easier versions of the above (e.g., Tireworld from IPPC-2006) or have features not supported by our implementation of LRTDP (e.g., rewards, universal quantification, etc.) so we were not able to test on them. Additionally, we perform a brief comparison of LRTDP+GOTH against FFReplan, since this planner shares some insights with GOTH. In all experiments except measuring the effect of generalization, the planners had a 24-hour limit to solve each problem. All experiments for GOTH, as well as those for ReTrASE and SixthSense, described in sections 3.5.2 and 3.6.2 respectively, were performed on a dual-core 2.8 GHz Intel Xeon processor with 2GB of RAM.

Comparison against h_{FF}

In this subsection, we use each of the domains to illustrate various aspects and modes of GOTH's behavior and compare it to the behavior of h_{FF}. As shown below, on five of the six test domains LRTDP+GOTH substantially outperforms LRTDP+h_{FF}.

We start the comparison by looking at a domain whose structure is especially inconvenient for h_{FF}, Machine Shop. Problems in this set involve two machines and a number of objects equal to the ordinal of the corresponding problem. Each object needs to go through a series of manipulations, of which each machine is able to do only a subset. The effects of some manipulations may cancel the effects of others (e.g., shaping an object destroys the layer of paint sprayed on it). Thus, the order of actions in a trajectory is critical. This domain illuminates the drawbacks of h_{FF}, which ignores delete effects and does not distinguish good and bad action sequences as a result. Machine Shop has no dead ends.

Figures 3.3 and 3.4 show the speed and memory performance of LRTDP equipped with the two heuristics on those problems from MachineShop (and two other domains) that at least one these planners could solve without running out of memory. As implied by the preceding discussion of GOTH's space requirements, the memory consumption of LRTDP+GOTH is measured by the number of states, basis functions, and nogoods whose values need to be maintained (GOTH caches the basis functions and LRTDP caches the states). In the case of LRTDP+h_{FF} all memory used is only due to LRTDP's state caching because h_{FF} by itself does not memoize anything. On Machine Shop, the advantage of LRTDP+GOTH is clearly vast, reaching several orders of magnitude. In fact, LRTDP+h_{FF} runs out of memory on the three hardest problems, whereas LRTDP+GOTH is far from that.

Concerning policy quality, we found the use of GOTH to yield optimal or near-optimal policies on Machine Shop. This contrasts with h_{FF} whose policies were on average 30% more costly than the optimal ones.

The structure of the Triangle Tireworld domain, unlike Machine Shop's, is not particularly adversarial for h_{FF}. However, LRTDP+GOTH noticeably outperforms LRTDP+h_{FF} on it too, as Figures 3.3 and 3.4 indicate. Nonetheless, neither heuristic saves enough memory to let LRTDP solve past problem 8. In terms of solution quality, both planners find optimal policies on the prob-

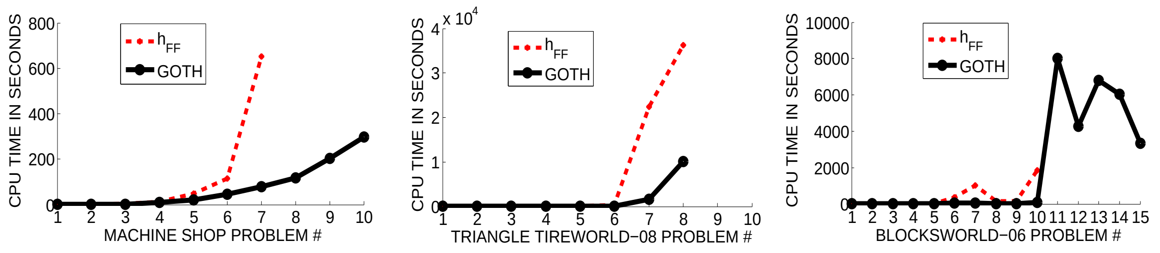


Figure 3.3: GOTH outperforms h_{FF} on Machine Shop, Triangle Tireworld, and Blocksworld in speed by a large margin.

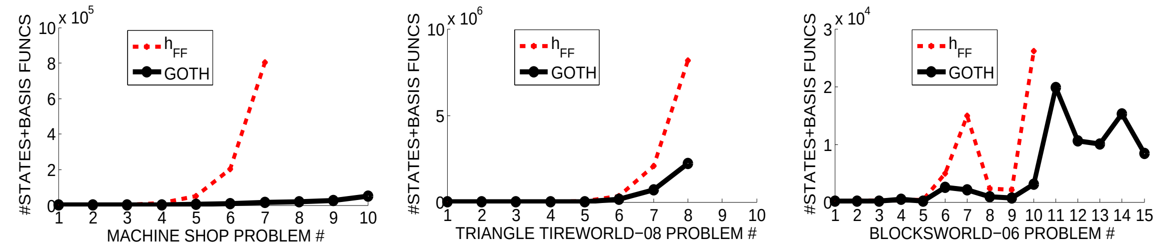


Figure 3.4: GOTH's advantage over h_{FF} on Machine Shop, Triangle Tireworld, and Blocksworld in memory is large as well.

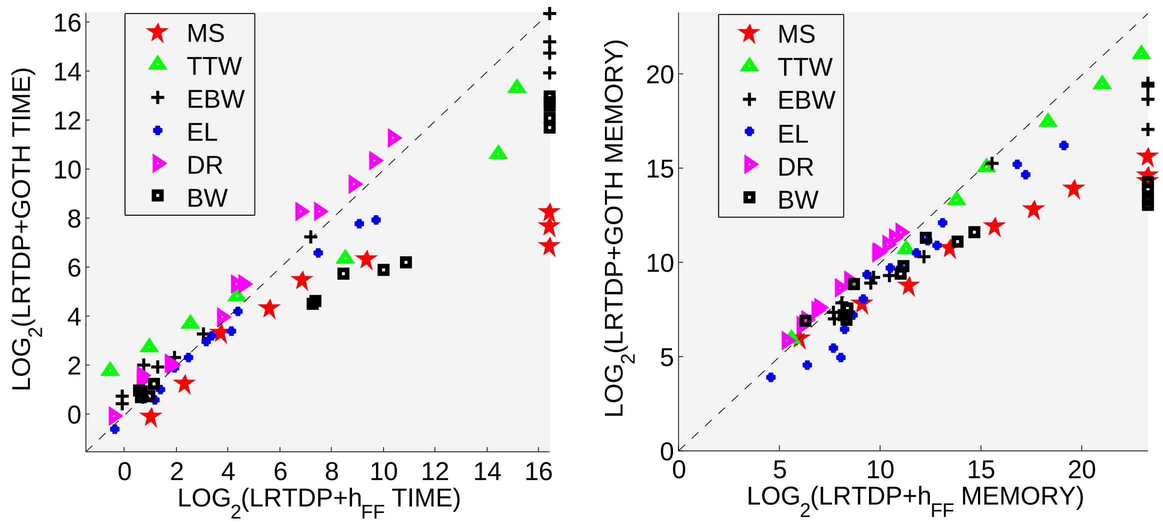


Figure 3.5: The big picture: GOTH provides a significant advantage on large problems. Points below the dashed diagonals correspond to problem instances on which LRTDP+GOTH did better than LRTDP+h_{FF}, i.e. used less time/memory. Note that the axes are on a Log scale.

lems they can solve.

The results on Exploding Blocksworld (EBW, Figure 3.5) are similar to those on Triangle Tireworld, where the LRTDP+GOTH's more economical memory consumption eventually translates to a speed advantage. Importantly, however, on several EBW problems LRTDP+GOTH is superior to LRTDP+h_{FF} in a more illustrative way: it manages to solve four problems on which LRTDP+h_{FF} runs out of space. The policy quality under the guidance of either heuristic is nearly identical.

The Drive domain is small, and using GOTH on it does not provide significant benefit. On Drive problems, planners spend most of the time in decision-theoretic computation (rather than the computation of heuristic values) but explore no more than around 2000 states. LRTDP under the guidance of GOTH and h_{FF} explores roughly the same number of states, but since this number is small, generalization does not play a big role and GOTH incurs the additional overhead of maintaining the basis functions without getting a significant benefit from them. Perhaps surprisingly, however, GOTH sometimes leads LRTDP to find policies with higher success rates (coverage), while never causing it to find worse policies than h_{FF}. The difference in policy quality reaches 50% on the Drive domain's largest problems. Reasons for this are a topic for future investigation.

On the remaining test domains, Elevators and Blocksworld, LRTDP+GOTH dominates LRTDP+h_{FF} in both speed and memory while providing policies of equal or better quality. Figures 3.3 and 3.4 show the performance on Blocksworld as an example. Classical planners in our portfolio cope with determinized versions of these domains very quickly, and abstraction ensures that the obtained heuristic values are spread over many states. Similar to the case of EBW, the effectiveness of GOTH is such that LRTDP+GOTH can solve even the five hardest problems of Blocksworld, which LRTDP+h_{FF} could not.

Figure 3.5 provides the big picture of the comparison. For each problem we tried, it contains a point whose coordinates are the logarithms of the amount of time/memory that LRTDP+GOTH and LRTDP+h_{FF} took to solve that problem. Thus, points that lie below the $Y = X$ line correspond to problems on which LRTDP+GOTH did better according to the respective criterion. The axes of the time plot of Figure 3.5 extend to $\log_2(86400)$, the logarithm of the time cutoff (86400s, i.e., 24 hours) that we used. The points that lie on the extreme right or top of these plots denote problems that could not be solved under the guidance of at least one of the two heuristics. Overall, the time plot shows that, while LRTDP+GOTH ties with or is slightly beaten by LRTDP+h_{FF} on Drive and smaller problems of other domains, it enjoys a comfortable advantage on most large problems. In terms of memory, this advantage extends to most medium-sized and small problems as well, and sometimes translates into a qualitative difference, allowing LRTDP+GOTH to handle problems that

EBW	EL	TTW	DR	MS	BW
2.07	4.18	1.71	1.00	14.40	7.72

Table 3.1: Average ratio of the number of states memoized by LRTDP under the guidance of h_{FF} to the number under GOTH across each test domain. The bigger these numbers, the more memory GOTH saves the MDP solver compared to h_{FF}.

LRTDP+h_{FF} cannot.

Why does GOTH's and h_{FF}'s comparative performance differ from domain to domain? For an insight, refer to Table 3.1. It displays the ratio of the number of states explored by LRTDP+h_{FF} to the number explored by LRTDP+GOTH, averaged for each domain over the problems that could be solved by both planners. As mentioned in Section 3.2.2, we take the number of states explored under the guidance of a heuristic to be a measure of the heuristic's informativeness, Thus, Table 3.1 reflects the relative informativeness of GOTH and h_{FF}. Note the important difference between the data in this chart and memory usage as presented on the graphs: the information in the table disregards memory consumption due to the heuristics, thereby separating the description of heuristics' informativeness from a characterization of their efficiency. Associating the data in the table with the relative speeds of LRTDP+h_{FF} and LRTDP+GOTH on the test domains reveals a clear trend; the size of LRTDP+GOTH's speed advantage is strongly correlated with its memory advantage, and hence with its advantage in informativeness. In particular, GOTH's superiority in informativeness is not always sufficient to compensate for its computation cost. Indeed, the $1.71\times$ average reduction (compared to h_{FF}) in the number of explored states on Triangle Tireworld is barely enough to make good the time spent on deterministic planning (even with generalization). In contrast, on domains like Blocksworld, where GOTH causes LRTDP to visit many times fewer states than h_{FF}, LRTDP+GOTH consistently solves the problems much faster.

Benefit of Generalization

Our main hypothesis regarding GOTH has been that generalization is vital for making GOTH computationally feasible. To test it and measure the importance of basis functions and nogoods for GOTH's operation, we ran a version of GOTH with generalization turned off on several domains, i.e., with the classical planner being invoked from every state passed to GOTH for evaluation.

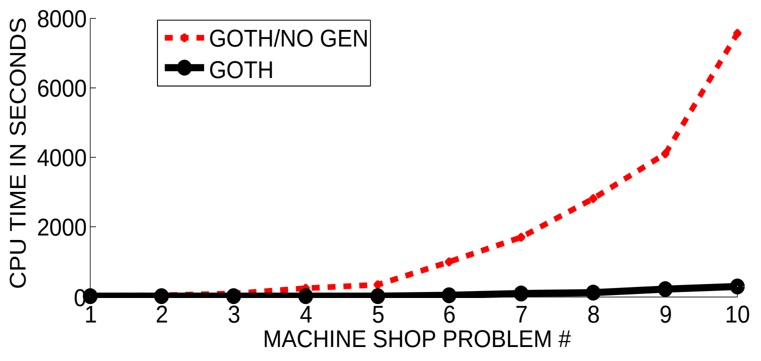


Figure 3.6: GOTH is much faster with generalization than without.

(As an aside, note that this is akin to the strategy of FFReplan, with the fundamental difference that GOTH's state values are eventually overridden by the decision-theoretic training process of LRTDP. We explore the relationship between FFReplan and GOTH further in the next subsection.)

As expected, GOTH without generalization proved to be vastly slower than full GOTH. For instance, on Machine Shop LRTDP+GOTH with generalization turned off is approximately 30-40 times slower (Figure 3.6) by problem 10, and the gap is growing at an alarming rate, implying that without our generalization technique the speedup over h_{FF} would not have been possible at all. On domains with implicit dead ends, e.g., Exploding Blocksworld, the difference is even more dramatic, reaching over two orders of magnitude.

Furthermore, at least on the relatively small problems on which we managed to run LRTDP +GOTH/NO GEN, we found the quality of policies (measured by the average trajectory length) yielded by "generalized" GOTH to be typically *better* than with generalization off. This result is somewhat unexpected, since generalization is an additional layer of approximation on top of determinizing the domain. We attribute this phenomenon to "generalized" GOTH's weight-averaging update strategy (line 19 of Algorithm 3.1; see also the corresponding discussing in subsection *Weight Updates* of Section 3.4.1). As pointed out earlier, the weight of a basis function (i.e., the length of a plan, in the case of GOTH/NO GEN) from any single classical planner invocation may not be reflective of the basis function's quality, and GOTH/NO GEN will suffer from such noise more than regular GOTH. In any event, even if GOTH without generalization yielded better policies, its slowness would make its use unjustifiable in practice.

One may wonder whether generalization can also benefit h_{FF} the way it helped GOTH. While we have not conducted experiments to verify this, we believe the answer is “No”. Unlike full deterministic plan construction, finding a relaxed plan sought by h_{FF} is much easier and faster. Considering that the generalization mechanism involves iterating over many of the available basis functions to evaluate a state, any savings that may result from avoiding h_{FF}'s relaxed plan computation will likely be negated by this iteration.

Computational Profile

An interesting aspect of GOTH's modus operandi is the fraction of the computational resources an MDP solver uses that is due to GOTH. E.g., across the Machine Shop domain, LRTDP+GOTH spends 75-90$\%$ of the time in heuristic computation, whereas LRTDP+h_{FF} only 8-17$\%$. Thus, GOTH is computationally much heavier than h_{FF} but causes LRTDP to spend drastically less time exploring the state space.

Comparison with FFReplan

Before continuing, we would like to emphasize again that, in contrast to FFReplan, GOTH is a heuristic function meant to guide a FIND-AND-REVISE MDP solver and is not a standalone planner by itself. Nonetheless, one can find similarities between GOTH's and FFReplan's underlying ideas. Indeed, both employ deterministic planners, FFReplan — for action selection directly, while GOTH — for state evaluation. However, since GOTH is not a self-contained planner, it lets a dedicated MDP solver correct its judgment. As a consequence, even though GOTH per se ignores probabilistic information in the domain, probabilities are (or can be) taken into account during the solver's search for a policy. FFReplan, on the other hand, ignores them entirely. Due to this discrepancy, performances of FFReplan and a planner guided by GOTH are typically very distinct. For instance, FFReplan is faster than most decision-theoretic planners. At the same time, FFReplan has difficulty dealing with probabilistic subtleties. It is known to come up with very low success rate policies on *probabilistically interesting* problems, e.g., on almost all problems of Triangle Tireworld'06 [65]. In these MDPs, “cheap” trajectories are easy to accidentally deviate from into costly areas of the state space, a pitfall that determinization-based planners like FFReplan often fail to recognize. LRTDP+GOTH can handle such scenarios much better: as stated above, it produces optimal, 100% success-rate policies on the first eight out of ten problems of the even harder version of Triangle Tireworld that appeared at IPPC-2008.

3.4.3 Summary

GOTH is a heuristic function that provides a heuristic search-based MDP solver with informative state value estimates using costs of plans in the deterministic version of the given MDP. Computing such plans is expensive. To amortize the time spent on their computation, GOTH employs basis functions, which generalize the cost of one plan/trajectory to many states. As the experiments show, this strategy and the informativeness of state value estimates make GOTH into a more effective heuristic than the state of the art, h_{FF}.

3.5 RETRASE

In GOTH, the role of information transfer via basis functions and nogoods was primarily to reuse computation in the form of classical planner invocations and thus save time. In this section, we present an MDP solver called RETRASE, **Re**gressing **Tr**ajectories for **A**pproximate **S**tate **E**valuation [55, 58] that employs basis functions in a similar way but this time chiefly for the purpose of drastically reducing the memory footprint.

As we already discussed, many dynamic programming-based MDP algorithms such as VI suffer from the same critical drawback — they represent the state value function extensionally, i.e., as a table, thus requiring memory (and time) exponential in the number of state variables of a factored MDP. Since this extensional representation grows very rapidly, these approaches do not scale to handle large problems, e.g., the largest ones from the IPPC.

Two broad approaches have been proposed for avoiding creation of a state-value table. One method consists in computing the policy *online* with the help of a domain determinization, e.g., the all-outcomes one. In online settings, the policy needs to be decided on-demand, only for the current state at each time step. Once an action for the current state is selected, the agent executes it, transitions to another state, and the process repeats. This makes maintaining a state-value table unnecessary (although potentially useful). Running a classical planner on a domain determinization helps choose an action in the current state without resorting to this table. Determinization-based planners, e.g., FFHop [100], are often either slow due to invoking a classical planner many times or, as in the case of FFReplan, disregard the probabilistic nature of actions and have trouble with probabilistically interesting domains [65].

The other strategy, dimensionality reduction, maps the MDP state space to a parameter space of lower dimension. Typically, the mapping is done by constructing a small set of basis functions, learning weights for them, and combining the weighted basis function values into the values of states. Researchers have successfully applied dimensionality reduction by manually defining a domain-specific basis function set in which basis functions captured some human intuition about the domain at hand. It is relatively easy to find such a mapping in domains with *ordinal* (e.g., numeric) state variables, especially when the numeric features correlate strongly with the value of the state, as in Gridworlds, Sysadmin, and FreeCraft [38, 39, 36]. In contrast, dimensionality reduction is difficult to use in *nominal* (also known as *discrete* or *logical*) domains, such as those used in the IPPC. Besides not having metric quantities, there is often no valid distance function between states (indeed, the distance between states is usually asymmetric and violates the triangle equality). It is extremely hard for a human to devise basis functions or a reduction mapping in nominal domains. The focus of section is an automatic procedure for doing so.

To our knowledge, there has been little work on mating decision theory, determinization, and dimensionality reduction. With the RETRASE algorithm, we are bridging the gap, proposing a fusion of these ideas that removes the drawbacks of each. RETRASE learns a compact value function approximation successful in a range of nominal domains. As with GOTH, the cornerstone of RETRASE is the abstraction framework described in earlier sections. In particular, in the same way as GOTH, RETRASE constructs the approximation by obtaining a set of basis functions automatically with the help of planning in a determinized version of the domain at hand. However, being a full probabilistic planner, unlike GOTH, it also *learns* the weights for these basis functions by the decision-theoretic means and aggregates them to compute state values as other dimensionality-reduction methods do. Thus, as opposed to GOTH, RETRASE tries to incorporate the probabilistic information lost at the determinization stage back into the solution. The set of basis functions is normally much smaller than the set of reachable states, thus giving our planner a large reduction in memory requirements as well as in the number of parameters to be learned, while the implicit reuse of classical plans thanks to basis functions makes it fast.

We demonstrate the practicality of RETRASE by comparing it to the top IPPC-2004, 06 and 08 performers and other state-of-the-art planners on challenging problems from these competitions. RETRASE demonstrates orders of magnitude better scalability than the best optimal planners, and frequently finds significantly better policies than the top-performing approximate solvers.

3.5.1 RETRASE *Description*

The main intuition underlying RETRASE is that extracting basis functions defined in Section 3.3 from an MDP is akin to mapping the MDP to a lower-dimensional parameter space. In practice, this space is much smaller than the original state space, since only the relevant causal structure is retained[3], giving us large reduction in space requirements. Solving this new problem amounts to learning weights, a quantitative measure of each basis function's quality. There are many imaginable ways to learn them; in this dissertation, we explore one such method — a modified version of RTDP.

The weights reflect the fact that basis functions differ in the total expected cost of goal trajectories they enable as well as in the total probability of these trajectories. At this point, we stress that RETRASE makes two approximations on its way to computing an MDP's value function, and one of them is related to the semantics of basis function weights and importance. Any given basis function enables only some subset W of the goal trajectories in a given state and is oblivious to all other trajectories originating in that state. The other trajectories may or may not be preferable to the ones in W (e.g., because the former may lead the agent to the goal with 100% probability). Therefore, the importance of the trajectories (and hence of corresponding basis functions!) depends on the state. Our intuitive notion of weights ignores this subtlety, since in RETRASE, the weight of a basis function does not vary with states in which this basis function holds. Thus, a weight as we compute it here is, in effect, a reflection of the "average" importance of the corresponding basis function across the states this basis function represents. This is the first approximation made by RETRASE.

The above details notwithstanding, the differences among basis function weights exist also because each trajectory considers only one outcome of each of its actions. The sequence of outcomes

[3]We can also strictly limit the size of this space by putting a bound on the number of basis functions we are willing to handle.

Algorithm 3.2: RETRASE

```
Input: factored goal-oriented MDP consisting of domain $D = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$ and
problem $P = \langle \mathcal{G}, s_0 \rangle$, trial length $L$, timeout $T$
Output: a policy closed w.r.t. $s_0$

$D_d \leftarrow \underline{Det}(D)$ // $\underline{Det}$ is a determinization routine, omitted from the pseudocode
$Map \leftarrow$ empty map from basis functions to weights
$DE \leftarrow$ empty set of dead ends

function RETRASE(domain $D$, problem $P$, trial length $L$, timeout $T$)
begin
    // Do modified RTDP over the basis functions
    repeat
        $s \leftarrow s_0$
        foreach num_steps $= 1$ through $L$ do
            $\hat{a} \leftarrow \arg\min_a \{\mathcal{C}(a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')$Value$(s', P, T)\}$
            ModifiedBellmanBackup$(s, \hat{a}, D, P, T)$
            $s \leftarrow$simulate action $\hat{a}$ in $s$
        end
    until until stopped;
    return a policy $\pi_{s_0}$ closed w.r.t. $s_0$ and greedy w.r.t. the value function induced by Value(.)
end

function Value(state $s$, problem $P$, timeout $T$)
begin
    if $s \in DE$ then return a large penalty
    else if some member $f'$ of $Map$ holds in $s$ then return $\min_{\text{b.f.s } f \text{ that hold in } s}\{Map[f]\}$
    else
        $\langle cost\_of\_plan, Map' \rangle \leftarrow$ GetBasisFunctionsForS$(s, P, T)$ // see Algorithm 3.1
        if $cost\_of\_plan == \infty$ then
            insert $s$ into $DE$
            return a large penalty
        else
            foreach $\langle f, weight \rangle$ in $Map'$ do
                if $f$ is not in $Map$ then insert $\langle f, weight \rangle$ into $Map$
            end
            return $cost\_of\_plan$
        end
    end
end

function ModifiedBellmanBackup(state $s$, action $a$, domain $D$, problem $P$, timeout $T$)
begin
    foreach b.f. $f$ that holds in $s$ and enables $a$ do
        $Map[f] \leftarrow \mathcal{C}(a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')$Value$(s', P, T)$
    end
end
```

the given trajectory considers may be quite unlikely. In fact, getting some action outcomes that the trajectory does not consider may prevent the agent from ever getting to the goal. Thus, since each trajectory has an associated basis function, it may be much "easier" to reach the goal in the presence of some basis functions than others.

Now, given that each state is generally represented by several basis functions, what is the connection between the state's value and their weights? In general, the relationship is quite complex: under the optimal policy, trajectories enabled by several basis functions may be possible. However, determining the subset of basis functions enabling these trajectories is at least as hard as solving the MDP exactly. Instead, we approximate the value of a state by the *minimum* weight among all basis functions that represent the state. This amounts to saying that the "better" a state's "best" basis function is, the "better" is the state itself, and is the second approximation RETRASE makes.

Thus, deriving useful basis functions and their weights gives us an approximation to the optimal value function.

Algorithm's Operation

The pseudocode of RETRASE is presented in Algorithm 3.2. For a step-by-step example of RETRASE's operation please refer to the proof of Theorem 3.1. RETRASE starts by computing the determinization D_d of the domain (line 5). Like GOTH, it uses D_d to rapidly compute the basis functions. The algorithm explores the state space by running modified RTDP trials (lines 12-19), memoizing all the dead ends and basis functions it learns along the way. Whenever during state evaluation (line 15) RETRASE finds a state that is neither a known dead end nor has any basis functions that hold in it, RETRASE uses the regression procedure **GetBasisFuncsForS(.)** (line 28) outlined in Algorithm 3.1 to generate a basis function for it. Regression yields not only the basis functions but also an approximate cost of reaching the goal in D_d from any state with a given basis function via the given plan. We use these values to initialize the corresponding basis functions' weights (lines 33-35). As in GOTH, if the deterministic planner can prove the non-existence of a plan or simply cannot find a plan before some timeout, the state in question is deemed to be a dead end (lines 30-31).

For each state s visited by the modified RTDP, the **ModifiedBellmanBackup(.)** routine updates the weight of each basis function that enables the execution of the currently optimal action a (lines 41-46). The expected cost of taking a becomes the new weight of each such basis function. The intuitive reason for updating the basis functions enabling a is that a can be executed in any state where these basis functions hold; hence, the quality of a should be reflected in these basis functions' weights. Conversely, a cannot be executed wherever basis functions that do not enable it hold, so the expected cost of taking a is irrelevant to determining the weights of those basis functions.

Theoretical Properties

A natural question about ReTrASE is that of convergence. To answer it, we prove the following negative result:

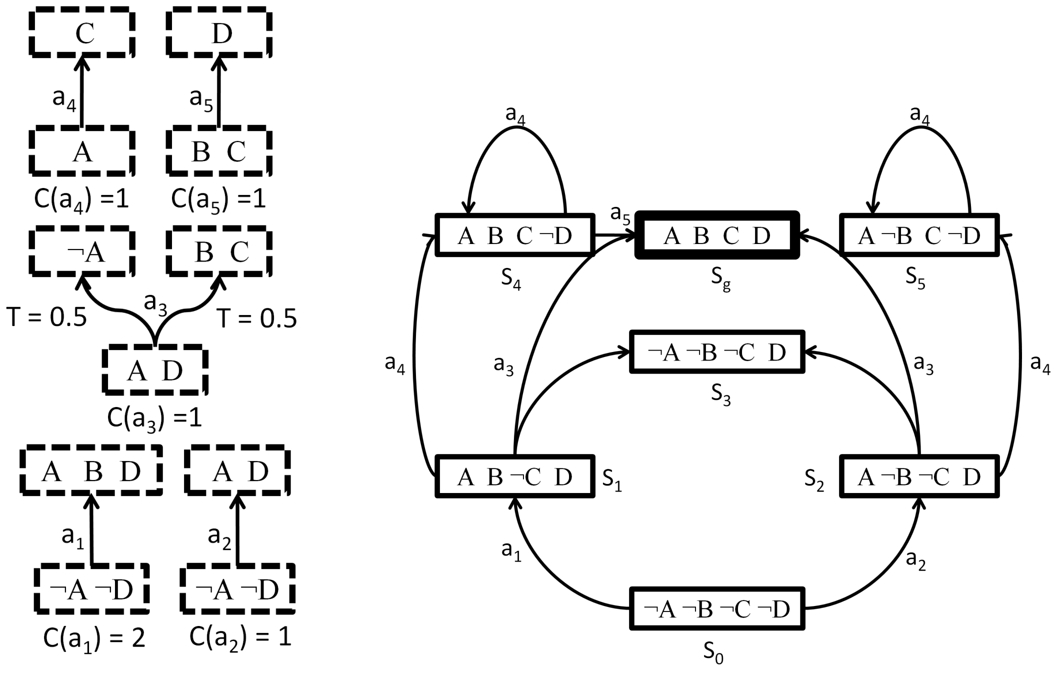
Theorem 3.1. *There are problems on which* ReTrASE *may not converge.* ◇

Proof. By the theorem statement we mean that, on some problems, depending on the order in which basis functions are discovered, ReTrASE may indefinitely oscillate over a set of several policies with different expected costs. One such MDP M is presented in Figure 3.7, which shows M's transition graph and action set. Solving M amounts to finding a policy of minimum expected cost that takes the agent from state s_0 to state s_g and uses actions a_1 — a_5. The optimal solution to M is a linear plan $s_0 - a_1 - s_1 - a_4 - s_4 - a_5 - s_g$.

To see that ReTrASE fails to converge on M, we simulate ReTrASE's operation on this MDP. Recall that ReTrASE executes a series of trials, all originating at s_0.

Trial 1. To choose an action in s_0, ReTrASE needs to evaluate states s_1 and s_2. It does not yet have any basis functions to do that, so it uses the **GetBasisFuncsForS(.)** procedure in Algorithm 3.1 to generate them, together with initial estimates for their weights.

Suppose the procedure first looks for a basis function for s_1 and finds the plan $s_1 - a_4 - s_4 - a_5 - s_g$. Regressing it yields the following basis function-weight pairs: $weight(A \wedge B \wedge C \wedge D) = 0$, $weight(A \wedge B \wedge C) = 1$, $weight(A \wedge B) = 2$. $A \wedge B$ is the only basis functions that holds in s_1 so far. Therefore, the current estimate for the value of s_1, $V(s_1)$, is 2. Accordingly, the current estimate for the value of action a_1 in s_0, $Q^V(s_0, a_1)$, becomes $\mathcal{C}(a_1) + V(s_1) = 4$.



(a) Actions (b) Transition graph

Figure 3.7: An example MDP on which RETRASE fails to converge that has four state variables, A, B, C, and D, five actions shown in part a) of the figure, and the transition graph induced by them shown in part b) of the figure.

Next, suppose that for state s_2, **GetBasisFuncsForS(.)** finds the plan $s_2 - a_3 - s_g$. Regressing it yields one basis function-weight pair in addition to the already discovered ones, $weight(A \wedge D) = 1$. Function $A \wedge D$ is the only one that holds in s_2, so we get $V(s_2) = 1$ and $Q^V(s_0, a_2) = 2$.

Now RETRASE can choose an action in s_0. Since at the moment $Q^V(s_0, a_1) > Q^V(s_0, a_2)$, it picks a_2 and executes it, transitioning to s_2.

In s_2, RETRASE again needs to evaluate two actions, a_3 and a_4. Notice that a_4 leads to s_5, which is a dead end. **GetBasisFuncsForS(.)** discovers this fact by failing to produce any basis functions for s_5. Thus, $V(s_5)$ is a very large dead-end penalty, e.g. 1000000, yielding $Q^V(s_2, a_4) = 1000001$. However, a_3 may also lead to a dead end, s_3, with $P = 0.5$, so $Q^V(s_2, a_3) = 500001$. Nonetheless, a_3 is more preferable, so this is the action that RETRASE picks in s_2.

At this time, RETRASE performs a modified Bellman backup in s_2. The only known basis function that holds in s_2 and enables the chosen action a_3 is $A \wedge D$. Therefore, RETRASE sets

$weight(A \wedge D) = Q^V(s_2, a_3) = 500001$.

Executing a_3 in s_2 completes the trial with a transition either to goal s_g or to dead end s_3.

Trial 2. This time, ReTrASE can select an action in s_0 without resorting to regression. Currently, $V(s_1) = 2$, since $A \wedge B$ with $weight(A \wedge B) = 2$ is the minimum-weight basis function in s_1. However, $V(s_2) = 500001$ due to the backup performed during trial 1. Therefore, $Q^V(s_0, a_1) = 4$ but $Q^V(s_0, a_2) = 500002$, making a_1 look more attractive. So, ReTrASE chooses a_1, causing a transition to s_1.

In s_1, the choice is between a_3 and a_4. The values of both are easily calculated with known basis functions, $Q^V(s_1, a_3) = 500001$ and $Q^V(s_1, a_4) = 2$.

The natural choice is a_4, and ReTrASE performs the corresponding backup. The basis functions enabling a_4 in s_1 are $A \wedge B$ and $A \wedge D$. Their weights become $Q^V(s_1, a_4) = 2$ after the update.

The rest of the trial does not change any weights and is irrelevant to the proof.

Trial $n \geq 3$. Crucially, basis function $A \wedge D$, whose weight changed in the previous trials, holds both in state s_1 and in state s_2. Due to the update in s_2 during trial 1, $weight(A \wedge D)$ became large and made s_1 look beneficial. On the other hand, thanks to the update in s_1 during trial 2, $weight(A \wedge D)$ became small and made s_2 look beneficial. It is easy to see that this cycle will continue in subsequent trials: in the odd ones, ReTrASE will prefer action a_1 in state s_0, and in the even ones it will prefer a_2 in s_0. As a result, ReTrASE will keep on switching between two policies, one of which is suboptimal. ■

Overall, the classes of problems on which ReTrASE may diverge is difficult to characterize. Predicting whether ReTrASE may diverge on a particular problem is an area for future work. We maintain, however, that a lack of theoretical guarantees is not indicative of a planner's practical performance. Indeed, several IPPC winners, including FFReplan, have a weak theoretical profile. The experimental results show that ReTrASE too performs very well on many of the planning community's benchmark problems.

3.5.2 Experimental Results

Our goal in this subsection is to demonstrate two important properties of ReTrASE – (1) scalability and (2) quality of solutions in complex, probabilistically interesting domains. We start by showing that ReTrASE easily scales to problems on which the state-of-the-art optimal and non-determinization-based approximate planners run out of memory. Then, we illustrate ReTrASE's ability to compute better policies for hard problems than state-of-the-art approximate solvers.

Implementation Details

ReTrASE is implemented in C++ and uses the miniGPT [10] package's code for RTDP as its basis. Our implementation is in the prototype stage and does not fully support some of the PPDDL language features used to describe IPPC problems (e.g. universal quantification, disjunctive goals, rewards, etc.).

Experimental Setup

We report results on six problem sets — Triangle Tireworld (TTW) from IPPC-2006 and -2008, Drive from IPPC-2006, Exploding Blocksworld (EBW) from IPPC-2006 and -2008, and Elevators from IPPC-2006. In addition, we ran ReTrASE on a few problems from IPPC-2004. Since our implementation does not yet support such PPDDL features as universal quantification, we were unable to test on the remaining domains from these competitions. However, we emphasize that most of the six domains we evaluate on are probabilistically interesting and hard. Even the performance of the best IPPC participants on most of them leaves a lot of room for improvement, which attests to their informativeness as testbeds for our planner.

To provide a basis for comparison, for each of the above domains we also present the results of the best IPPC participants. Namely, we give the results of the IPPC winner on that domain, of the overall winner of that IPPC, and ours. For the memory consumption experiment, we ran two VI-family planners, LRTDP with inadmissible h_{FF} (LRTDP+h_{FF}), and LRTDP+OPT — LRTDP with the admissible Atom-Min-1-Forward|Min-Min heuristic [10]. Both are among the top-performing decision-theoretic planners.

We ran ReTrASE on the test problems under the restrictions resembling those of IPPC-2006

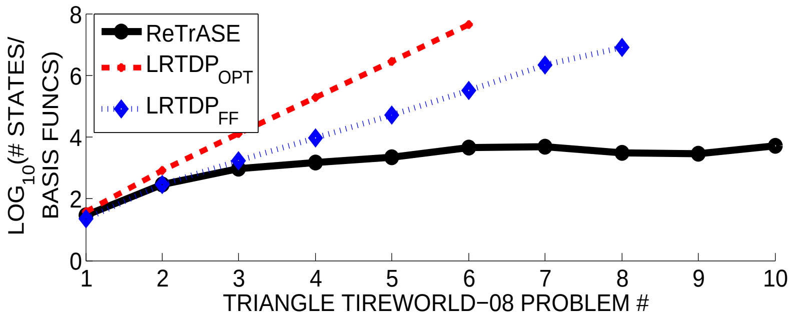


Figure 3.8: Memory usage on logarithmic scale: RETRASE is much more efficient than both LRTDP+OPT and LRTDP+h_{FF}.

and -2008. Namely, for each problem, RETRASE had a maximum of 40 minutes for training, as did all the planners whose results we present here. RETRASE then had 30 attempts to solve each problem. In IPPC-2006 and -2008, the winner was decided by the *success rate* of their policy — the percentage of 30 execution trials of their policy that finished in a goal state. Accordingly, on the relevant graphs we present both RETRASE's success rate and that of its competitors.

While analyzing the results, it is important to be aware that our RETRASE implementation is not optimized. Consequently, RETRASE's potential efficiency is likely even better than indicated by the experiments.

Comparing Scalability

We begin by showcasing the memory savings of RETRASE over traditional, decision-theoretic planners exemplified by LRTDP+OPT and LRTDP+h_{FF} on the Triangle Tireworld domain. Figure 3.8 demonstrates the savings of RETRASE to increase dramatically with problem size. In fact, neither LRTDP variant is able to solve past problem 8 as both run out of memory, whereas RETRASE copes with all ten problems. Scalability comparisons for other domains we tested on yield generally similar results.

Non-decision-theoretic approximate algorithms (e.g., the determinization-based ones) do not suffer from the scalability issues as much as LRTDP. Thus, it is more meaningful to compare RETRASE against them in terms of the quality of produced solutions. As we show, RETRASE's scalability allows it to successfully compete on IPPC problems with any participant.

Comparing Solution Quality: Success Rate

Continuing with the Triangle Tireworld domain, we compare the success rates of ReTrASE, RFF [95] — the overall winner of IPPC-2008, and HMDPP [50] — the winner of IPPC-2008 on this particular domain. Note that Triangle Tireworld, perhaps the most famous probabilistically interesting domain, was designed largely to confound solvers that rely on domain determinization [65], e.g., FFReplan; therefore, performance on it is particularly important for evaluating a new determinization-based planner. As Figure 3.9 shows, on this domain ReTrASE ties with HMDPP by achieving the maximum possible success rate, 100%, on all ten problems and outperforms the competition winner, which cannot solve problem 10 at all and achieves only 83%-success rate on problem 9.

On the IPPC-2006 Drive domain, ReTrASE also fares well (Figure 3.10). Its average success rate is just ahead of the unofficial domain winner (FFReplan) and of the IPPC-2006 winner (FPG [19]), but the differences among all three are insignificant.

For the Exploding Blocksworld domain on the IPPC-2006 version (Figure 3.11), ReTrASE dominates other planners by a considerable margin on almost every problem. Its edge is especially noticeable on the hardest problems, 11 through 15. On the most recent EBW problem set, from IPPC-2008 (Figure 3.12), ReTrASE performs well too. Even though its advantage is not as apparent as in IPPC-2006, it is nonetheless ahead of its competition in terms of the average success rate.

The Elevators and Triangle Tireworld-06 domains are easier than the ones presented above. Surprisingly, on many of the Elevators problems ReTrASE did not converge within the allocated 40 minutes and was outperformed by several planners. We suspect this is due to bad luck ReTrASE has with basis functions in this domain. However, on TTW-06 ReTrASE was the winner on every problem.

Comparing Solution Quality: Expected Cost

On problems where ReTrASE achieves the maximum success rate it is interesting to ask how close the expected trajectory cost that its policy yields is to the optimal. The only way we could find out the expected cost of an optimal policy for a problem is by running an optimal planner on

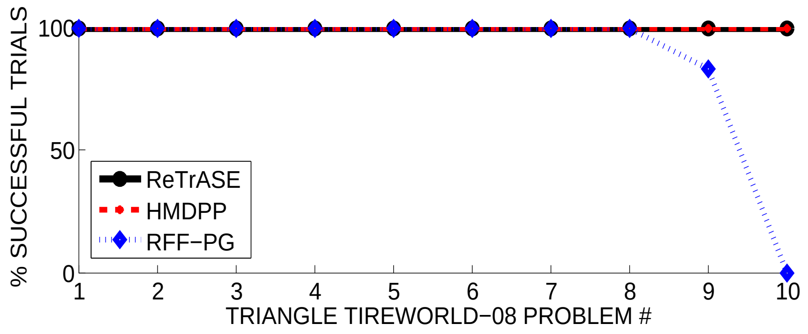


Figure 3.9: RETRASE achieves perfect success rate on Triangle Tireworld-08.

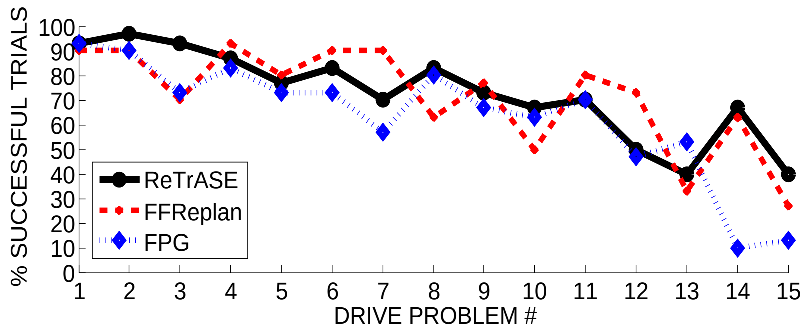


Figure 3.10: RETRASE is at par with the competitors on Drive.

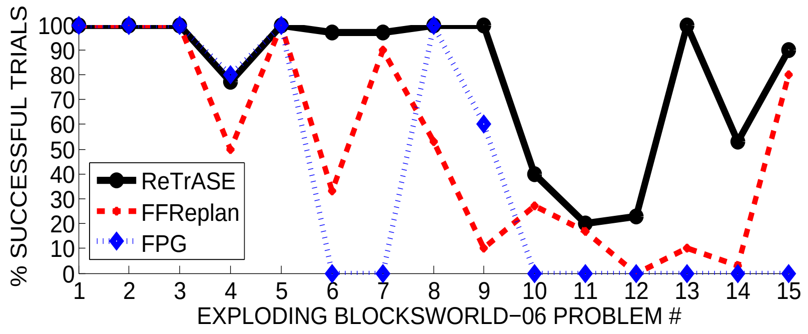


Figure 3.11: RETRASE dominates on Exploding Blocksworld-06.

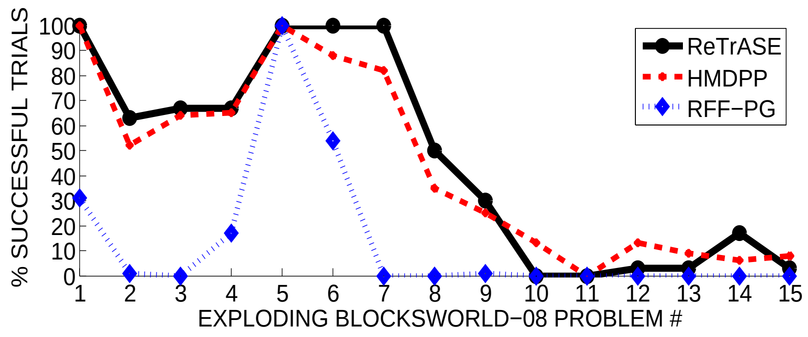


Figure 3.12: RETRASE outmatches all competitors on Exploding Blocksworld-08, although by a narrow margin.

it. Unfortunately, the optimal planner we used, LRTDP+OPT, scales enough to solve only relatively small problems (at most a few million states). On such problems we found RETRASE to produce trajectories of expected cost within 5% of the optimal.

Comparison to FFHop

FFReplan has been a very powerful planner and a winner of at least one IPPC. However, recent benchmarks defeat it by exploiting its near-complete disregard for probabilities when computing a policy. Researchers have proposed a powerful improvement to FFReplan, FFHop [100], and demonstrated its capabilities on problems from IPPC-2004. Due to the current lack of support for some PPDDL language features we were not able to run RETRASE on most IPPC-2004 domains. Table 3.2 compares the success rates of the two planners on the IPPC-2004 problems we did test. Even though RETRASE performs better on these problems, the small size of the experimental base makes the comparison of RETRASE and FFHop inconclusive.

Problem name	FFHop	RETRASE
exploding-block	93.33%	100%
g-tire-problem	60%	70%

Table 3.2: Success rates on some IPPC-2004 problems.

To conclude, while we did not test on all IPPC domains, our current experimental evaluation clearly demonstrate RETRASE's scalability improvements over the VI-family planners and its at-par or better performance on many competition problems compared to state-of-the-art approximate systems.

3.5.3 Summary

RETRASE is an MDP solver based on a combination of state abstraction and dimensionality reduction. It automatically extracts basis functions, which provide a compact representation of the given MDP while retaining its causal structure. Simultaneously with discovering basis functions, it learns weights for the already discovered ones using modified Bellman backups. These weights let ReTrASE evaluate states without memoizing state values explicitly. Such an approach allows ReTrASE to solve larger problems than the best performers of several recent IPPCs.

3.6 SixthSense

Although basis functions efficiently generalize information about states from which reaching the goal is possible, they have nothing to say about dead ends. As a result, algorithms that use only basis functions for information transfer cannot avoid either caching dead ends or rediscovering them every time they run into them. In fact, the issue of quickly and reliably recognizing dead ends plagues virtually all modern MDP solvers. For instance, in IPPC-2008 [17], the domains with a complex dead-end structure, e.g., Exploding Blocksworld, have proven to be some of the most challenging. Surprisingly, however, there has been little research on methods for effective discovery and avoidance of dead ends in MDPs. Of the two types of dead ends, implicit ones confound planners the most, since they do have executable actions. However, explicit dead ends can be a resource drain as well, since verifying that none of the available actions are applicable in a state can be costly if the number of actions and such states is large.

Broadly speaking, existing planners use one of two approaches for identifying dead ends. Recall that in this chapter we are working with a variant of factored SSP_{s_0} MDPs with dead end states, entering which incurs a high user-defined penalty. When faced with a yet-unvisited state, many planners (e.g., LRTDP) apply a heuristic value function, which hopefully recognizes dead ends and immediately assigns this penalty to them. This method is fast to invoke but often fails to catch many implicit dead ends due to the problem relaxation inevitably used by the heuristics. Failure to detect them causes the planner to waste much time in exploring the states reachable from implicit dead ends, these states being dead ends themselves. Other MDP solvers use state value estimation approaches that recognize dead ends reliably but are very expensive; for example, RFF, HMDPP, and ReTrASE employ full deterministic planners. When a problem contains many dead ends, these MDP solvers may spend a lot of their time launching classical planners from dead ends. Indeed, most probabilistic planners would run faster if recognizing dead ends was not so computationally expensive.

In this section, we complete our abstraction framework by presenting a novel mechanism, SixthSense [57, 58], to do exactly this — quickly and reliably identify dead-end states in MDPs. Underlying SixthSense is a key insight: large sets of dead-end states can usually be characterized by a compact logical conjunction, a nogood, which "explains" why no solution exists. For example, a Mars rover that flipped upside down will be unable to achieve its goal, regardless of its location, the orientation of its wheels, etc. Knowing this explanation lets a planner quickly recognize millions of states as dead ends. Crucially, dead ends in most MDPs can be described with a small number of nogoods.

SixthSense learns nogoods by generating candidate conjunctions with a bottom-up greedy search (resembling that used in rule induction [22]) and testing them to avoid false positives with a planning graph-based procedure. A vital input to this learning algorithm are basis functions, derived as shown in the previous sections. SixthSense is provably sound — every nogood output represents a set of true dead-end states. We empirically demonstrate that SixthSense speeds up two different types of MDP solvers on several IPPC domains with implicit dead ends and show the performance improvements SixthSense gives to GOTH and ReTrASE. Overall, SixthSense tends to identify most of the dead ends that the solvers encounter, reducing memory consumption by as much as 90%. Because SixthSense runs quickly, on large problems with dead ends it also gives a 30-50% speedup. With these savings, it enables planners to solve problems they could not previously handle.

3.6.1 SixthSense *Description*

To discover nogoods, we devise a machine learning generate-and-test algorithm that is an integral part of SixthSense. The "generate" step proposes a *candidate* conjunction, using some of the dead ends the planner has found so far as training data. For the testing stage, we develop a novel planning graph-based algorithm that tries to prove that the candidate is indeed a nogood. Nogood discovery happens in several attempts called *generalization rounds*. First we outline the generate-and-test procedure for a single round in more detail and then describe the scheduler that decides when a generalization round is to be invoked.

Algorithm 3.3 shows SixthSense's pseudocode. We remind the reader the meaning of two pieces of notation established in Section 3.3 that this pseudocode uses: for a set of literals c, $\bigwedge c$

stands for the conjunction of all literals in c; for a conjunction of literals c, $lit(c)$ stands for the set of all literals in c.

Generation of Candidate Nogoods

There are many ways to generate a candidate but if, as we conjecture, the number of explanations/nogoods in a given problem is indeed very small, naively constructed hypotheses, e.g., conjunctions of literals picked uniformly at random, are very unlikely to be nogoods and pass the test stage. Instead, our procedure makes an "educated guess" by employing basis functions according to one crucial observation. Recall that, by definition, basis functions are preconditions for goal trajectories. Therefore, no state represented by them can be a dead end. On the other hand, any state represented by a nogood, by the nogoods' definition, *must* be a dead end. These facts combine into the following observation: *a state may be represented by a basis function or by a nogood but not both.*

Of more practical importance to us is its corollary that any conjunction that has no conflicting pairs of literals (a literal and its negation) and contains the negation of at least one literal in every basis function (i.e., *defeats* every basis function) is a nogood. This fact provides a guiding principle — form a candidate by going through each basis function in the problem and, if the candidate does not defeat it, picking the negation of one of the basis function's literals. By the end of the run, the candidate provably defeats all basis functions and hence can only hold either in states with no trajectories to the goal, i.e., dead ends, or in no states at all (if it contains conflicting literals, e.g., A and $\neg A$). The idea has a big drawback though: finding *all* basis functions in an MDP is prohibitively expensive. Fortunately, it turns out that making sure a candidate defeats only a few randomly selected basis functions (100-200 for the largest problems we encountered) is enough in practice for the candidate to be a nogood with reasonably high probability (although not for certain, motivating the need for verification). Therefore, to use the learning algorithm, a caller needs to provide it with a "training" set of basis functions, denoted as BF in Algorithm 3.3. Candidate generation in lines 10-17 of SIXTHSENSE's pseudocode proceeds as just described: the candidate's set of literals c is formed by picking a defeating literal for each known basis function.

So far, we have not specified how exactly the defeating literals should be chosen. Here as well

Algorithm 3.3: SIXTHSENSE

```
Input: factored goal-oriented MDP consisting of domain $D = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$ and
problem $P = \langle \mathcal{G}, s_0 \rangle$, set of all domain literals $\mathcal{L}$, training set of non-generalized dead-end states $DE$,
training set of basis functions $BF$
Output: a nogood or $null$

$D_d \leftarrow \underline{Det}(D)$ // $\underline{Det}$ is a determinization routine, omitted from the pseudocode

function LearnNogood(set of d.e.s $DE$, set of b.f.s $BF$, set of all literals $\mathcal{L}$, goal conjunction $\mathcal{G}$)
begin
    // construct a candidate set of literals
    $c \leftarrow \{\}$
    foreach b.f. $f \in BF$ do
        if $c$ does not defeat $f$ then
            $L \leftarrow$ SampleDefeatingLiteral$(DE, f, c)$
            $c \leftarrow c \cup \{L\}$
        end
    end
    // check candidate with planning graph, and perform pruning if it is a nogood
    if CheckWithPlanningGraph$(c, \mathcal{L}, \mathcal{G})$ then
        foreach literal $L \in c$ do if CheckWithPlanningGraph$(c \setminus \{L\}, \mathcal{L}, \mathcal{G})$ then $c \leftarrow c \setminus \{L\}$
    else return $null$
    // if we got here then the candidate set of literals forms a valid nogood
    // upon receiving the nogood below, the caller should clear the dead-end set $DE$
    return $\bigwedge c$
end

function CheckWithPlanningGraph(literal conjunction $c'$, set of all literals $\mathcal{L}$, goal conjunction $\mathcal{G}$)
begin
    $\overline{c}' \leftarrow$ set of negations of all literals in $c'$
    foreach literal $G$ in $(lit(\mathcal{G}) \setminus c')$ do
        $c'' \leftarrow c' \cup \{\neg G\} \cup (\mathcal{L} \setminus (\overline{c}' \cup \{G\}))$
        if $\underline{PlanningGraph}(c'', D_d, \mathcal{G}) == success$ then return $false$
    end
    return $true$
end

function SampleDefeatingLiteral(set of dead ends $DE$, basis function $f$, literal conjunction $c'$)
begin
    foreach literal $L \in lit(f)$ s.t. $\neg L \notin c'$ do $cntr_{\neg L} \leftarrow 0$
    foreach dead-end state $d \in DE$ do
        if $\bigwedge c'$ holds in $d$ then
            foreach literal $L \in lit(f)$ s.t. $\neg L \notin c'$ and $\neg L$ holds in $d$ do $cntr_{\neg L} \leftarrow cntr_{\neg L} + 1$
        end
    end
    return a literal $L'$ sampled according to $P(L') \sim cntr_{L'}$
end
```

we can do better than naive uniform sampling. Intuitively, the frequency of a literal's occurrence in the dead ends that the MDP solver has encountered by any given point in its running time correlates with the likelihood of this literal's presence in nogoods. Therefore, as SIXTHSENSE is iterating over the basis functions in lines 12-17, for every basis function f its **SampleDefeatingLiteral(.)** subroutine samples a literal defeating f from the distribution induced by literals' frequencies of occurrence in the dead ends represented by the constructed portion of the nogood candidate (lines 37-46).

Nogood Verification

As already established, if in the above candidate generation procedure we used the set of *all* basis functions that exist for a given MDP, verifying the resulting candidate would not be necessary (other than making sure that the candidate does not contain pairs of conflicting literals). However, in general we do not have all possible basis functions at our disposal. Consequently, we need to make sure that the candidate created by SIXTHSENSE from the available basis functions is indeed a nogood. Let us denote the problem of establishing whether a given conjunction is a nogood as *NOGOOD-DECISION*.

Theorem 3.2. *NOGOOD-DECISION is $PSPACE$-complete.* ♢

Proof. See the Appendix. The proof is via a reduction from the deterministic plan existence problem in factored goal-oriented MDPs. ■

In the light of Theorem 3.2, we may realistically expect an efficient algorithm for *NOGOOD-DECISION* to be either sound or complete, but not both. A sound algorithm would never conclude that a candidate is a nogood when it is not. A complete one would pronounce a candidate to be a nogood whenever the candidate is in fact a nogood. A key component of our work is a sound algorithm for identifying nogoods. It is based on the observation that all the per-state checks in the naive scheme in the above proof can be replaced by only a few, whose running time is polynomial in the problem size. Although sound, this operation is incomplete, i.e., may reject some candidates that are actually nogoods. Nonetheless, it is effective at identifying nogoods in practice.

To verify a nogood candidate consisting of a set c of literals efficiently, we group all *non-goal* states represented by $\bigwedge c$ into several *superstates of* c. We define a superstate of a literal set c to be a set consisting of:

- All of c's literals,
- The negation of one of the goal literals that are not present in c, i.e., the negation of a literal in $lit(\mathcal{G}) \setminus c$;
- All literals over all other variables in the domain, i.e., the variables not participating in any of the literals added above.

As an example, suppose the complete set of literals in our problem is $\{A, \neg A, B, \neg B, C, \neg C, D, \neg D, E, \neg E\}$, the goal is $A \wedge \neg B \wedge E$, and the candidate nogood literal set is $\{A, C\}$. Then the superstates our algorithm constructs for this candidate are $\{A, \mathbf{B}, C, D, \neg D, E, \neg E\}$ and $\{A, B, \neg B, C, D, \neg D, \neg \mathbf{E}\}$ (the negation of a goal literal in each superstate is highlighted **in bold**).

The intuition behind this definition of superstates of c is as follows. Every non-goal state s represented by $\bigwedge c$ is "contained" in one of superstates of c in the sense that there is a superstate of c containing all of s's literals. Moreover, if a superstate has no trajectory to the goal, no such trajectory exists for any state contained in the superstate, implying that these states are all dead ends. Combining these two observations, if *no* goal trajectory exists from *any* superstate of c then *all* the states represented by the candidate are dead ends. By definition, such a candidate is a nogood.

Accordingly, to find out whether the candidate is a nogood, our procedure runs the planning graph algorithm on each of the candidate's superstates (lines 27-35) in the all-outcomes determinization of the MDP. Each planning graph instance returns $success$ if and only if it can reach all the goal literals and resolve all mutexes between them. The initial set of mutexes fed to the planning graph for a given superstate are just the mutexes between each literal and its negation, if both are present in that superstate.

Theorem 3.3. *A candidate set of literals forms a nogood if the planning graph expansion on each of its superstates in the all-outcomes determinization either a) fails to achieve at least one goal literal or b) fails to resolve mutexes between any two of the goal literals.* ♢

Proof. Since the planning graph is sound when it reports that the goal is *not* reachable from a given set of literals and mutexes between them, its failure on *all* superstates in the all-outcomes determinization of the given MDP indicates the candidate is a true nogood. ■

Our verification procedure is incomplete for two reasons. First, since each superstate has more literals than any single state it contains, it may have a goal trajectory that is impossible to execute from any state. Second, the planning graph algorithm is incomplete by itself; it may declare plan existence when no plan actually exists.

At the cost of incompleteness, our algorithm is only polynomial in the problem size. To see this, recall that each planning graph expansion from a superstate is polynomial in the number of domain literals, and note that the number of superstates is linear in the number of goal literals.

If the verification test is passed, we try to prune away unnecessary literals (lines 20-20) that may have been included into the candidate during sampling. This analog of Occam's razor strives to reduce the candidate to a *minimal* nogood and often gives us a much more general conjunction than the original one at little extra verification cost.

If the learning algorithm returns a nogood, the MDP solver should discard the set of dead ends DE that served as training data and, when learning needs to be invoked again, pass in a new dead-end set. The motivation for this step will become clear once we discuss scheduling of SIXTHSENSE invocations.

Scheduling

Since we do not know the number of nogoods in the problem a priori, we need to perform several generalization rounds (learning procedure invocations). Optimally deciding when to do that is hard, if not impossible, but we have designed an adaptive scheduling mechanism that works well in practice. It tries to estimate the size of the training set likely sufficient for learning an extra nogood, and invokes learning when that much data has been accumulated. When generalization rounds start failing, the scheduler calls them exponentially less frequently. Thus, very little computation time is wasted after all nogoods that could reasonably be discovered have been discovered. (There are certain kinds of nogoods whose discovery by SIXTHSENSE, although possible, is highly improbable. We elaborate on this point in Section 3.8.)

Our algorithm is inspired by the following tradeoff. The sooner a successful generalization round happens, the earlier SIXTHSENSE can start using the resulting nogood, saving time and memory. On the other hand, trying to learn a nogood too soon, with hardly any training data available, is unlikely to succeed. The exact balance is difficult to pinpoint even approximately, but our empirical trials indicate three helpful trends: (1) The learning algorithm is capable of operating successfully with surprisingly little training data, as few as 10 dead ends. The number of basis functions does not play a big role provided there is more than about 100 of them. (2) If a generalization round fails with statistics collected from a given number of dead ends, their number usually needs to be increased drastically. However, because learning is probabilistic, such a failure could also be accidental, so it is justifiable to return to the "bad" training data size occasionally. (3) A typical successful generalization round saves the planner enough time and memory to compensate for many failed ones. These three regularities suggest the following strategy:

- Initially, the scheduler waits for a small batch of basis functions, BF in Algorithm 3.3, and a small number of dead ends, DE, to be accumulated before invoking the first generalization round. For the reasons above, our implementation used the initial settings of $|BF| = 100$ and $|DE| = 10$ for all problems.

- After the first round and including it, whenever a round succeeds the scheduler waits for a number of dead ends *not represented by the known nogoods* equal to half of the previous batch size to arrive before invoking the next round. Decreasing the batch size in this way is usually worth the risk according to observations (2) and (3) and because the round before succeeded. If a round fails, the scheduler waits for the accumulation of twice the previous number of unrecognized dead ends before trying generalization again.

Perhaps unexpectedly, in many cases we have seen very large training dead-end sets decrease the probability of learning a nogood. This phenomenon can be explained by training sets of large sizes sometimes containing subcollections of dead ends "caused" by different nogoods. Consequently, the literal occurrence statistics induced by such a mix make it hard to generate reasonable candidates. This finding has led us to restrict the training batch size (DE in Algorithm 3.3) to $10,000$. If, due to exponential backoff, the scheduler is forced to wait for the arrival of $n > 10,000$ dead ends, it skips the first $(n - 10,000)$ and retains only the latest $10,000$ for training. For the same locality considerations, the dead-end training set should be emptied at the end of each successful generalization round.

Theoretical Properties

Before presenting the experimental results, we analyze SIXTHSENSE's properties. The most important one is that the procedure of identifying dead ends as states in which at least one nogood holds is sound. It follows directly from the nogood's definition.

Importantly, SIXTHSENSE puts no bounds on the nogood length, being theoretically capable of discovering any nogood. One may ask: are there any nontrivial bounds on the amount of training data for SIXTHSENSE to generate a nogood of a given length with at least a given probability? As the following argument indicates, even if such bounds exist they are likely to be of no use in practice. For SIXTHSENSE to generate any given nogood, the training data must contain many dead ends caused by this nogood. However, depending on the structure of the problem, most such dead ends may be unreachable from the initial state. If the planning algorithm that uses SIXTHSENSE (e.g., LRTDP) never explores those parts of the state space, no amount of *practically collectable* training data will help SIXTHSENSE discover some of the nogoods with high probability.

At the same time, we can prove another important property of SIXTHSENSE:

Theorem 3.4. *Suppose the set of dead ends DE passed to* SIXTHSENSE *as training data contains only those dead ends that are not represented by any nogood from a particular set NG. If* SIXTHSENSE *succeeds in learning a nogood using the set DE, this nogood will not be in NG.* $\diamondsuit$

Proof. Stated differently, this theorem says that if SIXTHSENSE's training data excludes dead ends represented by previously discovered nogoods, those nogoods will not be rediscovered again. According to Algorithm 3.3, each nogood candidate is built up iteratively by sampling literals from a distribution induced by training dead ends that are represented by the constructed portion of the candidate. Also, by the theorem's assumption, no training dead end is represented by any previously learned nogood. Therefore, the probability of sampling a known nogood (lines 12-17) is 0. ■

Regarding SIXTHSENSE's speed, the number of frequently encountered nogoods in any given problem is rather small, which makes identifying dead ends by iterating over the nogoods very quick. Moreover, a generalization round is polynomial in the amount of training data. We point out, however, that obtaining the training data theoretically takes exponential time. Nevertheless, since training dead ends are identified as a part of the usual planning procedure in most MDP solvers, the only extra work to be done for SIXTHSENSE is obtaining a few basis functions. Their required number is so small that in nearly every probabilistic problem they can be quickly obtained by invoking a speedy deterministic planner from several states. This explains why in practice SIXTHSENSE is very fast.

Last but not least, we believe that SIXTHSENSE can be incorporated into nearly any existing solver for factored goal-oriented MDPs, since, as explained above, the training data required by SIXTHSENSE is either available in these solvers and can be cheaply extracted, or can be obtained independently of the solver's operation by invoking a deterministic planner.

3.6.2 Experimental Results

Our goal in the experiments was to explore the benefits SIXTHSENSE brings to different types of planners, as well as to gauge the effectiveness of nogoods and the amount of computational resources taken to generate them. We used three IPPC domains as benchmarks: Exploding Blocksworld-08 (EBW-08), Exploding Blocksworld-06 (EBW-06), and Drive-06. IPPC-2006 and -2008 contained several more domains with dead-end states, but, with respect to dead ends, their structure is similar to that of the domains we chose. The only exception is the Triangle Tireworld-08 domain, whose representational artifacts do not allow for generalizing dead ends with SIXTHSENSE. In all experiments, we restricted each participating MDP solver to use no more than 2 GB of memory.

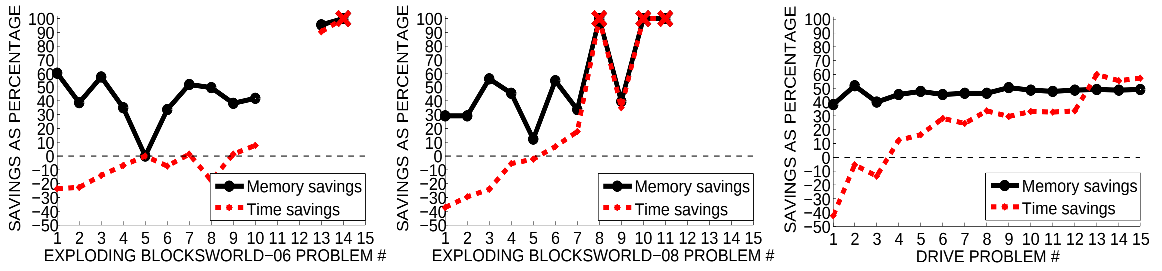


Figure 3.13: Time and memory savings due to nogoods for LRTDP+h_{FF} (representing the "Fast but Insensitive" type of planners) on 3 domains, as a percentage of resources needed to solve these problems without SIXTHSENSE (higher curves indicate bigger savings; points below zero require more resources with SIXTHSENSE). The reduction on large problems can reach over 90% and even enable more problems to be solved (their data points are marked with a $\times$).

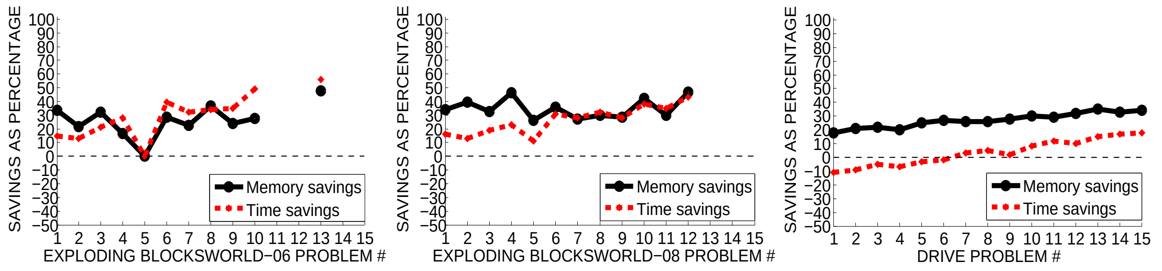


Figure 3.14: Resource savings from SIXTHSENSE for LRTDP+GOTH/NO 6S (representing the "Sensitive but Slow" type of planners).

Structure of Dead Ends in IPPC Domains

Among the IPPC benchmarks, we found domains with only two types of implicit dead ends. In the Drive domain, which exemplifies the first of them, the agent's goal is to stay alive and reach a destination by driving through a road network with traffic lights. The agent may die trying but, because of the domain formulation, this does not necessarily prevent the car from driving. Thus, all of the implicit dead ends in the domain are generalized by the singleton conjunction $(not\ alive)$. A few other IPPC domains, e.g., Schedule, resemble Drive in having one or several exclusively single-literal nogoods representing all the dead ends. Such nogoods are typically easy for SIXTHSENSE to derive.

EBW-06 and -08's dead ends are much more complex. In the EBW domain, the objective is to rearrange a number of blocks from one configuration to another, and each block might explode in the process. For each goal literal, EBW has two multiple-literal nogoods explaining when this literal cannot be achieved. For example, if block $b4$ needs to be on block $b8$ in the goal configuration, then any state in which $b4$ explodes before being picked up by the manipulator or in which $b8$ explodes is a dead end, represented either by nogood $(not\ (no - destroyed\ b4)) \wedge (not\ (holding\ b4)) \wedge (not\ (on\ b4\ b8))$ or by $(not\ (no - destroyed\ b8)) \wedge (not\ (on\ b4\ b8))$. We call such nogoods *statistically identifiable* and point out that EBW also has others too, described in Section 3.8. The variety and structural complexity of EBW nogoods makes them challenging to learn.

Planner Types

As we discussed at the beginning of this section, MDP solvers can be divided into two groups according to the way they handle dead ends. Some of them identify dead ends using fast but unreliable means like heuristics, which miss a lot of dead ends, causing the planner to waste time and memory exploring useless parts of the state space. We will call such planners "fast but insensitive" with respect to dead ends. Most others use more accurate but also more expensive dead-end identification mechanisms. We term these planners "sensitive but slow" in their treatment of dead ends. The monikers for both types apply only to the way these solvers handle dead ends and not to their overall performance. With this in mind, we demonstrate the effects SIXTHSENSE has on each type.

Benefits to Fast but Insensitive Planners

This group of planners is represented in our experiments by LRTDP with the h_{FF} heuristic. We will call this combination LRTDP+h_{FF}, and LRTDP+h_{FF} equipped with SIXTHSENSE — LRTDP+h_{FF} +6S for short. Implementationwise, SIXTHSENSE is incorporated into h_{FF}. When evaluating a newly encountered state, h_{FF} first consults the available nogoods produced by SIXTHSENSE. Only when the state fails to match any nogood does h_{FF} resort to its traditional means of estimating the state value. Without SIXTHSENSE, h_{FF} misses many dead ends, since it ignores actions' delete effects.

Figure 3.13 shows the time and memory savings due to SIXTHSENSE across three domains as the percentage of the resources LRTDP+h_{FF} took to solve the corresponding problems (the higher the curves are, the bigger the savings). No data points for some problems indicate that neither LRTDP+h_{FF} nor LRTDP+h_{FF}+6S could solve them with only 2GB of RAM. There are a few large problems that could only be solved by LRTDP+h_{FF}+6S. Their data points are marked with a $\times$ and savings for them are set at 100% (e.g., on problem 14 of EBW-06) as a matter of visualization, because we do not know how much resources LRTDP+h_{FF} would need to solve them. Additionally, we point out that as a general trend, problems grow in complexity within each domain with the increasing ordinal. However, the increase in difficulty is not guaranteed for any two adjacent problems, especially in domains with a rich structure, causing the jaggedness of graphs for EBW-06 and -08.

As the graphs demonstrate, the memory savings on average grow very gradually but can reach staggering 90% on the largest problems. In fact, on the problems marked with a $\times$, they enable LRTDP+h_{FF}+6S to do what LRTDP+h_{FF} cannot. This qualitative advantage of LRTDP+h_{FF}+6S is due to the fact that, since nogoods help it recognize many states as dead ends, it does not explore (and hence memoize) these states' successors, which are also dead ends. In other words, LRTDP+h_{FF}+6S does not visit nearly as many fruitless states as LRTDP+h_{FF}. Notably, the time savings are lagging for the smallest and some medium-sized problems (approximately 1-7). Each of them takes only a few seconds to solve, so the overhead of SIXTHSENSE is noticeable. However, on large problems, SIXTHSENSE fully comes into its element and saves 30% or more of the planning time.

Benefits to Sensitive but Slow Planners

Planners of this type include top IPPC performers RFF and HMDPP, as well as RETRASE and others. Most of them use a deterministic planner, e.g., FF, on a domain determinization to find plans from the given state to the goal and use such plans in various ways to construct a policy. Whenever the deterministic planner can prove nonexistence of a path to the goal or simply fails to find one within a certain time, these MDP solvers consider the state from which the planner was launched to be a dead end. Due to the properties of classical planners, this method of dead-end identification is reliable but rather expensive. To model it, we employed LRTDP with the GOTH heuristic. GOTH evaluates states with classical planners, so including or excluding SIXTHSENSE from GOTH allows for simulating the effects SIXTHSENSE has on the above algorithms. As SIXTHSENSE is part of the standard GOTH implementation, GOTH without it is denoted as GOTH/NO 6S. Figure 3.14

illustrates LRTDP+GOTH's behavior. Qualitatively, the results look similar to LRTDP+h_{FF}+6S's but there is a subtle critical difference — the time savings in the case of LRTDP+h_{FF}+6S grow faster. This is a manifestation of the fundamental distinction of SIXTHSENSE in the two settings. For the "Sensitive but Slow", SIXTHSENSE helps recognize implicit dead ends faster (and obviates memoizing them). For the "Fast but Insensitive", it in addition obviates exploring many of the implicit dead ends' successors, causing a faster savings growth with problem size.

Benefits to ReTrASE.

RETRASE is perhaps the most natural MDP solver to be augmented with SIXTHSENSE. It already uses basis functions to store information about non-dead-end states, and utilizing nogoods would allow it to capitalize on the abstraction framework even more, providing additional insights into the benefits for other planners that might employ the abstraction framework to serve all of their state space representation needs.

To measure the effect of SIXTHSENSE on RETRASE and get a different perspective on the role of SIXTHSENSE than in the previous experiments, we ran RETRASE and RETRASE+6S for at most 12 hours on each of the 45 problems of the EBW-06, -08, and Drive sets, and noted the policy quality, as reflected by the success rate, at fixed time intervals. For smaller problems, we measured policy quality every few seconds, whereas for larger ones — every 5-10 minutes. Qualitatively, the trends on all the problems were similar, so here we study them on the example of problem 12 from the EBW-06 set, one of the hardest problems attempted. For this problem, after 12 hours of CPU running time, RETRASE+6S extracted 62267 basis functions and learned their weights; it also discovered 79623 dead ends. Out of these dead ends, 18392 were identified by RETRASE+6S running a deterministic planner starting at them having this planner fail to find a path to the goal. The remainder, i.e., 77%, were discovered with 15 nogoods that SIXTHSENSE derived. Since every deterministic planner call from a non-dead-end state typically yields several basis functions, SIXTHSENSE saved RETRASE at least $(79623 - 18392)/(62267 + 79623) \approx 43\%$ of classical planner invocations, with accompanying time savings. On the other hand, RETRASE's running time is not occupied solely by basis function extraction — a significant fraction of it consists of basis function weight learning and state space exploration. Besides, SIXTHSENSE, although fast, was not instan-

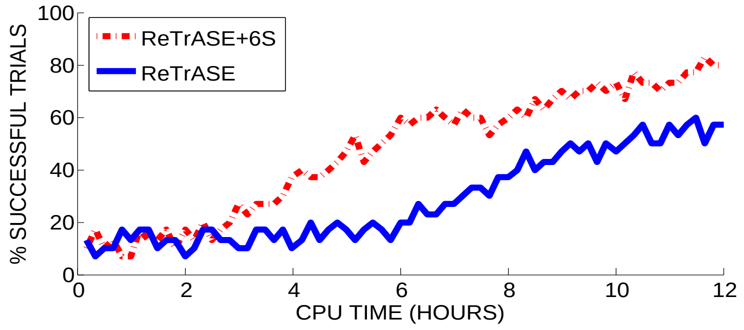


Figure 3.15: SIXTHSENSE speeds up RETRASE by as much as 60% on problems with dead ends. The plot shows this trend on the example of problem 12 of EBW-06.

taneous. Therefore, based on this model we expected the overall speedup caused by SIXTHSENSE to be less than 40% and likely also less than 30%.

With this in mind, please refer to Figure 3.15 showing the plots of policy quality yielded by RETRASE and RETRASE+6S versus time. As expected intuitively, the use of SIXTHSENSE does not change RETRASE's pattern of convergence, and the shape of the two plots are roughly similar. (If allowed to run for long enough both planners should converge to policies of the same quality, although the plots do not show this.) However, surprisingly, the time it takes RETRASE+6S to arrive at a policy of the quality RETRASE gets after 12 hours of execution turns out to be about 5.5 hours. Thus, the speedup SIXTHSENSE has yielded is considerably larger than predicted by our model, roughly 60% versus the expected 30% or less.

Additional code instrumentation revealed an explanation for this discrepancy. The model just sketched implicitly assumes that the time cost of a successful deterministic planner call (one that yields basis functions) and one that proves the state to be a dead end to be the same. This appears to be far from reality; the latter, on average, was over 4 times more expensive. With this factor taken into account, the model would forecast a 64% time savings on classical planner calls due to employment of SIXTHSENSE, which agrees with the actual data much better.

Regarding memory savings, SIXTHSENSE helps RETRASE as well, but the picture here is much clearer. Indeed, since RETRASE memoizes only basis functions (with weights), dead ends, and nogoods, a 43% reduction in the total number of these as predicted by our model straightforwardly translates to the corresponding memory reduction our experiments showed. We point out, however, that even without SIXTHSENSE, RETRASE's memory requirements are very low compared to other MDP solvers, and reducing them even further is a less significant performance gain than the boost in speed.

Last but not least, we found that SIXTHSENSE almost never takes more than 10% of LRTDP+h_{FF} +6S's or LRTDP+GOTH's running time. For LRTDP+h_{FF}+6S, this fraction includes the time spent on deterministic planner invocations to obtain the basis functions, whereas in LRTDP+GOTH, the classical plans are available to SIXTHSENSE for free. In fact, as the problem size grows, SIXTHSENSE eventually gets to occupy less than 0.5% of the total planning time. As an illustration of SIXTHSENSE's operation, we found out that it always finds the single nogood in the Drive domain after using just 10 dead ends for training, and manages to acquire most of the statistically identifiable nogoods in EBW. In the available EBW problems, their number is always less than several dozens, which, considering the space savings they bring, attests to nogoods' high efficiency.

3.6.3 Summary

SIXTHSENSE is a machine learning algorithm for discovering the counterpart of basis functions, nogoods. The presence of a nogood in a state guarantees the state to be a dead end. Thus, nogoods help a planner quickly identify dead ends without memoizing them, helping save memory and time. SIXTHSENSE serves as a submodule of a planner that periodically attempts to "guess" nogoods using dead ends the planner visited and basis functions the planner discovered as training data. It checks each guess using a sound planning graph-based verification procedure. Depending on the type of MDP solver, SIXTHSENSE vastly speeds it up, reduces its memory footprint, or both, on MDPs with dead-end states.

3.7 Related Work

In spirit, the concept of extracting useful state information in the form of basis functions is related to explanation-based learning (EBL) [47, 51]. In EBL, the planner would try to derive control rules for action selection by analyzing its own execution traces. In practice, EBL systems suffer from accumulating too much of such information, whereas the approaches we have presented do not. The idea of using determinization followed by regression to obtain basis functions has parallels to some research on relational MDPs, which uses first-order regression on optimal plans in small problem instances to construct a policy for large problems in a given domain [37, 87]. However, our function aggregation and weight learning methods are completely different from theirs. In most of the literature on using basis functions to solve MDPs, the basis functions are constructed manually by domain experts [36, 38, 39]. One of the main strengths of our approaches lies in generating them automatically, without human intervention.

RETRASE, in essence, exploits basis functions to perform dimensionality reduction, but basis functions are not the only known alternative to serve this purpose. Other flavors of dimensionality reduction include algebraic and binary decision diagrams (ADDs/BDDs), and principle component analysis (PCA)-based methods. SPUDD, Symbolic LAO*, and Symbolic RTDP are optimal algorithms that exploit ADDs and BDDs for a compact representation and efficient backups in an MDP [29, 43]. While they are a significant improvement in efficiency over their non-symbolic counterparts, these optimal algorithms still do not scale to large problems. APRICODD, an approximation scheme developed over SPUDD [92], showed promise, but it is not clear whether it is competitive with today's top methods since it has not been applied to the competition domains.

Some researchers have applied non-linear techniques like exponential PCA and NCA for dimensionality reduction [48, 84]. These methods assume the original state space to be continuous and hence are not applicable to typical planning benchmarks.

In fact, most basis function-based dimensionality reduction techniques are not applied in nominal domains. A notable exception is FPG [19], which performs policy search and represents the policy compactly with a neural network. Our experiments demonstrate that RETRASE outperforms FPG consistently on several domains.

The use of determinization for solving MDPs in general was inspired by advances in classical planning, most notably the FF solver [44]. The practicality of the new technique was demonstrated by FFReplan [99] that used the FF planner on an MDP determinization for direct selection of an action to execute in a given state. More recent planners to employ determinization that are, in contrast to FF-Replan, successful at dealing with probabilistically interesting problems include RFF-RG/BG [95]. At the same time, the latter kind of algorithms typically invokes a deterministic planner many more times than our techniques do. This forces them to avoid the all-outcomes determinization, as these invocations would be too costly otherwise. Other related planners include Temptastic [102], precautionary planning [32], and FFHop [100].

The employment of determinization for heuristic function computation was made famous by the FF heuristic, h_{FF}[44], originally part of a classical planner by the same name. LRTDP [13] and HMDPP [50] adopted this heuristic with no modifications as well. In particular, HMDPP runs h_{FF} on a "self-loop determinization" of an MDP, thereby forcing h_{FF}'s estimates to take into account some of the problem's probabilistic information.

To our knowledge, there have been no previous attempts to handle identification of dead ends in MDPs. The "Sensitive but Slow" and "Fast but Insensitive" mechanisms were not actually designed specifically for the purpose of identifying dead ends and are unsatisfactory in many ways. One possible reason for this omission may be that most MDPs studied by the Artificial Intelligence and Operations Research communities until recently had no dead ends. However, MDPs with dead ends have been receiving attention in the past few years as researchers realized their probabilistic interestingness [65]. Besides the analogy to EBL, SIXTHSENSE can also be viewed as a machine learning algorithm for rule induction, similar in purpose, for example, to CN2 [22]. While this analogy is valid, SIXTHSENSE operates under different requirements than most such algorithms, because we demand that SIXTHSENSE-derived rules (nogoods) have zero false-positive rate. Last but not least, our term "nogood" shares its name with and closely mirrors the concept from the areas of truth maintenance systems (TMSs) [25] and constraint satisfaction problems (CSPs) [27]. However, our methodology for finding nogoods has little in common with algorithms used in that literature.

3.8 Future Research Directions

The experiments indicate that the proposed abstraction framework is capable of advancing the state of the art in planning under uncertainty. Nonetheless, there are several promising directions for future improvement.

Making Structure Extraction Faster

Even though the employment of basis functions in GOTH renders GOTH much faster than otherwise, the relatively few classical planner invocations that have to be made are still expensive, and GOTH's advantage in informativeness is not always sufficient to secure an overall advantage in speed for the MDP solver that uses it. Incidentally, we noticed that on some of the domains ReTrASE spends a lot of time discovering basis functions that end up having high weights (i.e., are not very "important"). We see two ways of handling the framework's occasional lack of speed in discovering useful problem structure.

The first approach is motivated by noticing that the speed of basis function extraction depends critically on how fast the available deterministic planners are on the deterministic version of the domain at hand. Therefore, the speed issue can be alleviated by adding more modern classical planners to the portfolio and launching them in parallel in the hope that at least one will be able to cope quickly with the given domain. Of course, this method may backfire when the number of employed classical planners exceeds the number of cores on the machine where the MDP solver is running, since the planners will start contending for resources. Nonetheless, up to that limit, increasing the portfolio size should help. In addition, using a reasonably-sized portfolio of planners may help reduce the variance in the time it takes to arrive at a deterministic plan.

The above idea is an extensional approach to accelerate the domain structure extraction, one that increases the performance of the algorithm by making more computational resources available to it. There is also an intensional one, that improves the algorithm itself. The ultimate reason for the frequent discovery of "useless" basis functions via deterministic planning is the fact that a basis function's importance is largely determined by the probabilistic properties of the corresponding trajectory, something the all-outcomes determinization completely discards. An alternative would be to give classical planners a domain determinization that retains at least some of its probabilistic structure. Although seemingly paradoxical, such determinizations exist, e.g., the one proposed by the authors of HMDPP [50]. Its use could improve the quality of obtained basis functions and thus reduce the deterministic planning time spent on discovering subpar ones. Different determinization strategies may also ease the task of the classical planners provided that the determinization avoids enumerating all outcomes of every action without significant losses in solution quality.

Lifting Representation to First-Order Logic

Another potentially fruitful research direction is increasing the power of abstraction by lifting the representation of basis functions and nogoods to first-order logic. Such representation's benefits are apparent, for example, in the EBW domain. In EBW, besides the statistically identifiable nogoods, there are others of the form "block b is not in its goal position and has an exploded block somewhere in the stack above it". Indeed, to move b one would first need to remove all blocks, including the exploded one, above it in the stack, but in EBW exploded blocks cannot be relocated. Expressed in first-order logic, the above statement would clearly capture many dead ends. In propositional logic, however, it would translate to a disjunction of many ground conjunctions, each of which is a nogood corresponding to one possible position of the exploded block in the stack above b. Each such ground nogood separately accounts for a small fraction of dead ends in the MDP and is almost undetectable statistically, preventing SIXTHSENSE from discovering it.

Handling Conditional Effects

So far, we have assumed that an action's precondition is a simple conjunction of literals. PPDDL's most recent versions allow for a more expressive way to describe an action's applicability via conditional effects. Figure 3.16 shows an action with this feature. In addition to the usual precondition, this action has a separate precondition controlling each of its possible effects. Depending on the state, any subset of the action's effects can be executed.

```
(:action be-evil
 :parameters ()
 :precondition (and (gremlin-alive))
 :effect (and
             (if (and (has Screwdriver) (has Wrench))
                 (and (plane-broken)))
             (if (and (has Hammer))
                 (and (plane-broken)
                      (probabilistic 0.9
                      (and (not (gremlin-alive)))))))
```

Figure 3.16: Action with conditional effects

The presented algorithms currently do not handle problems with this construct for two reasons. First, regression as defined in the Section 2.2 does not work for conditional effects. However, its definition can be easily extended to such cases. As a starting step, consider a goal trajectory $t(e)$ and suppose that outcome $out(a_i, i, e)$, part of $t(e)$, is the result of applying action a_i in state s_{i-1} of e. Denote the precondition of k-th conditional effect of a_i as $cond\_prec_k(a_i)$. When e was sampled, conjunction $out(a_i, i, e)$ was generated in the following way. For every k, it was checked whether $cond\_prec_k(a_i)$ holds in s_{i-1}. If it did, the dice were rolled to select the outcome of the corresponding conditional effect. Denote this outcome as $cond\_out_k(a_i, i, e)$. Furthermore, let $lit(cond\_out_k(a_i, i, e)) = \emptyset$ (i.e., let $cond\_out_k(a_i, i, e)$ be an empty conjunction) if $cond\_prec_k(a_i)$ does not hold in s_i.

By definition, $cond\_prec_k(a_i)$ can be empty in either of two cases:

- If $cond\_prec_k(a_i)$ does not hold in s_{i-1};
- If $cond\_prec_k(a_i)$ holds in s_{i-1} but while sampling $cond\_out_k(a_i, i, e)$ we happened to pick an outcome that does not modify s_{i-1}.

In light of this fact, define the *cumulative precondition* of $out(a_i, i, e)$ as

$$cu\_prec(out(a_i, i, e)) = prec(a_i) \wedge \left[\bigwedge_k \{cond\_prec_k(a_i) \text{ s.t. } lit(cond\_out_k(a_i, i, e)) \neq \emptyset\} \right]$$

and observe that

$$out(a_i, i, e) = \bigwedge_k \{cond\_out_k(a_i, i, e)\}.$$

Thus, $cu\_prec(out(a_i, i, e))$ is a conjunction of preconditions of those conditional effects of a_i that contributed at least one literal to $out(a_i, i, e)$. In other words, it is the *minimum necessary precondition* of $out(a_i, i, e)$. Therefore, to extend regression to actions with conditional effects we simply substitute $cu\_prec(out(a_i, i, e))$ for $prec(a_i)$ into the formulas for generating basis functions from Section 2.2.

Unfortunately, there is a second, practical difficulty with making GOTH, RETRASE, and SIXTHSENSE work in the presence of conditional effects. Recall that our primary way of obtaining goal trajectories for regression is via deterministic planning. Determinizing an ordinary probabilistic action yields the number of deterministic actions equal to the number of original action's outcomes. In the presence of conditional effects, this statement needs to be qualified. Each conditional effect can be thought of as describing an "action within an action" with its own probabilistic outcomes. These "inside actions" need not be mutually exclusive. Therefore, the number of outcomes of an action with conditional effects is generally *exponential* in the latter's number. As a consequence, determinizing such actions may lead to a blowup in problem representation size. Further research is needed to identify special cases in which the determinization of conditional effects can be done efficiently.

Beyond Goal-Oriented MDPs

So far, the probabilistic planning community has predominantly concentrated on goal-oriented MDPs. However, there are interesting problems of other types as well. As an example, consider the Sysadmin domain [17], in which the objective is to keep a network of computers running for as long as possible. Such reward-maximization problems have received little attention up until now (possibly with the exception of Sysadmin itself and a few other benchmark domains). Extending the automatic abstraction framework to reward-maximization problems is a potentially impactful research direction. However, it meets with a serious practical as well as theoretical difficulty. Recall that the natural deterministic analog of SSP MDPs are shortest-path problems. Researchers have studied them extensively and developed a wide range of very efficient tools for solving them, such as FF, LPG, LAMA, and others. As shown earlier, the techniques presented here critically rely on these tools for extracting the basis functions and estimating their weights. In contrast, the closest classical counterpart of probabilistic reward-maximization scenarios are *longest*-path problems. Known algorithms for various deterministic formulations of this setting are at best exponential *in the state space size*, explaining the lack of fast solvers for them. In their absence, the invention of alternative efficient ways of automatically extracting important causal information is an important step on the way to extending abstraction beyond goal-oriented MDPs.

Abstraction Framework and Existing Planners

Despite improvements being possible, our abstraction framework is useful even in its current state, as evidenced by both the experimental and theoretical results. Moreover, it has a property that makes its use very practical; the framework is complementary to the other powerful ideas incorporated in successful solvers of the recent years, e.g., HMDPP, RFF, FFHop, and others. Thus, abstraction can greatly benefit many of these solvers and also inspire new ones. As an example, note that FFReplan could be enhanced with abstraction in the following way. It could extract basis functions from deterministic plans it is producing while trying to reach the goal and store each of them along with their weight and the last action regressed before obtaining that particular basis function. Upon encountering a state represented by at least one of the known basis functions, "generalized FFReplan" would select the action corresponding to the basis function with the smallest weight. Besides an accompanying speed boost, which is a minor point in the case of FFReplan since it is very fast as is, FFReplan's robustness could be greatly improved, since this way its action selection would be informed by several trajectories from the state to the goal, as opposed to just one. Employed analogously, basis functions could speed up FFHop, an MDP solver that has great potential but is somewhat slow in its current form. In fact, we believe that virtually any algorithm for solving factored goal-oriented MDPs could have its convergence accelerated if it regresses the trajectories found during policy search and carries over information from well explored parts of the state space to the weakly probed ones with the help of basis functions and nogoods. We hope to verify this conjecture in the future. At the same time, solvers of discounted-reward MDPs are unlikely to gain much from the kind of abstraction proposed in this chapter, even though, mathematically, the described techniques will work even on this MDP class. In Chapter 2, we mentioned that discounted-reward MDPs can be viewed as SSP MDPs where each action has some probability of leading directly to the goal [6]. As a result, any sequence of actions in a discounted-reward MDP is a goal trajectory. This leads to an overabundance of discoverable basis functions, potentially making their number comparable to the number of states in the problem (in theory, the number of basis functions can reach $3^{|\mathcal{X}|}$, where $|\mathcal{X}|$ is the number of state variables).

A different approach for making abstraction benefit existing planners is to let RETRASE produce a value function estimate and to allow another planner, e.g. LRTDP, complete the solution of the problem starting from this estimate. This idea is reminiscent of hybridized planning [70] and is motivated by the fact that it is hard to know when RETRASE has converged on a given problem (and whether it ever will). Therefore, it makes sense to have an algorithm with convergence guarantees take over from RETRASE at a certain point. Empirical research is needed to determine when the switch from RETRASE to another solver should happen.

3.9 Summary

A central issue that limits practical applicability of automated planning under uncertainty is the scalability of available techniques. In this chapter, we have presented a powerful approach to tackle this fundamental problem — an abstraction framework that extracts problem structure and exploits it to spread information gained by exploring one region of the MDP's state space to others.

The components of the framework are the elements of problem structure called *basis functions* and *nogoods*. The basis functions are preconditions for those sequences of actions (*trajectories*) that take the agent from some state to the goal with positive probability. As such, each applies in many of the MDP's states, sharing associated reachability information across them. Crucially, basis functions are easy to come by via fast deterministic planning or even as a byproduct of the normal probabilistic planning process. While basis functions describe only MDP states from which reaching the goal is possible, their counterparts, nogoods, identify dead ends, from which the goal cannot be reached. Crucially, the number of basis functions and nogoods needed to characterize the problem space is typically vastly smaller than the problem's state space. Thus, the framework can be used in a variety of ways that increase the scalability of the state of the art methods for solving MDPs.

We have described three approaches illustrating the framework's operation, GOTH, RETRASE, and SIXTHSENSE. The experimental results show the promise of the outlined abstraction idea. Although we describe several ways to enhance our existing framework, even as is it can be utilized to qualitatively improve scalability of virtually any modern MDP solver and inspire the techniques of tomorrow.

Chapter 4
PLANNING EFFICIENTLY WITHOUT A GOAL

The increase in the scalability of algorithms for solving factored goal-oriented MDP over the past decade has been largely due to the extensive use of classical planners as subroutines in MDP approximation techniques. Indeed, nearly all successful planners of the past years, from FFReplan to RETRASE introduced in the previous chapter, have critically relied on the ability to find a trajectory to a goal state in a determinization of a given MDP. Unfortunately, while powerful on many problems, this strategy implicitly biases the approximation methods towards goal-oriented MDPs with particular characteristics and renders them completely ineffective on others.

4.1 Overview

The weaknesses of the existing solution techniques have been demonstrated by the new benchmark MDPs of the 2011 International Probabilistic Planning Competition [86] (IPPC-2011). These problems belong to the class of factored finite-horizon scenarios with an initial state, denoted as FH_{s_0} and described by transforming the FH MDP definition (2.10) in the same way as the definition of SSP MDPs (2.16) is transformed into the factored SSP$_{s_0}$ MDP definition (2.20). Like SSP_{s_0}, FH_{s_0} models many important problems, from inventory management [80] to teaching a skill to a student over a given number of lessons [86]. Formally, factored FH_{s_0} is a subclass of factored SSP_{s_0}, so we would expect all the algorithms for the latter to extend to the former. However, this turns out not to be the case in practice.

For example, the determinization-based algorithms (including those introduced in Chapter 3) that can solve SSP$_{s_0}$ MDPs are inapplicable to FH$_{s_0}$ MDPs, due to the goal structure of the latter. In SSP MDPs in general, finding a path to the goal is nontrivial. In fact, the probability of reaching the goal from s_0 serves as a reasonable proxy optimization criterion when solving SSP$_{s_0}$ MDPs approximately, since optimizing the expected cost of doing so is hard and because in goal-oriented scenarios reaching the goal is semantically more important than cost minimization. It is the goal attainment probability that was the measure of policy quality in the previous IPPCs and in the experiments in Chapter 3, and determinization-based planning excelled at optimizing it. In contrast, finding the goal from any given state in FH MDPs is very easy. Recall from the proof of Theorem 2.5 that if a FH MDP is viewed as a goal-oriented problem, then all states at the problem's horizon are goals; the agent will get to the goal eventually no matter what policy it follows. Thus, goal attainment probability is not meaningful in FH MDPs. Moreover, it is not clear how to use deterministic planning to optimize the expected reward over a fixed number of steps, the main policy quality criterion in FH MDPs. Fast deterministic planners are usually highly suboptimal in terms of the cost of the plans that they find, and are not designed to look for plans of a specific length.

Another aspect of IPPC-2011 MDPs preventing both determinization-based and the more conventional dynamic programming-based solvers is their high-entropy transition functions. For probabilistic planning problems, we define the entropy of the transition function as the average entropy of the successor state distribution of a state-action pair. Intuitively, problems with a high-entropy transition function not only have a high average number of successors per state-action pair, but also a large number of highly likely ones among them. In real-world systems, high-entropy transition functions are common and are often caused by *natural dynamics*, effects of exogenous events or forces of nature that cannot be directly controlled but that need to be taken into account during planning. The exogenous events, e.g., changes in weather, alter the state in parallel with an agent's actions, enabling the world to transition to many states with a high probability in just one time step. This thwarts determinization-based solvers because the all-outcome determinizations of such problems have very large branching factors. The branching factor cannot always be easily reduced by omitting low-probability action outcomes, because the pruned problem can become an overly crude approximation of the original one. For instance, in the Mars rover example, disregarding various possibilities of equipment failure, which generally have low probabilities, may render the planning problem trivial. Due to the large branching factors, solving such determinizations becomes difficult even for classical planners, negating the main strength of determinization-based techniques. For dynamic programming-based solvers, high-entropy transition functions create similar complications. At the core of all these algorithms is a Bellman backup operator similar to the one in Equation 2.9, evaluating which for a single state requires iterating over all positive-probability successors of all corresponding state-action pairs — a prohibitively expensive step in the cases when the transition function is high-entropy.

An algorithm with a great potential to tackle large factored FH$_{s_0}$ MDPs is UCT [52], a Monte Carlo Tree Search technique. Originally invented as a reinforcement learning strategy, it does not explicitly manipulate the transition function's probabilities and is optimal in the limit. A version of it, PROST [49], was the winner of IPPC-2011. At the same time, to be successful, UCT needs several of its parameters and components to be tuned well. For a specific problem, this process may be justifiable. However, UCT's parameter values carry over poorly from one scenario to the next, making UCT fairly brittle as a general autonomous MDP solver.

The contribution of this dissertation to the state of the art in solving large factored FH$_{s_0}$ MDPs is a series of three techniques culminating in a top-performing, easily tunable algorithm for these problems:

- The algorithm that forms a theoretical basis for the other two is LR^{2}TDP [54]. LR^{2}TDP is founded on a crucial observation that for many FH$_{s_0}$ MDPs with initial state s_0 and horizon H, which we will henceforth denote as $M(s_0, H)$, one can produce a successful policy by solving $M(s_0, h)$, the same MDP but with a much smaller horizon h, and "extending" its solution up to horizon H. Therefore, under time constraints, trying to solve the sequence of MDPs $M(s_0, 1), M(s_0, 2), \cdots$ with increasing horizon will often yield a near-optimal policy even if the computation is interrupted long before the planner gets to tackle MDP $M(s_0, H)$. This strategy, which we call *reverse iterative deepening*, is at the heart LR^{2}TDP. Its crucial distinction from iterative deepening is that it obtains a solution for MDP $M(s_0, h)$ by augmenting the solution for $M(s_0, h-1)$ obtained earlier, as opposed to building it from scratch. This gives LR^{2}TDP higher speed and better anytime performance than that of its forerunner, LRTDP.

- Although LR^{2}TDP's intuition addresses the issue of anytime performance, by itself it does not enable LR^{2}TDP to handle large branching factors. For this purpose, we introduce GLUTTON [54], a planner derived from LR^{2}TDP and our entry in IPPC-2011. GLUTTON endows LR^{2}TDP with optimizations that help achieve competitive performance on difficult problems with large branching factors – subsampling the transition function, separating out natural dynamics, caching transition function samples, and using primitive cyclic policies as a fall-back solution.

- Both LR^{2}TDP and GLUTTON do most of their planning *offline*: they try to find a policy closed w.r.t. s_0, which potentially involves many states, and only then execute it. This is wasteful if the policy needs to be executed only a few times, since only a few of the states will end up getting visited. The last algorithm for factored FH$_{s_0}$ MDPs that we propose, GOURMAND [59], is an *online* version of LR^{2}TDP. It incorporates many of the same optimizations as GLUTTON and plans as the agent travels through the state space, thereby saving valuable computational resources. PROST, the winner of IPPC-2011, is also an online algorithm, but GOURMAND has many fewer parameters and outperforms PROST thanks to generalizing better across different scenarios.

4.2 Preliminaries

Before delving into the details of LR^{2}TDP, GLUTTON, and GOURMAND, we slightly change the FH MDP-related notation introduced in Chapter 2 and discuss the main competitor of the techniques we propose, the UCT algorithm.

4.2.1 Notation

As for SSP$_{s_0}$ problems, an optimal policy for FH$_{s_0}$ MDPs must satisfy a special version of the Bellman equations (Equations 2.1). To make subsequent explanations clearer, we introduce a change of variable in these equations by expressing value functions not in terms of the current time t but in terms of the number of steps, h, remaining to reach the horizon. In an FH$_{s_0}$ MDP $M(s_0, H)$, we have $h = H - t + 1$. For simplicity, in this chapter we also assume that the transition and reward functions of the MDPs we are dealing with are stationary and denote them as $\mathcal{T}(s, a, s')$ and $\mathcal{R}(s, a, s')$, respectively, although the algorithms we will discuss can be easily adapted to nonstationary transitions and rewards as well. With these alterations, the Bellman equations for FH MDPs become

$$V^*(s,0) = 0 \qquad \forall s \in \mathcal{S} \tag{4.1}$$

$$V^*(s,h) = \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s')[\mathcal{R}(s,a,s') + V^*(s',h-1)] \right] \quad \forall h \in [1,H],\ s \in \mathcal{S}. \tag{4.2}$$

When discussing solution techniques, we will reason about the *augmented state space* of an FH$_{s_0}$ MDP $M(s_0, H)$, which is a set $\mathcal{S} \times \{0, \ldots, H\}$ of state-number of steps-to-go pairs. In these terms, an optimal solution of $M(s_0, H)$ is a policy $\pi^*_{s_0,H}$ maximizing the expected reward that can be collected starting from augmented state (s_0, H).

4.2.2 Additional Background

In the probabilistic planning community, finite-horizon problems have not yet received as much attention as goal-oriented MDPs, and the main algorithms developed specifically for them have been the mathematically fundamental, but not very efficient, VI and PI. At the same time, these problems can be viewed as acyclic MDP with a special structure, implying that heuristic search algorithms for acyclic MDP algorithms, such as AO* [75], apply to them without modification. A more recent method for solving FH$_{s_0}$ MDPs is UCT, which appears to have a number of advantages over VI/PI and AO*. Below, we review a version of VI specialized to finite-horizon scenarios and then concentrate on UCT.

Value Iteration

VI for finite-horizon MDPs is simpler and more efficient than for SSP problems (Algorithm 2.3). For the former, unlike for the latter, the maximum length of any trajectory equals the MDP's horizon H. This allows the optimal value function to be computed in a single pass over the state space by setting $V^*(s,0) = 0$ as in Equation 4.1 and computing $V^*(s,h)$ as h ranges from 1 to H using Equation 4.2. However, space-wise this flavor of VI is more wasteful, as it has to store a number for each state in the *augmented* state space.

UCT

At a high level, UCT [52] works in a similar way to RTDP: it samples a series of trials. Each trial consists of choosing an action in the current state, simulating the action, virtually transitioning to a new state, and repeating this process in that new state. However, UCT's state value update strategy is distinctly different. It does not use Bellman backups, instead estimating the state value as an aggregate of the rewards obtained starting from this state so far. Moreover, in each visited state UCT picks an action based on the action's current quality estimate (an approximation of the action's Q-value) and an "exploration term". The exploration term forces UCT to choose actions that have been tried rarely in the past, even if their estimated quality is low. The avoidance of Bellman backups makes UCT suitable for many scenarios other MDP algorithms cannot handle (e.g., when the transition function is not known explicitly or has a high entropy). However, it also extorts a price: unlike in VI or RTDP, UCT's action quality estimates do not improve monotonically. This leads UCT to try suboptimal actions from time to time and prevents it from having a reliable termination condition indicating when UCT is near convergence.

Online versus Offline Planning

Many planning algorithms, including VI, LRTDP, and UCT, allow offline planning mode, when the planner tries to find a complete optimal policy, or at least an optimal policy closed w.r.t. (s, H), before executing it. In many cases, doing so is infeasible and unnecessary — the problem may have so many states that they cannot all be visited over the lifetime of the system, so finding π^* for them is a waste of time. E.g., over its lifetime a robot will ever find itself in only a small fraction of possible configurations. Such problems may be better solved *online*, i.e., by finding π^* or its approximation for the current state, executing the chosen action, and so on. In many scenarios, the MDP needs to be solved online and under time constraint. Ways of adapting MDP solvers to the online setting vary depending on the algorithm.

4.3 LR^{2}TDP

We begin by introducing LR^{2}TDP, an extension of LRTDP for finite-horizon problems. Like its predecessor, LR^{2}TDP solves an MDP for the given initial state s_0 optimally in the limit and has a good anytime performance. Extending LRTDP to finite-horizon problems may seem an easy task, but its most straightforward extension performs worse than the one we propose, LR^{2}TDP.

4.3.1 LR^{2}TDP *Description*

As a reminder, LRTDP for goal-oriented MDPs (Algorithm 2.7) operates in a series of trials starting at the initial state s_0. Each trial consists of choosing the greedy best action in the current state according to the current value function, performing a Bellman backup on the current state, sampling an outcome of the chosen action, transitioning to the corresponding new state, and repeating the cycle. A trial continues until it reaches a goal or a converged state. At the end of each trial, LRTDP performs a special convergence check on all states in the trial to prove, whenever possible, the convergence of these states' values. Once it can prove that s_0 has converged, LRTDP halts.

Thus, the most straightforward adaptation of LRTDP to an FH$_{s_0}$ MDP $M(s_0, H)$, which we call LRTDP$_{FH}$, is to let each trial start at (s_0, H) and run for at most H time steps. Indeed, if we convert a finite-horizon MDP to its goal-oriented counterpart, all states H steps away from s_0 are goal states. However, as we explain below, LRTDP$_{FH}$'s anytime performance is not very good, so we turn to a more sophisticated approach.

Our algorithm, LR^{2}TDP, follows a strategy that we name *reverse iterative deepening*. As its pseudocode in Algorithm 4.1 shows, it uses LRTDP$_{FH}$ in a loop to solve a sequence of MDPs $M(s_0, 1), M(s_0, 2), \cdots, M(s_0, H)$, in that order. In particular, LR^{2}TDP first decides how to act optimally in $(s_0, 1)$, i.e., assuming there is only one more action to execute — this is exactly equivalent to solving $M(s_0, 1)$. Then, LR^{2}TDP runs LRTDP$_{FH}$ to decide how to act optimally starting at $(s_0, 2)$, i.e., two steps away from the horizon — this amounts to solving $M(s_0, 2)$. Then it runs LRTDP$_{FH}$ again to decide how to act optimally starting in $(s_0, 3)$, thereby solving $M(s_0, 3)$, and so on. Proceeding this way, LR^{2}TDP either eventually solves $M(s_0, H)$ or, if operating under a time limit, runs out of time and halts after solving $M(s_0, h')$ for some $h' < H$.

Crucially, in the spirit of dynamic programming, LR^{2}TDP reuses state values computed while

Algorithm 4.1: LR^{2}TDP

```
Input: FH$_{s_0}$ MDP $M(s_0, H) = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, H, s_0 \rangle$, heuristic $V_0$, $\epsilon > 0$, (optional) timeout $T$
Output: a policy closed w.r.t. $s_0$, optimal if $V_0$ is admissible, $\epsilon$ is sufficiently small, and $T$ is
sufficiently large

$V \leftarrow V_0$

function LR$^2$TDP(FH$_{s_0}$ MDP $M(s_0, H)$, $\epsilon > 0$, (optional) timeout $T$)
begin
    foreach $s \in \mathcal{S}$ do $V(s, 0) \leftarrow 0$
    foreach $h = 1, \ldots, H$ or until time $T$ runs out do
        $t_{start} \leftarrow$ current time
        $\pi \leftarrow$ LRTDP$_{FH}$($M(s_0, h)$, $\epsilon$, $T$)
        $t_{end} \leftarrow$ current time
        $T \leftarrow T - (t_{end} - t_{start})$
    end
    return $\pi$
end

function LRTDP$_{FH}$(FH$_{s_0}$ MDP $M(s, h)$, $\epsilon > 0$, (optional) timeout $T'$)
begin
    // Convert $M(s, h)$ into the equivalent goal-oriented MDP $M_g^h$
    $\mathcal{S}' \leftarrow \mathcal{S} \times \{0, \ldots, h\}$
    $\mathcal{G} \leftarrow \{(s', 0) \mid s' \in \mathcal{S}\}$
    $M_g^h \leftarrow \langle \mathcal{S}', \mathcal{A}, \mathcal{T}, -\mathcal{R}, \mathcal{G}, s_0 \rangle$
    // Run LRTDP (Algorithm 2.7), memoizing the values of all the encountered augmented states
    $\pi_{s_0,h} \leftarrow$ LRTDP($M_g^h$, $\epsilon$, $T'$)
    return $\pi_{s_0,h}$
end
```

solving $M(s_0, 1), M(s_0, 2), \ldots, M(s_0, h{-}1)$ when tackling the next MDP in the sequence, $M(s_0, h)$. Namely, observe that any (s, h') in the augmented state space of any MDP $M(s_0, h'')$ also belongs to the augmented state spaces of all MDPs $M(s_0, h''')$, $h''' \geq h''$, and $V^*(s, h')$ is the same for all these MDPs. Therefore, by the time LR^{2}TDP gets to solving $M(s_0, h)$, values of many of its states will have been updated or even converged as a result of handling some $M(s_0, i)$, $i < h$. Accordingly, LR^{2}TDP memoizes values and convergence labels of all augmented states ever visited by LRTDP$_{FH}$ while solving for smaller horizon values, and reuses them to solve subsequent MDPs in the above sequence. Thus, solving $M(s_0, h)$ takes LR^{2}TDP only an incremental effort over the solution of $M(s_0, h-1)$.

LR^{2}TDP can be viewed as backchaining from the goal in an acyclic goal-oriented MDP. Indeed, a finite-horizon MDP $M(s_0, H)$ is simply a goal-oriented MDP whose state space is the augmented state space of $M(s_0, H)$, and whose goals are all states of the form (s, H). It has no loops because executing any action leads from some state (s, h) to another state $(s', h-1)$. LR^{2}TDP essentially solves such MDPs by first assuming that the goal is one step away from the initial state, then two steps from the initial state, and so on, until it addresses the case when the goal is H steps away from the initial state. Compare this with LRTDP$_{FH}$'s behavior when solving $M(s_0, H)$. LRTDP$_{FH}$ does not backtrack from the goal; instead, it tries to forward-chain from the initial state to the goal (via trials) and propagates state values backwards whenever it succeeds. As an alternative perspective, LRTDP$_{FH}$ iterates on the search depth, while LR^{2}TDP iterates on the distance from the horizon. The benefit of the latter is that it allows for the reuse of computation across different iterations. In a sense, LR^{2}TDP is a natural generalization of VI to finite-horizon problems with an initial state: VI for FH$_{s_0}$ MDPs essentially relies on reverse iterative deepening too, but uses it to compute actions for *all* states for each distance h to horizon, as opposed to only those reachable from a particular starting state s_0 in h steps as LR^{2}TDP does.

Clearly, both LRTDP$_{FH}$ and LR^{2}TDP eventually arrive at the optimal solution. So, what are the advantages of LR^{2}TDP over LRTDP$_{FH}$? We argue that if stopped before convergence, the policy of LR^{2}TDP is likely to be much better for the following reasons:

- In many MDPs $M(s_0, H)$, the optimal policy for $M(s_0, h)$ for some $h << H$ is optimal or near-optimal for $M(s_0, H)$ itself. E.g., consider a manipulator that needs to transfer blocks regularly arriving on one conveyor belt onto another belt. The manipulator can do one pick-up, move, or put-down action per time step. It gets a unit reward for moving each block, and needs to accumulate as much reward as possible over 50 time steps. Delivering one block from one belt to another takes at most 4 time steps: move manipulator to the source belt, pick up a block, move manipulator to the destination belt, release the block. Repeating this sequence of actions over 50 time steps clearly achieves maximum reward for $M(s_0, 50)$. In other words, $M(s_0, 4)$'s policy is optimal for $M(s_0, 50)$ as well.

  Therefore, explicitly solving $M(s_0, 50)$ for all 50 time steps is a waste of resources — solving $M(s_0, 4)$ is enough. However, LRTDP$_{FH}$ will try to do exactly the former: it will spend a lot of effort trying to solve M for horizon 50 at once. Since it "spreads" its effort over many time steps, it will likely fail to completely solve $M(s_0, h)$ for any $h < H$ by the deadline. Contrariwise, LR^{2}TDP solves the given problem incrementally, and may have a solution for $M(s_0, 4)$ (and hence for $M(s_0, 50)$) if stopped prematurely.

- When LRTDP$_{FH}$ starts running, many of its trials are very long, since each trial halts only when it reaches a converged state, and at the beginning reaching a converged state takes about H time steps. Moreover, at the beginning, each trial causes the convergence of only a few states (those near the horizon), while the values of augmented states with small time step values change very little. Thus, the time spent on executing the trials is largely wasted. In contrast, LR^{2}TDP's trials when solving an MDP $M(s_0, h)$ are very short, because they quickly run into states that converged while solving $M(s_0, h-1)$ and before, and often lead to the convergence of most of a trial's states. Hence, we can expect LR^{2}TDP to be faster.

- As a consequence of large trial length, LRTDP$_{FH}$ explores (and therefore memorizes) many augmented states whose values (and policies) will not have converged by the time the planning process is interrupted. Thus, it risks using up available memory before it runs out of time, and to little effect, since it will not know well how to behave in most of the stored states anyway. In contrast, LR^{2}TDP typically knows how to act optimally in a large fraction of augmented states in its memory.

Note that, incidentally, LR^{2}TDP works in much the same way as VI, raising a question: why not use VI in the first place? The advantage of asynchronous dynamic programming over VI in finite-horizon settings is similar to its advantage in goal-oriented settings. A large fraction of the state space may be unreachable from s_0 in general and by the optimal policy in particular. LR^{2}TDP avoids storing information about many of these states, especially if guided by an informative heuristic. In addition, in finite-horizon MDPs, many states are not reachable from s_0 *within H steps*, further increasing potential savings from using LR^{2}TDP.

So far, we have glossed over a subtle question: if LR^{2}TDP is terminated after solving $M(s_0, h)$, $h < H$, what policy should it use in augmented states (s, h') that it has never encountered? There are two cases to consider — a) LR^{2}TDP has labeled s as solved for some $h'' < \min\{h, h'\}$, and b) LR^{2}TDP has not solved (or even visited) s for any time step. In the first case, LR^{2}TDP can simply find the largest value $h'' < \min\{h, h'\}$ for which (s, h'') is solved and return the optimal action for (s, h''). This is the approach we use in GLUTTON, a further development of LR^{2}TDP, and it works well in practice. Case b) is more complicated and may arise, for instance, when s is not reachable from s_0 within h steps. One possible solution is to fall back on some simple *default policy* in such situations, an approach analogous to using rollout policies in UCT. We discuss this option when describing the implementation of GLUTTON.

4.3.2 Max-Reward Heuristic

LR^{2}TDP is a FIND-AND-REVISE algorithm (Section 2.3.1), and will converge to an optimal solution if initialized with an admissible heuristic, i.e., an upper bound on V^*. For this purpose, LR^{2}TDP uses an estimate we call the *Max-Reward* heuristic. Its computation hinges on knowing the maximum reward R_{max} any action can yield in any state, or an upper bound on it. R_{max} can be automatically derived for an MDP using simple domain analysis.

To produce a heuristic value $V_0(s, h)$ for (s, h), Max-Reward finds the largest horizon value $h' < h$ for which LR^{2}TDP already has an estimate $V(s, h')$. Recall that LR^{2}TDP is likely to have $V(s, h')$ for some such h', since it solves the given MDP in the reverse iterative deepening fashion. If so, Max-Reward sets $V_0(s, h) = V(s, h') + (h - h')R_{max}$; otherwise, it sets $V_0(s, h) = hR_{max}$. The bound obtained in this way is often loose but is guaranteed to be admissible.

4.4 **GLUTTON**

In spite of its good anytime behavior, LR^{2}TDP by itself does not perform well on finite-horizon MDPs with high-entropy transition functions. As discussed earlier, high-entropy transition functions can be induced by the natural dynamics of a problem, especially if the scenario involves exogenous events. To remedy LR^{2}TDP's weaknesses, we present GLUTTON, our entry at the IPPC-2011 competition that endows LR^{2}TDP with mechanisms for efficiently handling natural dynamics and other optimizations.

4.4.1 GLUTTON *Description*

At a high level, GLUTTON can be viewed as a version of LR^{2}TDP for *factored* FH$_{s_0}$ MDPs, augmented with several approximations for increased efficiency. Pure LR^{2}TDP does not make use of an MDP's state variable factorization and, as a consequence, is oblivious to a lot of problem structure. GLUTTON fills that gap; below we describe each of its factorization-based optimizations in detail.

Subsampling the Transition Function

GLUTTON's way of dealing with a high-entropy transition function is to subsample it. For each encountered state-action pair (s, a), GLUTTON samples a set $U_{s,a}$ of successors of s under a, and performs Bellman backups using states in $U_{s,a}$:

$$V^*(s,h) \approx \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \sum_{s' \in U_{s,a}} \mathcal{T}(s,a,s') V^*(s', h-1) \right\} \tag{4.3}$$

The size of $U_{s,a}$ is chosen to be much smaller than the number of states to which a could transition from s. There are several heuristic ways of setting this value, e.g. based on the entropy of the transition function. At IPPC-2011 we chose $|U_{s,a}|$ for a given problem to be a constant: GLUTTON first tried to solve a problem with $|U_{s,a}| = 20$ and then, if there was time remaining, with $|U_{s,a}| = 30$. Overall, we found GLUTTON to be fairly insensitive to $|U_{s,a}|$ values past 20. We purposefully did not make an attempt to carefully tune this parameter, in order to demonstrate that GLUTTON's performance is robust across a wide range of benchmarks even with crudely picked $|U_{s,a}|$.

Subsampling can give an enormous improvement in efficiency for GLUTTON at a reasonably small reduction in the solution quality compared to full Bellman backups. However, subsampling alone does not make solving many of the IPPC benchmarks feasible for GLUTTON. Consider, for instance, the Sysadmin problem, introduced in Section 2.1.9, that involves maintaining a network of servers. In its instance with 50 servers (and hence 50 state variables), there are a total of 51 ground actions, one for restarting each server plus a *noop* action. Each action can potentially change all 50 variables, and the value of each variable is sampled independently from the values of others. Suppose we set $|U_{s,a}| = 30$. Even for such a small size of $U_{s,a}$, determining the current greedy action in just one state could require $51 \cdot (50 \cdot 30) = 76,500$ variable sampling operations. Considering that the procedure of computing the greedy action in a state may need to be repeated billions of times, the need for further improvements, such as those describe next, quickly becomes evident.

Separating Out Natural Dynamics

One of our key observations is the fact that the efficiency of sampling successor states for a given state can be drastically increased by reusing some of the variable samples when generating successors for multiple actions. To do this, we separate each action's effect into those due to natural dynamics (*exogenous* effects), those due to the action itself (*pure* effects), and those due to some interaction between the two (*mixed* effects). More formally, assume that an MDP with natural dynamics has a special action *noop* that captures the effects of natural dynamics when the controller does nothing. In the presence of natural dynamics, for each non-*noop* action a, the set $\mathcal{X}$ of a problem's state variables can be represented as a disjoint union

$$\mathcal{X} = \mathcal{X}_a^{ex} \cup \mathcal{X}_a^{pure} \cup \mathcal{X}_a^{mixed} \cup \mathcal{X}_a^{none}$$

Moreover, for the *noop* action we have

$$\mathcal{X} = (\cup_{a \neq noop}(\mathcal{X}_a^{ex} \cup \mathcal{X}_a^{mixed})) \cup \mathcal{X}_{noop}^{none}$$

where X_a^{ex} are variables acted upon only by the exogenous effects, X_a^{pure} — only by the pure effects, X_a^{mixed} — by both the exogenous and pure effects, and X_a^{none} are not affected by the action at all. For example, in a Sysadmin problem with n machines, for each action a other than the *noop*, $|X_a^{pure}| = 0$, $|\mathcal{X}_a^{ex}| = n-1$, and $|\mathcal{X}_a^{none}| = 0$, since natural dynamics acts on any machine unless the administrator restarts it. $|X_a^{mixed}| = 1$, consisting of the variable for the machine the administrator restarts. Notice that, at least in the Sysadmin domain, for each non-*noop* action a, $|X_a^{ex}|$ is much larger than $|X_a^{pure}|+|X_a^{mixed}|$. Intuitively, this is true in many real-world domains as well — natural dynamics affects many more variables than any single non-*noop* action. These observations suggest generating $|U_{s,noop}|$ successor states for the *noop* action, and then modifying these samples in order to obtain successors for other actions by resampling some of the state variables using each action's pure and mixed effects.

We illustrate this technique on the example of approximately determining the greedy action in some state s of the Sysadmin-50 problem. Namely, suppose that for each action a in s we want to sample a set of successor states $U_{s,a}$ to evaluate Equation 4.3. First, we generate $|U_{s,noop}|$ $noop$ sample states using the natural dynamics (i.e., the *noop* action). Setting $|U_{s,noop}| = 30$ for the sake of the example, this takes $50 \cdot 30 = 1500$ variable sampling operations, as explained previously. Now, for each resulting $s' \in U_{s,noop}$ and each $a \neq noop$, we need to re-sample variables $\mathcal{X}_a^{pure} \cup \mathcal{X}_a^{mixed}$ and substitute their values into s'. Since $|\mathcal{X}_a^{pure} \cup \mathcal{X}_a^{mixed}| = 1$, this takes one variable sampling operation per action per $s' \in U_{s,noop}$. Therefore, the total number of additional variable sampling operations to compute sets $U_{s,a}$ for all $a \neq noop$ is 30 *noop* state samples $\cdot$ 1 variable sample per non-*noop* action per *noop* state sample $\cdot$ 50 non-*noop* actions $= 1500$. This gives us 30 state samples for each non-*noop* action. Thus, to evaluate Equation 4.3 in a given state with 30 state samples per action, we have to perform $1500 + 1500 = 3000$ variable sampling operations. This is about 25 times fewer than the 76,500 operations we would have to perform if we subsampled naively. Clearly, in general the speedup will depend on how "localized" actions' pure and mixed effects in the given MDP are compared to the effects of natural dynamics.

The caveat of sharing the natural dynamics samples for generating non-*noop* action samples is that the resulting non-*noop* action samples are not independent, i.e., are biased. However, in our experience, the speedup from this strategy (as illustrated by the above example) and associated gains in policy quality when planning under time constraints outweigh the disadvantages due to the bias in the samples.

Caching the Transition Function Samples

In spite of the already significant speedup due to separating out the natural dynamics, we can compute an approximation to the transition function even more efficiently. Notice that nearly all the memory used by algorithms such as LR^{2}TDP is occupied by the state-value table containing the values for the already visited augmented states (s, h). Since LR^{2}TDP populates this table lazily (as opposed to VI), when LR^{2}TDP starts running the table is almost empty and most of the available memory on the machine is unused. GLUTTON uses this memory as a cache for samples from the transition function. That is, when GLUTTON analyzes a state-action pair (s, a) for the first time, it samples successors of s under a as described above and stores them in this cache (in Section 4.2.1, we assumed the MDP to be stationary, so the samples for (s, a) do not need to be cached separately for each number of steps-to-go h). When GLUTTON encounters (s, a) again, it retrieves the samples for it from the cache, as opposed to re-generating them. Initially, the GLUTTON process is CPU-bound, but due to caching it quickly becomes memory-bound as well. Thus, the cache helps GLUTTON make the most of the available resources. When all of the memory is filled up, GLUTTON starts gradually shrinking the cache to make room for the growing state-value table. Currently, it chooses state-action pairs for eviction and replacement randomly.

Default Policies

Since GLUTTON subsamples the transition function, it may terminate with an incomplete policy — it may not know a good action in states it missed due to subsampling. To pick an action in such a state (s, h'), GLUTTON first attempts to use the trick discussed previously, i.e., to return either the optimal action for some solved state (s, h''), $h'' < h'$, or a random one. However, if the branching factor is large or the amount of available planning time is small, GLUTTON may need to do such random "substitutions" for so many states that the resulting policy is very bad, possibly worse than the uniformly random one.

As it turns out, for many MDPs there are simple *cyclic policies* that do much better than the completely random one. A cyclic policy consists in repeating the same sequence of steps over and over again. Consider, for instance, the robotic manipulator scenario from Section 4.3. The optimal policy for it repeats an action cycle of length 4. In general, near-optimal cyclic policies are difficult to discover. However, it is easy to evaluate the set of *primitive cyclic policies* for a problem, each of which repeats a *single* action.

This is exactly what GLUTTON does. For each action, it evaluates the cyclic policy that repeats that action in any state by simulating this policy several times and averaging the reward. Then, it selects the best such policy and compares it to three others, also evaluated by simulation: (1) the "smart" policy computed by running LR^{2}TDP with substituting random actions in previously unencountered states, (2) the "smart" policy with substituting the action from the best primitive cyclic policy in these states, and (3) the completely random policy. For the actual execution, GLUTTON uses the best of these four. As we show in the Experiments section, for several domains pure primitive cyclic policies turned out to be surprisingly effective.

Our concept of default policies, e.g., the cyclic ones, parallels the notion of rollout policies in UCT. UCT uses rollout policies in states that it has not encountered previously and hence has no information about, similar to GLUTTON. The effectiveness of cyclic policies in GLUTTON suggests that they may benefit UCT and its variants as well.

4.5 GOURMAND

In spite of its optimizations, across the factored FH$_{s_0}$ MDP benchmarks of the IPPC-2011 competition the LRTDP-based GLUTTON gets slightly outperformed by a system called PROST [49]. PROST is based on the UCT algorithm, a technique that, among other achievements, has drastically improved machines' ability to play Go [33] and Solitaire [8]. Recently, UCT has been closely studied as a probabilistic planning tool: in addition to PROST, all other IPPC-2011 participants except GLUTTON were derived from it as well. Perhaps unexpectedly, despite being beaten by PROST, GLUTTON dominated all of these other UCT-based planners. Meanwhile, besides the use of UCT and LRTDP respectively, another major difference between PROST and GLUTTON was *how* they used these algorithms. While PROST employed UCT in an *online* manner, interleaving planning and execution, GLUTTON constructed policies *offline*.

Critically analyzing these results, we ask: is the nascent trend of using UCT as the dominant probabilistic planning algorithm justified? Which, of UCT and LRTDP, performs better if both are used online? Does LRTDP have any practical advantages over UCT in online mode? To start answering these questions, we compare the suitability of UCT and LRTDP to solving finite-horizon MDPs online under time constraints.

In an online setting, the planner decides on the best or near-optimal action in the current state, executes it, decides on an action in the state where it ends up, and so on. As mentioned in Section

4.2.2, the advantage of this approach is the that planner spends much less resources on analyzing states the agent never visits — they are only analyzed as a side-effect of computing an action for a state it does visit. Of course, online planning is feasible only if the agent can afford to plan in the middle of acting. In this section, we assume this to be the case, and, in fact, many real-life scenarios do conform to this assumption. E.g., consider a robot trying to navigate an environment with a lot of people. Instead of computing a policy offline by considering the probabilities of people showing up in its path, it can decide on the direction in which to move until the next decision epoch, e.g., for 1 second, in order to avoid running into anyone. Executing the action will bring it to a different state, where it can repeat the decision process.

There are many possible ways of choosing good actions online for the states the agent encounters without solving the entire MDP. A principled approach to doing this is illustrated by Figure 1.1. At the t-th decision epoch in a finite-horizon MDP $M(s_0, H)$, when the agent is in a state s, the agent finds an optimal policy π^*_{s,L_t} for the MDP $M(s, L_t)$, a problem that is identical to the original one except for the start state, which is now s, and the horizon $L_t < h$, where $h = H - t + 1$. The agent can then choose action $\pi^*_{s,L_t}(s, L_t)$ recommended by $M(s, L_t)$'s optimal policy. This action selection rule has the intuitive property that if the number L_t, which we call *lookahead*, is as large as the number of decision epochs h remaining till the end of the process, an optimal policy resulting from it, an *L_t-lookahead policy*, is optimal for the original MDP starting from augmented state (s, h). Note that this *does not* imply that as L_t approaches h, the quality of an optimal L_t-lookahead policy, as measured by its value function, *monotonically* approaches that of $\pi^*_{s,h}$. Indeed, one can construct pathological examples in which increasing lookahead up to a certain point results in policies of deteriorating quality. However, in many non-contrived real-life examples, larger lookahead generally translates to a better policy. Thus, given a time constraint, obtaining the best approximation in practice according to this scheme requires determining the optimal, or at least a good, value of L_t for each decision epoch. Unfortunately, these lookahead values are usually unknown a-priori for the problem at hand.

We claim that because of the difficulty of choosing the L_t values, LRTDP is generally better suited for solving finite-horizon MDPs under time constraints than UCT. In particular, UCT does not have a convergence condition, making it hard to determine the time it takes it to converge for a given lookahead L and effectively forcing the practitioner to specify the same L for all decision epochs based on a guess. In the meantime, a good L is heavily problem-dependent. Setting it too high may prevent UCT from converging to a good action for the state for this lookahead within a reasonable time and will force it to pick an action largely at random. Setting it too low may make UCT's behavior too myopic. On the other hand, LRTDP, thanks to its convergence condition, can help determine a good lookahead value automatically via a reverse iterative deepening strategy of the kind used in GLUTTON. Moreover, if the time constraint is specified for executing the entire process, not on a per-epoch-basis, iterative deepening LRTDP allows for a strategy that distributes the available computation time among different decision epochs in the process in a problem-independent manner and without human intervention. We implement these observations in a novel LRTDP-based planner called GOURMAND [59] that robustly solves finite-horizon MDP online and performs better than both GLUTTON and PROST.

4.5.1 GOURMAND *Description*

GOURMAND is analogous to PROST in that both use a version of a basic algorithm, LRTDP and UCT respectively, to choose an action in an augmented state (s, h) encountered at the t-th decision epoch of the process as described above, by trying to solve the state for some lookahead. However, while PROST needs an engineer to specify the value of the lookahead and the timeout to devote to choosing an action at epoch t, GOURMAND determines both of these values without human intervention. GOURMAND is also related to GLUTTON — they share the same flavor of LRTDP, LR^{2}TDP, and some of the engineering optimizations such as subsampling the transition function. A major difference between the two is the mode in which they use LR^{2}TDP. GLUTTON uses it in an offline fashion. As a result, when the time T allocated for solving the MDP runs out, GLUTTON may not have solved all the states reachable by its policy from s_0 and has to resort to ad-hoc methods of action selection, e.g., the primitive cyclic policies, when it encounters such states during the computed policy's execution. GOURMAND does not have this difficulty: thanks to its online use of LR^{2}TDP and its time allocation strategy, it makes an informed action choice in any state where it ends up.

Algorithm 4.2 shows GOURMAND's pseudocode. Initially, GOURMAND distributes the total available time T equally among the H decision epochs (lines 9, 15) and, while choosing an action

Algorithm 4.2: GOURMAND

```
Input: FH$_{s_0}$ MDP $M(s_0, H) = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, H, s_0 \rangle$, heuristic $V_0$, $\epsilon > 0$, timeout $T$
Output: none (executes actions for states encountered at time steps $1, \ldots, H$)

$V \leftarrow V_0$

// Running averages of the amount of time it takes to solve a state for lookahead $L$
$Ts_0 \leftarrow 0$
$Ts_L \leftarrow \infty$ for all $L = 1, \ldots, H$
$\overline{T} \leftarrow T$

function GOURMAND(FH$_{s_0}$ MDP $M(s_0, H)$, $\epsilon > 0$, timeout $T$)
begin
    $s \leftarrow s_0$
    foreach $t = 1, \ldots, H$ do
        $T_t \leftarrow \frac{\overline{T}}{H-t+1}$
        if $t == 1$ then
            $\pi \leftarrow$ LR$^2$TDP_TIMED$(M(s_0, H), \epsilon, T_t)$
            $\overline{T} \leftarrow \overline{T} - T_t$
            $\hat{L}_t \leftarrow$ largest $L$ for which augmented state $(s_0, L)$ has been solved
            $\pi^*_{s,\hat{L}_t} \leftarrow \pi$
        else
            $L_t \leftarrow$ largest $L \leq H - t + 1$ s.t. $Ts_L < T_t$
            $\hat{T}_t \leftarrow T_t + (T_t - Ts_{L_t})(H - t)$
            $\hat{L}_t \leftarrow L_t$
            if $(\hat{L}_t < H - t + 1)$ $and$ $(Ts_{L_t+1} < \hat{T}_t$ $or$ $Ts_{L_t+1} == \infty)$ then $\hat{L}_t \leftarrow L_t + 1$
            $t_{start} \leftarrow$ current time
            $\pi^*_{s,\hat{L}_t} \leftarrow$ LR$^2$TDP_TIMED$(M(s, \hat{L}_t), \epsilon, \hat{T}_t)$;
            $t_{end} \leftarrow$ current time
            $\overline{T} \leftarrow \overline{T} - (t_{end} - t_{start})$
        end
        $s \leftarrow$ execute action $\pi^*_{s,\hat{L}_t}(s, \hat{L}_t)$ in $s$
    end
end

function LR$^2$TDP_TIMED(FH$_{s_0}$ MDP $M(s'_0, \hat{L})$, $\epsilon > 0$, timeout $T$)
begin
    $t_{start} \leftarrow$ current time
    $t_{end} \leftarrow$ current time
    foreach $L = 1, \ldots, \hat{L}$ or until time $T$ runs out do
        $\pi \leftarrow$LRTDP$_{FH}(M(s'_0, L), \epsilon, T - (t_{end} - t_{start}))$ // see Algorithm 4.1
        $t_{end} \leftarrow$ current time
        if $Ts_L == \infty$ then $Ts_L \leftarrow 0$
        $Ts_L \leftarrow$ update average with $(t_{end} - t_{start})$
    end
    return $\pi$
end
```

during the initial decision epoch (time step 1), tries to estimate how long solving a state takes, on average, for different lookahead values (lines 16-21, 39-44). In subsequent decision epochs, these estimates will let GOURMAND determine the largest lookahead for which a given state can be solved within a given time constraint. To obtain the estimates, GOURMAND runs a specially instrumented version of LR^{2}TDP (Algorithm 4.1) called LR^{2}TDP_TIMED from the initial state, timing how long LR^{2}TDP takes to solve for lookahead values $L = 1, 2, \ldots$. At some point, the time $T_0 = \frac{T}{H}$ allocated to the first decision epoch runs out. By then, GOURMAND achieves two things. First, it solves s_0 for some lookahead $\hat{L}_0$ (line 19), and can select an action in s_0 according to the optimal policy for $M(s_0, \hat{L}_0)$ (line 31). Second, in the process of solving for lookahead $\hat{L}_0$ it gets estimates Ts_L of the time it takes to solve a state completely for lookaheads $L = 1, 2, \ldots, \hat{L}_0$ (lines 41 - 43). Note that getting these estimates is possible ultimately due to LR^{2}TDP's stopping condition, which lets us know when LR^{2}TDP has (nearly) converged for a given lookahead L and hence how long solving for this lookahead takes. Thus, GOURMAND's entire strategy hinges on this property of LR^{2}TDP.

In each epoch t past the initial one, GOURMAND figures out the largest finite lookahead value L_t for which it *should* be able to solve for the current and all subsequent epochs if it divided the remaining time equally among them, i.e., allocated time $T_t = \frac{\overline{T}}{H-t+1}$ to each (line 22). It does this based on the estimates Ts_L it has obtained previously. Then, GOURMAND decides whether it realistically *may be able to* solve the current decision epoch for an even larger lookahead $\hat{L}_t$ without impacting performance guarantees for future decision epochs, i.e. while ensuring that it can solve them for lookahead L_t. To see the intuition for how GOURMAND can achieve this, observe that since $Ts_{L_t} < T_t$, if GOURMAND solved the current epoch just for lookahead L_t, there would probably be some extra time of approximately $(T_t - Ts_{L_t})$ left. By itself, this extra time chunk does not let GOURMAND solve for a lookahead bigger than L_t. However, if GOURMAND "borrows" similar extra time chunks from *future* decision epochs $t' > t$, solving for a larger lookahead *now* may well be possible. Since there are $(H - t)$ decision epochs after t, the total amount of additional time GOURMAND can gain via such borrowing is $(T_t - Ts_{L_t})(H - t)$. Accordingly, GOURMAND adds $(T_t - Ts_{L_t})(H - t)$ to T_t (line 23) and determines whether it can increase the target lookahead to $L_t + 1$ thanks to the borrowed time. Establishing this may be complicated by the fact that GOURMAND does not necessarily know how long solving for $L_t + 1$ takes, in which case its estimate for Ts_{L_t+1} is ∞ (line 8) . However, both in the case when Ts_{L_t+1} is unknown and in the case when it is known to be less that $\hat{T}_t$, GOURMAND takes the risk and sets the target lookahead $\hat{L}_t$ to $L_t + 1$ (line 25) anyway.

GOURMAND then sets off solving MDP $M(s, \hat{L}_t)$ until it either manages to solve the current state s for lookahead $\hat{L}_t$ or the allocated time $\hat{T}_t$ runs out (line 27). Throughout this operation it has LR^{2}TDP_TIMED measure how long solving s takes for lookaheads $L = 1, 2, \ldots, \hat{L}_t$ and update the running averages Ts_L accordingly (lines 41 - 43).

Although not shown in the pseudocode explicitly, GOURMAND uses many of the optimizations to the basic LR^{2}TDP introduced in GLUTTON (Section 4.4): subsampling the transition function, separating out natural dynamics, and caching. They allow GOURMAND to cope with the previously discussed challenges of many finite-horizon MDPs, such as high-entropy transition functions.

4.6 Experimental Results

In the previous sections, we introduced a succession of four increasingly sophisticated algorithms: LRTDP$_{FH}$, LR^{2}TDP, GLUTTON, and GOURMAND. We have claimed that each of these approaches is superior to its predecessor in one way or another. In this section, we provide an empirical foundation for these claims. We start by showing the advantage of LR^{2}TDP over LRTDP$_{FH}$. Then we study the effects of the individual optimizations added to LR^{2}TDP on the performance of GLUTTON. Last but not least, we compare the performance of GOURMAND to that of GLUTTON and PROST across all IPPC-2011 problems, demonstrating that GOURMAND dominates GLUTTON thanks to its online nature and outmatches PROST thanks to its robustness. C++ implementations of GLUTTON and GOURMAND similar to those used in the experiments are available at `http://www.cs.washington.edu/ai/planning/glutton.html` and `http://www.cs.washington.edu/ai/planning/gourmand.html`, respectively.

4.6.1 Experimental Setup

The results are reported using the setting of IPPC-2011 [86]. At IPPC-2011, the competitors needed to solve 80 problems. The problems came from 8 domains, 10 problems each. Within each domain, problems were numbered 1 through 10, with problem size/difficulty roughly increasing with its number. All problems were reward-maximization finite-horizon MDPs with the horizon of 40. They were described in the RDDL language [85] (see Section 2.20), but translations to the older format, PPDDL, were available and participants could use them instead. The participants had a total of 24 hours of wall clock time to allocate in any way they wished among all the problems. Each participant ran on a separate large instance of Amazon's EC2 node (4 virtual cores on 2 physical cores, 7.5 GB RAM).

The 8 benchmark domains at IPPC-2011 were Sysadmin (abbreviated as Sysadm in some figures in this section), Game of Life (GoL), Traffic, Skill Teaching (Sk T), Recon, Crossing Traffic (Cr Tr), Elevators (Elev), and Navigation (Nav). Sysadmin, Game of Life, and Traffic domains are very large (many with over 2^{50} states). Recon, Skill Teaching, and Elevators are smaller but require a larger planning lookahead to behave near-optimally. Navigation and Crossing Traffic essentially consist of goal-oriented MDPs. The goal states are not explicitly marked as such; instead, they are the only states visiting which yields a reward of 0, whereas the highest reward achievable in all other states is negative.

A planner's solution policy for a problem was assessed by executing the policy it produced 30 times on a special server. Each of the 30 rounds would consist of the server sending the problem's initial state, the planner sending back an action for that state, the server executing the action, noting down the reward, and sending a successor state, and so on. After 40 such exchanges, another round would start. A planner's performance was judged by its average reward over 30 rounds.

In most of the experiments, we show planners' normalized scores on various problems. The normalized score of planner Pl on problem p always lies in the $[0, 1]$ interval and is computed as follows:

$$score_{norm}(Pl, p) = \frac{\max\{0, s_{raw}(Pl, p) - s_{baseline}(p)\}}{\max_i\{s_{raw}(Pl_i, p)\} - s_{baseline}(p)} \tag{4.4}$$

where $s_{raw}(Pl, p)$ is the average reward of the planner's policy for p over 30 rounds, $max_i\{s_{raw}(Pl_i, p)\}$ is the maximum average reward of *any* IPPC-2011 participant on p, and $s_{baseline}(p) = \max\{ s_{raw}(random, p), s_{raw}(noop, p)\}$ is the baseline score, the maximum of expected rewards yielded by repeating the $noop$ action in any state and choosing actions randomly. Roughly, a planner's score is its policy's reward as a fraction of the highest reward of any participant's policy on the given problem.

In the experiments that follow next and illustrate the benefits of reverse iterative deepening and GLUTTON's optimizations, we gave different variants of GLUTTON at most 18 minutes to solve each of the 80 problems (i.e., divided the 24 hours given under the competition conditions equally among all problem instances).

4.6.2 Reverse Iterative Deepening

Running either LR^{2}TDP or LRTDP$_{FH}$ on IPPC-2011 problems is infeasible without the optimizations included in the GLUTTON planner. Therefore, to demonstrate the power of iterative deepening, we built a version of GLUTTON denoted GLUTTON-NO-ID that uses LRTDP$_{FH}$ instead of LR^{2}TDP. A-priori, we may expect two advantages of GLUTTON over GLUTTON-NO-ID. First, according to the intuition in Section 4.3, GLUTTON should have a better anytime performance. That is, if GLUTTON and GLUTTON-NO-ID are interrupted T seconds after starting to solve a problem, GLUTTON's solution should be at least as good as GLUTTON-NO-ID's. Second, GLUTTON should be faster because GLUTTON's trials are on average shorter than GLUTTON-NO-ID's. The length of the latter's trials is initially equal to the horizon, while most of the former's end after only a few steps. Under limited-time conditions such as those of IPPC-2011, both of these advantages should translate to better solution quality for GLUTTON. To verify this prediction, we ran GLUTTON and GLUTTON-NO-ID under IPPC-2011 conditions (i.e., on a large instance of Amazon EC2 with a 24-hour limit) with 18 minutes per problem and calculated their normalized scores on all the problems using Equation 4.4. The scores were averaged for each domain.

Figure 4.1 compares GLUTTON's and GLUTTON-NO-ID's results. On most domains, GLUTTON-NO-ID performs worse than GLUTTON, and on Sysadmin, Elevators, and Recon the difference is very large. This is a direct consequence of the above theoretical predictions. Both GLUTTON-NO-ID and GLUTTON are able to solve small instances on most domains within allocated time. However, on larger instances, both GLUTTON-NO-ID and GLUTTON typically use up all of the allocated time for solving the problem, and both are interrupted while solving. Since GLUTTON-NO-ID has worse anytime performance, its solutions on large problems tend to be worse than GLUTTON's. In fact, the Recon and Traffic domains are so complicated that GLUTTON-NO-ID and GLUTTON are almost always stopped before finishing to solve them. As we show when analyzing cyclic policies in Section 4.6.3, on Traffic both planners end up falling back on such policies, so their scores are the same. However, on Recon cyclic policies do not work very well, causing GLUTTON-NO-ID to fail dramatically due to its poor anytime performance.

4.6.3 Effects of GLUTTON*'s Optimizations*

The role of each of GLUTTON's optimizations is difficult to evaluate precisely, since they often work in concert with each other. As an approximation, to provide an idea of a given optimization's merit, we created a version of GLUTTON with that optimization turned off and pitted it against full-fledged GLUTTON. The results are presented in the next few subsections.

Separating out Natural Dynamics

In particular, GLUTTON-NO-SEP-ND is a version of GLUTTON that does not separate out the natural dynamics of a problem. Namely, when computing the greedy best action for a given state, GLUTTON-NO-SEP-ND samples the transition function of each action independently. For any given problem, the number of generated successor state samples N per state-action pair was the same for GLUTTON and GLUTTON-NO-SEP-ND, but varied slightly from problem to problem around the value of 30. To gauge the performance of GLUTTON-NO-SEP-ND, we ran it on all 80 problems under the IPPC-2011 conditions. We expected GLUTTON-NO-SEP-ND to perform worse overall — without factoring out natural dynamics, sampling successors should become more expensive, so GLUTTON-NO-SEP-ND's progress towards the optimal solution should be slower.

Figure 4.2 compares the performance of GLUTTON and GLUTTON-NO-SEP-ND. As predicted, GLUTTON-NO-SEP-ND's scores are noticeably lower than GLUTTON's. However, we discovered the performance pattern to be richer than that. As it turns out, GLUTTON-NO-SEP-ND solves small problems from small domains (such as Elevators, Skill Teaching, etc.) almost as fast as GLUTTON. This effect is due to the presence of caching. Indeed, sampling the successor function is expensive during the first visit to a state-action pair, but the samples get cached, so on subsequent visits to this pair neither planner incurs any sampling cost. Crucially, on small problems, both GLUTTON and

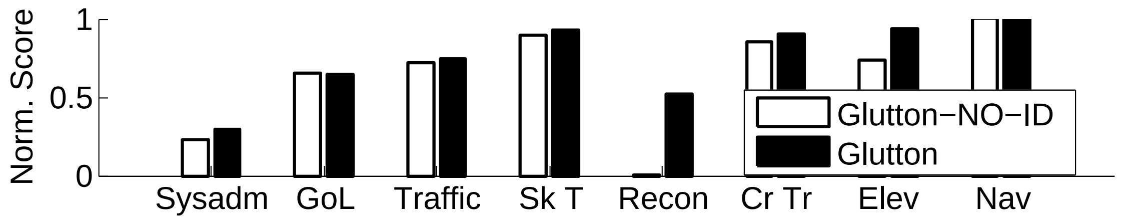


Figure 4.1: Average normalized scores of GLUTTON with and without iterative deepening (denoted as GLUTTON and GLUTTON-NO-ID in the plot, respectively) on all of the IPPC-2011 domains.

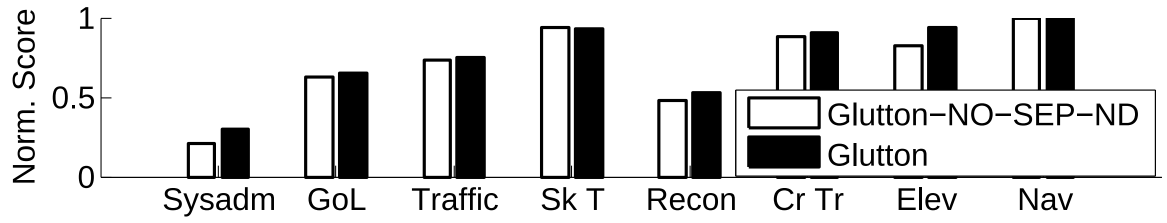


Figure 4.2: Average normalized scores of GLUTTON with and without separation of natural dynamics (denoted as GLUTTON and GLUTTON-NO-SEP-ND in the plot, respectively) on all of the IPPC-2011 domains.

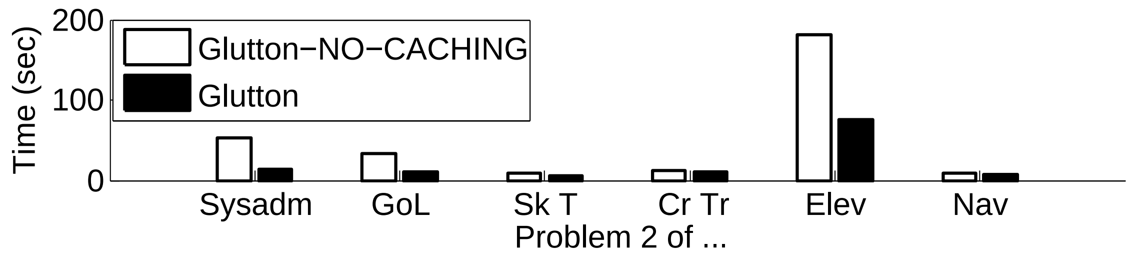


Figure 4.3: Time it took GLUTTON with and without caching to solve problem 2 of six IPPC-2011 domains.

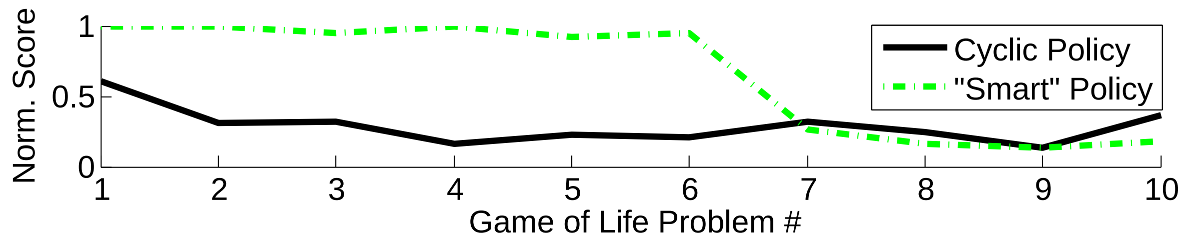


Figure 4.4: Normalized scores of the best primitive cyclic policies and of GLUTTON's "smart" policies on Game of Life.

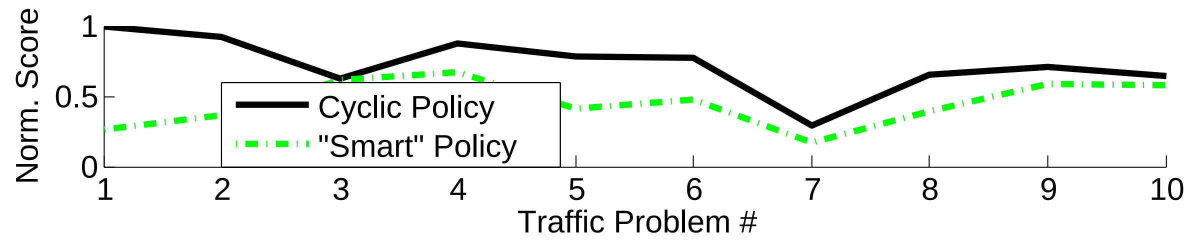


Figure 4.5: Normalized scores of the best primitive cyclic policies and of the "smart" policies produced by GLUTTON on Traffic.

GLUTTON-NO-SEP-ND have enough memory to store the samples for *all* state-action pairs they visit in the cache. Thus, GLUTTON-NO-SEP-ND incurs a higher cost only at the initial visit to a state-action pair, which results in an insignificant speed increase overall.

In fact, although this is not shown explicitly in Figure 4.2, GLUTTON-NO-SEP-ND occasionally performs better than GLUTTON on small problems. This happens because for a given state, GLUTTON-NO-SEP-ND-produced samples for all actions are independent. This is not the case with GLUTTON since these samples are derived from the same set of samples from the *noop* action. Consequently, GLUTTON's samples have more bias, which makes the set of samples somewhat unrepresentative of the actual transition function.

The situation is quite different on larger domains such as Sysadmin. On them, both GLUTTON and GLUTTON-NO-SEP-ND at some point have to start shrinking the cache to make space for the state-value table, and hence may have to resample the transition function for a given state-action pair over and over again. For GLUTTON-NO-SEP-ND, this causes an appreciable performance hit, immediately visible in Figure 4.2 on the Sysadmin domain.

Caching Transition Function Samples

GLUTTON's clone without the caching feature is called GLUTTON-NO-CACHING. GLUTTON-NO-CACHING is so slow that it cannot handle most IPPC-2011 problems. Therefore, to show the effect of caching we run GLUTTON and GLUTTON-NO-CACHING on instance 2 of six IPPC-2011 domains (all domains but Traffic and Recon, whose problem 1 is already very hard), and record the amount of time it takes them to solve these instances. Instance 2 was chosen because it is harder than instance 1 and yet is easy enough that GLUTTON can solve it fairly quickly on all six domains both with and without caching.

As Figure 4.3 shows, even on problem 2 the speed-up due to caching is significant, reaching about $2.5\times$ on the larger domains such as Game of Life, i.e., where it is most needed. On domains with big branching factors, e.g., Recon, caching makes the difference between success and utter failure.

Cyclic Policies

The cyclic policies evaluated by GLUTTON are seemingly so simple that it is hard to believe they ever beat the policy produced after several minutes of GLUTTON's "honest" planning. Indeed, on most problems GLUTTON does not resort to them. Nonetheless, they turn out to be useful on a surprising number of problems. Consider, for instance, Figures 4.4 and 4.5. They compare the normalized scores of GLUTTON's "smart" policy produced at IPPC-2011, and the best primitive cyclic policy across various problems from these domains.

On Game of Life (Figure 4.4), GLUTTON's "smart" policies for the easier instances clearly win. At the same time, notice that as the problem size increases, the quality of cyclic policies nears and eventually exceeds that of the "smart" policies. This happens because the increase in difficulty of problems within the domain is not accompanied by a commensurate increase in time allocated for solving them. Therefore, the quality of the "smart" policy GLUTTON can come up with within allocated time keeps dropping, as seen on Figure 4.4. Granted, on Game of Life the quality of cyclic policies is also not very high, although it still helps GLUTTON score higher than 0 on all the problems. However, the Traffic domain (Figure 4.5) proves that even primitive cyclic policies can be very powerful. On this domain, they dominate anything GLUTTON can come up with on its own, and approach in quality the policies of PROST, the winner on this set of problems. It is due to them that GLUTTON performed reasonably well at IPPC-2011 on Traffic. Whether the success of primitive cyclic policies is particular to the structure of IPPC-2011 scenarios or generalizes beyond them is a topic for future research.

4.6.4 UCT vs. LRTDP

The objective of our last set of experiments was to compare the performance of online LRTDP as used in GOURMAND to that of online UCT as used in PROST and offline LRTDP as used in GLUTTON across a diverse collection of finite-horizon MDPs, and to analyze patterns in their behavior. To this end, we present these planners' results on all IPPC-2011 problems but, unlike in previously discussed experiments, pay special attention to intra-domain performance trends.

As in previous experiments, each planner ran for 24 hours on an Large Instance of Amazon EC2 server, but instead of allocating an equal amount of time to all problems, they used a more

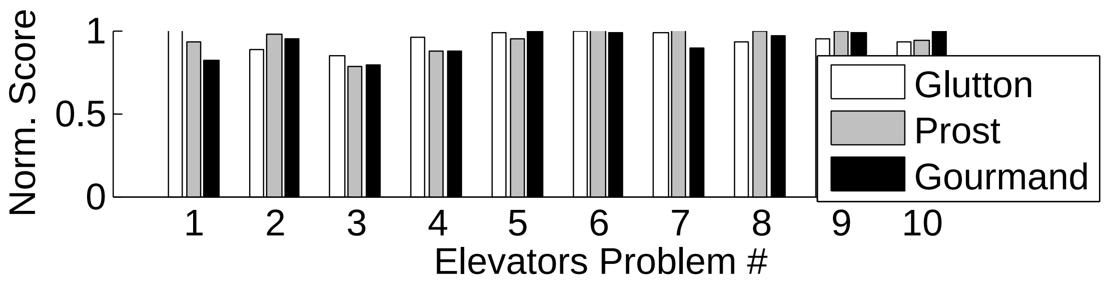


Figure 4.6: All planners are tied on the Elevators domain. Planners' normalized scores are computed as in Equation 4.4.

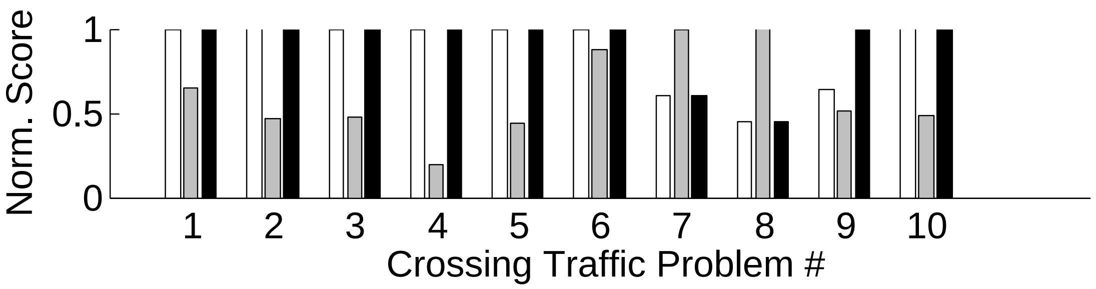


Figure 4.7: GOURMAND (avg. score 0.9052) and GLUTTON vastly outperform PROST (0.6099) on the Crossing Traffic domain.

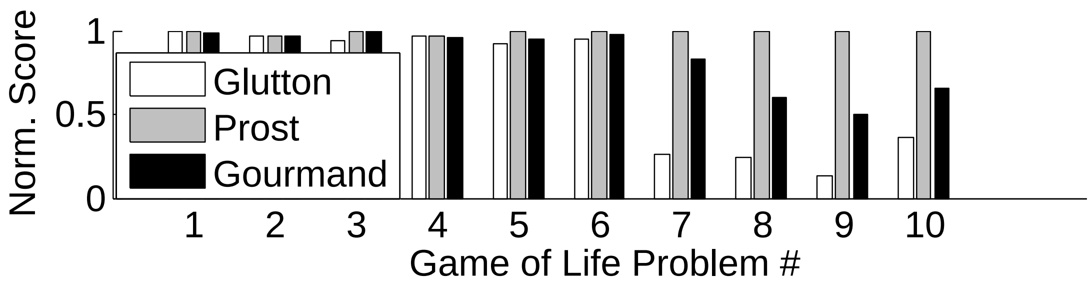


Figure 4.8: PROST (avg. score 0.9934) outperforms GOURMAND (0.8438) on the Game of Life domain.

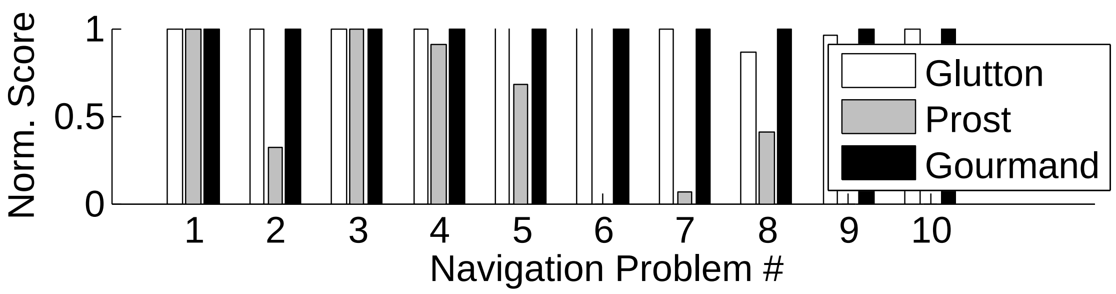


Figure 4.9: GOURMAND (avg. score 1.0) and GLUTTON vastly outperform PROST (0.4371) on the Navigation domain.

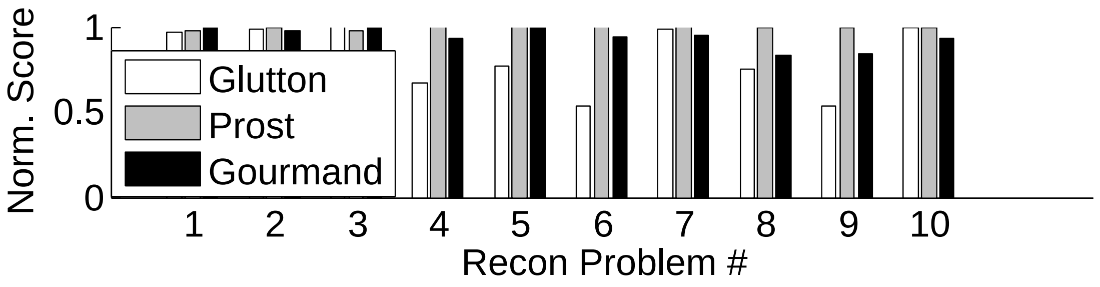


Figure 4.10: GOURMAND and PROST are tied on the Recon domain. Planners' normalized scores are computed as in Equation 4.4

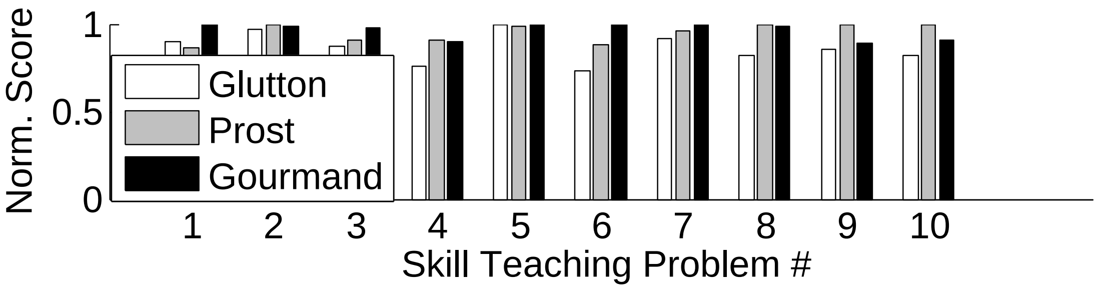


Figure 4.11: GOURMAND and PROST are tied on the Skill Teaching domain.

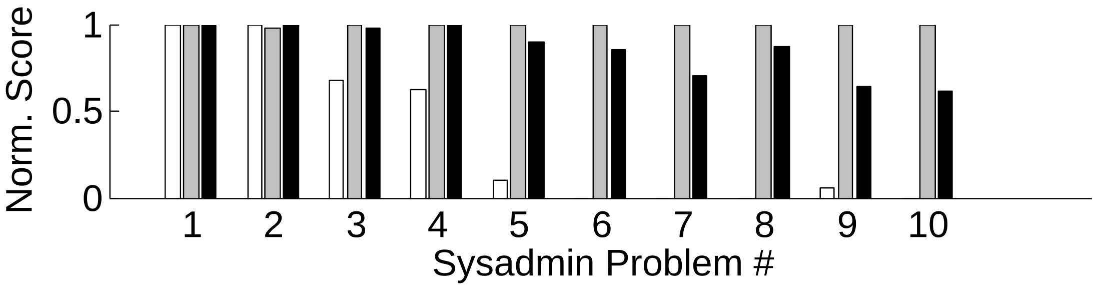


Figure 4.12: PROST (avg. score 0.9978) outperforms GOURMAND (0.8561) on the Sysadmin domain.

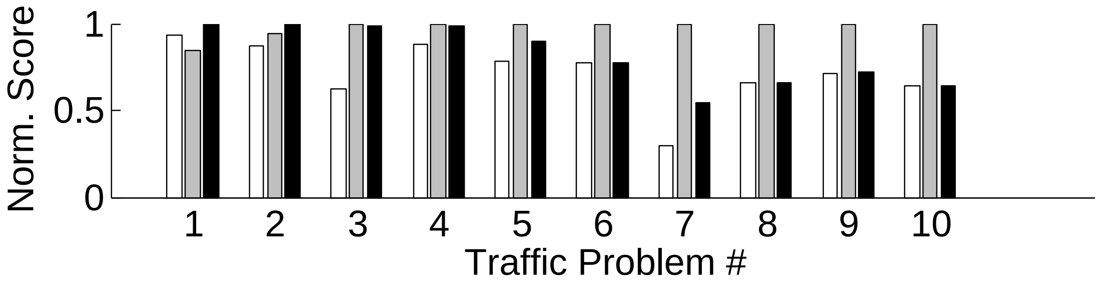


Figure 4.13: PROST (avg. score 0.9791) outperforms GOURMAND (0.8216) on the Traffic domain.

sophisticated strategy. Its high-level idea was to solve easy problems first and devote more time to harder ones. To do so, GLUTTON and GOURMAND first solved problem 1 from each domain. Then they kept redistributing the remaining time equally among the remaining problems and picking the next problem from the domain whose instances on average had been the fastest to solve. As a result, the hardest problems got 40-50 minutes of planning. This strategy was feasible for them because LR^{2}TDP's convergence criterion indicated when a given problem was solved and the next one could be tackled. The IPPC-2011 winner, PROST, did not have this luxury because it was using UCT, but was able to use a similar time allocation strategy thanks to several preset parameters. In particular, PROST's authors specified a single tuned lookahead value, $L = 15$, for all of the benchmark problems. PROST also had a per-epoch timeout. To compute a policy, in each decision epoch PROST would run UCT with $L = 15$ for the time specified by the timeout, return the best action (according to the value function upon termination) to the server, which simulated the action and sent PROST a new state. If UCT happened to re-visit a state across the 30 policy execution attempts, it returned an action for it immediately, without waiting for the timeout. In this case, the freed-up time was redistributed among subsequent epochs. Because of this, PROST could also execute all 30 rounds before the time allocated to this problem was up. When this happened, the remaining time was distributed among the remaining problems. The ultimate effect was similar to the case of GLUTTON and GOURMAND: larger problems got more time.

The overall results for each planner and domain are presented in Figures 4.7 - 4.13. Across all domains, GOURMAND earned the average score of 0.9183 ± 0.0222, PROST— 0.8608 ± 0.0220, and GLUTTON— 0.7701 ± 0.0235, i.e., GOURMAND outperformed PROST by a statistically significant amount, and PROST, in turn, did noticeably better than GLUTTON.

Several performance patterns deserve a special note. First, we revisit the intuition we stated at the beginning that if the value of L chosen for UCT is too large, by the timeout UCT will still be very far from convergence and pick an action largely at random, whereas online LRTDP will converge completely for a smaller lookahead and make a more informed decision. While it may be true for vanilla UCT implementations without a heuristic, PROST's performance on the IPPC-2011 benchmarks does not confirm this. In particular, consider the Sysadmin, Game of Life, and Traffic domains. All of them require a very small lookahead, typically up to 8, to come up with a near-optimal policy. Moreover, they have extremely large branching factors (around 2^{50} for some states of the largest Sysadmin instances). Since UCT used $L = 15$, one might expect it to make hardly any progress due to the enormous number of extra states it has to explore. Nonetheless, PROST wins on these domains overall, despite the fact that on many instances GOURMAND routinely solves states for $L = 6$. We hypothesize that on these problems, UCT may be arriving at a good policy much sooner than its value function converges, possibly due to the help of the heuristic with which PROST endows it. The fact that UCT does not need to perform Bellman backups, which are expensive in MDPs with large branching factors as in these domains, probably also contributed to its convergence speed. Nonetheless, more experimentation is needed for a more conclusive explanation.

Second, the figures show GOURMAND's performance to be more uniform than PROST's. The lookahead parameter for PROST was empirically picked to give good results across many of the competition domains (competition rules allowed this). Indeed, PROST performed very well on average and even outperformed GOURMAND on three domains above. Yet, due to its adaptive strategy, online LRTDP implemented by GOURMAND does not suffer sharp drops in performance on some problems sets as UCT implemented by PROST does, and is robust across all benchmark domains.

In fact, UCT's overall defeat was caused by very poor performance on two domains, Navigation (Figure 4.9) and Crossing Traffic (Figure 4.7). Incidentally, both of them are in effect goal-oriented domains — the agent incurs a cost for every decision epoch it is not in one of the special states staying in which is "free". Crucially, to reach these states successfully, one needs to select actions very carefully during the first decision epochs of the process. For instance, in Crossing Traffic, the agent is trying to cross a motorway. It can do this either safely, by making detours, or by boldly dashing across the moving stream of cars, which can kill the agent. Getting to the other side via detours takes longer, and the agent has to plan with a sufficient lookahead during the first few decision epochs of the process to realize that this is the safer course of action. This highlights the main drawback of guessing a value for L — even within the same domain, $L = 15$ is sufficient for some problems but not others, leading to catastrophic consequences for the agent in the latter case. Online and offline LR^{2}TDP, as used in GOURMAND and GLUTTON respectively, eventually arrives at a sufficiently large lookahead thanks to reverse iterative deepening and solves many such problems successfully.

In comparison to GLUTTON, both GOURMAND and PROST demonstrate a pronounced overall advantage. Since GLUTTON attempts to solve problems offline, by the timeout it often fails to visit many states that its policy can visit from the initial state. This is not very noticeable on domains with relatively small problems, such as Navigation, Recon, Skill Teaching, and Elevators, which can usually be solved nearly completely by the timeout even in the offline mode. However, on very large problems of Game of Life, Sysadmin, and Traffic, offline planning does not pay off: by the timeout, GLUTTON's policy on them is far from being closed with respect to s_0. In order to compensate for this, during policy execution GLUTTON uses the default policies, which in many cases do not help enough. To exacerbate the situation, GLUTTON spends considerable effort on states it never encounters during the evaluation rounds. This is precisely the weakness of offline planning in situations when the produced policy needs to be executed only a few times. Indeed, since each IPPC-2011 problem has horizon 40 and needs to be attempted 30 times during evaluation, the number of distinct states for which performance "really matters" is at most $30 \cdot 39 + 1 = 1171$ (the initial state is encountered 30 times). The number of states GLUTTON visits and tries to learn a policy for during training is typically many orders of magnitude larger. Thus, GLUTTON's main undoing is resource misallocation. GOURMAND and PROST do not suffer from this issue to such a significant degree, because by planning online they always make an informed choice of action in states that they visit.

Last but not least, we point out that the presence of a termination condition in LRTDP can give rise to many adaptive time allocation strategies, of which GOURMAND exploits only one. Our objective in designing and evaluating GOURMAND was not to pick the best such strategy. Rather, it was to demonstrate that at least some of them can yield an online planner that is significantly more robust, performant, and easier to deploy than UCT. GOURMAND's results on IPPC-2011 domains showcase this message.

4.7 Related Work

Several techniques similar to subsampling and separating natural dynamics have been proposed in the reinforcement learning and concurrent MDP literature, e.g., [79] and [72], respectively. An alternative way of increasing the efficiency of Bellman backups is performing them on a symbolic value function representation, e.g., as in symbolic RTDP [30]. A great improvement over Bellman backups with explicitly enumerated successors, it nonetheless does not scale to many IPPC-2011 problems.

There has not been much literature on analyzing the performance of UCT in solving MDPs with known transition and reward functions, other than the aforementioned publications on GLUTTON [54], GOURMAND [59], and PROST [49], and work on a modification of UCT called BRUE [28]. The latter points out that the basic UCT tries to optimize cumulative regret, whereas in probabilistic planning with a known model minimizing simple regret is more appropriate. BRUE, a version of UCT for minimizing simple regret, has superior convergence properties compared to UCT but still provides no principled mechanism for detecting convergence. There have also been attempts to examine UCT's properties in the context of adversarial planning [82].

A promising class of approaches for solving finite-horizon MDPs offline that could potentially have much lower resource consumption than any existing techniques is automatic dimensionality reduction [19, 58]). We have showed its power on goal-oriented MDPs in Chapter 3, but compactifying the value function of finite-horizon MDPs appears to follow different intuitions. We are not aware of any such algorithms for this type of problems. The methods for solving finite-horizon MDPs online have been studied fairly little.

4.8 Future Research Directions

The AI research on efficient methods for solving large finite-horizon MDPs is still in its infancy. Although the use of UCT, LRTDP, and, more recently, of an improved version of AO* [15] has provided considerable insights into this type of problems, none of these approaches are fully satisfactory in their current form.

In particular, none of the existing methods sufficiently exploit the structure of factored finite-horizon MDPs. In Chapter 3, we have demonstrated that for goal-oriented scenarios, extracting and using their latent structure can yield fully automatic speedy planning techniques with a small memory footprint. However, these algorithms rely on the presence of non-trivial goal states and therefore are not applicable to finite-horizon MDPs. Dimensionality reduction methods created for (similarly goal-less) infinite-horizon discounted-reward MDPs [39] can probably be extended to finite horizons, but require hand-crafted basis functions and thus are not human-independent. The discovery of autonomous dimensionality reduction algorithms for factored FH MDPs would mark an important research advance for this problem type.

An orthogonal research direction concerns combining the advantages of model-free and dynamic programming-based state value updates in MDPs with high-entropy transition functions. Note that difficulties with such transition functions, which, as a reminder, model the effects of exogenous events, are not particular to finite-horizon problems. Exogenous events are present in many goal-oriented scenarios too, and ways of dealing with them in those settings are equally poorly studied. GLUTTON's and GOURMAND's subsampling of the transition function as a way of adapting Bellman backups to large numbers of possible state transitions is not ideal, because it breaks the theoretical guarantees inherent to methods such as VI and LRTDP. On the other hand, completely ignoring the explicitly given state transition probabilities and relying purely on Monte Carlo simulations, as UCT does, is wasteful. Moreover, UCT gives only asymptotic quality guarantees, which cannot be used to detect convergence reliably. This is a significant drawback, as our experimental comparison of GOURMAND and PROST has shown. It appears that further progress in efficient methods for solving FH MDPs hinges on inventing a new state update mechanism that would perform well in the face of high-entropy transition functions and at the same time would allow for implementing a principled convergence check.

Besides algorithms based on iteratively improving the value function or policy of an MDP, there is also a fundamentally different solution strategy. It consists in *guessing* a general parametrized form of a good policy for the problem at hand and choosing its parameter values appropriately. This approach has successfully tackled MDPs with infinite state or action spaces [80], where more conventional algorithms are simply inapplicable. An example of a general class of policies that work well in many scenarios is the class of cyclic policies. Intuitively, cyclic policies repeat the same sequence of actions over and over again. In the scenario discussed earlier in this chapter involving a robotic arm that transfers objects from one conveyor belt to another, a cyclic policy is optimal. Unfortunately, it is not clear how to easily compute a reasonable cyclic policy automatically or how to verify that policies of this type are suitable for a given problem. However, in light of success of even very primitive cyclic policies on some benchmark MDP domains (e.g., Traffic), research in this area can potentially yield a high payoff.

4.9 Summary

Although any finite-horizon MDP can technically be converted to a goal-oriented one, for a variety of reasons the approximate determinization-based dimensionality reduction techniques of the previous chapter do not apply to the finite-horizon case. This prompts the question: how do we solve these goal-less MDPs efficiently? The lack of meaningful goal states in FH MDPs turns out to defeat many other existing approaches as well, leaving us only with the most basic techniques for this problem class, e.g., VI.

To fill this gap, we have introduced three algorithms for handling large FH_{s_0} MDPs. The first of them, LR^2TDP, is an adaptation of LRTDP to FH_{s_0} MDPs based on reverse iterative deepening. The strategy gives LR^2TDP anytime performance superior to that of the more straightforward LRTDP version for finite-horizon problems. However, by itself it does not allow LR^2TDP to scale to FH_{s_0} MDPs with complex characteristics, e.g., with high-entropy transition functions. In the meantime, high-entropy transition functions arise in many scenarios as a way to model exogenous events or other components of a scenario's natural dynamics. To allow LR^2TDP to cope with complicated FH_{s_0} problems, we equip it with a number of optimizations and implement them in the second system described in this chapter, GLUTTON.

GLUTTON participated in the IPPC-2011 competition and performed well, but was topped by a UCT-based solver, PROST. UCT is a powerful technique naturally suited to dealing with high-entropy transition functions. All participants of IPPC-2011 other than GLUTTON were based on it. The fact that GLUTTON lost to one of them but outperformed the other three made us examine the advantages and drawbacks of LR^2TDP that served as GLUTTON's foundation when compared to UCT. It turns out that LR^2TDP has at least one practically important feature than UCT lacks — a convergence criterion. If, unlike in GLUTTON, LR^2TDP is used in the online mode, its convergence criterion makes it more adaptable and easier to tune than online UCT. To test this observation, we have introduced a third planner for FH_{s_0} MDPs, GOURMAND, which is built around an online version of LR^2TDP. An extensive experimental comparison shows that online LR^2TDP indeed performs better overall than online UCT, with LR^2TDP's advantage being largely due to its convergence criterion.

In spite of UCT's and LR^2TDP's successes, the current state of the art in solving large finite-horizon MDPs leaves a lot of room for improvement. The invention of automatic dimensionality reduction methods for this class of problems, along with a more flexible state value update rule and mechanisms for constructing good cyclic policies for such scenarios, will result in vastly more capable FH MDP algorithms.

Chapter 5
BEYOND STOCHASTIC SHORTEST-PATH MDPS

While the scalability of the available solution techniques is an important factor that determines the usefulness of MDPs as a modeling tool, it is not the only one. Many scenarios whose state and action space sizes are well within the scalability limits of modern planning algorithms cannot be solved only because they do not fit the assumptions of any known MDP class. For instance, there is only one widely accepted MDP type that can model goal-oriented settings, the stochastic shortest path MDPs (Definitions 2.16 and 2.19). It comes with two restrictions:

- SSP MDPs must have a complete proper policy, one that can reach the goal from any state with probability 1.

- Every improper policy must incur an infinite cost from any state from which it has a positive probability of never reaching the goal.

Many scenarios with very natural characteristics violate one of both of these clauses. The first restriction essentially confines SSP MDPs to problems with no catastrophic events that could prevent an agent from reaching the goal. Such catastrophic events are a possibility to be reckoned with in many settings, e.g., robotics, and ignoring them is sometimes completely unacceptable. Moreover even if a given goal-oriented MDP has no dead ends, verifying this fact can be nontrivial, which further complicates the use of the SSP model. The requirement of improper policies accumulating infinite cost if they do not reach the goal forbids the situations in which an agent is interested in *maximizing the probability* of reaching the goal, as opposed to minimizing the expected cost of doing so. These settings could be modeled by assigning the cost of 0 to each action and the reward of 1 for reaching the goal. However, under this reward function, policies that never lead to the goal have a cost of 0, which is unacceptable according to the SSP MDP definition. Researchers have tried to develop planning formulations where reasoning about dead ends is possible (e.g., the aforementioned criterion of maximizing the probability of reaching the goal), but the models proposed to date do not cover many interesting cases, and their mathematical properties are still understood relatively poorly.

5.1 Overview

The final contribution of this dissertation is a set of SSP MDP extensions that remove this model's constraints, along with optimal algorithms for solving them. These new MDP classes show how far the SSP formalism can be extended without turning goal-oriented MDPs into a theoretically interesting but computationally infeasible construct. Their exposition is organized as follows:

- We begin with **g**eneralized **SSP** MDPs (*$GSSP_{s_0}$*) [61], a class that allows a more general action reward model than SSP. Although it does not admit completely unrestricted action rewards (an issue we rectify later in the chapter), it properly contains several notable classes of infinite-horizon problems. We define the semantics of optimal solutions for $GSSP_{s_0}$ problems and propose a new heuristic search framework for them, called FRET (**F**ind, **R**evise, **E**liminate **T**raps). We also show that the previously discussed scenarios where an agent wants to maximize the probability of reaching the goal form a subclass of *$GSSP_{s_0}$* that we call *MAXPROB*. Since *MAXPROB* is contained in *$GSSP_{s_0}$*, FRET can solve it as well and is, to our knowledge, the first heuristic search framework to do so. Our investigation of *MAXPROB*'s mathematical properties completes with a derivation of a VI-like algorithm that can solve MAXPROB MDPs *independently of initialization* (previously, VI was known to yield optimal solutions to MAXPROB only if initialized strictly inadmissibly [80]).

- Although *MAXPROB* forms the basis for our theory of goal-oriented MDPs with dead ends, by itself it evaluates policies in a rather crude manner, completely disregarding their *cost*. Our next *SSP* extension, one that takes costs into account as well, is **SSP** MDPs with **a**voidable **d**ead **e**nds (*$SSPADE_{s_0}$*) [60]. $SSPADE_{s_0}$ MDPs always include an initial state and have well-defined easily computable optimal solutions if dead ends are present but avoidable from it. Besides defining *$SSPADE_{s_0}$*, we describe the modifications required for the existing FIND-AND-REVISE algorithms to work correctly on these problems.

- Next, we introduce cost-aware classes of MDPs with dead ends that admit that dead ends may exist and the probability of running into them from the initial state may be positive no matter how hard the agent tries. Mathematically, there are two ways of dealing with such situations. The first is to assume that entering a dead end, while highly undesirable, carries only a finite penalty. This is the approach we take in **SSP** MDPs with **u**navoidable **d**ead **e**nds and a **f**inite penalty (*fSSPUDE*$_{s_0}$) [60]. As with *SSPADE*$_{s_0}$, we show that existing heuristic search algorithms need only slight adjustments to work with *fSSPUDE*$_{s_0}$.

- The other way of treating dead ends is to view them not only as unavoidable but also as extorting an infinitely high cost if an agent hits one. We model such scenarios with **SSP** MDPs with **u**navoidable **d**ead **e**nds and an **i**nfinite penalty (*iSSPUDE*$_{s_0}$) [60, 96]. Conceptually, iSSPUDE$_{s_0}$ MDPs represent the most difficult settings: since every policy in them reaches an infinite-cost state from s_0, the expected cost of any policy at s_0 is also infinite. This makes SSP's cost-minimization criterion uninformative. The most recent attempt to take both policies' goal probability and cost into account assumed these criteria to be independent, and therefore constructed a Pareto set of non-dominated policies as a solution to this optimization problem [18]. Computing such a set is generally intractable. Instead, we claim that a natural *primary* objective for scenarios with unavoidable infinitely costly dead ends is to maximize the probability of getting to the goal (i.e., to minimize the chance of getting into a lethal accident, a dead-end state). However, of all policies maximizing this chance we would prefer those that reach the goal in the least costly way (in expectation). This is exactly the multiobjective criterion we propose for *iSSPUDE*$_{s_0}$. Solving *iSSPUDE*$_{s_0}$ is conceptually much more involved than handling the *SSP* extensions above, and in this chapter we present an optimal tractable algorithm for it.

- In conclusion of our work on SSP extensions, we present **s**tochastic **s**imple **l**ongest **p**ath MDPs (*SSLP*$_{s_0}$). Their definition imposes no restrictions whatsoever on action costs/rewards or the existence of proper policies. Because of this, *SSLP*$_{s_0}$ includes all of the aforementioned SSP extensions as special cases. When defining the notion of an optimal policy for *SSLP*$_{s_0}$, we explicitly concentrate only on Markovian policies, since no general policy for them may dominate all others in terms of expected reward. Discovering the best Markovian policy for an

SSLP$_{s_0}$ MDP is a probabilistic counterpart of computing a simple longest path between two nodes in a graph, known to be NP-hard in the size of the state space [89] (in fact, this deterministic problem is a special case of the probabilistic one). Thus, the most efficient algorithms for SSLP$_{s_0}$ MDPs are exponential in SSLP$_{s_0}$ problems' flat representation unless $P = NP$. Nonetheless, *SSLP*$_{s_0}$ has a theoretical value of showing that removing *all* restrictions from the *SSP* definition can be a liability.

The mathematical treatment of goal-oriented MDPs with dead ends complements the methods introduced in Chapter 3 for identifying dead ends in factored planning problems in practice. Namely, recall that the SIXTHSENSE algorithm described in Section 3.6 provides MDP solvers with a mechanism that recognizes dead-end states quickly and efficiently. As the experiments we present in this chapter demonstrate, SIXTHSENSE greatly benefits the algorithms for the MDP classes above, serving as a source of informative heuristic values for them. Thus, our work provides a complete set of tools for problems with dead ends: fundamental optimal algorithms for solving them as well as a powerful optimization in the form of SIXTHSENSE to make these algorithms' implementations efficient.

5.2 Preliminaries

All the background material relevant to understanding this chapter has been covered previously, but we introduce several new pieces of notation to facilitate the explanations.

5.2.1 Notation

In Section 2.2.3, we described Bellman backup as a procedure that updates the value of a state by applying Equation 2.9 (or its analogue for IHDR or FH MDPs) to it. In this chapter and its theorems' proofs contained in the Appendix, we view Bellman backup as an operator on the set of all value functions of an MDP. Namely, fix an ordering on the set $\mathcal{S}$ of an MDP's states. A value function V of this MDP can be regarded as a vector whose j-th component is the value of the j-th state of $\mathcal{S}$ in the chosen ordering. In this sense, the set of all value functions $\mathcal{V}$ of an MDP forms a Banach space [80]. By Bellman backup we will mean the application of Equation 2.9 or its analogue for the MDP class at hand to some $V \in \mathcal{V}$. We will distinguish between two kinds of Bellman backup. The *local* Bellman backup operator applies to the value of a single state s, leaving the values of other untouched, and will be denoted as

$$\mathscr{B}_{(s)} : \mathcal{V} \to \mathcal{V}. \tag{5.1}$$

The *full* Bellman backup applies at *all* states $s \in \mathcal{S}$ simultaneously, and will be denoted as

$$\mathscr{B} : \mathcal{V} \to \mathcal{V}. \tag{5.2}$$

Viewed alternatively, full Bellman backup is equivalent to local Bellman backups synchronously applied at all states, as in standard VI.

We will be discussing Bellman backup operators and VI algorithms based on them for several MDP classes, and will distinguish between these different versions by denoting the Bellman backup and VI version for MDP class X as $\mathscr{B}_X$ (or $\mathscr{B}_{X(s)}$) and VI$_X$, respectively. E.g., the VI for SSP MDPs in Section 2.2.3 will be denoted as VI$_{SSP}$; this algorithm uses full Bellman backup $\mathscr{B}_{SSP}$.

When discussing *GSSP*$_{s_0}$ and *SSLP*$_{s_0}$, we will be working with reward functions and hence in the reward-maximization setting to emphasize that actions' rewards in these classes do not have to be primarily negative, as is SSP MDPs. On the other hand, for MDPs with dead ends we will take a cost-based view, since the cost function in these problems must obey similar requirements as for SSP.

A lot of the material in the chapter will focus on the existence and properties of optimal stationary deterministic Markovian policies for various MDP types. We will use the term "stationary deterministic Markovian" often in the process and therefore abbreviate it to *s.d.M.* for conciseness.

5.3 Generalized SSP MDPs: Enabling Zero-Reward Cycles

In the introduction to this chapter, we briefly mentioned problems in which the criterion of interest is maximizing the probability of reaching the goal from another state. For instance, consider planning the process of powering down a nuclear reactor. In this setting, actual action costs (insertion of control rods, adjusting coolant level, penalty for unsuccessful shutdown, etc.) required for performing cost optimization are difficult to obtain. Optimizing for the successful shutdown probability obviates the need for them and is therefore a more advantageous approach. As we already discussed, to formulate a probability-maximization MDP it is enough to set the action costs to 0 and let the reward for reaching the goal be 1 (or any other positive number). The main reason why the resulting MDP does not belong to *SSP* is that its transition graph contains goal-free "loops" of zero-cost actions (e.g., "insert control rods", "raise control rods"). Another example of an MDP with such zero-cost loops (or, more generally, strongly connected components) is shown in Figure 5.1, where the loops are formed by pairs of actions $\{s_1, s_2\}$ and $\{s_3, s_4\}$. These zero-cost regions of the state space allow an agent to stay there without paying anything and without ever going to the goal — a situation disallowed by the SSP MDP definition.

In this section, we define generalized stochastic shortest-path (GSSP$_{s_0}$) MDPs, a class of problems that relaxes *SSP*'s restrictions on action rewards and admits problems with zero-cost regions in their transition graph. While seemingly a small extension of *SSP*$_{s_0}$, *GSSP*$_{s_0}$ has vastly more complicated mathematical properties. They will force us to develop a new heuristic search framework, FRET, for this MDP class. We will also show that *GSSP*$_{s_0}$ contains several notable classes of MDPs (see Figure 1.2): *SSP*$_{s_0}$ itself, positive-bounded MDPs (*POSB*), and negative MDPs (NEG) [80]. The theory of GSSP$_{s_0}$ MDPs will be a stepping stone to our analysis of planning in the presence of dead ends in later sections.

5.3.1 Definition

Definition 5.1. ***GSSP*$_{s_0}$ *MDP.*** *A generalized stochastic shortest path (GSSP$_{s_0}$) MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{G}, s_0 \rangle$, where $\mathcal{S}$, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{R} = -\mathcal{C}$, $\mathcal{G}$, and s_0 are as in the SSP$_{s_0}$ MDP definition (2.19), under the following conditions:*

1. *There exists a policy π that is proper w.r.t. s_0.*

2. *The expected sum of* nonnegative *rewards yielded by any policy π, denoted as V_+^π, is bounded*

from above starting from any state s, time step t and execution history $h_{s,t}$, i.e.,

$$V_+^\pi(h_{s,t}) = \mathbb{E}\left[\sum_{t'=0}^{\infty} \max\{0, R_{t'+t}^{\pi_{h_{s,t}}}\}\right] < \infty. \tag{5.3}$$

♣

The objective is to find a reward-maximizing policy $\pi^*_{s_0}$ that reaches the goal from s_0 with probability 1, i.e., [1]

$$\pi^*_{s_0} = \underset{\pi \text{ } proper \text{ } w.r.t. \text{ } s_0}{\text{argsup}} V^\pi(s_0) \tag{5.4}$$

As before, we denote the value function of an optimal policy as V^*. By the definition of $\pi^*_{s_0}$, this value function satisfies

$$V^*(s_0) = \sup_{\pi \text{ } proper \text{ } w.r.t. \text{ } s_0} V^\pi(s_0) \tag{5.5}$$

The *GSSP*$_{s_0}$ definition has two notable aspects. The first of them is the definition's condition 2 that bounds the expected nonnegative reward of all policies. Without it, GSSP$_{s_0}$ MDPs could have artifacts such as two states s and s' with an action a s.t. $\mathcal{T}(s,a,s') = 1, \mathcal{T}(s',a,s), \mathcal{R}(s,a,s') = 1$, and $\mathcal{R}(s',a,s) = -1$. Effectively, s and s' form a cycle with alternating transition costs. For any policy π that uses a both in s and in s', $V^\pi(s)$ and $V^\pi(s')$ would be ill-defined, since the expected reward series starting at these states would fail to converge. Condition 2 makes such cycles impossible, because for policies like π, $V_+^\pi(s) = V_+^\pi(s') = \infty$.

The second important subtlety concerns Equation 5.4. When selecting the optimal policy, this equation considers only *proper* policies. In contrast, when computing optimal policies for SSP MDPs, both in the Optimality Principle (Theorem 2.3) and in the VI algorithm (Algorithm 2.3) we

[1]The "argsup" expression in Equation 5.4 is well-defined, because, as a consequence of Theorem 5.1, there exists a policy whose value function is $\sup_{\pi \text{ } proper \text{ } w.r.t. \text{ } s_0} V^\pi(s_0)$.

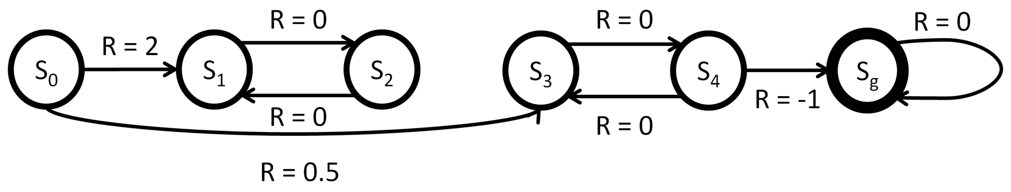


Figure 5.1: An example GSSP$_{s_0}$ MDP presenting multiple challenges for computing the optimal value function and policy efficiently. State s_g is the goal.

considered *all* existing policies. Why do we need to explicitly restrict ourselves to proper policies as candidate solutions in the case of *GSSP*$_{s_0}$? The simple answer is that in SSP MDPs, a reward-maximizing policy is *always* proper, whereas in GSSP$_{s_0}$ MDPs this may not be so. Intuitively, since SSP MDPs disallow zero- and positive-reward cycles, the faster the agent reaches a goal state, the less cost it will incur, i.e., going for the goal is the best thing to do in an SSP problem. In GSSP$_{s_0}$ scenarios, zero-reward cycles *are* possible. As a consequence, if reaching the goal requires incurring a cost but a zero-reward cycle is available, the reward-optimal course of action for the agent is to stay in the zero-reward cycle. However, semantically, we want the agent to go for the goal. Considering only proper policies during optimization, as Equation 5.4 requires, enforces these semantics.

One may ask whether it is natural in a decision-theoretic framework to prefer a policy that reaches the goal over one that maximizes reward, as the presented semantics may do. Note, however, that this ostensible clash of optimization criteria in GSSP$_{s_0}$ MDPs can always be avoided: it is intuitively clear and can be shown formally that if attaining the goal in a GSSP$_{s_0}$ MDP has a sufficiently high reward, the best goal-striving policy will also be reward-maximizing. That is, in this case, both optimization criteria will yield the same solution. At the same time, determining the "equalizing" goal reward value can be difficult, and *GSSP*$_{s_0}$ removes the need for doing this. To sum up, the traditional decision-theoretic solution semantics and the semantics of GSSP$_{s_0}$ MDP solutions are largely equivalent, but the latter makes the modeling process simpler by needing fewer parameters in the model.

As another consequence of the *GSSP*$_{s_0}$ optimal solution definition, for any state s from which no policy reaches the goal with probability 1, $V^*(s) = -\infty$. This follows from Equation 5.5, because for such states, $\sup_{\pi \text{ } proper \text{ } w.r.t. \text{ } s_0} V^\pi(s) = \sup_\emptyset V^\pi(s) = -\infty$. Moreover, no such state is reachable from s_0 by any policy proper w.r.t. s_0.

5.3.2 Mathematical Properties

GSSP$_{s_0}$ problems have the following mathematical properties, illustrated by the MDP in Figure 5.1, some of which drastically change the behavior of the known SSP$_{s_0}$ MDP solution techniques.

Theorem 5.1. ***The Optimality Principle for Generalized SSP MDPs.*** *For a GSSP$_{s_0}$ MDP, define $V^\pi(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{\infty} R_{t'+t}^{\pi_{h_{s,t}}}]$ for any state s, time step t, execution history $h_{s,t}$, and policy π proper w.r.t. $h_{s,t}$. Let $V^\pi(h_{s,t}) = -\infty$ if π is improper w.r.t. $h_{s,t}$. The optimal value function V^* for this MDP exists, is stationary Markovian, and satisfies, for all $s \in \mathcal{S}$,*

$$V^*(s) = \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s')[\mathcal{R}(s,a,s') + V^*(s')] \right] \tag{5.6}$$

and, for all $s_g \in \mathcal{G}$, $V^*(s_g) = 0$. Moreover, at least one optimal policy $\pi^*_{s_0}$ proper w.r.t. s_0 and greedy w.r.t. the optimal value function is stationary deterministic Markovian and satisfies, for all $s \in \mathcal{S}$,

$$\pi^*_{s_0}(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s')[\mathcal{R}(s,a,s') + V^*(s')] \right]. \tag{5.7}$$

$\diamondsuit$

Proof. Although the full proof is postponed until the Appendix, its intuition is important for explaining subsequent material, so we describe it here. Its main insight is that a GSSP$_{s_0}$ MDP can be converted to an equivalent SSP$_{s_0}$ MDP problem, for which a similar Optimality Principle has already been established (Theorem 2.3). Namely, consider the transition graph of a GSSP$_{s_0}$ MDP such as the one in Figure 5.1, and observe that for any policy π, $V^\pi(s_1) = V^\pi(s_2)$ and $V^\pi(s_3) = V^\pi(s_4)$. This is because $\{s_1, s_2\}$ and $\{s_3, s_4\}$ form zero-reward cycles (or, more generally, zero-reward strongly connected components of the transition graph), and the agent can travel within them "for free". Therefore, for the purpose of finding the optimal value function, each of the sets $\{s_1, s_2\}$ and $\{s_3, s_4\}$ can be treated as a single state. Now, imagine the MDP in Figure 5.1 where the set $\{s_1, s_2\}$ along with transitions between s_1 and s_2 has been replaced by a new state $\hat{s}_{1,2}$ and $\{s_3, s_4\}$

has been similarly replaced by $\hat{s}_{3,4}$. Further, suppose we identified $\hat{s}_{1,2}$ as a dead end and eliminated it from the MDP entirely — the corresponding GSSP$_{s_0}$ MDP states s_1 and s_2 are dead ends and therefore are not reachable from s_0 by any optimal policy anyway. Recalling that action rewards can be treated as negative costs, the resulting problem would clearly be an SSP$_{s_0}$ MDP, since it would have no zero-reward regions and no dead ends. The optimal value function $\hat{V}^*$ for this MDP would be equivalent to V^* for the original GSSP$_{s_0}$ MDP, with $V^*(s_3) = V^*(s_4) = \hat{V}^*(\hat{s}_{3,4})$, $V^*(s_0) = \hat{V}^*(s_0)$, and $V^*(s_g) = \hat{V}^*(s_g)$. With a more detailed justification, this lets us conclude that the Optimality Principle holds for GSSP$_{s_0}$ MDPs simply because it holds for SSP$_{s_0}$ problems.

Despite the possibility of transforming a GSSP$_{s_0}$ MDP problem into an SSP$_{s_0}$ problem, the former class is not a subclass of the latter. As implied by Theorem 5.5, deriving an optimal policy from the optimal value function of a GSSP$_{s_0}$ MDP is significantly more complicated (conceptually as well as computationally) than for its SSP$_{s_0}$ counterpart. ■

Theorem 5.2. *Define local Bellman backup for GSSP$_{s_0}$ MDPs to be the operator $\mathscr{B}_{GSSP_{s_0}(s)} : \mathcal{V} \to \mathcal{V}$ that applies the transformation*

$$V'(s) \leftarrow \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')[\mathcal{R}(s, a, s') + V(s')] \right] \tag{5.8}$$

to a value function V at a state s. The optimal value function V^* of a GSSP$_{s_0}$ MDP is a fixed point of $\mathscr{B}_{GSSP_{s_0}(s)}$, i.e., $\mathscr{B}_{GSSP_{s_0}(s)} V^* = V^*$, for all $s \in \mathcal{S}$. ♢

Proof. It is a direct consequence of the Optimality Principle stating that V^* must satisfy Equation 5.6. ■

As an example, for the GSSP$_{s_0}$ MDP in Figure 5.1, it can be easily verified that the optimal value function $V^*(s_0) = -0.5$, $V^*(s_1) = V^*(s_2) = -\infty$, $V^*(s_3) = V^*(s_4) = -1$, $V^*(s_g) = 0$ indeed satisfies the Bellman equation and is a fixed point of $\mathscr{B}_{GSSP_{s_0}(s)}$. Unfortunately, the Optimality Principle for GSSP$_{s_0}$ MDPs does not allow us to directly apply Bellman backups in a VI-like algorithm to solve this class of problems, due to three complications described next.

Theorem 5.3. *For a GSSP$_{s_0}$ MDP, $\mathscr{B}_{GSSP_{s_0}(s)}$ can have suboptimal fixed points $V \geq V^*$.* ♢

Proof. For the GSSP$_{s_0}$ MDP in Figure 5.1, consider an admissible value function $V(s_0) = 4$, $V(s_1) = V(s_2) = 2$, $V(s_3) = V(s_4) = 1$, $V(s_g) = 0$. All policies greedy w.r.t. V are improper w.r.t. s_0 and hence suboptimal, but V stays unchanged under $\mathscr{B}_{GSSP_{s_0}(s)}$ for any s. ■

The issues with $\mathscr{B}_{GSSP_{s_0}(s)}$ do not end here. Returning to the example in Figure 5.1, note that for $V^*(s_1) = V^*(s_2) = -\infty$, since no policy can reach the goal from these states, so their values can be updated with Bellman backups ad infinitum but never converge. The situation with s_3 and s_4 is even trickier. Like s_1 and s_2, they are also part of a cycle, but reaching the goal from them *is* possible. However, staying in the loop forever accumulates more reward (0) than going to the goal (-1), frustrating $\mathscr{B}_{GSSP_{s_0}(s)}$ as well. The situation with full Bellman backup is even more dire:

Theorem 5.4. *Define full Bellman backup for GSSP$_{s_0}$ MDPs to be the operator $\mathscr{B}_{GSSP_{s_0}} : \mathcal{V} \to \mathcal{V}$ that applies the transformation*

$$V'(s) \leftarrow \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')[\mathcal{R}(s, a, s') + V(s')] \right]$$

to a value function V at all states s simultaneously. If initialized with an arbitrary admissible V, $\mathscr{B}_{GSSP_{s_0}}$ may never converge. ♢

Proof. For the GSSP$_{s_0}$ MDP in Figure 5.1, consider an admissible value function $V(s_0) = 2.5$, $V(s_1) = -5, V(s_2) = -6$, $V(s_3) = 1$, $V(s_4) = 2$, $V(s_g) = 0$. Applying $\mathscr{B}_{GSSP_{s_0}}$ to V yields V' s.t. $V'(s_0) = 1.5$, $V'(s_1) = -6$, $V'(s_2) = -5$, $V'(s_3) = 2$, and $V'(s_4) = 1$, and applying $\mathscr{B}_{GSSP_{s_0}}$ to V' yields $V'' = V$. Thus, $\mathscr{B}_{GSSP_{s_0}}$ indefinitely cycles between two value functions that have distinct values at all states except the goal. ■

The last complication that prevents us from using the Optimality Principle for solving GSSP$_{s_0}$ MDPs is the fact that, even if V^* is known, an optimal policy cannot be computed simply by taking greedy actions w.r.t. V^*:

Theorem 5.5. *For a GSSP$_{s_0}$ MDP, there can be s.d.M. policies greedy w.r.t. V^* but improper w.r.t. s_0.* ♢

Proof. For the GSSP$_{s_0}$ MDP in Figure 5.1, consider the policy that reaches s_3 from s_0 and then loops between s_3 to s_4 indefinitely. It is greedy w.r.t. V^* but never reaches the goal. ■

Due to the existence of multiple value functions that are fixed points for local Bellman backup for *all* states, heuristic search algorithms based on the FIND-AND-REVISE framework will also yield suboptimal solutions for this class of problems. A potential optimal approach for tackling GSSP$_{s_0}$ MDPs would follow up on the intuition of Theorem 5.1 by finding the problematic sets of states like $\{s_1, s_2\}$ and $\{s_3, s_4\}$ in Figure 5.1, replacing each of them with a single state, and then solving the resulting SSP$_{s_0}$ MDP. Unfortunately, the conversion of a GSSP$_{s_0}$ MDP into an SSP$_{s_0}$ problem requires discovering *all* such problematic regions. This involves visiting the entire state space and makes the procedure too expensive.

Thus, the failure of Bellman backup leaves us with no known optimal but space- and time-efficient techniques capable of solving GSSP$_{s_0}$ MDPs. Next, we present an algorithmic framework that resolves this difficulty.

5.3.3 *FRET — a Schema for Heuristic Search for GSSP$_{s_0}$ MDPs*

Our framework, called FRET (**F**ind, **R**evise, **E**liminate **T**raps), encompasses algorithms that can solve GSSP$_{s_0}$ MDPs when initialized with an admissible heuristic. At the highest level, FRET starts with a GSSP$_{s_0}$ MDP $\hat{M}_i = M$ and an admissible heuristic $\hat{V}_i = V_0$ for it, and, given an $\epsilon > 0$,

- *Runs* FIND-AND-REVISE *initialized with $\hat{V}_i$ to obtain a value function $\hat{V}'_i \leq \hat{V}_i$ for $\hat{M}_i$ that is ϵ-consistent over all states s reachable from s_0 via at least one $\hat{V}'_i$-greedy policy.* This $\hat{V}'_i$ may highlight problematic regions in $\hat{M}_i$ such as $\{s_1, s_2\}$ and $\{s_3, s_4\}$ in Figure 5.1, where an agent "wants" to stay by following a policy greedy w.r.t. $\hat{V}'_i$ from the initial state.

- *Performs an ELIMINATE-TRAPS step by detecting some of the problematic regions and changing $\hat{V}'_i$ and $\hat{M}_i$ to remove them.* Specifically, this step modifies $\hat{M}_i$ to produce another MDP, $\hat{M}_{i+1}$, in which each problematic region highlighted by $\hat{V}'_i$ is either replaced by a single state or, if the goal is unreachable from it, completely discarded. For instance, in $\hat{M}_{i+1}$, the region $\{s_3, s_4\}$ in Figure 5.1 would be replaced by a new state $\hat{s}_{3,4}$, with all transitions that previously used to lead to s_3 or s_4 leading to $\hat{s}_{3,4}$ in $\hat{M}_{i+1}$; the $\{s_1, s_2\}$ region would be eliminated entirely. ELIMINATE-TRAPS also turns $\hat{V}'_i$ into $\hat{V}_{i+1}$, an admissible value function for $\hat{M}_{i+1}$.

- *Iterates the two previous steps, using the output of ELIMINATE-TRAPS as input to* FIND-AND-REVISE. The process stops when ELIMINATE-TRAPS fails to modify the current MDP $\hat{M}_i$ and its value function $\hat{V}'_i$. If ϵ is small enough, $\hat{V}'_i$ is the optimal value function for $\hat{M}_i$. Moreover, $\hat{V}'_i$ can be "expanded" into an optimal (or near-optimal) value function for the the *original* MDP M for all of M's states reachable from s_0 by an optimal policy. That is, like FIND-AND-REVISE on SSP$_{s_0}$ MDPs, FRET is guaranteed to converge to V^* for M over the states visited by optimal policies from s_0 if run for a sufficiently long time.

- *Extracts a policy proper w.r.t. s_0 from the resulting value function for M.* If FRET is allowed to run long enough to converge to V^*, the extracted policy is guaranteed to be optimal.

To explain the technique in more detail, we first review several definitions that proved useful in explaining the FIND-AND-REVISE framework, FRET's analogue for SSP$_{s_0}$ MDPs, and introduce several new ones.

Definitions

A central notion for FRET is that of the *greedy graph of a value function rooted at s* (Definition 2.34). G^V_s is the combined reachability graph of all policies greedy w.r.t. V and rooted at s. As with FIND-AND-REVISE on SSP$_{s_0}$ problems, the key to FRET's efficiency is updating the current value function V only at the states in $G^V_{s_0}$. This lets FRET avoid visiting the entire state space when computing a policy rooted at s_0.

A concept related to the greedy graph is an MDP's *transition graph* (also known as the *reachability graph*) *rooted at s* (Definition 2.32), a directed graph G_s of all states that can be reached from s. It can be expressed in terms of value functions' greedy graphs as $G_s = \cup_{V \in \mathcal{V}} G^V_s$.

We now formalize the concept of the problematic zero-reward regions such as $\{s_1, s_2\}$ and $\{s_3, s_4\}$ in Figure 5.1 that were mentioned in the high-level overviews of Theorem 5.1's proof and of FRET:

Definition 5.2. ***Trap.*** *In the transition graph G_{s_0} of a GSSP$_{s_0}$ MDP, a trap is a strongly connected component (SCC) $C = \{S_C, A_C\}$ consisting of a set S_C of states and a set A_C of hyperedges with the following properties:*

- *S_C contains no goal states, i.e., $\mathcal{G} \cap S_C = \emptyset$.*
- *S_C is "closed" w.r.t. A_C, i.e., for every hyperedge in A_C, its source and all of its destinations are in S_C.*
- *If $s \in S_C$ is the source of a hyperedge A_C and $s_1, \ldots s_n \in S_C$ are this A_C's destinations, then for the action a corresponding to A_C, $\mathcal{R}(s, a, s_i) = 0$ for all $1 \leq i \leq n$.* ♣

Informally, a trap is a strongly connected component of G all of whose internal hyperedges correspond to actions with exclusively zero-reward outcomes.

Definition 5.3. ***Potential Permanent Trap.*** *In the transition graph G_{s_0} of a GSSP$_{s_0}$ MDP, a potential permanent trap is a trap C with the following properties:*

- *G_{s_0} has no hyperedges whose source is in C and at least one of whose destinations is not in C.*
- *C is maximal, in the sense that no trap of G_{s_0} properly contains it.* ♣

Definition 5.4. ***Permanent Trap.*** *A permanent trap w.r.t. a GSSP$_{s_0}$ MDP's value function V and state s is a trap C with the following properties:*

- *C is contained in V's greedy graph G_{s_0} rooted at s_0.*

- *G_{s_0} has no hyperedges whose source is in C and at least one of whose destinations is not in C.* ♣

Put simply, a potential permanent trap is a trap escaping from which (and, in particular, reaching the goal) is impossible. If an agent enters it, it will stay there forever no matter what policy it uses. Thus, a potential permanent trap consists entirely of dead-end states. The region $\{s_1, s_2\}$ in Figure 5.1 is an example of a potential permanent trap. At the same time, no policy greedy w.r.t. a given value function may visit a given potential permanent trap from s_0 (hence the word "potential" in its name). A permanent trap, on the other hand, is a set of dead ends that can be visited by an agent if the agent starts at s_0 and uses actions greedy w.r.t. a particular value function. For instance, in Figure 5.1, $\{s_1, s_2\}$ is a permanent trap w.r.t. s_0 and V s.t. $V(s_1) = 10$, $V(s_3) = 2$, but is not a permanent trap w.r.t. s_0 and V s.t. $V(s_1) = -3$, $V(s_3) = 2$. It is easy to see that a permanent trap is always a strongly connected subcomponent of some potential permanent trap. Observe that, since there is no way to reach the goal from any state s of a permanent trap, $V^*(s) = -\infty$.

Definition 5.5. ***Potential Transient Trap.*** *In the transition graph G_{s_0} of a GSSP$_{s_0}$ MDP, a potential transient trap is a trap C with the following properties:*

- *G_{s_0} has a hyperedge whose source is in C and at least one of whose destinations is not in C.*
- *C is maximal, in the sense that no trap of G_{s_0} properly contains it.* ♣

Definition 5.6. ***Transient Trap.*** *A transient trap w.r.t. a GSSP$_{s_0}$ MDP's value function V and state s is a trap C with the following properties:*

- *C is contained in V's greedy graph $G^V_{s_0}$.*
- *G_{s_0} has a hyperedge whose source is in C and at least one of whose destinations is not in C.*
- *$G^V_{s_0}$ has no hyperedges whose source is in C and at least one of whose destinations is not in C.* ♣

Potential transient traps are regions like $\{s_3, s_4\}$ in Figure 5.1. All their internal transitions bring zero reward, so an agent can stay in them forever without incurring any cost. However, unlike for potential permanent traps, escaping from a potential transient trap is possible. A transient trap is a zero-reward state space region that an agent that starts at s_0 can run into and will not be able to leave *if this agent uses only V-greedy actions*. However, the agent *can* escape from a transient trap w.r.t. V if it uses an action that, according to V, looks suboptimal. Such traps are called "transient" because they look like traps (i.e., appealing state space regions from which reaching the goal is impossible using seemingly optimal actions) only if the agent evaluates its actions w.r.t. V. If the agent updates V to a new value function V', it may be able to exit the trap via an action that appeared suboptimal according to V but is optimal according to V'. More concretely, for the MDP in Figure 5.1, consider a value function V s.t. $V(s_1) = -\infty$, $V(s_3) = V(s_4) = 1$, $V(s_g) = 0$. W.r.t. s_0 and V, $\{s_3, s_4\}$ is a transient trap, because V-greedy actions lead an agent from s_0 to s_4 and then back to s_3, forcing the agent to stay in $\{s_3, s_4\}$. However, if the agent revises its state estimates to a value function V' s.t. $V'(s_1) = -\infty$, $V'(s_3) = V'(s_4) = -2$, $V'(s_g) = 0$, $\{s_3, s_4\}$ will not be a transient trap anymore: V'-greedy actions will lead the agent from s_0 straight to the goal.

Algorithm Description

We can now cast the operation of FRET in terms of these definitions. Throughout the explanation, we will be referring to the pseudocode in Algorithms 5.1 and 5.2.

As already mentioned, FRET iteratively applies two transformations (lines 9-13 of Algorithm 5.1) to a GSSP$_{s_0}$ MDP M and its admissible value function V_0. The first of them, FIND-AND-REVISE, behaves as described in Section 2.3.1. It repeatedly searches the greedy graph $G_{\hat{s}_0}^{\hat{V}_i}$ of the current value function $\hat{V}_i$ and the current MDP $\hat{M}_i$ for states whose values have not reached ϵ-consistency yet and revises their values with local Bellman backups $\mathscr{B}_{GSSP_{s_0}(s)}$, possibly changing $G_{\hat{s}_0}^{\hat{V}_i}$ in the process. The properties of FIND-AND-REVISE guarantee that if FIND-AND-REVISE's initialization function in the i-th iteration of FRET, $\hat{V}_i$, is admissible, then so is FIND-AND-REVISE's output function in that iteration, $\hat{V}_i'$. However, the examples in the previous subsection have demonstrated that in the presence of zero-reward actions FIND-AND-REVISE's Bellman backups may never arrive at the optimal value function for $\hat{M}_i$ and that the greedy graph $G_{\hat{s}_0}^{\hat{V}_i'}$

Algorithm 5.1: FRET

```
1  Input: GSSP$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{G}, s_0 \rangle$, admissible heuristic $V_0$, $\epsilon > 0$
2  Output: a policy closed w.r.t. $s_0$, optimal if $V_0$ is admissible and $\epsilon$ is sufficiently small
3
4  $T2OrigS \leftarrow \emptyset$ // global map from transient traps to their sets of states; modified by the call in line 37
5
6  function FRET(GSSP$_{s_0}$ MDP $M$, heuristic $V_0$, $\epsilon > 0$)
7  begin
8      $\hat{M}_i \leftarrow M, \hat{V}_i \leftarrow V_0$
9      repeat
10         $\hat{V}_i' \leftarrow$ Find-and-Revise$(\hat{M}_i, \hat{V}_i, \epsilon)$ // see Algorithm 2.5
11         $\langle \hat{M}_{i+1}, \hat{V}_{i+1} \rangle \leftarrow$ Eliminate-Traps$(\hat{M}_i, \hat{V}_i')$
12         $\hat{M}_i \leftarrow \hat{M}_{i+1}, \hat{V}_i \leftarrow \hat{V}_{i+1}$
13     until $\hat{V}_{i+1} = \hat{V}_i'$;
14     foreach $s \in \mathcal{S}$ do $V(s) \leftarrow -\infty$ // lines 14-16 build the equivalent of $\hat{V}_i$ for the original MDP $M$
15     foreach $s \in \mathcal{S} \cap \hat{\mathcal{S}}_i$ do $V(s) \leftarrow \hat{V}_i(s)$ // $\hat{\mathcal{S}}_i$ is the state set of MDP $\hat{M}_i$
16     foreach $\hat{s} \in \hat{\mathcal{S}}_i \setminus \mathcal{S}$ do foreach $s \in T2OrigS[\hat{s}]$ do $V(s) \leftarrow \hat{V}_i(\hat{s})$
17     $Processed \leftarrow \mathcal{G}$
18     $G^V_{s_0} \leftarrow \{S^V, A^V\}$ // $V$'s greedy graph rooted at $s_0$
19     while $Processed \neq S^V$ do
20         choose $s \in S^V \setminus Processed$, $a \in \mathcal{A}$ s.t. $\mathcal{T}(s, a, s') > 0$ for some $s' \in Processed$
21         $\pi_{s_0}(s) \leftarrow a$
22         $Processed \leftarrow Processed \cup \{s\}$
23     end
24     return $\pi_{s_0}$
25 end
26
27 function Eliminate-Traps(GSSP$_{s_0}$ MDP $M$, value function $V$)
28 begin
29     $G^V_{s_0} \leftarrow \{S^V, A^V\}$ // $V$'s greedy graph rooted at $s_0$
30     $\mathcal{SCC} \leftarrow \underline{TarjanZero}(G^V_{s_0})$ // finds SCCs with only zero-reward hyperedges; pseudocode
       omitted
31     $Traps \leftarrow \emptyset$
32     foreach SCC $C = \{S, A\} \in \mathcal{SCC}$ do
33         if ($\nexists$ $V$-greedy $a$, $s \in S, s' \notin S$ s.t. $\mathcal{T}(s, a, s') > 0$) and ($\mathcal{G} \cap S == \emptyset$) then
34             $Traps \leftarrow Traps \cup \{C\}$
35         end
36     end
37     $\hat{M} \leftarrow$ Transform-MDP$(M, Traps)$ // $\hat{M} = \langle \hat{\mathcal{S}}, \hat{\mathcal{A}}, \hat{\mathcal{T}}, \hat{\mathcal{R}}, \hat{\mathcal{G}}, \hat{s}_0 \rangle$
38     foreach $s \in \hat{\mathcal{S}} \cap \mathcal{S}$ do $\hat{V}(s) \leftarrow V(s)$
39     foreach $C = \{S, A\} \in Traps$ do
40         if $\exists a \in \mathcal{A}, s \in S, s' \notin S$ s.t. $\mathcal{T}(s, a, s') > 0$ // if $C$ is a transient trap then
41             $\hat{s} \leftarrow$ state in $\hat{\mathcal{S}}$ s.t. $T2S[\hat{s}] = S$
42             $\hat{V}(\hat{s}) \leftarrow \min_{s \in S} V(s)$
43         end
44     end
45     return $\langle \hat{M}, \hat{V} \rangle$
46 end
```

Algorithm 5.2: Transform-MDP

```
// see Algorithm 5.1 for the definition of global map $T2OrigS$

function Transform-MDP(GSSP$_{s_0}$ MDP $M$, set of traps $Traps$)
begin
    $T2S \leftarrow \emptyset$ // a mapping from traps to sets of states contained in each trap
    $\hat{\mathcal{S}} \leftarrow \mathcal{S} \setminus \bigcup_{C \in Traps}[S$ s.t. $C = \{S, A\}], \hat{\mathcal{A}} \leftarrow \mathcal{A}, \hat{\mathcal{G}} \leftarrow \mathcal{G}$
    foreach $C = \{S, A\} \in Traps$ do
        if $\exists a \in \mathcal{A}, s \in S, s' \notin S$ s.t. $\mathcal{T}(s, a, s') > 0$ then
            $OrigStateSet \leftarrow \emptyset$
            foreach $s \in S$ do
                if $T2OrigS[s] \neq null$ then
                    $OrigStateSet \leftarrow OrigStateSet \cup T2OrigS[s]$
                    Remove the entry for $s$ from $T2OrigS$
                else $OrigStateSet \leftarrow OrigStateSet \cup \{s\}$
            end
            $\hat{\mathcal{S}} \leftarrow \hat{\mathcal{S}} \cup \{\hat{s}\}$ // $\hat{s}$ is new, $T2OrigS[\hat{s}] \leftarrow OrigStateSet$, $T2S[\hat{s}] \leftarrow S$
        else foreach $s \in S$ do if $T2OrigS[s] \neq null$ then Remove entry for $s$ from $T2OrigS$
    end
    foreach $s \in \hat{\mathcal{S}}$ do
        if $s \in \mathcal{S}$ then
            foreach $a \in \mathcal{A}$ do
                if $\exists s_d$ in a permanent trap in $Traps$ s.t. $\mathcal{T}(s, a, s_d) > 0$ then
                    foreach $s' \in \hat{\mathcal{S}}$ do $\hat{\mathcal{T}}(s, a, s') \leftarrow 0$
                    continue
                end
                foreach $s' \in \hat{\mathcal{S}}$ do
                    if $s' \in \mathcal{S}$ then $\hat{\mathcal{T}}(s, a, s') \leftarrow \mathcal{T}(s, a, s'), \hat{\mathcal{R}}(s, a, s') \leftarrow \mathcal{R}(s, a, s')$
                    else $\hat{\mathcal{T}}(s, a, s') \leftarrow \sum_{s'' \in T2S[s']} \mathcal{T}(s, a, s''), \hat{\mathcal{R}}(s, a, s') \leftarrow \sum_{s'' \in T2S[s']} \frac{\mathcal{T}(s,a,s'')\mathcal{R}(s,a,s'')}{\hat{\mathcal{T}}(s,a,s')}$
                end
            end
        else
            foreach $s' \in T2S[s]$ do
                foreach $a \in \mathcal{A}$ do
                    if $\exists s_d$ in a permanent trap in $Traps$ s.t. $\mathcal{T}(s', a, s_d) > 0$ then continue
                    if $\neg(\sum_{s'' \in T2S[s]} \mathcal{T}(s', a, s'') < 1 \vee \exists s'' \in T2S[s]$ s.t. $\mathcal{R}(s', a, s'') \neq 0)$ then
                        continue
                    else
                        $\hat{\mathcal{A}} \leftarrow \hat{\mathcal{A}} \cup \{a_{s'}\}$ // $a_{s'}$ is a newly created action
                        foreach $s_1, s_2 \in \hat{\mathcal{S}}$ do $\hat{\mathcal{T}}(s_1, a_{s'}, s_2) \leftarrow 0$
                        foreach $\hat{s} \in \hat{\mathcal{S}}$ do
                            if $\hat{s} \in \mathcal{S}$ then $\hat{\mathcal{T}}(s, a_{s'}, \hat{s}) \leftarrow \mathcal{T}(s', a, \hat{s}), \hat{\mathcal{R}}(s, a_{s'}, \hat{s}) \leftarrow \mathcal{R}(s', a, \hat{s})$
                            else
                                $\hat{\mathcal{T}}(s, a_{s'}, \hat{s}) \leftarrow \sum_{s'' \in T2S[\hat{s}]} \mathcal{T}(s', a, s'')$
                                $\hat{\mathcal{R}}(s, a_{s'}, \hat{s}) \leftarrow \sum_{s'' \in T2S[\hat{s}]} \frac{\mathcal{T}(s',a,s'')\mathcal{R}(s',a,s'')}{\hat{\mathcal{T}}(s,a_{s'},\hat{s})}$
                            end
                        end
                    end
                end
            end
        end
    end
    if $s_0 \in \mathcal{S} \cap \hat{\mathcal{S}}$ then $\hat{s}_0 \leftarrow s_0$ else $\hat{s}_0 \leftarrow \hat{s} \in \hat{\mathcal{S}}$ s.t. $s_0 \in T2S[\hat{s}]$
    return $\hat{M} = \langle \hat{\mathcal{S}}, \hat{\mathcal{A}}, \hat{\mathcal{T}}, \hat{\mathcal{R}}, \hat{\mathcal{G}}, \hat{s}_0 \rangle$
end
```

of FIND-AND-REVISE's output value function in the i-th iteration of FRET, $\hat{V}_i'$, may contain permanent and transient traps.

The second step, ELIMINATE-TRAPS (lines 27-46 of Algorithm 5.1), searches $G_{\hat{s}_0}^{\hat{V}_i'}$ for traps and changes $\hat{M}_i$ and $\hat{V}_i'$ into a new MDP $\hat{M}_{i+1}$ and value function $\hat{V}_{i+1}$ s.t. $\hat{M}_{i+1}$ has none of the discovered traps of $\hat{M}_i$, and $\hat{V}_{i+1}$ is the admissible "version" of $\hat{V}_i'$ for $\hat{M}_{i+1}$. To find traps, ELIMINATE-TRAPS employs Tarjan's algorithm [94] (its pseudocode is omitted) to identify all maximal SCCs of $G_{\hat{s}_0}^{\hat{V}_i'}$ all of whose internal hyperedges have zero reward (line 30). It then considers only those of the discovered zero-reward SCCs that have no goal states and outgoing edges in this graph (lines 32-36), i.e., satisfy the definition of either a permanent or a transient trap (Definitions 5.4 and 5.6).

The logic for using $\hat{M}_i$ to construct a new MDP $\hat{M}_{i+1}$ without the discovered traps is captured in the Transform-MDP routine (Algorithm 5.2). We will call $\hat{M}_{i+1}$ a *contraction* of $\hat{M}_i$:

Definition 5.7. ***Contraction of a GSSP$_{s_0}$ MDP.*** *For a GSSP$_{s_0}$ MDP M, a contraction of M is an MDP $\hat{M}$ that results from eliminating some traps from M using Algorithm 5.2.* ♣

Abstracting away from Transform-MDP's many technical details, its high-level idea is very straightforward. Transform-MDP takes the state space of $\hat{M}_i$, excludes from it all the states in the identified traps, and, for each removed *transient* trap, adds a single new state to replace the discarded states of that trap. These newly added states *represent* the states of the eliminated transient traps in the contraction $\hat{M}_{i+1}$:

Definition 5.8. ***Representative of a state from a trap.*** *For a GSSP$_{s_0}$ MDP M and a trap $C = \{S, A\}$ in it, suppose $\hat{M}$ is a contraction of M where the set of states S of M has been replaced by a single state $\hat{s}$. For any state $s \in S$ of M, $\hat{s}$ is the representative of s in $\hat{M}$.* ♣

The states of eliminated permanent traps do not have representatives in the state space of $\hat{M}_{i+1}$. As discussed previously, the ultimate aim of trying to solve $\hat{M}_{i+1}$ with FIND-AND-REVISE or transforming it into yet another MDP is computing a (near-) optimal policy for the original MDP M from s_0. States from permanent traps cannot be part of it, since, by definition, M has a proper policy w.r.t. s_0 that avoids such states entirely. Therefore, there is no need to consider them any further once they have been recognized as belonging to permanent traps.

Once the state space $\hat{\mathcal{S}}$ of $\hat{M}_{i+1}$ has been constructed as above, Transform-MDP modifies other components of $\hat{M}_i$ to be consistent with $\hat{\mathcal{S}}$. Among other changes, for every state of $\hat{\mathcal{S}}$, it eliminates any action that in $\hat{M}_i$ leads to a state in a permanent trap. It also modifies the transition function of other actions so that any transitions to a state in a transient trap in $\hat{M}_i$ lead to the representative of that trap's states in $\hat{M}_{i+1}$.

We now consider the operation of Transform-MDP in more detail. Transform-MDP starts by setting $\hat{M}_{i+1}$'s state space $\hat{\mathcal{S}}$ to $\hat{M}_i$'s state space less the discovered traps, $\hat{M}_{i+1}$'s action space $\hat{\mathcal{A}}$ to $\hat{M}_i$'s action space, and $\hat{M}_{i+1}$'s goal set $\hat{\mathcal{G}}$ to $\hat{M}_i$'s goal set (line 6 of Algorithm 5.2). Then Transform-MDP adds to $\hat{\mathcal{S}}$ (line 16) a representative of each eliminated transient (line 8) trap. At this stage, some work needs to be done to maintain a mapping $T2OrigS$ from the states of $\hat{M}_{i+1}$ that represent the original MDP M's traps to the sets of states in those traps. $T2OrigS$ is necessary to transform the solution of a contraction of M to a solution of M itself once FRET exits its main loop (line 16 of Algorithm 5.1). For each transient trap C of $\hat{M}_i$, the maintenance (lines 9-16 of Algorithm 5.2) involves determining the set $OrigStateSet$ of states of the *original* MDP M that are represented by the states of that trap (lines 11-14) and entering that set into $T2OrigS$ for $\hat{M}_{i+1}$'s representative $\hat{s}$ of C (line 16). Note that if C turns out to be a permanent trap in $\hat{M}_i$, all of its states representing traps of M are removed from $T2OrigS$ (line 17), because C will not have a representative in $\hat{M}_{i+1}$ or $\hat{M}_{i+1}$'s contractions.

The purpose of the bulk of Transform-MDP's pseudocode is construct $\hat{M}_{i+1}$'s transition function $\hat{\mathcal{T}}$ and reward function $\hat{\mathcal{R}}$ for each state-action pair of $\hat{M}_{i+1}$'s (lines 19-51 of Algorithm 5.2). For each state $s \in \hat{\mathcal{S}}$, the construction process depends on whether the state has been inherited from $\hat{M}_i$ (lines 20-31) or whether it is a representative of a trap of $\hat{M}_i$ (lines 31-50), i.e., is not in $\hat{M}_i$'s state space. In the former case, all actions that in $\hat{M}_i$ used to lead from s to a state of an eliminated permanent trap are rendered inapplicable in s (line 22-25). For the remaining actions, their transition probabilities from s to other states inherited from $\hat{M}_i$, along with rewards for such transitions, remain the same as in $\hat{M}_i$ (line 27). However, the probabilities and rewards for their transitions from s to states of eliminated transient traps of $\hat{M}_i$ are summed up for $\hat{M}_{i+1}$'s representatives of these traps (line 28).

The initialization of the transition and reward function for state-action pairs where the state s is a representative of $\hat{M}_i$'s transient trap is analogous to the case when s has been inherited from $\hat{M}_i$, but the details are slightly more complicated. Specifically, since s is not a state of $\hat{M}_{i+1}$, no actions inherited from $\hat{M}_i$ are applicable in it. Instead, for each state s' of $\hat{M}_i$ represented by s in $\hat{M}_{i+1}$ and every action a of $\hat{M}_i$, Transform-MDP creates a new action $a_{s'}$ in $\hat{M}_{i+1}$ (line 38), applicable only in s (line 39). There are two exceptions to this rule:

- If a in $\hat{M}_i$ leads from a state s' represented by s to a state in an eliminated permanent trap of $\hat{M}_i$, no new action in $\hat{M}_{i+1}$ is created for the pair (s', a) (line 34).

- No new action in $\hat{M}_{i+1}$ is created for $\hat{M}_i$'s state-action pairs (s', a) where s' is represented by s and a leads from s' exclusively to states represented by s via zero-reward transitions (line 36). The analogue of a for s in $\hat{M}_{i+1}$ would be a deterministic self-loop with zero reward, which would turn s into a single-state transient trap of $\hat{M}_{i+1}$, contrary to our intent of introducing s in order to eliminate a trap in the first place.

For each created action $a_{s'}$, its transition probabilities and rewards for s are initialized (lines 40-46) in the same way as in lines 26-29, i.e., as if they would be for s' if s' was inherited from $\hat{M}_i$.

Transform-MDP finishes the construction of $\hat{M}_{i+1}$ by initializing its initial state by either letting it be the same as for $\hat{M}_i$ or, if s_0 is in an eliminated transient trap of $\hat{M}_i$, to a state representing that trap in $\hat{M}_{i+1}$.

Returning to ELIMINATE-TRAPS (Algorithm 5.1), after producing $\hat{M}_{i+1}$ using Transform-MDP it converts $\hat{V}_i'$ into an admissible value function for this new MDP. Observe that $\hat{M}_{i+1}$'s state space $\hat{S}$ is different from $\hat{M}_i$'s state space in only two ways: some transient traps have been replaced by single representative states, and states of some permanent traps are missing. Thus, to convert $\hat{V}_i'$ to an admissible value function for $\hat{M}_{i+1}$, all we need to do is to admissibly initialize the values of the representative states and remove the values of states in the eliminated traps; the values of states that are shared between $\hat{M}_{i+1}$ and $\hat{M}_i$ can stay as they are under $\hat{V}_i'$. This is exactly what ELIMINATE-TRAPS does in lines 38-43 of Algorithm 5.1. ELIMINATE-TRAPS sets the values of trap representatives to the lowest value under $\hat{V}_i'$ of any state in the corresponding trap. In fact, the value of any state in that trap would work just as well. The intuition behind this step is that in a transient trap, any state can be reached "for free" (i.e., by paying zero cost) from any other state of that trap, implying that under the *optimal* value function, the values of all states in a given trap must be equal. Therefore, since $\hat{V}_i'$ is admissible for $\hat{M}_i$ (the properties of FIND-AND-REVISE guarantee this, due to the admissibility of the heuristic V_0), $\hat{V}_i'$'s value of any state in any of $\hat{M}_i$'s traps is an upper bound on the value of the representative of that trap in $\hat{M}_{i+1}$.

FRET's repetition of FIND-AND-REVISE and ELIMINATE-TRAPS steps cannot continue forever: each ELIMINATE-TRAPS invocation produces an MDP whose state space is smaller than that of the current MDP, and each FIND-AND-REVISE invocation is guaranteed to halt after a finite number of REVISE operations. Thus, eventually FRET exits its main loop with an MDP $\hat{M}_i$ and a near-optimal value function $\hat{V}_i$ for it. What is the relationship between $\hat{M}_i$ and the input MDP M? It is easy to see that $\hat{M}_i$ is M's contraction, obtained by successively eliminating traps of M. A crucial implication of this insight is the fact that a value function for $\hat{M}_i$ can be "expanded" into a value function for M:

Definition 5.9. ***Expansion of a value function.*** *Let $\hat{V}$ be a value function for a contraction $\hat{M} = \langle \hat{\mathcal{S}}, \hat{\mathcal{A}}, \hat{\mathcal{T}}, \hat{\mathcal{R}}, \hat{\mathcal{G}}, \hat{s}_0 \rangle$ of a GSSP$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{G}, s_0 \rangle$. An expansion of $\hat{V}$ for M is a value function V s.t.*

$$V(s) = \begin{cases} \hat{V}(s) & \text{if } s \in \mathcal{S} \cap \hat{\mathcal{S}} \\ \hat{V}(\hat{s}) & \text{if } s \text{ is in a trap of } M \text{ whose set of states has been replaced by state } \hat{s} \text{ in } \hat{M} \\ -\infty & \text{otherwise} \end{cases} \tag{5.9}$$

♣

Moreover, the expansion of $\hat{M}_i$'s optimal value function is clearly the optimal value function for M. FRET performs such an expansion as its penultimate step (lines 14-16 of Algorithm 5.1), right before computing a policy for M.

The convergence characteristics of FRET are summarized by the results below. Although some of these theorems require FRET's heuristic to be not only admissible but also monotonic (Definition 2.29), their versions hold even when the heuristic does not have this property. However, their proofs for the case of a non-monotonic admissible heuristic are significantly more complicated than those presented here. In the proofs, $\mathscr{E}$ denotes the operator implemented in FRET's ELIMINATE-TRAPS step.

Theorem 5.6. *For a GSSP$_{s_0}$ MDP and any $\epsilon > 0$, if FRET has a systematic FIND procedure and is initialized with an admissible heuristic that is finite and monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s, FRET converges to a value function that is ϵ-consistent over the states in its greedy graph rooted at s_0 after a finite number of REVISE and ELIMINATE-TRAPS steps.* ♢

Proof. See the Appendix. The proof is somewhat technical and relies on several lemmas that are also presented in the Appendix, but its underlying idea is simple and has been discussed above: ELIMINATE-TRAPS keeps decreasing MDP's size, and each call to FIND-AND-REVISE halts after a finite time for any $\epsilon > 0$. ■

Theorem 5.7. *For a GSSP$_{s_0}$ MDP, if FRET has a systematic FIND procedure and is initialized with an admissible heuristic that is finite and monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s, as ϵ goes to 0 the value function and policy computed by FRET approaches, respectively, the optimal value function and an optimal policy over all states reachable from s_0 by at least one optimal s.d.M. policy.* ♢

Proof. See the Appendix. Like the proof of the Optimality Principle (Theorem 5.1), the proof of the above claim is based on the observation that a GSSP$_{s_0}$ problem can be turned into an equivalent SSP$_{s_0}$ MDP. In particular, the proof shows that running FRET on a GSSP$_{s_0}$ MDP amounts to lazily performing such a conversion. If FRET is run long enough (i.e., ϵ is sufficiently small), it reduces to executing FIND-AND-REVISE on the resulting SSP$_{s_0}$ MDP, a process known to converge to an optimal solution (Theorem 2.15). ■

Upon convergence to an ϵ-consistent V, FRET constructs a policy greedy w.r.t. V as in lines 17-23 of the pseudocode. Theorem 5.5 showed that for V^*, constructing $\pi^*_{s_0}$ by simply taking arbitrary V^*-greedy actions would not be enough. Instead, FRET builds an output policy by using V-greedy actions to connect states in $G^V_{s_0}$ either directly to the goal states or to states already connected to the goal states, as long as this is possible. If the value function V at which FRET halts is either V^* or sufficiently close to V^*, the policy π_{s_0} produced at the end of this process has an important property: every trajectory of π_{s_0} originating at every state of $G^V_{s_0}$ eventually terminates in a goal state. Thus, π_{s_0} is proper w.r.t. every state in $G^V_{s_0}$ and, if V is close to V^*, optimal:

Theorem 5.8. *For a GSSP$_{s_0}$ MDP, any policy π_{s_0} derived by FRET from the optimal value function is optimal and proper w.r.t. s_0.* $\diamondsuit$

Proof. See the Appendix. Intuitively, the result holds because, by construction, π_{s_0} is greedy w.r.t. V^* and from every state reachable by π_{s_0} from s_0, executing π_{s_0} with a positive probability results in a trajectory that leads to the goal. Thus, the agent that follows π_{s_0} starting at s_0 is bound to arrive at a goal state eventually. ■

As for FIND-AND-REVISE for SSP$_{s_0}$ MDPs, in general a policy constructed by FIND-AND-REVISE w.r.t. an ϵ-consistent V may be suboptimal. However, for ϵ sufficiently close to 0 this does not happen.

Example

To illustrate FRET's operation, we simulate its pseudocode on the GSSP$_{s_0}$ MDP in Figure 5.1. Suppose FRET starts with $\hat{V}_0(s_0) = 4, \hat{V}_0(s_1) = \hat{V}_0(s_2) = 2, \hat{V}_0(s_3) = \hat{V}_0(s_4) = 1, \hat{V}_0(s_g) = 0$. This function already satisfies $\hat{V}_0 = \mathscr{B}_{GSSP_{s_0}(s)}\hat{V}_0$, so the FIND-AND-REVISE step in the zeroth iteration finishes immediately with $\hat{V}'_0 = \hat{V}_0$. ELIMINATE-TRAPS then builds $G^{\hat{V}'_0}_{s_0}$, which includes only states s_0, s_1, and s_2. The only (permanent) trap is formed by s_1, s_2. Thus, ELIMINATE-TRAPS produces an MDP $\hat{M}_1$ with s_1 and s_2 and transitions to them from s_0 missing, and constructs a value function $\hat{V}_1(s_0) = 4, \hat{V}_1(s_3) = \hat{V}_1(s_4) = 1, \hat{V}_1(s_g) = 0$. In the first round, FIND-AND-REVISE starts with $\hat{M}_1$ and $\hat{V}_1$ and converges to $\hat{V}'_1(s_0) = 1.5, \hat{V}'_1(s_3) = \hat{V}'_1(s_4) = 1, \hat{V}'_1(s_g) = 0$.

$G_{s_0}^{\hat{V}_1'}$, consisting of s_0, s_3, and s_4, again contains a trap, this time a transient one formed by s_3, s_4. The ELIMINATE-TRAPS step discovers this trap and constructs a new MDP $\hat{M}_2$ whose state space consists of s_0, s_g, and a representative $\hat{s}_{3,4}$ of the eliminated transient trap. It also puts together an admissible value function $\hat{V}_2$ for $\hat{M}_2$ s.t. $\hat{V}_2(s_0) = 1.5, \hat{V}_2(\hat{s}_{3,4}) = 1$, and $\hat{V}_2(s_g) = 0$. Finally, in the second round, FIND-AND-REVISE converges to $\hat{V}_2'(s_0) = -0.5, \hat{V}_2'(\hat{s}_{3,4}) = -1, \hat{V}_2'(s_g) = 0$. $G_{s_0}^{\hat{V}_2'}$ contains no traps, so FRET has converged; in fact, in this example, $\hat{V}_2'$ is optimal for $\hat{M}_2$. Therefore, its expansion $V(s_0) = -0.5, V(s_1) = V(s_2) = -\infty, V(s_3) = V(s_4) = -1, V(s_g) = 0$ is optimal for the MDP in Figure 5.1.

5.3.4 $GSSP_{s_0}$ and Other MDP Classes

Despite *$GSSP_{s_0}$* relaxing the conditions on action rewards, one might view the remaining requirements of its definition as still too restrictive. In this section, we try to allay these concerns by showing that several established MDP types as well as at least one newly formulated class can be viewed as subclasses of *$GSSP_{s_0}$*, with the benefit of FRET being applicable to them as well. Figure 1.2 displays the hierarchy of MDP classes derived in this section.

SSP_{s_0} MDPs

We begin the analysis by showing that *$GSSP_{s_0}$* contains *SSP_{s_0}*, the class that motivated us to define *$GSSP_{s_0}$* in the first place:

Theorem 5.9. *SSP_{s_0} under the weak definition of its components (Definition 2.15) is contained in $GSSP_{s_0}$.* ♢

Proof. *SSP_{s_0}* satisfies both requirements of the *$GSSP_{s_0}$* definition (5.1). Namely, every SSP$_{s_0}$ MDP has a complete proper policy. Also, under the weak definition (2.15), all actions of an SSP$_{s_0}$ MDP have a positive cost (negative reward), so the expected sum of nonnegative rewards of any of its policies must be 0 from any state. ■

In light of Theorem 2.5, this result implies that *FH_{s_0}* and *$IHDR_{s_0}$* are subclasses of *$GSSP_{s_0}$* too; in addition, due to Theorem 2.6, *$GSSP_{s_0}$* also contains the versions of all these classes without the initial state. As we demonstrate next, these are are at least two more known MDP classes that can be regarded as special cases of *GSSP*$_{s_0}$.

Positive-Bounded MDPs

Definition 5.10. ***Positive-bounded (POSB) MDP.*** *A POSB MDP [80] is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, with components as described in Definition 2.1, under the following conditions:*

- *For every state $s \in \mathcal{S}$, there must exist an action $a \in \mathcal{A}$ whose immediate expected reward at s is nonnegative, i.e., $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')\mathcal{R}(s, a, s') \geq 0$.*
- *$V_+^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \max\{0, R_t^{\pi_s}\}\right] < \infty$ for all policies π for all states $s \in \mathcal{S}$ (GSSP$_{s_0}$ has an identical requirement).* ♣

Solving a POSB MDP means finding a policy that maximizes the undiscounted expected reward over an infinite number of steps starting from every state. The POSB MDP definition implies that such a policy always exists and has a nonnegative value everywhere (see Proposition 7.2.1b in [80]). Although at first glance *POSB* appears to be rather different from *GSSP*$_{s_0}$, the former is a subclass of the latter:

Theorem 5.10. *POSB $\subset$ GSSP$_{s_0}$.* ♢

Proof. See the Appendix. The proof consists in embedding a POSB instance into a GSSP$_{s_0}$ MDP of a comparable size. ■

Negative MDPs

Definition 5.11. ***Negative (NEG) MDP.*** *A NEG MDP [80] is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, with components as described in Definition 2.1, under the following conditions:*

- *For all $s \in \mathcal{S}, a \in \mathcal{A}$ the expected reward of a must be nonpositive: $\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s')\mathcal{R}(s, a, s') \leq 0$.*

- *There must exist at least one policy π s.t. $V^{\pi}(s) > -\infty$ for all $s \in \mathcal{S}$.* ♣

As with POSB MDPs, an optimal policy for a NEG problem maximizes long-term expected reward at every state, and every NEG instance is also a GSSP$_{s_0}$ instance:

Theorem 5.11. *NEG $\subset$ GSSP$_{s_0}$.* ♢

Proof. The proof, analogous to the one for Theorem 5.10, is presented in the Appendix. ■

MAXPROB MDPs

There is yet another interesting type of MDPs that FRET can solve. We have seen examples of them before: these are goal-oriented MDPs in which an agent wants a policy leading to the goal with the highest probability. The nuclear reactor shutdown scenario described at the beginning of this section is an instance of this MDP kind, as are other problems where action costs are irrelevant or too hard to model and where no policy proper w.r.t. s_0 may exist. The evaluation measure that assesses policy quality by the policy's probability of eventually reaching the goal was used at IPPC-2006 and -2008 [17] to judge the performance of various MDP solvers. It is also the criterion we employed to evaluate the RETRASE and GOTH algorithms in Chapter 3. We call this optimization criterion *MAXPROB* and define a corresponding MDP class:

Definition 5.12. ***MAXPROB MDP.*** *Given a goal-oriented MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{G}, (s_0) \rangle$ with an optional initial state s_0, let s'_g be a new state that does not belong to M's state space $\mathcal{S}$, and let a'_g be a new action that does not belong to $\mathcal{A}$. The MAXPROB MDP M_{MP} derived from M is a tuple $\langle \mathcal{S}', \mathcal{A}', \mathcal{T}', \mathcal{R}', \mathcal{G}', (s_0) \rangle$, where*

- *$\mathcal{S}' = \mathcal{S} \cup \{s'_g\}$;*

- *$\mathcal{A}' = \mathcal{A} \cup \{a'_g\}$;*
- *$\mathcal{T}'(s, a, s') = \mathcal{T}(s, a, s')$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, $s' \in \mathcal{S}$; $\mathcal{T}'(s, a, s'_g) = 0$ for all $s \in \mathcal{S} \setminus \mathcal{G}$ and $a \neq a'_g$; $\mathcal{T}'(s, a'_g, s'_g) = 1$ for all $s \in \mathcal{S}$; $\mathcal{T}'(s'_g, a, s'_g) = 1$ for all $a \in \mathcal{A}'$;*
- *$\mathcal{R}'(s_g, a'_g, s'_g) = 1$ whenever $s_g \in \mathcal{G}$, and 0 for all other triplets;*
- *$\mathcal{G}' = \{s'_g\}$;*
- *s_0 is the same as in M, if M has it.* ♣

Note that M does not need to be an SSP$_{s_0}$ MDP — it only needs to have a set of goal states. M_{MP}, the MAXPROB derivative of M, has largely the same state space and transition function as M, but adds a special goal state s'_g and action a'_g. The state s'_g is the sole goal in M_{MP}. Transitioning to s'_g is possible from any state of M, but only via a'_g. However, in M_{MP}, the only transitions that bring a non-zero reward, the reward of 1, are those from a goal state of M to s'_g; all others, including the transitions to s'_g from non-goals of M, bring no reward. As an upshot, the only way to get some reward in M_{MP} is by reaching the goal of M_{MP} via a goal state of M.

The MAXPROB MDP M_{MP} derived from a goal-oriented MDP M has an important property: even if M has no proper policy w.r.t. its initial state, M_{MP} always does, due to the extra action a'_g leading to the goal from any state in M_{MP}'s state space. In fact, by the construction of M_{MP}, the following theorem holds:

Theorem 5.12. *Consider a policy π in a goal-oriented MDP M. For the MAXPROB MDP M_{MP} derived from M, the value $V^\pi(s)$ for any state $s \in \mathcal{S}_{M_{MP}}$ w.r.t. which π is closed is the probability that π, if executed in M, eventually reaches the goal from s.* ♢

Thus, the *MAXPROB* criterion gives us a way to reason about the "best" policy for problems that, due to the lack of a proper policy, do not conform to any of the MDP class definitions discussed so far.

Crucially, MAXPROB MDPs derived from problems with an initial state do *not* belong to SSP_{s_0}, because *MAXPROB*, unlike SSP_{s_0}, admits MDPs whose value functions can have transient traps (Definition 5.6). Because of this, the heuristic search machinery developed for SSP_{s_0} does not work for MAXPROB MDPs. Fortunately, MAXPROB MDPs can be solved by FRET as a consequence of the following result:

Theorem 5.13. *MAXPROB* $\subset$ *POSB* $\subset$ $GSSP_{s_0}$. $\diamondsuit$

Proof. See the Appendix. The proof shows that, even though MAXPROB MDPs have goal states and POSB MDPs do not, mathematically the structure of the former is a special case of the structure of the latter. ■

To test FRET's effectiveness, in our empirical evaluation we concentrate on MAXPROB MDPs derived from hard problems without a proper policy used at IPPC. We show that in terms of the MAXPROB criterion, FRET can solve them optimally and much more efficiently than the only other optimal algorithm available for solving them, the inadmissibly initialized VI. We also point out that *MAXPROB* will play a central role in our analysis of goal-oriented MDPs with dead ends in the next section.

5.3.5 Heuristics for $GSSP_{s_0}$

Unsurprisingly, due to the complete prior lack of heuristic search methods for solving MDPs with traps (such MDPs abound even in previously established classes like $POSB$ and NEG), research in admissible heuristics for them has also been nonexistent. To prompt it, we suggest a technique that can both serve as a standalone admissible heuristic for GSSP_{s_0} MDPs in a factored form and help other admissible heuristics for this MDP class when they are invented.

The heuristic we propose is based on the SIXTHSENSE module [57] (Section 3.6). SIXTHSENSE operates by learning nogoods, sound but incomplete rules that help identify states as dead ends. The following observation motivates the use of SIXTHSENSE for factored GSSP_{s_0} problems. In GSSP_{s_0} MDPs, many of the dead ends are located in permanent traps. As we have demonstrated, permanent traps are the regions of the state space where value functions are difficult for Bellman backup to improve. Consequently, an admissibly initialized FIND-AND-REVISE algorithm is likely to visit them and enter them in the state value table before ELIMINATE-TRAPS corrects their values. Employing SIXTHSENSE during the FIND-AND-REVISE step should detect dead-end states early and set the correct values for them, thereby helping FIND-AND-REVISE avoid actions that may lead to them. These "dangerous" actions typically have other states as potential outcomes as well, so in the long run identifying dead ends helps prevent unnecessary visits to large parts of the state space. Moreover, since SIXTHSENSE's nogoods efficiently identify dead ends on-the-fly, these states' values do not need to be memorized. The empirical evaluation presented next supports the validity of this reasoning.

5.3.6 Experimental Results

To evaluate the performance of FRET, we compare it to VI. VI and, with some modifications, PI are the only two other major algorithms able to solve $GSSP_{s_0}$ optimally. However, since they do not use heuristics to exclude states from consideration, they necessarily memorize the value of every state in an MDP's state space. Therefore, their space efficiency, the main limiting factor of MDP solvers' scalability, is the same, and the results of our experimental comparison apply to PI as well.

In the experiments, we use problems from the Exploding Blocksworld (EBW) set of IPPC-2008 [17]. As was done during the IPPC itself and the performance evaluation on these problems in Chapter 3, we assess policy quality of FRET and VI in terms of their policies' probability of reaching the goal from s_0. We stress once again that finding an optimal policy according to this criterion amounts to solving the corresponding MAXPROB MDP, which belongs to the $GSSP_{s_0}$ class. Unlike the participants of IPPC, FRET and VI can solve this optimization problem directly.

We use LRTDP in the FIND-AND-REVISE step of FRET, aided by the following SIXTHSENSE-based heuristic V_0:

$$V_0(s) = \begin{cases} 0 & \text{if } s = s'_g \text{ in the derived MAXPROB MDP } M_{MP} \\ 0 & \text{if } s \text{ if SIXTHSENSE can prove that } s \text{ is a dead end in the original MDP } M \\ 1 & \text{if } s \text{ is neither a goal nor can be proved a dead end in the original MDP } M \end{cases} \tag{5.10}$$

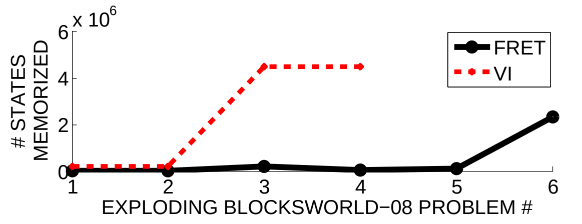


Figure 5.2: Under the guidance of an admissible heuristic, FRET uses much less memory than VI.

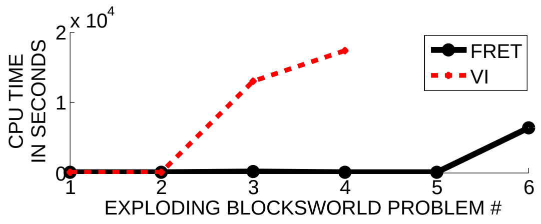


Figure 5.3: VI lags far behind FRET in speed.

In the MAXPROB MDP M_{MP} derived from a goal-oriented MDP M, the former dead ends of M are states from which getting to the former goals of M, and hence getting any reward other than 0, is impossible. Therefore, their value under V^* in M_{MP} is 0. Since SIXTHSENSE can point out a large fraction of M's dead ends, we can assign V_0 for these states to be 0. Similarly, V_0 will be 0 at the sole goal state s'_g of M_{MP}, where the action a'_g leads from M's goal states. For all other states, V_0 will assign 1, the highest reward any policy possibly can obtain in M_{MP}. Since SIXTHSENSE is provably sound, V_0 is guaranteed to be admissible.

All MDPs in EBW have a specified initial state s_0. FRET uses this information to compute a policy rooted at s_0, but VI by default still tries to compute a value function and policy for all states, even those not reachable from the initial state. To make the comparison fair, we modified VI to operate only on states reachable from s_0 as well. In the experiments, VI uses an inadmissible heuristic that assigns value 0 to all non-goal states of M, and 1 to M's goals.

The experiments were run with 2GB of RAM. The results are presented in Figures 5.2 and 5.3. EBW problems increase in state space size from the lowest- to highest-numbered one. Out of the 15 available, FRET solved the first 6 before it ran out of memory, whereas VI managed to solve only 4. (As a side note, some IPPC participants managed to solve more problems; however, this is not surprising because those algorithms are suboptimal). Also, FRET solved problem 6 faster than VI solved problem 4. These data indicate that even with such a crude heuristic as in Equation 5.10, FRET significantly outperforms VI in time and memory consumption. Moreover, no heuristic can help VI avoid memorizing the entire reachable state space and thus allow it to solve bigger problems, whereas with the advent of more powerful admissible GSSP$_{s_0}$ MDP heuristics the scalability of FRET will only increase.

5.3.7 Summary

The MDP class we have presented, *GSSP*$_{s_0}$, extends *SSP* by relaxing *SSP*'s restriction on the reward model. Unlike *SSP*, *GSSP*$_{s_0}$ allows regions where an agent can stay forever without reaching the goal or incurring any reward. Although this extension is seemingly small, sophisticated machinery such as the new heuristic search framework FRET is required to cope with it. Moreover, thanks to admitting such zero-reward regions, *GSSP*$_{s_0}$ contains other established MDP classes, $POSB$ and NEG, in addition to *SSP*. Most importantly, *GSSP*$_{s_0}$ subsumes and provides algorithms for *MAXPROB*, a class of problems where the main objective is maximizing an agent's *probability* of reaching the goal. *MAXPROB* will prove vital to the analysis of MDPs with dead ends, the main subject of the next section.

5.4 SSP MDPs with Dead Ends

MAXPROB MDPs (Definition 5.12) are one way to think about policy quality optimization in the presence of dead ends. At the same time, maximizing the probability of reaching the goal without any concern for the expected cost is often an overly crude approach. A remedy lies in extending the cost-oriented *SSP*$_{s_0}$ class to allow dead-end states.

Straightforwardly introducing dead ends into an SSP MDP's state space under the expected cost minimization criterion breaks existing decision-theoretic solution methods such as VI, preventing them from ever converging. As an alternative, researchers have suggested approximate algorithms that are aware of the possibility of dead ends and try to avoid them when computing a policy — RETRASE [55, 58] from Chapter 3 as well as HMDPP [50], RFF [95], and several other techniques behave in this way. However, these attempts have lacked a theoretical analysis of how to incorporate dead ends into SSP MDPs in a principled way, and optimize more for the probability of reaching the goal rather than policy cost.

5.4.1 Overview

We begin developing a theory of goal-oriented MDPs with dead ends by introducing *SSPADE*$_{s_0}$, a small extension of *SSP*$_{s_0}$ that has well-defined policies optimal in terms of expected cost as long as dead ends can be avoided from the initial state.

The other two classes proposed in this section analyze the case when dead ends cannot be dodged with 100%-probability. Besides computational challenges mentioned above, these situations present semantic difficulties. Consider an *improper* SSP$_{s_0}$ MDP, one that conforms to the *SSP*$_{s_0}$ definition (2.19) except for the requirement of proper policy existence, and hence has unavoidable dead ends. In such an MDP, the objective of finding a policy that minimizes the expected cost of reaching the goal becomes ill-defined. That objective assumes that for at least one policy, the cost incurred by all of the policy's trajectories is finite; however, this cost is finite only for proper policies, all of whose trajectories terminate at the goal. Thus, all policies in an improper SSP may have an infinite expected cost, making the cost criterion unhelpful for selecting the "best" policy. The key question becomes: can we repair cost-based optimization in the presence of unavoidable dead ends, or should we take the dead ends' infinite penalty as a given and choose policies that reduce the risk of failure above all else?

The former option is a possibility in scenarios where the penalty of hitting a dead end can be approximated with a high but finite number. For instance, suppose the agent buys an expensive ticket for a concert of a favorite band in another city, but remembers about it only on the day of the event. Getting to the concert venue requires a flight, either by hiring a business jet or by a regular airline with a layover. The first option is very expensive but almost guarantees making the concert on time. The second is much cheaper but, since the concert is so soon, missing the connection, a somewhat probable outcome, means missing the concert. Nonetheless, the cost of missing the concert is only the price of the ticket, so a rational agent would choose to travel with a regular airline. Accordingly, the MDP class we propose for this case, *fSSPUDE*$_{s_0}$, assumes that the agent can put a price (penalty) on ending up in a dead end state and wants to compute a policy with the least expected cost (including the possible penalty). While seemingly straightforward, this intuition is tricky to operationalize because of several subtleties. We overcome these subtleties and show how *fSSPUDE*$_{s_0}$ can be solved with easy modifications to existing *SSP*$_{s_0}$ algorithms.

For the cases with infinite dead-end penalty we introduce another MDP class, *iSSPUDE*$_{s_0}$. Consider, for example, the task of planning an ascent to the top of Mount Everest for a group of human alpinists. Such an ascent is fraught with inherent lethal risks, and to any human, the price of their own life can be taken as infinite. In this setting, *SSP*'s cost-minimization criterion is uninformative, as discussed above, since every policy reaches an infinite-cost state. Instead, a natural *primary* objective here is to maximize the probability of getting to the goal (i.e., to minimize the chance of getting into a lethal accident, a dead-end state), in the same way as in MAXPROB MDPs. However, of all policies maximizing this probability, we would prefer the least costly one (in expectation). This is exactly the multiobjective criterion we propose for this MDP class.

Intuitively, the objectives of fSSPUDE$_{s_0}$ and iSSPUDE$_{s_0}$ MDPs are related — as *fSSPUDE$_{s_0}$*'s dead-end penalty gets bigger, the optimal policies for both classes become equal in terms of their probability of reaching the goal. We provide a theoretical and an empirical analysis of this insight, showing that solving an fSSPUDE$_{s_0}$ MDP yields a *MAXPROB*-optimal policy if the dead-end penalty is high enough.

5.4.2 SSPADE$_{s_0}$: SSP MDPs with Avoidable Dead Ends

Although the definition of *SSP$_{s_0}$* [12] requires the existence of a complete proper policy, note that an optimal solution for these MDPs exists even if there is a policy proper merely *w.r.t. the initial state*. In other words, the requirement of a complete proper policy is there to make computing an optimal solution easy or at all possible, but not to ensure its existence. Based on this insight, we leave the issue of solution computability aside for the moment and formulate the definition of a goal-oriented MDP with avoidable dead ends.

Definition 5.13. ***SSPADE$_{s_0}$* MDP.** *An SSP$_{s_0}$ MDP with avoidable dead ends (SSPADE$_{s_0}$ MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ where $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}$, and s_0 are as in the SSP$_{s_0}$ MDP definition (2.19), under the following conditions:*

- *There exists at least one proper policy w.r.t. s_0.*
- *For every improper s.d.M. policy π, for every $s \in \mathcal{S}$ where π is improper, $V^{\pi}(s) = \infty$.* ♣

Solving a SSPADE$_{s_0}$ MDP means finding a policy $\pi^*_{s_0} = \arg\min_{\pi} V^{\pi}(s_0)$.

In SSPADE$_{s_0}$ MDPs, dead ends may abound in parts of the state space not visited by policies proper w.r.t. s_0, making this class more general than *SSP$_{s_0}$*. However, returning to the question of finding an optimal solution for a SSPADE$_{s_0}$ MDP, the *SSPADE$_{s_0}$* definition's lax requirements predictably break several algorithms that work for SSP$_{s_0}$ MDPs.

Value Iteration

VI$_{SSP}$, value iteration for *SSP* and *SSP$_{s_0}$*, does not converge on *SSPADE$_{s_0}$*, because the optimal costs for dead ends are infinite. One might think that we may be able to adapt VI$_{SSP}$ to *SSPADE$_{s_0}$* by restricting computation to the subset of states reachable from s_0. However, even this is not true, because *SSPADE$_{s_0}$* requirements do not preclude dead ends reachable from s_0. To enable VI$_{SSP}$ to work on *SSPADE$_{s_0}$*, we would need to detect divergence of state value sequences generated by the application of Bellman backup — an unsolved problem, to our knowledge.

Heuristic Search

Although VI$_{SSP}$ does not terminate for *SSPADE$_{s_0}$*, heuristic search algorithms do. The FIND-AND-REVISE results for *SSPADE$_{s_0}$* are similar to those for *SSP$_{s_0}$* (Theorems 2.14 and 2.15), although their proofs are somewhat more complicated:

Theorem 5.14. *For a SSPADE$_{s_0}$ MDP and an $\epsilon > 0$, if* FIND-AND-REVISE *has a systematic FIND procedure and is initialized with an admissible monotonic heuristic, it converges to a value function that is ϵ-consistent over the states in its greedy graph rooted at s_0 after a finite number of REVISE steps.* $\diamondsuit$

Proof. See the Appendix. The result follows from the observation that each update increases the value function by at least ϵ at some state but never makes it exceed the optimal value function V^*. Since V^* is finite over the states in its greedy graph, which does not include any states without a proper policy, the total number of updates before FIND-AND-REVISE halts must be finite. ■

Theorem 5.15. *For a SSPADE$_{s_0}$ MDP, if* FIND-AND-REVISE *has a systematic FIND procedure and is initialized with an admissible monotonic heuristic, as ϵ goes to 0 the value function and* *policy computed by* FIND-AND-REVISE *approaches, respectively, the optimal value function and an optimal policy over all states reachable from* s_0 *by at least one optimal policy.* $\diamondsuit$

Proof. See the Appendix. Continuing the reasoning in the proof of Theorem 5.14, if FIND-AND-REVISE is run for long enough (i.e., given a very small ϵ), all the states without a proper policy drop out of the greedy graph due to their values becoming very high, and the values of the rest approach their optimal values from below. $\blacksquare$

Unfortunately, this does not mean that all the heuristic search algorithms developed for SSP$_{s_0}$ MDPs, such as LRTDP (Section 2.3.3), will eventually arrive at an optimal policy on any SSPADE$_{s_0}$ problem. The subtle reason for this is that the FIND procedure of some of these algorithms is *not* systematic when dead ends are present. We consider the applicability to SSPADE$_{s_0}$ MDPs of the most widely used heuristic search algorithms, LAO*, ILAO*, RTDP, and LRTDP:

- **LAO*.** LAO* [41] is a heuristic search algorithm that, in the process of computation, analyzes candidate partial policies with PI. If dead ends are present, PI, if run on them, never halts, so neither does LAO*. To fix LAO* for SSPADE$_{s_0}$ MDPs, one can, for instance, artificially limit the maximum number of iterations in PI's policy evaluation step.

- **ILAO*.** Unlike LAO*, ILAO* [41] converges on SSPADE$_{s_0}$ problems without any modification.

- **RTDP.** Observe that if an RTDP trial enters a dead end whose only action leads deterministically back to that state, the trial will not be able to escape that state and will continue forever. Thus, RTDP will never approach the optimal value function no matter how much time passes. Artificially limiting the trial length (the number of state transitions in a trial) with a finite N works, but the magnitude of N is very important. Unless $N \geq |\mathcal{S}|$, convergence of RTDP to the optimal value functions cannot be a priori guaranteed, because some states may not be reachable from s_0 in less than N steps and hence will never be chosen for updates. On the other hand, in MDPs with large state spaces setting $N = |\mathcal{S}|$ may make trials wastefully long.

- **LRTDP.** LRTDP fails to converge to the optimal solution on SSPADE$_{s_0}$ MDPs for the same reason as RTDP, and can be amended in the same way.

As illustrated by the examples of the above algorithms, SSPADE$_{s_0}$ MDPs do lend themselves to heuristic search, but designing FIND-AND-REVISE algorithms for them warrants more care than for SSP$_{s_0}$ MDPs.

5.4.3 fSSPUDE$_{s_0}$: The Case of a Finite Dead-End Penalty

The ostensibly easiest way to handle unavoidable dead ends is to assign a finite positive penalty D for visiting one. The semantics of this approach would be that the agent pays D when encountering a dead end, and the process stops. However, this straightforward modification to the MDP cannot be directly operationalized, because the set of dead ends is not known a-priori and needs to be inferred while planning. Moreover, this definition also has a caveat — it may cause non-dead-end states that lie on potential paths to a dead end to have higher costs than dead ends themselves. For instance, imagine a state s whose only action leads with probability $(1 - \epsilon)$ to a dead end, with probability $\epsilon > 0$ to the goal, and costs $\epsilon(D + 1)$. A simple calculation shows that $V^*(s) = D + \epsilon > D$, even though reaching the goal from s *is* possible.

Therefore, we change the semantics of the finite-penalty model as follows. Whenever the agent reaches *any* state with the expected cost of reaching the goal equaling D or greater, the agent simply pays the penalty D and "gives up", i.e., the process stops. Intuitively, this setting describes scenarios where the agent can put a price on how desirable reaching the goal is. For instance, in the previously discussed example involving a concert in another city, paying the penalty corresponds to deciding not to go to the concert, i.e., foregoing the pleasure the agent would have derived from attending the performance.

The mathematical benefit of putting a "cap" on any state's cost as described above is that the cost of any policy becomes finite under its usual definition in terms of expected utility:

$$\mathrm{VF}^{\pi}(h_{s,t}) \equiv \min\left\{D, \mathbb{E}\left[\sum_{t'=0}^{\infty} C_{t'+t}^{\pi_{h_{s,t}}}\right]\right\}, \qquad (5.11)$$

where $C^{\pi_{h_{s,t}}}_{t'+t}$ is a random variable for the cost paid by the agent starting from time step t and state s if the execution history by that point was $h_{s,t}$ and the agent followed policy π afterwards. This policy evaluation criterion lets us find an optimal policy in goal-oriented MDPs with unavoidable dead ends with finite penalty:

Definition 5.14. ***fSSPUDE$_{s_0}$* *MDP.*** *A stochastic shortest-path MDP with unavoidable dead ends and a finite penalty (fSSPUDE$_{s_0}$ MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, D, s_0 \rangle$, where $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}$, and s_0 are as in the SSP$_{s_0}$ MDP definition (2.19) and $D \in \mathbb{R}^+ \cup \{\infty\}$ is a penalty incurred if a dead-end state is visited, under the following condition:*

- *The expected sum of unpenalized costs incurred by any policy π from any state s where π is improper is infinite, i.e.,*

$$V^{\pi}(h_{s,t}) = \mathbb{E}\left[\sum_{t'=0}^{\infty} C^{\pi_{h_{s,t}}}_{t'+t}\right] = \infty. \tag{5.12}$$

♣

An optimal solution to a fSSPUDE$_{s_0}$ MDP is a policy $\pi^*_{s_0}$ that minimizes Equation 5.11 for all states it reaches from s_0:

$$\pi^*_{s_0} = \underset{\pi}{\text{argmin}}\, \mathrm{VF}^{\pi}(s_0) \tag{5.13}$$

The definitions of both *SSP$_{s_0}$* and *SSPADE$_{s_0}$* have a requirement identical to Equation 5.12. The main difference is that policies in *fSSPUDE$_{s_0}$* are evaluated in terms of *penalized* cost (Equation 5.11), instead of the criterion mentioned in that requirement.

The fact that, as Equation 5.11 implies, the cost of every policy in an fSSPUDE$_{s_0}$ MDP is finite suggests that *fSSPUDE$_{s_0}$* should be no harder to solve than *SSP$_{s_0}$*. This intuition is confirmed by the following result:

Theorem 5.16. *fSSPUDE*$_{s_0}$ $= SSP_{s_0}$. ♢

Proof. See the Appendix. The proof's main idea of converting an fSSPUDE$_{s_0}$ problem to an SSP$_{s_0}$ MDP is to add a special action to the former's action space. That action deterministically takes an agent from any state to the goal and costs D. Mathematically, it makes D the price of the worst choice available to the agent, exactly corresponding to the semantics of *fSSPUDE*$_{s_0}$. ■

Corollary. *The Optimality Principle for SSP MDPs (Theorem 2.3) holds for fSSPUDE*$_{s_0}$ *MDPs as well.* ♢

The conversion of fSSPUDE$_{s_0}$ MDPs to their SSP$_{s_0}$ counterparts immediately enables solving *fSSPUDE*$_{s_0}$ with modified versions of standard *SSP*$_{s_0}$ algorithms, as described next.

Value Iteration

Theorem 5.16 implies that V<span>F</span>*, the optimal cost function of an fSSPUDE$_{s_0}$ MDP, must satisfy the following modified Bellman equation:

$$V(s) = \min \left\{ D, \min_{a \in \mathcal{A}} \left[\mathcal{C}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s')V(s') \right] \right\} \tag{5.14}$$

It also tells us that a $\pi^*_{s_0}$ for an fSSPUDE$_{s_0}$ problem must be greedy w.r.t. VF*. Thus, an fSSPUDE$_{s_0}$ MDP can be solved with VI$_{SSP}$ that uses Equation 5.14 for updates.

Heuristic Search

By the same reasoning as above, all FIND-AND-REVISE algorithms for *SSP*$_{s_0}$ and their guarantees apply to fSSPUDE$_{s_0}$ MDPs if they use Equation 5.14 instead of Equation 2.9 as Bellman backup. Thus, heuristic search for *SSP*$_{s_0}$ works for *fSSPUDE*$_{s_0}$ with hardly any modification.

We point out that, although the theoretical result in Theorem 5.16 is new, some existing MDP solvers use Equation 5.14 implicitly to cope with goal-oriented MDPs that have unavoidable dead ends. One example is the miniGPT package [10]; it allows a user to specify a value D and then uses it to implement Equation 5.14 in several algorithms, including VI and LRTDP.

5.4.4 iSSPUDE$_{s_0}$: The Case of an Infinite Dead-End Penalty

The second way of dealing with unavoidable dead ends is to view them as truly irrecoverable situations and assign $D = \infty$ for getting into them. As a motivation, recall the example of planning a climb to the top of Mount Everest. Since dead ends here cannot be avoided with certainty and the penalty of visiting them is ∞, comparing policies based on the expected cost of reaching the goal breaks down — they all have an infinite expected cost. Instead, we would like to find a policy that maximizes the probability of reaching the goal and whose expected cost *over the trajectories that reach the goal* is the smallest.

To formulate this policy evaluation criterion more precisely, we will work with stationary deterministic Markovian policies because the notation for them is simpler than for the more general history-dependent policies and because, as we show later, at least one optimal policy according to this criterion is stationary deterministic Markovian. For the history-dependent policies, the criterion and its derivation has an analogous form.

In accordance with the criterion's informal description above, each policy is assessed by two characteristics: its probability of reaching the goal and its expected cost *given that it reaches the goal*. To crystallize the former, we define a policy's *goal-probability value function* as

$$P^{\pi}(s) = \sum_{t=1}^{\infty} P[(S_t^{\pi_s} = s_g) \in (\mathcal{G} \wedge S_{t'}^{\pi_s} = s \notin \mathcal{G} \; \forall \, 1 \leq t' < t)], \tag{5.15}$$

where $S_t^{\pi_s}$ is a random variable for the state in which the agent ends up t time steps after starting to follow policy π from state s. Intuitively, $P^{\pi}(s)$ is a sum of probabilities that an agent, initially in state s, will reach the goal for the first time after $1, 2, \ldots$ time steps of executing π. Crucially, we have already seen a class of MDPs whose objective is finding $P^*(s_0)$, the highest probability of reaching the goal from s_0. This class is *MAXPROB* (Section 5.3.4), and P^{π} is just a different notation for its policies' value functions.

Once a policy's goal-probability function P^{π} has been computed, we can calculate its expected cost of reaching the goal. Let $S_t^{\pi_s +}$ be a random variable that gives a distribution over states s' for which $P^{\pi}(s') > 0$ and in which an agent can end up after t steps of executing π starting from state

s. $S_t^{\pi_s+}$ differs from the variable $S_t^{\pi_s}$ in Equation 5.15 by ranging only over states from which π can reach the goal with a positive probability. In other words, $S_t^{\pi_s+}$ assigns a non-zero probability to a state s' only if it can be reached by π from s in t steps *and* the goal can be reached by π from s' afterwards. In terms of these variables, we can define π's *goal-conditioned expected cost* as

$$[V^\pi|P^\pi](s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \mathcal{C}(S_t^{\pi_s+}, A_t^{\pi_s}, S_{t+1}^{\pi_s+})\right] \tag{5.16}$$

The importance of $[V^\pi|P^\pi]$ is in giving us an idea of a policy's expected cost even if the policy fails to avoid dead ends entirely. *SSP*'s standard definition of policy utility breaks down in such situations because some of the policy's trajectories never reach the goal and incur an infinite cost. $[V^\pi|P^\pi]$ avoids this pitfall by taking into account only those trajectories that terminate in a goal state.

Using P^π and $[V^\pi|P^\pi]$, we can define a policy's value under our criterion as an ordered pair

$$\text{VI}^\pi(s) = (P^\pi(s), [V^\pi|P^\pi](s)) \tag{5.17}$$

Specifically, we write $\pi(s) \prec \pi'(s)$, meaning π' is preferable to π at s, whenever $\text{VI}^\pi(s) \prec \text{VI}^{\pi'}(s)$, i.e., when either $P^\pi(s) < P^{\pi'}(s)$, or $P^\pi(s) = P^{\pi'}(s)$ and $[V^\pi|P^\pi](s) > [V^{\pi'}|P^{\pi'}](s)$. Notice that the goal-conditioned expected cost criterion is used only if two policies are equal in terms of the probability of reaching the goal, since maximizing this probability is the foremost priority. For the case when $P^\pi(s) = P^{\pi'}(s) = 0$, i.e., when neither π nor π' can reach the goal from s, we let $[V^\pi|P^\pi](s) = [V^{\pi'}|P^{\pi'}](s) = 0$ and hence $\text{VI}^\pi(s) = \text{VI}^{\pi'}(s)$. With this optimization criterion in mind, we define the following MDP class:

Definition 5.15. ***iSSPUDE$_{s_0}$ MDP.*** *A stochastic shortest-path MDP with unavoidable dead ends and an infinite penalty (iSSPUDE$_{s_0}$ MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, \infty, s_0\rangle$, where $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}$, and s_0 are as in the SSP$_{s_0}$ MDP definition (2.19) and $D = \infty$ is a penalty incurred if a dead-end state is visited, under the following condition:*

- *Under any policy, the cost of any trajectory that never reaches the goal is infinite.*

♣

An optimal solution to a iSSPUDE$_{s_0}$ MDP is a policy $\pi^*_{s_0}$ that maximizes Equation 5.17 for all states it reaches from s_0 under the $\prec$-ordering:

$$\pi^*_{s_0} = \underset{\prec\pi}{\text{argmax}}\ \text{VI}^{\pi}(s_0) \tag{5.18}$$

In contrast to fSSPUDE$_{s_0}$ MDPs, no existing algorithm can solve iSSPUDE$_{s_0}$ problems, and we develop completely new techniques to tackle them.

Value Iteration

As for the finite-penalty case, we begin by deriving a VI-like algorithm for solving *iSSPUDE*$_{s_0}$. Finding a policy satisfying Equation 5.18 may seem hard, since we are effectively dealing with a multicriterion optimization problem. Note, however, that the optimization criteria are, to a certain degree, independent — we can *first* find the set of policies whose probability of reaching the goal from s_0 is optimal, and *then* select from them the policy minimizing the expected cost of goal trajectories. This amounts to calculating the optimal goal-probability function P^*, then computing the optimal goal-conditional value function $[V^*|P^*]$, and finally deriving an optimal policy from $[V^*|P^*]$. We consider these subproblems in order.

Finding P^*. The task of finding, for every state, the highest probability with which the goal can be reached by any policy in a given goal-oriented MDP M amounts to solving that MDP w.r.t. to the MAXPROB criterion (Section 5.3.4). Its solution is the optimal goal-probability function P^* satisfying

$$\begin{aligned} P^*(s) &= 1 && \forall s \in \mathcal{G} \text{ of } M \\ P^*(s) &= \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{T}(s,a,s')P^*(s') && \forall s \in \mathcal{S}\setminus\mathcal{G} \text{ of } M \end{aligned} \tag{5.19}$$

In Section 5.3.4, we showed how to construct a MAXPROB version of the given goal-oriented MDP and tackle it with the heuristic search framework FRET in order to compute P^* for all states reachable from s_0 by a P^*-optimal policy.

P^* for the complete state space of an MDP can be found in a similar manner by observing that eliminating all potential traps in a MAXPROB problem turns it into an SSP$_{s_0}$ MDP. Thus, a flavor of VI that first applies the ELIMINATE-TRAPS operator to a MAXPROB MDP and then iterates over the resulting problem with full Bellman backup (Equation 2.9) eventually finds P^* for all states. Formally, we call this algorithm VI_{MP} and summarize its behavior with the following theorem:

Theorem 5.17. *On a MAXPROB MDP M_{MP} derived from a goal-oriented MDP M, VI$_{MP}$ converges to the optimal value function P^* as its number of iterations tends to infinity.* ♢

Proof. See the Appendix. The result follows from the convergence properties of VI for SSP MDPs. ■

Finding $[V^*|P^*]$. We could derive optimality equations for calculating $[V^*|P^*]$ from first principles and then develop an algorithm for solving them. However, instead we present a more intuitive approach. Essentially, given an iSSPUDE$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, \infty, s_0 \rangle$ and its optimal goal-probability function P^*, we will build a modification M^{P^*} of M, called the *MAXPROB-optimal derivative of M*, whose optimal value function is exactly M's goal-conditional value function $[V^*|P^*]$. M^{P^*} will have no dead ends, have only actions greedy w.r.t. P^*, and have a transition function favoring transitions to states with higher probabilities of successfully reaching the goal. Crucially, M^{P^*} will turn out to be an SSP$_{s_0}$ MDP, so we will be able to find $[V^*|P^*]$ with something as straightforward as VI$_{SSP}$.

To construct M^{P^*}, observe that an optimal policy π^* for an iSSPUDE$_{s_0}$ MDP, one whose cost function is $[V^*|P^*]$, must necessarily use only actions greedy w.r.t. P^*, i.e., those maximizing the right-hand side of Equation 5.19. For each state s, denote the set of such actions as $\mathcal{A}_s^{P^*}$. We focus on non-dead-end states s, because for dead ends $P^*(s) = 0$, and they will not be part of M^{P^*}. By Equation 5.19, for each non-dead-end non-goal s, each $a^* \in \mathcal{A}_s^{P^*}$ satisfies $P^*(s) = \sum_{s' \in \mathcal{S}} T(s, a^*, s')P^*(s')$, or, in a slightly rewritten form,

$$\sum_{s' \in \mathcal{S}} \frac{\mathcal{T}(s, a^*, s') P^*(s')}{P^*(s)} = 1 \tag{5.20}$$

This equation effectively says: *given that the goal is reached from s by executing a^* in s and following an optimal policy onwards*, with probability $\frac{\mathcal{T}(s,a^*,s_1)P^*(s_1)}{P^*(s)}$ action a^* must have caused a transition from s to s_1, with probability $\frac{\mathcal{T}(s,a^*,s_2)P^*(s_2)}{P^*(s)}$ it must have caused a transition to s_2, and so on.

This means that if we want to find the vector $[V^*|P^*]$ of expected costs over goal-reaching trajectories under π^*, then it is enough to find the optimal cost function of MDP $M^{P^*} = \langle \mathcal{S}^{P^*}, \mathcal{A}^{P^*}, \mathcal{T}^{P^*}, \mathcal{C}^{P^*}, \mathcal{G}^{P^*}, s_0^{P^*} \rangle$, where:

- $\mathcal{S}^{P^*} = \mathcal{S} \setminus \{s \in \mathcal{S} | P^*(s) = 0\}$, i.e., $\mathcal{S}^{P^*}$ is the same as $\mathcal{S}$ for M but does not include M's dead ends. The dead ends can be omitted because we are interested only in the costs of trajectories that reach the goal, and no such trajectory, by definition, goes through a dead end.

- $\mathcal{A}^{P^*} = \cup_{s \in \mathcal{S}} \mathcal{A}_s^{P^*}$, i.e., the set of actions consists of all P^*-greedy actions in each state.

- For each $s, s' \in \mathcal{S}^{P^*}$ and $a^* \in \mathcal{A}_s^{P^*}$, $\mathcal{T}^{P^*}(s, a^*, s') = \frac{\mathcal{T}(s,a^*,s')P^*(s')}{P^*(s)}$, as explained in the discussion of Equation 5.20. For each $s, s' \in \mathcal{S}^{P^*}$ and $a^* \notin \mathcal{A}_s^{P^*}$, $\mathcal{T}^{P^*}(s, a^*, s') = 0$, i.e., only actions optimal in s under P^* can be executed in s in the MDP M^{P^*}.

- $\mathcal{C}^{P^*} = \mathcal{C}$ over states in $\mathcal{S}^{P^*}$ and actions in $\mathcal{A}^{P^*}$.

- $\mathcal{G}^{P^*} = \mathcal{G}$.

- $s_0^{P^*} = s_0$.

As it turns out, like M_{MP}, M^{P^*} also belongs to a familiar class of MDPs:

Theorem 5.18. *For an iSSPUDE$_{s_0}$ MDP M with $P^*(s_0) > 0$, the MAXPROB-optimal derivative M^{P^*} constructed from M as above is an SSP$_{s_0}$ MDP.* ◇

Proof. See the Appendix. Intuitively, M^{P^*} is an *SSP*$_{s_0}$ MDP because in it, like in M, all improper policies incur an infinite cost but it, unlike M, does not have any dead ends. ■

Moreover, optimal solutions of M^{P^*} have a crucial property:

Theorem 5.19. *For an iSSPUDE*$_{s_0}$ *MDP* M *with* $P^*(s_0) > 0$*, every optimal s.d.M. policy for* M*'s MAXPROB-optimal derivative* M^{P^*} *is optimal w.r.t. the goal-reaching probability in* M*, i.e.,* $P^{\pi^*}(s) = P^*(s)$ *for all states* s *of* M *s.t.* $P^*(s) > 0$. ♢

Proof. See the Appendix. The ultimate reason why this is the case is that any π^* of M^{P^*} uses only M's P^*-greedy actions and is proper, i.e., uses the P^*-greedy actions in such a way that it does not get stuck in "loops". As a result, it reaches the goal in M from any state s with the maximum possible probability, $P^*(s)$. ■

Thus, solving an iSSPUDE$_{s_0}$ problem M boils down to solving an SSP$_{s_0}$ MDP whose optimal value function, by construction, is $[V^*|P^*]$, and whose optimal policy minimizes not only the expected cost over the goal-reaching trajectories but also the probability of reaching the goal in M. Therefore, the above theorems as well as the Optimality Principle for SSP MDPs (Theorem 2.3) give us the following corollary:

Corollary. *For an iSSPUDE*$_{s_0}$ *MDP with the optimal goal-probability function* P^**, the optimal goal-conditional value function* $[V^*|P^*]$ *satisfies*

$$[V^*|P^*](s) = 0 \; \forall s \text{ s.t. } P^*(s) = 0 \tag{5.21}$$

$$[V^*|P^*](s) = \min_{a \in \mathcal{A}^{P^*}} \left\{ \sum_{s' \in \mathcal{S}} \mathcal{C}(s, a, s') + \frac{\mathcal{T}(s, a, s') P^*(s')}{P^*(s)} [V^*|P^*](s') \right\}$$

An optimal solution to an iSSPUDE$_{s_0}$ *MDP is a policy* $\pi^*_{s_0}$ *greedy w.r.t.* P^* *and* $[V^*|P^*]$*. At least one such policy is s.d.M..* ♢

Algorithm 5.3: IVI

```
1 Input: iSSPUDE$_{s_0}$ MDP $M$
2 Output: an optimal policy $\pi^*_{s_0}$ rooted at $s_0$

    1. Find $P^*$ by running VI$_{MP}$ on MAXPROB MDP $M_{MP}$ derived from $M$
    2. Find $[V^*|P^*]$ by running VI$_{SSP}$ on SSP$_{s_0}$ MDP $M^{P^*}$ derived from $M$ and $P^*$

  return $\pi^*_{s_0}$ greedy w.r.t. $P^*$ and $[V^*|P^*]$
```

Algorithm 5.4: IHS

```
1 Input: iSSPUDE$_{s_0}$ MDP $M$
2 Output: an optimal policy $\pi^*_{s_0}$ rooted at $s_0$

    1. Find $P^*_{s_0}$ by running FRET initialized with admissible heuristic $P_0$ on MAXPROB MDP $M_{MP}$
       derived from $M$
    2. Find $[V^*_{s_0}|P^*_{s_0}]$ by running FIND-AND-REVISE initialized with an admissible heuristic $V_0$
       on SSP$_{s_0}$ MDP $M^{P^*_{s_0}}$ derived from $M$ and $P^*_{s_0}$

  return $\pi^*_{s_0}$ greedy w.r.t. $P^*_{s_0}$ and $[V^*_{s_0}|P^*_{s_0}]$
```

Putting It All Together. As just showed, solving an iSSPUDE$_{s_0}$ MDP ultimately reduces to solving a derived MAXPROB MDP and then, using its optimal value function P^*, a specially constructed SSP$_{s_0}$ MDP. The overall process is captured in a procedure called IVI (**I**nfinite-penalty **V**alue **I**teration), whose high-level pseudocode is presented in Algorithm 5.3.

Heuristic Search for iSSPUDE$_{s_0}$ MDPs

Our heuristic search schema IHS (**I**nfinite-penalty **H**euristic **S**earch) for iSSPUDE$_{s_0}$ MDPs, presented in Algorithm 5.4, follows the same principles as IVI but uses heuristic search instead of full-fledged VI in each stage. There are two main differences between IHS and IVI. The first one is that IHS produces functions $P^*_{s_0}$ and $[V^*_{s_0}|P^*_{s_0}]$ that are guaranteed to be optimal only over the states visited by some optimal policy $\pi^*_{s_0}$ starting from the initial state s_0. Accordingly, the SSP$_{s_0}$ MDP $M^{P^*_{s_0}}$ required to solve a given iSSPUDE$_{s_0}$ MDP M is constructed only over states reachable from s_0 by at least one $P^*_{s_0}$-optimal policy. Second, IHS requires two admissible heuristics, one (P_0)

being an *upper* bound on P^* and the other (V_0) being a *lower* bound on $[V^*_{s_0}|P^*_{s_0}]$.

5.4.5 Equivalence of Optimization Criteria

We have presented three ways of dealing with MDPs having unavoidable dead ends:

- Solving them according to the MAXPROB criterion.
- Solving them according to the fSSPUDE$_{s_0}$ criterion.
- Solving them according to the iSSPUDE$_{s_0}$ criterion.

Although the proposed algorithms for MAXPROB and iSSPUDE$_{s_0}$ MDPs are significantly more complicated than those for MDPs with a finite dead-end penalty, intuition tells us that solving a given tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ under all three criteria should yield "similar" policies, provided that in the finite-penalty case the penalty D is very large. This intuition can be formalized as a theorem:

Theorem 5.20. *For a given tuple of MDP components $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ satisfying the conditions of the SSP$_{s_0}$ MDP definition (2.19), there exists a finite penalty D_{thres} s.t. every optimal s.d.M. policy of every fSSPUDE$_{s_0}$ MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, D, s_0 \rangle$ for any $D > D_{thres}$ is optimal for the MAXPROB MDP derived from this fSSPUDE$_{s_0}$ MDP as well.* ♢

Proof. See the Appendix. The proof's main insight is the fact that a very large penalty, even a finite one, is such a strong deterrent against visiting dead ends that an optimal policy for an fSSPUDE$_{s_0}$ MDP does everything possible to avoid them, thereby maximizing the probability of reaching the goal. ■

Corollary. For an fSSPUDE$_{s_0}$ MDP with a dead-end penalty D exceeding the threshold penalty D_{thres} for the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ of this MDP's components, for every optimal policy π^* and $s \in \mathcal{S}$, π^*'s goal-probability value function P^{π^*} equals P^* for the iSSPUDE$_{s_0}$ MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, \infty, s_0 \rangle$. ♢

As a consequence, if we choose $D > D_{thres}$, we can be sure that at any given state s, all optimal

(V<span style="font-variant:small-caps">F</span>*-greedy) policies of the resulting fSSPUDE$_{s_0}$ MDP will have the same probability of reaching the goal, and this probability is $P^*(s)$, the optimal one under the *MAXPROB* and *iSSPUDE*$_{s_0}$ criteria as well.

This prompts a question: what can be said about the goal-probability function P of an fSSPUDE$_{s_0}$ MDP's optimal policies if $D \leq D_{thres}$? Unfortunately, in this case different VF-optimal policies may not only be suboptimal in terms of P, but, even for a fixed D, each VF-optimal policy may have a different, arbitrarily low chance of reaching the goal. For example, consider an MDP with three states: s_0 (the initial state), s_d (a dead end), and s_g (a goal). Action a_d leads from s_0 to s_d with probability 0.5 and to s_g with probability 0.5 and costs 1 unit. Action a_g leads from s_0 to s_g also with probability 1, and costs 3 units. Finally, suppose we solve this setting as an fSSPUDE$_{s_0}$ MDP with $D = 4$. It is easy to see that both policies, $\pi(s_0) = a_d$ and $\pi(s_0) = a_g$, have the same expected cost, 3. However, the former reaches the goal only with probability 0.5, while the latter always reaches it. The ultimate reason for this discrepancy is that the policy evaluation criterion of *fSSPUDE*$_{s_0}$ is oblivious to policies' probabilities of reaching the goal, and optimizes for this characteristic only indirectly, via policies' expected costs.

The equivalence of optimization criteria, as stated in Theorem 5.20, implies two ways of finding a MAXPROB-optimal policy in the unavoidable-dead-end case: either by directly solving the corresponding MAXPROB instance or by choosing a sufficiently large penalty D and solving the finite-penalty fSSPUDE$_{s_0}$ MDP. We are not aware of a principled way to choose a finite D that would make optimal solutions in these two cases coincide, but it is often easy to guess by inspecting the MDP. Thus, although the latter method gives no a-priori guarantees, it frequently yields a MAXPROB-optimal policy in practice. We glean insights into which of the two methods of computing a MAXPROB-optimal policy is more efficient and under what circumstances by conducting an empirical study. Before describing it, we introduce one more MDP class, which subsumes all others proposed in this dissertation as well as several known *classical* planning problems.

5.4.6 Stochastic Simple Longest Path MDPs: the Ultimate Goal-Oriented Problems

Although some goal-oriented MDP types we have considered are at least as general as SSP$_{s_0}$ MDPs, every single one of them still imposes some restrictions on either the reward structure or existence

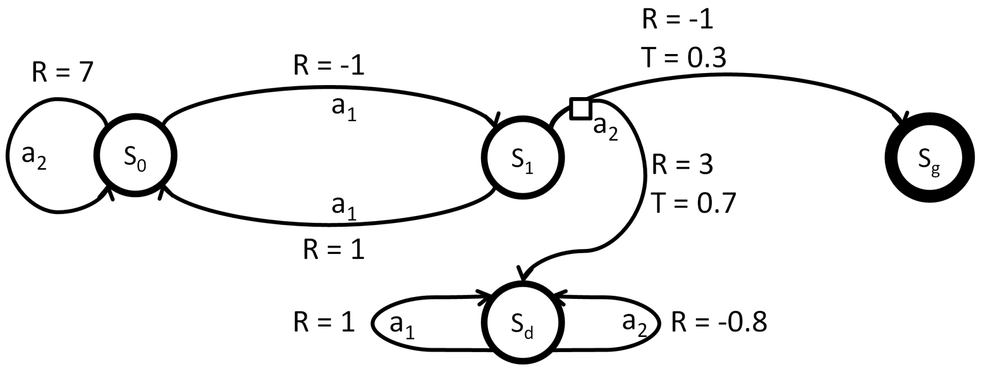


Figure 5.4: An example Stochastic Simple Longest-Path MDP.

of proper policies for the scenarios it models. This makes one wonder: can we formulate a well-defined optimization criterion for a class of goal-oriented MDPs completely free of such restrictions and find an optimal policy for it?

Figure 5.4 gives an overview of such MDPs' possible peculiarities:

- The MDP in this figure has a goal but no proper policy w.r.t. the initial state.
- Entering a dead end in it yields a higher reward than going to the goal (note the positive-reward self-loop of state s_d).
- In fact, if measured in terms of expected reward, the values of some policies for it are ill-defined: even though traversing from s_0 to s_1 and back yields a net reward of 0, the total expected reward of a policy that traps an agent in this cycle oscillates between -1 and 0 (if the agent starts in s_0) without settling down on either value.
- Last but not least, loitering in a non-dead-end state in such MDPs may yield a higher reward than other courses of action; a policy that chooses action a_2 in s_0 serves as an example.

To a first approximation, defining the notion of a "best" policy for goal-oriented MDPs with arbitrary reward functions is complicated by the same factor as for iSSPUDE$_{s_0}$ problems. If no policy leads to the goal with 100% probability, i.e., stops accumulating reward after a finite number of steps in expectation, expected reward alone becomes meaningless for evaluating policies' utility. The iSSPUDE$_{s_0}$ MDP definition suggests a way around this difficulty by defining policy utility as expected reward over only those of the policy's trajectories that reach the goal. An attempt to equip unrestricted goal-oriented problems with this optimization criterion has been made before in the definition of Stochastic Safest- and Shortest-Path MDPs (*S*^{3}*P*) [96].

Indeed, *iSSPUDE*$_{s_0}$'s policy evaluation criterion makes the value of every policy finite, and hence amenable to meaningful comparisons, even if the reward function is arbitrary. Paradoxically, however, it does not guarantee the existence of an optimal policy. For instance, for the MDP in Figure 5.4, consider an infinite family of policies in which the n-th member policy executes a_2 in s_0 n times, then picks a_1 in s_0, and finally chooses a_2 in s_1. All policies in this family have the optimal goal-probability value function for this MDP, $P^*(s_0) = 0.3$. However, the goal-conditional expected reward $\textsc{Vi}^*(s_0)$ of the n-th policy member is $7n - 2$, a value that grows without bound with increasing n. Thus, S^{3}P MDPs are undecidable.

The key insight for amending S^{3}P's optimization criterion is the observation that in the pathological example above, the policies are *non-Markovian*. Viewed differently, for goal-oriented MDPs with unconstrained rewards under, s.d.M. policies evaluated with $[V^\pi|P^\pi]$ are generally dominated by non-Markovian ones. However, if we consider only s.d.M. policies as solution candidates, the issue goes away, since there are finitely many of them and their value functions are comparable with each other. WE formalize this intuition in the following definition:

Definition 5.16. ***SSLP*$_{s_0}$ *MDP.*** *A stochastic simple longest-path MDP (SSLP*$_{s_0}$ *MDP) is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$*, where* $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}$*, and* s_0 *are as in the SSP*$_{s_0}$ *MDP definition (2.19).* ♣

An optimal solution to an SSLP$_{s_0}$ MDP is an s.d.M. policy $\pi^*_{s_0}$ that maximizes Equation 5.17 for all states it reaches from s_0 under the $\prec$-ordering:

$$\pi^*_{s_0} = \underset{\prec \pi}{\text{argmax}}\, \textsc{Vi}^\pi(s_0)$$

To emphasize it once again, the only difference between *iSSPUDE*$_{s_0}$'s and *SSLP*$_{s_0}$'s optimization criteria is *SSLP*$_{s_0}$'s consideration of exclusively s.d.M. policies.

Solving SSLP$_{s_0}$ *MDPs*

Our algorithms for solving SSLP$_{s_0}$ problems, SVI (SSLP$_{s_0}$ Value Iteration, Algorithm 5.5) and SHS (SSLP$_{s_0}$ Heuristic Search, Algorithm 5.6) are only partly based on dynamic programming. Specifically, dynamic programming in the form of synchronous VI or heuristic search can be used to find the goal-probability value function P^* or its part $P^*_{s_0}$ over the states reachable from s_0 by at least one P^*-optimal policy. However, computing $[V^*|P^*]$ afterwards cannot be done in the same way as for iSSPUDE$_{s_0}$ MDPs, because, unlike for these problems, the MAXPROB-optimal derivative of a SSLP$_{s_0}$ MDP is generally not as SSP$_{s_0}$ MDP, and we are trying to find an s.d.M. optimal solution it, not a globally optimal one.

Algorithm 5.5: SVI

```
1 Input: SSLP$_{s_0}$ MDP $M$
2 Output: an optimal s.d.M. policy $\pi^*_{s_0}$ rooted at $s_0$

    1. Find $P^*$ by running VI$_{MP}$ on MAXPROB MDP $M_{MP}$ derived from $M$
    2. For each s.d.M. $\pi$, compute $P^\pi$ and, if $P^\pi = P^*$, compute $[V^\pi|P^\pi]$

  return s.d.M. $\pi^*_{s_0} = argmax_{P^\pi = P^*}[V^\pi|P^\pi](s_0)$
```

Algorithm 5.6: SHS

```
1 Input: SSLP$_{s_0}$ MDP $M$
2 Output: an optimal s.d.M. policy $\pi^*_{s_0}$ rooted at $s_0$

    1. Find $P^*_{s_0}$ by running FRET initialized with admissible heuristic $P_0$ on MAXPROB MDP $M_{MP}$
       derived from $M$
    2. For each s.d.M. $\pi$, compute $P^\pi_{s_0}$ and, if $P^\pi_{s_0} = P^*_{s_0}$ over states in $G^\pi_{s_0}$, compute $[V^\pi_{s_0}|P^\pi_{s_0}]$

  return s.d.M. $\pi^*_{s_0} = argmax_{P^\pi = P^*}[V^*_{s_0}|P^*_{s_0}](s_0)$
```

Instead, both algorithms simply enumerate s.d.M. policies π (there are finitely many of them), evaluate P^π for each of them using policy evaluation (similar to Algorithm 2.1) and, for those whose P^π agrees with the optimal goal-probability function, compute $[V^\pi|P^\pi]$. A policy with the highest $[V^\pi|P^\pi]$ is returned as a solution.

It is easy to see that both algorithms are exponential in the MDP size, since they generally iterate over all s.d.M. policies. While seemingly inefficient, asymptotically it may be the best we can do:

Theorem 5.21. *SSLP$_{s_0}$ is NP-hard.* $\diamondsuit$

Proof. *SSLP$_{s_0}$* trivially contains the class of deterministic simple longest-path problems, known to be NP-hard [89]. ■

SSLP$_{s_0}$ and Other Classes

Theorem 24 serves as the motivation for *SSLP$_{s_0}$*'s name. Indeed, this MDP class is a probabilistic generalization of the simple longest-path problems, which consist in finding the heaviest loop-free path between two vertices in a graph. Just like simple longest-path problems include shortest-path problems as a special case, *SSLP$_{s_0}$* subsumes SSP$_{s_0}$ MDPs. More than that, *SSLP$_{s_0}$* encompasses all other classes we have introduced in this dissertation and, by transitivity, their subclasses, such as *IHDR*, *FH*, *POSB*, and *NEG*. To our knowledge, *SSLP$_{s_0}$* is the most general decidable class of planning problems studied in AI. A Venn diagram of the corresponding MDP class hierarchy is presented in Figure 1.2.

5.4.7 Experimental Results

The objective of our experiments was to determine which of the two methods for finding a probability-optimal policy for a goal-oriented problem with unavoidable dead ends is more efficient in practice, solving this problem as a MAXPROB MDP or as an fSSPUDE$_{s_0}$ MDP with a very large D parameter. To do a fair comparison between these approaches, we employed very similar algorithms to handle them. For *fSSPUDE$_{s_0}$* as well as *MAXPROB*, the most efficient optimal solution methods are heuristic search techniques, so in the experiments we used only algorithms of this type.

To solve fSSPUDE$_{s_0}$ MDPs, we used the implementation of the LRTDP algorithm from the miniGPT package [10]. As a source of admissible state value estimates, we chose the maximum of the atom-min-forward heuristic [42] and SIXTHSENSE (Section 3.6) [57, 58]. The sole purpose of the latter was to quickly and soundly identify many of the dead ends and assign the value of D to them. For MAXPROB MDPs, we used LRTDP with a SIXTHSENSE-derived heuristic, the same setup as in the experiments for GSSP$_{s_0}$ MDPs described in Section 5.3.6.

Our benchmarks were problems 1 through 6 of the Exploding Blocks World domain from IPPC-2008 and problems 1 through 15 of the Drive domain from IPPC-2006. Most problems in both domains have unavoidable dead ends and thus are appropriately modeled by fSSPUDE$_{s_0}$ and MAXPROB MDPs. To set the penalty D for the fSSPUDE$_{s_0}$ formulation, we examined each problem and tried to come up with an intuitive, easily justifiable value for it. For all problems, $D = 500$ yielded a policy that was optimal under both the VF (Equation 5.11) and MAXPROB criteria.

For all the benchmarks, solving their MAXPROB and fSSPUDE$_{s_0}$ versions, with $D = 500$ in the latter case, using the above heuristic search instantiations yielded the same qualitative outcome. In terms of speed, solving the fSSPUDE$_{s_0}$ versions was at least an order of magnitude faster than solving the MAXPROB versions. The difference in used memory was occasionally smaller, but only because the algorithms for both classes visited nearly the entire state space reachable from s_0 on some problems. Moreover, in terms of memory as well as speed, the difference between solving the fSSPUDE$_{s_0}$ and MAXPROB formulations was the largest (that is, solving MAXPROB MDPs was comparatively the *least* efficient) when the given problem had $P^*(s_0) = 1$, i.e., the MDP had no dead ends at all or had only avoidable ones.

Although surprising at the first glance, these performance patterns have a fundamental reason. Recall that FRET algorithms, those used for solving the MAXPROB versions, rely on the ELIMINATE-TRAPS operator. For every encountered fixed-point value function of Bellman backup, it needs to traverse this value function's transition graph involving all actions greedy w.r.t. it starting from s_0 (lines 27-46 of Algorithm 5.1). Also, FRET needs to be initialized with an admissible heuristic (in our experiments, based on SIXTHSENSE), which assigns the value of 0 to states it believes to be dead ends and 1 to the rest.

Now, consider how FRET operates on a MAXPROB MDP that does not have any dead ends — the kind of MAXPROB MDPs that, as our experiments show, is most problematic. For such a MAXPROB, there exists only one admissible heuristic function, $P_0(s) = 1$ for all s, because for these problems $P^*(s) = 1$ for all s and an admissible heuristic needs to satisfy $P_0(s) \geq P^*(s)$

everywhere. Thus, the heuristic FRET starts with is actually the optimal goal-probability function, and, as a consequence, is a fixed point of the local Bellman backup operators for all states in that value function's greedy graph rooted at s_0. Therefore, before concluding that P_0 is optimal, FRET needs to build its greedy graph. Observe, however, that since P_0 is 1 everywhere, this transition graph includes every state reachable from s_0, and uses every action in this part of the state space! Building and traversing it is very expensive if the MDP is large.The same performance bottleneck, although to a lesser extent, can also be observed on instances that do have unavoidable dead ends: building large transition graphs significantly slows down FRET even when P^* is far from being 1 everywhere.

This reasoning explains why solving MAXPROB MDPs is slow, but by itself does not clarify why solving fSSPUDE$_{s_0}$ is fast in comparison. For instance, we might expect the performance of FIND-AND-REVISE algorithms on fSSPUDE$_{s_0}$ to suffer in the following situations. Suppose state s is a dead end not avoidable from s_0 by any policy. This means that $V^*(s) = D$ under the finite-penalty optimization criterion, and that s is reachable from s_0 by any optimal policy. Thus, FIND-AND-REVISE will not halt until the value of s under the current value function reaches D. Furthermore, suppose that our admissible heuristic V_0 initializes the value of s with 0 — this is one of the possible admissible values for s. Finally, assume that all actions in s lead back to s with probability 1 and have the cost of 1 unit. In such a situation, a FIND-AND-REVISE algorithm will need to update the value of s D times before convergence. Clearly, this will make the performance of FIND-AND-REVISE very bad if the chosen value of D is very large. This raises the question: was solving fSSPUDE$_{s_0}$ MDPs in the above experiments so much more efficient than solving MAXPROB formulations due to our choice of (a rather small) value for D?

To dispel these concerns, we solved fSSPUDE$_{s_0}$ instances of the aforementioned benchmarks with $D = 5 \cdot 10^8$ instead of 500. On all of the 21 problems, the increase in speed compared to the case of fSSPUDE$_{s_0}$ with $D = 500$ was no more than a factor of 1.5. The cause for such a small discrepancy is the fact that, at least on our benchmarks, FIND-AND-REVISE almost never runs into the pathological case of having to update the value of a dead many times as described above, thanks to the atom-min-forward and, most importantly, SIXTHSENSE heuristics. They identify the majority of dead ends encountered by LRTDP and immediately set their values to D. Thus, instead of spending many updates on such states, LRTDP gets their optimal values in just one step. To test this explanation, we disabled these heuristics and assigned the value of 0 to all states at initialization. As predicted, the solution time of the fSSPUDE$_{s_0}$ instances skyrocketed by orders of magnitude.

The presented results appear to imply an unsatisfying fact: on MAXPROB MDPs that are in fact dead-end-free SSP$_{s_0}$ problems, FRET is not nearly as efficient as the algorithmic schema for *SSP*$_{s_0}$, FIND-AND-REVISE. The caveat of the *SSP*$_{s_0}$ algorithms, however, is that they pay for efficiency by *assuming* the existence of proper solutions. FRET, on the other hand, implicitly *proves* the existence of such solutions, and is therefore theoretically more robust.

5.4.8 Summary

The material in this section has covered the full range of assumptions one can make about the properties of dead ends in goal-oriented MDPs. The *SSPADE*$_{s_0}$ class assumes them to be present but entirely avoidable from the initial state. The theory of *SSPADE*$_{s_0}$ is not very different from that of SSP$_{s_0}$ problems. If the dead ends are present and unavoidable but the penalty for vising them can be quantified by a finite number, as in the *fSSPUDE*$_{s_0}$ class, the MDPs can also be solved largely using the existing methods. However, unavoidable dead ends with an infinite penalty, modeled by iSSPUDE$_{s_0}$ MDPs, require the formulation of a new optimization criterion and invalidate the previous approaches. Finally, *SSLP*$_{s_0}$ is a class that does not make any assumptions about the presence of dead ends, the penalty for entering them, and the action cost structure, but, because of that, is NP-hard. The optimal solutions of all of these classes are, in some sense equivalent: they have the same probability of reaching the goal (provided that the dead-end penalty in fSSPUDE$_{s_0}$ is high enough).

5.5 Related Work

Although goal-oriented MDPs with dead ends have been used as benchmarks in several IPPCs (see, e.g., [17]), there have been rather few attempts to formally define and study them from a theoretical standpoint. One such attempt is the work on the notion of “probabilistic interestingness” [65], which observes that the presence of dead ends makes goal-oriented MDPs less amenable to replanning approaches such as FFReplan [99]. Other existing approaches for MDPs with dead ends have mostly concentrated on optimizing what we have defined as the *MAXPROB* criterion [63, 67, 64]. However, these works have not studied the theoretical properties of solving *MAXPROB* in detail. In particular, the key realization that arbitrarily initialized VI in general cannot solve it optimally due to the presence of traps, along with heuristic search algorithms for doing so, has been missing.

In spite of the dearth of theoretical analysis of MDPs with dead ends, implementations of several heuristic search algorithms, e.g., LRTDP, for problems in which hitting a dead end incurs a finite penalty have been publicly available as part of the miniGPT package [10]. Nonetheless, little has been known so far about the dependence of their speed and solution quality on the dead-end penalty parameter. The latter, as our experiments show, can affect these algorithms' performance dramatically if they are not equipped with a heuristic for identifying dead ends.

The $iSSPUDE_{s_0}$ class can be viewed as a case of multi-objective MDPs, which model problems with several competing objectives, e.g., total time, monetary cost, etc. [21, 98]. Generally, their solutions are Pareto sets of all non-dominated policies. Unfortunately, such solutions are impractical due to high computational requirements of algorithms for finding them. Specifically for the case of goal-probability and expected-cost value functions, one such algorithm, MOLAO*, has been proposed in [18]. A criterion related to $iSSPUDE_{s_0}$'s has also been studied in robotics [53]. Around the time when $iSSPUDE_{s_0}$ was introduced [60], a nearly identical class was independently formulated in [96]. The algorithm in [96] for solving $iSSPUDE_{s_0}$ is similar to our IVI scheme, but that work describes no analogue for the more efficient heuristic search algorithm IHS proposed here and in [60].

5.6 Future Research Directions

The theory of goal-oriented MDPs presented in this dissertation has centered around policy utility measured by expected reward/cost. While it is an intuitive and mathematically convenient criterion, generally it gives a good idea of a policy's average performance only if the policy is executed many times. Indeed, since uncertainty makes it impossible to predict the outcome of an action before using it, executing a policy may force the agent to use a different sequence of actions every time, and hence get a different reward. Therefore, if an agent wants to use a policy just once or twice, or is simply risk-averse, it may be interested in the policy's variance, i.e., its propensity to yield rewards far above or below the mean. As a simple example of a situation where policy variance matters, consider selecting between two lotteries. In one, an agent can get a payoff of either \$6 or \$8 with equal probability, In another, it can get a payoff of either \$30 or -\$10 (i.e., incur a loss) with equal probability. Despite the fact that the expected payoff in the first lottery (\$7) is lower than in the second (\$10), many people will prefer the first if given only one chance to play either, because the variance in its outcomes will never cause them to lose money.

Although there has been a fair amount of work on defining policy utility to account for variance (e.g., [46], [68], [91], [69], and [74]), all or most of it, to our knowledge, applies only to FH and IHDR MDPs. In the meantime, these MDP types are merely special cases of goal-oriented classes, let alone of goal-oriented classes with dead ends. Research on optimizing for variance in the presence of goals and dead ends would increase the applicability of the MDPs studied in this chapter to real-world scenarios, where consistent performance of the chosen policies is almost always an important concern.

5.7 Summary

The work presented in this chapter lifts the restrictions imposed by the sole existing goal-oriented MDP class known to date, *SSP*, on the scenarios that can be modeled. In particular, problems with dead ends, i.e., states with no trajectories to the goal, violate *SSP*'s requirement of proper policy existence. Settings where one is interested in the optimal probability, as opposed to expected cost, of reaching the goal cannot be cast as SSP MDPs either.

Such probability maximization problems can be modeled by the *MAXPROB* class that we have introduced, which, in turn, is a subclass of $GSSP_{s_0}$, also presented in this chapter. The mathematical properties of these two MDP types are complex and prevent them from being solvable by the existing value iteration and heuristic search algorithms. To tackle $GSSP_{s_0}$, we have developed a new heuristic search framework called FRET.

For scenarios with dead ends, we have considered another three tractable MDP classes, $SSPADE_{s_0}$, $fSSPUDE_{s_0}$, and $iSSPUDE_{s_0}$. The first of them assumes that dead ends are avoidable from the initial state, the second — that entering them incurs a finite penalty, and the third one — that the dead ends are not only unavoidable but also infinitely costly. While $SSPADE_{s_0}$ and $fSSPUDE_{s_0}$ can be solved with minor modifications of existing techniques, $iSSPUDE_{s_0}$ requires a completely new approach.

The FRET framework for $GSSP_{s_0}$ plays a pivotal role in solving $iSSPUDE_{s_0}$, but is only of the components of the full algorithm.

Last but not least, we have proposed $SSLP_{s_0}$, an MDP class that includes all the known deterministic and probabilistic planning formulations studied in AI. Although general, $SSLP_{s_0}$ is NP-hard and is thus of mostly theoretical value.

Our mathematical analysis of goal-oriented MDP is valid for policy utility expressed as expected cost of getting to the goal. However, in many applications policies' variance is important as well. Developing variance-aware optimization criteria and algorithms for goal-oriented MDPs is a topic for future work.

Chapter 6

CONCLUSION

Planning under uncertainty, an indispensable area of AI, is facing two major challenges. The techniques it offers for solving its main set of models, the Markov decision processes (MDPs), are not sufficiently scalable to handle many realistically-sized problems. The fundamental reason for this shortcoming is the way existing MDP solvers operate: they analyze every configuration of the world separately, without generalizing plans across similar situations. The second challenge to probabilistic planning is the lack of expressiveness of the available MDP formulations. They fail to properly model many problem aspects that we consider common and natural, such as irrecoverable catastrophic failures. These two factors significantly limit the adoption of MDPs as a practical problem-solving tool.

This dissertation advances state of the art in addressing both of these challenges by making two broad groups of contributions. The first group consists of highly scalable algorithms for solving various types of MDPs. For goal-oriented MDPs, these algorithms are built on a novel method for extracting the problem structure and using it to solve planning problems by reasoning about their high-level regularities, not the low-level world states. RETRASE[55, 58], GOTH[56, 58], and SIXTHSENSE[57, 58] are approaches based on this underlying idea. RETRASE is a standalone MDP solver that, thanks to the use of generalization, outperforms the best of its competition. GOTH is an informative heuristic that brings the power of automatic abstract reasoning to the numerous class of heuristic search techniques. SIXTHSENSE can serve as a module in nearly any existing solver and help it identify dead-end states — those from which reaching the goal is impossible and that can otherwise drain a planner's computational resources in vain. For reward-oriented settings, this dissertation proposes two other performant solvers, GLUTTON [54] and GOURMAND [59], centered around the strategy of reverse iterative deepening. Their theoretical and empirical evaluation points out the crucial role of convergence criterion in their success — a heretofore underappreciated aspect of planning techniques. As a testimony to the merit of their core ideas, GLUTTON was the close runner-up at the 2011 International Probabilistic Planning Competition and GOURMAND topped that contest's winner.

The second set of this dissertation's contributions are new problem classes extending the modeling capabilities of the MDP paradigm. In particular, the presented work removes the restrictions of the sole known goal-oriented MDP class, the stochastic shortest-path (SSP) problems, exploring the mathematical properties of the resulting formulations and proposing optimal algorithms for them. The extensions begin with the GSSP$_{s_0}$ MDP class [61] and its subclass *MAXPROB*, which serve as the theoretical basis for the rest. The main results of this line of research [60] are classes that explore various assumptions about the existence of dead ends, which are completely disallowed in SSP MDPs. The proposed SSPADE$_{s_0}$, fSSPUDE$_{s_0}$, and iSSPUDE$_{s_0}$ MDPs describe scenarios where dead ends are, respectively, present but avoidable, unavoidable but carry a finite penalty, and, finally, unavoidable and infinitely damaging. The *SSLP*$_{s_0}$ class is the goal-oriented model devoid of any restrictions but intractable as a result.

The two sets of the dissertation's contributions complement each other. While the second develops the theory and optimal algorithms for MDPs with dead ends, the first introduces effective techniques for identifying dead-end states, e.g., SIXTHSENSE, thereby helping make the solvers for goal-oriented MDPs efficient in practice.

Although the dissertation contributes to several aspects of contemporary MDP knowledge, as well as summarizes the existing theory of this framework [71], MDPs have fundamental limitations that it does not address — the assumptions of full world observability and full awareness of the world dynamics. Partially observable MDPs and reinforcement learning models lift these limitations, and, as we hope, will greatly benefit from the extension of the ideas presented here.

BIBLIOGRAPHY

[1] Douglas Aberdeen, Sylvie Thiébaux, and Lin Zhang. Decision-theoretic military operations planning. In *Proceedings of the Second International Conference on Automated Planning and Scheduling*, pages 402–412, 2004.

[2] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[3] Richard Bellman. *Dynamic Programming*. Prentice Hall, 1957.

[4] D. Bertsekas and J. Tsitsiklis. Analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580595, 1991.

[5] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[7] Venkata Deepti Kiran Bhuma and Judy Goldsmith. Bidirectional LAO* algorithm. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 980–992, 2003.

[8] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding Klondike Solitaire with Monte-Carlo planning. In *Proceedings of the Seventh International Conference on Automated Planning and Scheduling*, 2009.

[9] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[10] B. Bonet and H. Geffner. mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, 24:933–944, 2005.

[11] Blai Bonet. Modeling and solving sequential decision problems with uncertainty and partial information. Technical Report R-315, Department of Computer Science, University of California, Los Angeles, 2004.

[12] Blai Bonet and Hector Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1233–1238, 2003.

[13] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proceedings of the First International Conference on Automated Planning and Scheduling*, pages 12–21, 2003.

[14] Blai Bonet and Hector Geffner. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *Proceedings of the Fourth International Conference on Automated Planning and Scheduling*, pages 3–23, 2006.

[15] Blai Bonet and Hector Geffner. Action selection for mdps: Anytime AO* versus UCT. In *Proceedings of the Twenty-sixth AAAI Conference on Artificial Intelligence*, 2012.

[16] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.

[17] Daniel Bryce and Olivier Buffet. International planning competition, uncertainty part: Benchmarks and results. In *http://ippc-2008.loria.fr/wiki/images/0/03/Results.pdf*, 2008.

[18] Daniel Bryce, William Cushing, and Subbarao Kambhampati. Probabilistic planning is multi-objective! Technical Report ASU CSE TR-07-006, Department of Computer Science and Engineering, Arizona State University, 2007.

[19] Olivier Buffet and Douglas Aberdeen. The factored policy-gradient planner. *Artificial Intelligence*, 173(5-6):722–747, 2009.

[20] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[21] Krishnendu Chatterjee, Rupak Majumdar, and Thomas A. Henzinger. Markov decision processes with multiple objectives. In *Proceedings of Twenty-third Annual Symposium on Theoretical Aspects of Computer Science*, pages 325–336, 2006.

[22] Peter Clark and Tim Niblett. The CN2 induction algorithm. In *Machine Learning*, pages 261–283, 1989.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2001.

[24] Peng Dai and Judy Goldsmith. LAO*, RLAO*, or BLAO*. In *Proceedings of AAAI workshop on heuristic search*, pages 59–64, 2006.

[25] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[26] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[27] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[28] Zohar Feldman and Carmel Domshlak. Simple regret optimization in online planning for markov decision processes. *arXiv:1206.3382v2*, 2012.

[29] Zhengzhu Feng and Eric A. Hansen. Symbolic heuristic search for factored Markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 455–460, 2002.

[30] Zhengzhu Feng, Eric A. Hansen, and Shlomo Zilberstein. Symbolic generalization for on-line planning. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 209–216, 2003.

[31] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, volume 19, pages 17–37, 1982.

[32] J. Foss, N. Onder, and D. Smith. Preventing unrecoverable failures through precautionary planning. In *ICAPS'07 Workshop on Moving Planning and Scheduling Systems into the Real World*, 2007.

[33] Sylvain Gelly and David Silver. Achieving master level play in 9x9 computer Go. In *Proceedings of the Twenty-third AAAI Conference on Artificial Intelligence*, pages 1537–1540, 2008.

[34] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

[35] Judy Goldsmith, Michael L. Littman, and Martin Mundhenk. The complexity of plan existence and evaluation in probabilistic domains. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, 1997.

[36] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, 1995.

[37] Charles Gretton and Sylvie Thiébaux. Exploiting first-order regression in inductive policy selection. In *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence*, pages 217–225, 2004.

[38] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1003–1010, 2003.

[39] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for finite MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2003.

[40] Eric A. Hansen. Suboptimality bounds for stochastic shortest path problems. In *Proceedings of the Twenty-seventh Conference on Uncertainty in Artificial Intelligence*, pages 301–310, 2011.

[41] Eric A. Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129:35–62, 2001.

[42] P. Haslum and H. Geffner. Admissible heuristic for optimal planning. In *AIPS*, page 140149, 2000.

[43] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.

[44] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[45] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[46] Stratton C. Jaquette. Markov decision processes with a new optimality criterion: Discrete time. *The Annals of Statistics*, 1(3):496–505, 1973.

[47] S. Kambhampati, S. Katukam, and Q. Yong. Failure driven dynamic search control for partial order planners: an explanation based approach. *Artificial Intelligence*, 88:253–315, 1996.

[48] Philipp Keller, Shie Mannor, and Doine Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *ICML'06*, pages 449–456, 2006.

[49] Thomas Keller and Patrick Eyerich. Probabilistic planning based on UCT. In *Proceedings of the Tenth International Conference on Automated Planning and Scheduling*, 2012.

[50] Emil Keyder and Hector Geffner. The HMDP planner for planning with probabilities. In *Sixth International Planning Competition at ICAPS'08*, 2008.

[51] C. Knoblock, S. Minton, and O. Etzioni. Integrating abstraction and explanation-based learning in PRODIGY. In *Ninth National Conference on Artificial Intelligence*, 1991.

[52] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of the Seventeenth European Conference on Machine Learning*, pages 282–293, 2006.

[53] S. Koenig and Y. Liu. The interaction of representations and planning objectives for decision-theoretic planning tasks. In *Journal of Experimental and Theoretical Artificial Intelligence*, volume 14, pages 303–326, 2002.

[54] Andrey Kolobov, Peng Dai, Mausam, and Daniel S. Weld. Reverse iterative deepening for finite-horizon MDPs with large branching factors. In *Proceedings of the Tenth International Conference on Automated Planning and Scheduling*, 2012.

[55] Andrey Kolobov, Mausam, and Daniel S. Weld. ReTrASE: Integrating paradigms for approximate probabilistic planning. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[56] Andrey Kolobov, Mausam, and Daniel S. Weld. Classical planning in MDP heuristics: with a little help from generalization. In *Proceedings of the Eighth International Conference on Automated Planning and Scheduling*, pages 97–104, 2010.

[57] Andrey Kolobov, Mausam, and Daniel S. Weld. SixthSense: Fast and reliable recognition of dead ends in MDPs. In *Proceedings of the Twenty-fourth AAAI Conference on Artificial Intelligence*, 2010.

[58] Andrey Kolobov, Mausam, and Daniel S. Weld. Discovering hidden structure in factored MDPs. *Artificial Intelligence*, 189:19–47, 2012.

[59] Andrey Kolobov, Mausam, and Daniel S. Weld. LRTDP vs. UCT for online probabilistic planning. In *Proceedings of the Twenty-sixth AAAI Conference on Artificial Intelligence*, 2012.

[60] Andrey Kolobov, Mausam, and Daniel S. Weld. A theory of goal-oriented MDPs with dead ends. In *Proceedings of the Twenty-eighth Conference on Uncertainty in Artificial Intelligence*, 2012.

[61] Andrey Kolobov, Mausam, Daniel S. Weld, and Hector Geffner. Heuristic search for generalized stochastic shortest path MDPs. In *Proceedings of the Ninth International Conference on Automated Planning and Scheduling*, 2011.

[62] R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97109, 1985.

[63] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelth National Conference on Artificial Intelligence*, pages 1073–1078, 1994.

[64] Iain Little, Douglas Aberdeen, and Sylvie Thiébaux. Prottle: A probabilistic temporal planner. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1181–1186, 2005.

[65] Iain Little and Sylvie Thiebaux. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, 2007.

[66] Michael L. Littman. Probabilistic propositional planning: representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 1997.

[67] S. Majercik and M. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147(1-2):119162, 2003.

[68] Petr Mandl. On the variance in controlled Markov chains. *Kybernetika*, 7(1), 1971.

[69] Shie Mannor and John Tsitsiklis. Mean-variance optimization in Markov decision processes. In *ICML*, 2011.

[70] Mausam, Piergiorgio Bertoli, and Daniel S. Weld. A hybridized planner for stochastic domains. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 1972–1978, 2007.

[71] Mausam and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[72] Mausam and Daniel S. Weld. Solving concurrent Markov decision processes. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 716–722, 2004.

[73] H. Brendan Mcmahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the Twenty-second International Conference on Machine Learning*, pages 569–576, 2005.

[74] Teodor Mihai Modovan and Pieter Abbeel. Risk aversion in Markov decision processes via near-optimal Chernoff bounds. In *NIPS*, 2012.

[75] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing, 1980.

[76] Nils J. Nilsson. Shakey the robot. Technical Report 323, SRI International, 1984.

[77] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[78] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial-order planner for ADL. In *Third International Conference on Knowledge Representation and Reasoning (KR-92)*, 1992.

[79] Scott Proper and Prasad Tadepalli. Scaling model-based average-reward reinforcement learning for product delivery. In *ECML*, pages 735–742, 2006.

[80] Martin L. Puterman. *Markov Decision Processes*. John Wiley & Sons, 1994.

[81] Martin L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24, 1978.

[82] Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In *ICAPS'11*, 2011.

[83] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[84] Nicholas Roy and Geoffrey Gordon. Exponential family PCA for belief compression in POMDPs. In *NIPS'02*, pages 1043–1049. MIT Press, 2003.

[85] Scott Sanner. Relational dynamic influence diagram language (RDDL): Language description. http://users.cecs.anu.edu.au/˜ssanner/IPPC_2011/RDDL.pdf, 2010.

[86] Scott Sanner. ICAPS 2011 international probabilistic planning competition. http://users.cecs.anu.edu.au/˜ssanner/IPPC_2011/, 2011.

[87] Scott Sanner and Craig Boutilier. Practical linear value-approximation techniques for first-order MDPs. In *Proceedings of the Twenty-second Conference on Uncertainty in Artificial Intelligence*, 2006.

[88] Scott Sanner, Robby Goetschalckx, Kurt Driessens, and Guy Shani. Bayesian real-time dynamic programming. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[89] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 1 of *Algorithms and Combinatorics*. Springer, 2003.

[90] Trey Smith and Reid G. Simmons. Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic. In *Proceedings of the Twenty-first National Conference on Artificial Intelligence*, 2006.

[91] Matthew J. Sobel. The variance of discounted Markov decision processes. *Journal of Applied Probability*, 19(4):794–802, 1982.

[92] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *Advances in Neural Information Processing Systems*, pages 1089–1095, 2000.

[93] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[94] R. Tarjan. Depth-first search and linear graph algorithms. In *SIAM Journal on Computing*, pages 1(2):146–160, 1972.

[95] Florent Teichteil-Königsbuch, Ugur Kuter, and Guillaume Infantes. Incremental plan aggregation for generating policies in MDPs. In *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems*, pages 1231–1238, 2010.

[96] Florent Teichteil-Königsbuch. Stochastic safest and shortest path problems. In *Proceedings of the Twenty-sixth AAAI Conference on Artificial Intelligence*, 2012.

[97] J. A. E. E. van Nunen. A set of successive approximation methods for discounted Markovian decision problems. *Mathematical Methods of Operations Research*, 20(5):203–208, 1976.

[98] K. Wakuta. Vector-valued Markov decisionprocesses and the systems of linear inequalities. *Stochastic Processes and their Applications*, 56:159–169, 1995.

[99] Sungwook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *Proceedings of the Fifth International Conference on Automated Planning and Scheduling*, 2007.

[100] Sungwook Yoon, Alan Fern, Subbarao Kambhampati, and Robert Givan. Probabilistic planning via determinization in hindsight. In *Proceedings of the Twenty-third AAAI Conference on Artificial Intelligence*, pages 1010–1016, 2008.

[101] Håkan L. S. Younes and Michael Littman. PPDDL1.0: The language for the probabilistic part of IPC-4. In *Fourth International Planning Competition at ICAPS'04*, 2004.

[102] Håkan L. S. Younes and Reid G. Simmons. Policy generation for continuous-time stochastic domains with concurrency. In *ICAPS*, pages 325–334, 2004.

Appendix A
THEOREM PROOFS

Theorem 3.2. *NOGOOD-DECISION is $PSPACE$-complete.* $\diamondsuit$

Proof. First, we show that *NOGOOD-DECISION* $\in PSPACE$. To verify that a conjunction is a nogood, we can verify that each state this conjunction represents is a dead end. For each state, such verification is equivalent to establishing plan existence in the all-outcomes determinization of the MDP. This problem is $PSPACE$-complete [20], i.e., is in $PSPACE$. Thus, nogood verification can be broken down into a set of problems in $PSPACE$, and is in $PSPACE$ itself.

To complete the proof, we point out that the aforementioned problem of establishing deterministic plan existence is an instance of *NOGOOD-DECISION*, providing a trivial reduction to *NOGOOD-DECISION* from a $PSPACE$-complete problem. $\blacksquare$

Theorem 5.1. ***The Optimality Principle for Generalized SSP MDPs.*** *For a GSSP$_{s_0}$ MDP, define $V^\pi(h_{s,t}) = \mathbb{E}[\sum_{t'=0}^{\infty} R_{t'+t}^{\pi_{h_{s,t}}}]$ for any state s, time step t, execution history $h_{s,t}$, and policy π proper w.r.t. $h_{s,t}$. Let $V^\pi(h_{s,t}) = -\infty$ if π is improper w.r.t. $h_{s,t}$. The optimal value function V^* for this MDP exists, is stationary Markovian, and satisfies, for all $s \in \mathcal{S}$,*

$$V^*(s) = \max_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s')[\mathcal{R}(s,a,s') + V^*(s')] \right] \quad (5.6)$$

and, for all $s_g \in \mathcal{G}$, $V^*(s_g) = 0$. Moreover, at least one optimal policy $\pi^*_{s_0}$ proper w.r.t. s_0 and greedy w.r.t. the optimal value function is stationary deterministic Markovian and satisfies, for all $s \in \mathcal{S}$,

$$\pi^*_{s_0}(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left[\sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s')[\mathcal{R}(s,a,s') + V^*(s')] \right].$$

$\diamondsuit$

Proof. The proof's main insight is that a GSSP$_{s_0}$ MDP can be converted to an equivalent SSP$_{s_0}$ MDP problem in which each of the GSSP$_{s_0}$ MDP's potential transient traps has been collapsed into a single state. Although the asymptotic complexity of the conversion procedure is polynomial and thus comparable to the cost of solving the GSSP$_{s_0}$ MDP, the procedure demonstrates that the optimal value function and policy for a GSSP$_{s_0}$ problem have some of the same properties as for the corresponding SSP$_{s_0}$ instance.

Given a GSSP$_{s_0}$ MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{G}, s_0 \rangle$, construct another MDP $\hat{M} = \langle \hat{\mathcal{S}}, \hat{\mathcal{A}}, \hat{\mathcal{T}}, \hat{\mathcal{R}}, \hat{\mathcal{G}}, \hat{s}_0 \rangle$ using the following procedure, implemented with minor modifications by the Transform-MDP method in Algorithm 5.2 if this method is provided with the set of all potential traps of M:

- Let $\hat{\mathcal{S}}$ be the same as $\mathcal{S}$, but for each potential transient trap in $\mathcal{S}$ replace the set $S \subseteq \mathcal{S}$ of all states in this trap by a single state $\hat{s}$ in $\hat{\mathcal{S}}$. Also, let $\hat{\mathcal{S}}$ contain no states of M that do not have a proper policy, i.e., states from which reaching the goal with probability 1 is impossible.

- Let $\hat{\mathcal{A}}$, $\hat{\mathcal{T}}$, and $\hat{\mathcal{R}}$ be constructed as in lines 6 and 19-51 of Algorithm 5.2, but for any state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, if a with a positive probability leads from s to a state without a proper policy in M, then skip a during the conversion procedure when iterating over actions for s.

- Let $\hat{\mathcal{G}} = \mathcal{G}$.

- Let $\hat{s}_0 = s_0$ (or, if s_0 is in a potential transient trap in M, let $\hat{s}_0$ be the state that replaces that trap in $\hat{M}$).

The above transformation of M into $\hat{M}$ throws away all states without a proper policy, along with actions that lead to them from the other states, and turns every potential transient trap into a single state without deterministic self-loops. As a result, $\hat{M}$ has a complete proper policy. Moreover, other than the goal, its transition graph contains no strongly connected components (SCCs) all of whose internal transitions have zero reward, because every such SCC would be part of a potential transient trap, and such traps have been eliminated during the conversion.

Last but not least, every policy improper w.r.t. a state has an infinitely negative reward from that state in $\hat{M}$. To see why this is so, note that in the original MDP M, for every policy π and every state s, $V_+^\pi(s) < \infty$, according to the second requirement of the GSSP$_{s_0}$ MDP definition (5.1), and the above transformation from M to $\hat{M}$ preserves this property. Now, suppose that π is improper w.r.t. a state s in $\hat{M}$. For this policy, the expected number of state transitions to reach the goal from s is infinite. Therefore, with a positive probability a given trajectory of π starting at s makes an infinite number of transitions without reaching the goal. Consider such an infinite trajectory of π. Since $V_+^\pi(s) < \infty$ and the reward for all transitions is bounded from above, this trajectory almost surely makes only a finite number of positive-reward transitions. This implies that the number of its negative-reward transitions is infinite with a positive probability. If the number of negative-reward transitions was almost surely finite, after a finite number of steps the trajectory would, with a positive probability, take only zero-reward transitions. This is possible only if the trajectory entered a goal state, a potential transient trap, or a potential permanent trap. However, by our assumption, this trajectory never reaches the goal, and $\hat{M}$ has no potential traps by construction. Thus, the number of zero-reward transitions in the trajectory has to be almost surely finite, and the number of negative-reward ones must be infinite with a positive probability. Since all negative rewards are bounded away from zero, this implies that with a positive probability the reward of a trajectory of π starting at s is infinitely negative, and hence $V^\pi(s) = -\infty$.

Thus, by construction, $\hat{M}$ meets all the conditions of the SSP$_{s_0}$ definition (2.19). Furthermore, note that for any policy π proper w.r.t s_0 for the GSSP$_{s_0}$ MDP M there is a corresponding policy proper $\hat{\pi}$ in $\hat{M}$. In the states that M and $\hat{M}$ have in common, $\hat{\pi}$ selects an action in the same way that π does. In the states of M that are in a potential transient trap S whose corresponding state in $\hat{M}$ is $\hat{s}$, $\hat{\pi}$ selects the same action in $\hat{s}$ as π does in M when π exits S. Thus, π behaves differently from $\hat{\pi}$ only within potential transient traps. However, the reward of all transitions within these traps is strictly zero; therefore, π and $\hat{\pi}$ have the same expected reward from corresponding states. In fact, the correspondence between policies goes the other way as well; for every policy $\hat{\pi}$ for $\hat{M}$, there is a policy π for M that behaves like $\hat{\pi}$ in the states that M and $\hat{M}$ have in common and in the states from which π exits potential transient traps. Within potential transient traps, π chooses actions so that an agent that follows π and enters such a trap reaches, with probability 1, a state from which the trap can be exited. In the rest of the proof, we will refer to such states as *exit states*. Selecting actions in this way is always possible, because a potential transient trap is an SCC in M's transition graph.

This two-way correspondence between the policies of M and $\hat{M}$, along with the corresponding policies having equivalent value functions, implies that the *optimal* value function V^* of M is equivalent to the optimal value function $\hat{V}^*$ of $\hat{M}$, a SSP$_{s_0}$ MDP. Namely,

- For the states s that M and $\hat{M}$ have in common, $V^*(s) = \hat{V}^*(s)$.
- For the states $s \in S$ where S is a potential transient trap of M and $\hat{s}$ is the trap's corresponding state in $\hat{M}$, $V^*(s) = \hat{V}^*(\hat{s})$.
- For all other states s of M (i.e., states without a proper policy, including states in M's potential permanent traps), $V^*(s) = -\infty$.

According to the Optimality Principle for SSP MDPs (Theorem 2.3), $\hat{V}^*$ of $\hat{M}$ satisfies Equation 5.6. Therefore, so does V^*. For the states of M outside of potential traps, this result is trivial because of the correspondence between M and $\hat{M}$. For the states of M without a proper policy, $V^*(s) = -\infty$, and hence also satisfies Equation 5.6. For the states of M in a potential transient trap S, there are two cases to consider:

1. States s all of whose successors s' under any V^*-greedy action a are in the same trap as s itself. The result for such states s follows because for all their V^*-greedy successors s', $V^*(s') = V^*(s)$ and $\mathcal{R}(s, a, s') = 0$.

2. States s some of whose successors are inside S and some are outside. The result for them follows immediately from the correspondence between M and $\hat{M}$.

Thus, the first part of the theorem, claiming the validity of Equation 5.6 for V^*, has been established.

We now prove the second part of the theorem, claiming the existence of at least one optimal s.d.M. policy for a GSSP$_{s_0}$ MDP. Since, according to the Optimality Principle for SSP MDPs, such a policy exists for the constructed MDP $\hat{M}$, for all states outside of M's traps and the exit states of M's potential transient traps, there exists a V^*-greedy s.d.M. policy that reaches either the goal or a state in a potential transient trap. A V^*-optimal s.d.M. policy also exists for states without a proper policy (including those in potential permanent traps), since all policies for these states are V^*-optimal. Therefore, it remains to show that for any non-exit state s and any exit state s'' in any potential transient trap S, there exists an s.d.M. policy that eventually reaches s'' from s with probability 1. To see why this is true, consider an MDP M_S that has S as the state space, any non-exit $s \in S$ as the initial state, any exit $s'' \in S$ as the goal state, the same transition function as M over the states in S, and a cost function that assigns a cost of 1 to every transition. Note that since a potential transient trap is an SCC in M's transition graph, M_S must have a proper policy w.r.t. s, i.e., a policy that reaches s'' with probability 1. This, along with M_S's cost function, implies that M_S is an SSP$_{s_0}$ MDP. Therefore, M_S has an s.d.M. policy greedy w.r.t. M_S's optimal value function that is guaranteed to reach s'' from s. However, since this policy uses only actions that transition within S and all such actions are greedy w.r.t. M's optimal value function V^* over S, this policy must be greedy also w.r.t. M's V^*. To summarize, M has a V^*-greedy s.d.M. policy that reaches from any state outside of potential traps either to the goal or to a potential transient trap, and, for every such trap, has a policy that reaches from any state in the trap to any exit from the trap with probability 1. Combining these policies yields an optimal s.d.M. policy for M. ■

Lemma A.1. *Any contraction $\hat{M}$ of a GSSP$_{s_0}$ MDP M is also a GSSP$_{s_0}$ MDP.* ◇

Proof. Like M, $\hat{M}$ satisfies both conditions of the GSSP$_{s_0}$ MDP definition (5.1). $\hat{M}$ has a policy $\hat{\pi}$ proper w.r.t. its initial state because such a policy π exists for M. If a trap of M that is visited by π is represented (Definition 5.8) by a state $\hat{s}$ of $\hat{M}$, then, by the construction of $\hat{M}$, the goal is reachable from $\hat{s}$ with probability 1; thus, replacing some of M's traps by single states does not prevent the existence of proper policies for $\hat{M}$. Also, every policy for $\hat{M}$ satisfies the second condition of the GSSP$_{s_0}$ MDP definition: if some policy π for $\hat{M}$ did not satisfy it, then there would be a policy for M that does not satisfy it, contradicting the assumption that M is a GSSP$_{s_0}$ MDP. ■

Lemma A.2. *For a GSSP$_{s_0}$ MDP, an admissible value function monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s must assign the same value to all states in a given potential transient or permanent trap.* ◇

Proof. Suppose for a contradiction that an admissible (Definition 2.36) monotonic (Definition 2.29) value function V assigns different values to states in some potential trap. Note that such a trap must necessarily have at least two states. Consider states $\underline{s}$ of this trap s.t. $V(\underline{s}) = V_{min}$, where V_{min} is the smallest value of V over this trap's states. At least one such state $\underline{s}^{\times}$ must have an action a s.t. $Q^V(\underline{s}^{\times}, a) > V(\underline{s}^{\times})$. This is because in each state s of this potential trap there is at least one action that leads only to states in the same trap and has a positive probability of transitioning from s to $s' \neq s$; moreover, if $V(s) = V_{min}$ then at least one of the other states in the trap has $V(s') > V(s)$, while the value of the rest is at least $V(s)$. Consider applying $\mathscr{B}_{GSSP_{s_0}(\underline{s}^{\times})}$ to V. Since $\underline{s}^{\times}$ has an action a s.t. $Q^V(\underline{s}^{\times}, a) > V(\underline{s}^{\times})$, $\mathscr{B}_{GSSP_{s_0}(\underline{s}^{\times})}V(\underline{s}^{\times}) > V(\underline{s}^{\times})$. This implies that V is not monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(\underline{s}^{\times})}$, contradicting the assumption of the theorem. ■

Lemma A.3. *For a GSSP$_{s_0}$ MDP, an admissible value function monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s is a fixed point for all $\mathscr{B}_{GSSP_{s_0}(s')}$ s.t. s' is in a potential transient or permanent trap.* ◇

Proof. The lemma essentially says that local Bellman backup cannot modify an admissible monotonic value function V at states in potential traps. By the definition of a potential trap, in every state s' of a potential traps at least one action leads only to the states of the same trap and has a reward of zero. As a consequence of Lemma A.2 and the definition of local Bellman backup (Equation 5.8), the Q-value of any such action under V must be $V(s')$. Moreover, at least one such action must necessarily be V-greedy in s': if no such action was V-greedy in s', then the Q-value of a V-greedy action in s' would have to be greater than $V(s')$, implying that V would not monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s')}$. Therefore, applying $\mathscr{B}_{GSSP_{s_0}(s')}$ to V would not change V. ■

Lemma A.4. *For a GSSP$_{s_0}$ MDP and any state s of that MDP, the local Bellman backup operator $\mathscr{B}_{GSSP_{s_0}(s)}$ for any state s preserves the admissibility and monotonicity of value functions that are monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s')}$ for all states s'.* ♢

Proof. Restated mathematically, the theorem claims that if $V \geq V^*$ and, for all $s \in \mathcal{S}$, $\mathscr{B}_{GSSP_{s_0}(s)}V \leq V$, then, for any $s, s' \in \mathcal{S}$, $\mathscr{B}_{GSSP_{s_0}(s)}V \geq V^*$ and $\mathscr{B}_{GSSP_{s_0}(s')}\mathscr{B}_{GSSP_{s_0}(s)}V \leq \mathscr{B}_{GSSP_{s_0}(s)}V$.

Recall from the proof of the Optimality Principle for GSSP$_{s_0}$ MDPs (Theorem 5.1) that a GSSP$_{s_0}$ MDP M with an optimal value function V^* can be converted to an SSP$_{s_0}$ MDP $\hat{M}$ with an optimal value function $\hat{V}^*$ that is "equivalent" to V^*. Similarly, any admissible (Definition 2.36) value function V for M monotonic (Definition 2.29) w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states of$M$ can be converted to an admissible value function $\hat{V}$ for $\hat{M}$ monotonic w.r.t. the local Bellman backup operator $\mathscr{B}_{SSP(s)}$ for all states of $\hat{M}$. When discussing the admissibility of SSP$_{s_0}$ MDPs' value functions in this proof, for convenience we will take the reward-oriented, not cost-oriented view and call an SSP$_{s_0}$ MDP's value function $\hat{V}$ admissible if $\hat{V} \geq \hat{V}^*$. To convert V to $\hat{V}$, for any state s'' that M and $\hat{M}$ have in common, let $\hat{V}(s'') = V(s'')$. For all states s'' in a potential transient trap of

M that are represented (Definition 5.8) by a state $\hat{s}$ in $\hat{M}$, let $\hat{V}(\hat{s}) = V(s'')$. This conversion is well-defined because, according to Lemma A.2, all states in a given potential transient trap of M have the same value under V. Since V is admissible and monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ in M, the equivalence of M and $\hat{M}$ and their optimal value functions V^* and $\hat{V}^*$ (see the proof of Theorem 5.1) implies that $\hat{V}$ is admissible and monotonic w.r.t. $\mathscr{B}_{SSP(s)}$ in $\hat{M}$. We will use the equivalence of V and $\hat{V}$ to show that the claim in the theorem holds for $\mathscr{B}_{GSSP_{s_0}(s)}$ and GSSP$_{s_0}$ MDPs because it holds for $\mathscr{B}_{SSP(s)}$ and SSP$_{s_0}$ MDPs.

In particular, to see that $\mathscr{B}_{GSSP_{s_0}(s)}$ preserves value function admissibility, observe that for SSP$_{s_0}$ MDPs, the local Bellman backup operator $\mathscr{B}_{SSP(s)}$ for any state s preserves admissibility, because the full Bellman backup operator for SSP MDPs does [5]. Therefore, for any state s of M that is outside of any potential traps, $\mathscr{B}_{GSSP_{s_0}(s)}V(s) = \mathscr{B}_{SSP(s)}\hat{V}(s) \geq \hat{V}^*(s) = V^*(s)$. If s is in a potential transient or permanent trap of M, then $\mathscr{B}_{GSSP_{s_0}(s)}V(s) = V(s)$, because at least one V-greedy action in s has zero reward and leads only to the states of the same trap, all of which have the same value as s under V. Recalling that $\mathscr{B}_{GSSP_{s_0}(s)}$ changes any value function only at s, these facts let us conclude that for any state s of M, $\mathscr{B}_{GSSP_{s_0}(s)}V$ is admissible.

Similarly, $\mathscr{B}_{SSP(s)}$ for any state s in $\hat{M}$ is known to preserve value function monotonicity [5]. Therefore, for any state s, $\hat{V}' = \mathscr{B}_{SSP(s)}\hat{V}$ is monotonic in $\hat{M}$. Now, let $V' = \mathscr{B}_{GSSP_{s_0}(s)}V$ for M. We will now show that for any s, s', $\mathscr{B}_{GSSP_{s_0}(s')}\mathscr{B}_{GSSP_{s_0}(s)}V = \mathscr{B}_{GSSP_{s_0}(s')}V' \leq V'$. First, suppose s is outside of potential traps in M. In this case, $V'(s') = \hat{V}'(s')$ for all s' outside of M's potential traps and $V'(s') = \hat{V}'(\hat{s}') = V(s')$ for all s' in M's potential transient traps, where $\hat{s}'$ is $\hat{M}$'s state representing (Definition 5.8) each s' in a potential trap. For s' outside of potential traps, $\mathscr{B}_{GSSP_{s_0}(s')}V' = \mathscr{B}_{SSP(s')}\hat{V}'$, and since, due to $\hat{V}'$'s monotonicity, $\mathscr{B}_{SSP(s')}\hat{V}' \leq \hat{V}'$ for $\hat{M}$, it must be that $\mathscr{B}_{GSSP_{s_0}(s')}V' \leq V'$ for M, i.e., V' is monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s')}$ for such states s'. If s' is in a potential trap of M then $\mathscr{B}_{GSSP_{s_0}(s')}V' = V'$, so V' is monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s')}$ for such states s' too. Thus, $\mathscr{B}_{GSSP_{s_0}(s)}$ preserves monotonicity of an admissible monotonic V if s is outside of potential traps. The case when s is in a potential trap is much easier: for these states, $\mathscr{B}_{GSSP_{s_0}(s)}V = V$. Therefore, $\mathscr{B}_{GSSP_{s_0}(s)}$ preserves monotonicity of an admissible monotonic V for all states s. ■

Lemma A.5. *Let $\mathscr{E}$ denote the ELIMINATE-TRAPS operator implemented in lines 27-46 of Algorithm 5.1. Let M be a GSSP$_{s_0}$ MDP, let V be a value function for M, and let $\langle \hat{M}, \hat{V} \rangle = \mathscr{E}\langle M, V \rangle$. If V is admissible and monotonic w.r.t. the local Bellman backup operator $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s of M, then $\hat{V}$ is admissible and monotonic w.r.t. the local Bellman backup operator $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s of $\hat{M}$.* $\diamondsuit$

Proof. ELIMINATE-TRAPS constructs $\hat{M}$ by eliminating all traps in the greedy graph $G^V_{s_0}$ of M and produces a value function $\hat{V}$ for $\hat{M}$ defined as in the proof of Lemma A.4 and of the Optimality Principle for GSSP$_{s_0}$ MDPs (Theorem 5.1). Namely, for each state $\hat{s}$ of $\hat{M}$ that represents an eliminated trap of M all of whose states s have value $V(s)$, $\hat{V}(\hat{s}) = V(s)$; for each state s that M and $\hat{M}$ have in common, $\hat{V}(s) = V(s)$. By the same reasoning as at the beginning of the proof of Lemma A.4 and of the Optimality Principle for GSSP$_{s_0}$ MDPs (Theorem 5.1), admissibility and monotonicity of V for M implies admissibility and monotonicity of $\hat{V}$ for $\hat{M}$. ■

Lemma A.6. *For a GSSP$_{s_0}$ MDP M, suppose FRET is run until ϵ-consistency and halts at a contraction $\hat{M}$ of M and a value function $\hat{V}$ for $\hat{M}$. Then the greedy graph $G^{\hat{V}}_{\hat{s}_0}$ of $\hat{M}$ does not contain any transient or permanent traps.* $\diamondsuit$

Proof. The claim follows directly from the fact that FRET halts only when its ELIMINATE-TRAPS step cannot find (and hence eliminate) any more traps in the greedy graph $G^{\hat{V}}_{\hat{s}_0}$ of the current value function $\hat{V}$ and MDP $\hat{M}$ (line 13 of Algorithm 5.1). ■

Theorem 5.6. *For a GSSP$_{s_0}$ MDP and any $\epsilon > 0$, if FRET has a systematic FIND procedure and is initialized with an admissible heuristic that is finite and monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states*

s, then FRET converges to a value function that is ϵ-consistent over the states in its greedy graph rooted at s_0 after a finite number of REVISE and ELIMINATE-TRAPS steps. $\diamondsuit$

Proof. The theorem statement consists of two parts. First, we show that for any $\epsilon > 0$, FRET converges after a finite number of REVISE and ELIMINATE-TRAPS steps, and then demonstrate that the resulting value function is ϵ-consistent over its greedy graph $G^V_{s_0}$. By Lemmas A.4, A.5, and A.1, for a GSSP$_{s_0}$ MDP M, operators $\mathscr{B}_{GSSP_{s_0}(s)}$ for all states s and $\mathscr{E}$ preserve admissibility (Definition 2.36) and monotonicity (Definition 2.29) of value functions. Therefore, for M, the sequence of expansions (Definition 5.9) of the value functions generated by the REVISE and ELIMINATE-TRAPS steps is monotonically decreasing (due to the monotonicity of the initializing value function V_0) and bounded from below by V^* at every state (due to the admissibility of V_0). Since REVISE updates only states at which the current value function $\hat{V}$ of the current contraction $\hat{M}$ of M (Definition 5.7) is ϵ-inconsistent, each REVISE step translates to a decrease of the expanded value function V by at least ϵ at some state of M. Therefore, in light of the expansion sequence being monotonically decreasing and bounded from below by V^*, FRET can apply at most $\frac{V_0(s)-V^*(s)}{\epsilon}$ REVISE steps to any single state s of M and to its representatives in contractions of M. The quantity $\frac{V_0(s)-V^*(s)}{\epsilon}$ is finite for some states but not others. Divide the set $\mathcal{S}$ of M's states into the set $\mathcal{S}'_I$ of states inside potential permanent traps, the set $\mathcal{S}_I$ of states outside of potential permanent traps for which there is no proper policy, and the set $\mathcal{S}_P$ of states for which such a policy exists (these sets are clearly disjoint). For all states $s \in \mathcal{S}'_I$, every monotonic value function is ϵ-consisent by Lemma A.3, so the REVISE step is never applied to them by FRET — their values change only due to the ELIMINATE-TRAPS updates. For any $s \in \mathcal{S}_P$, $V^*(s)$ is finite, so $\frac{V_0(s)-V^*(s)}{\epsilon}$, the number of REVISE operations applied to s and its representatives in contractions of M is finite as well. However, for any $s \in \mathcal{S}_I$, $V^*(s)$ is infinitely negative, so the expression $\frac{V_0(s)-V^*(s)}{\epsilon}$ cannot be used to bound the number of REVISE updates applied to such states and their represetatives by a finite value.

Nonetheless, the total number of REVISE operations FRET can perform on any state in $\mathcal{S}_I$ and its representatives before reaching ϵ-consistency must be finite. For contradiction, suppose it was infinite for some $s \in \mathcal{S}_I$. Since every REVISE step decreases the expansion value of s by at least ϵ, the expansion value of s would have to drop without bound. Moreover, since FRET only updates states in $G_{\hat{s}_0}^{\hat{V}}$, if FRET applied REVISE to s and its representatives an infinite number of times then s or its representatives would have to either stay in $G_{\hat{s}_0}^{\hat{V}}$ forever or reenter $G_{\hat{s}_0}^{\hat{V}}$ at some point after any finite number of REVISE steps. Due to the systematicity of the FIND procedure, s's or its representatives' presence in $G_{\hat{s}_0}^{\hat{V}}$ after any finite number of REVISE steps would mean that $V(s)$, the value of s under the expanded value function, would eventually influence $V(s_0)$. Also, since FRET updates only states reachable from s_0 via an s.d.M. policy and there is a finite number of these policies, there must exist a minimum positive probability with which s or its representatives can be reached from $\hat{s}_0$ in $G_{\hat{s}_0}^{\hat{V}}$ for any contraction $\hat{M}$ and value function $\hat{V}$ for it. This, in turn, implies that if s or its representatives were present in $G_{\hat{s}_0}^{\hat{V}}$ indefinitely and $V(s)$ dropped without bound, then $V(s_0)$ would have to drop without bound as well. However, this contradicts the fact that for any expansion V that FRET generates starting from an admissible heuristic, $V(s_0) \geq V^*(s_0) > -\infty$, because s_0 has a proper policy, by condition 1 of the GSSP$_{s_0}$ MDP definition (5.1). To summarize, before it halts, FRET can apply only finitely many REVISE operations.

Now, consider the number of ELIMINATE-TRAPS steps that FRET executes. Observe that every ELIMINATE-TRAPS operation yields a contraction of M that has at least one trap fewer than M. Any GSSP$_{s_0}$ MDP M has a finite number of traps, so FRET can apply only a finite number of ELIMINATE-TRAPS steps to M and its contractions before it yields a contraction that has no traps. As proved above, the total number of REVISE steps before FRET halts is finite as well, concluding the proof of the first part of the theorem.

The second part of the theorem claims that when FRET converges, its value function is ϵ-consistent over the states in its greedy graph rooted at s_0. For contradiction, suppose that FRET halted at a value function V whose $G_{s_0}^{V}$ contains a state s s.t. $Res^V(s) > \epsilon$. Since FRET halted at V, FRET's last ELIMINATE-TRAPS must have failed to find any traps in $G_{s_0}^{V}$, so the last FIND operation before the last ELIMINATE-TRAPS step must have acted on the same value function V at which FRET halted. Since the FIND procedure is systematic, it would have found s and caused a REVISE step to be applied to V at s, decreasing $V(s)$ by at least ϵ. This contradicts our assumption that FRET halted at V. ■

Theorem 5.7. *For a GSSP$_{s_0}$ MDP, if FRET has a systematic FIND procedure and is initialized with an admissible heuristic that is finite and monotonic w.r.t. $\mathscr{B}_{GSSP_{s_0}}(s)$ for all states s, as ϵ goes to 0 the value function and policy computed by FRET approaches, respectively, the optimal value function and an optimal policy over all states reachable from s_0 by at least one optimal s.d.M. policy.* $\diamondsuit$

Proof. The main insight of the proof is that running FRET on a GSSP$_{s_0}$ MDP amounts to gradually eliminating transient and permanent traps, which, if FRET is executed for long enough, turns the MDP into an SSP$_{s_0}$ problem. More concretely, the high-level approach will be to show that as ϵ gets smaller, the contraction of the original MDP M at which FRET halts "stabilizes", i.e., becomes equal to some MDP $\hat{M}^*$. $\hat{M}^*$ turns out to be an SSP$_{s_0}$ MDP, meaning that running FRET for a sufficiently small ϵ eventually reduces to executing FIND-AND-REVISE. Existing results about the behavior of FIND-AND-REVISE on SSP$_{s_0}$ MDPs imply that as $\epsilon \to 0$, the value functions produced by FIND-AND-REVISE approach the optimal value function $\hat{V}^*$ of $\hat{M}^*$. Moreover, $\hat{V}^*$ can be easily transformed to the optimal value function V^* for the original MDP M over $\mathcal{S}^*$, the set of states in the greedy graph of V^* rooted at s_0, completing the proof.

More specifically, pick a sequence $\{\epsilon_k\}_{k=0}^{\infty}$ s.t. $\epsilon_k > 0$ for all $k \geq 0$ and $\lim_{k\to\infty} \epsilon_k = 0$. For a given GSSP$_{s_0}$ MDP M and an admissible monotonic heuristic V_0, let $\{\hat{M}_k\}_{k=0}^{\infty}$ and $\{\hat{V}_k\}_{k=0}^{\infty}$ be sequences of MDPs and their value functions s.t., for each ϵ_k, if FRET is initialized with V_0, given M as input, and run until ϵ_k-consistency, FRET exits its main loop (lines 9-13 of Algorithm 5.1) with MDP $\hat{M}_k$ and this MDP's value function $\hat{V}_k$. (The fact that from any $\epsilon_k > 0$, FRET does exit its main loop after a finite number of steps is implied by Theorem 5.6.) Let $\{\hat{\mathcal{S}}_k\}_{k=0}^{\infty}$ be the sequence of subsets of $\hat{M}_k$s' state spaces s.t. $\hat{\mathcal{S}}_k$ is the set of states in the greedy graph $G_{\hat{s}_0}^{\hat{V}_k}$ (Definition 2.34) of $\hat{V}_k$ rooted at $\hat{M}_k$'s initial state.

Finally, let $\hat{\mathcal{S}}^*$ be the set of all states that appear in infinitely many sets $\hat{\mathcal{S}}_k$. Note that the sets in $\{\hat{\mathcal{S}}_k\}_{k=0}^{\infty}$ are generally *not* subsets of the state space $\mathcal{S}$ of the original MDP M. Rather, they contain states of $\mathcal{S}$ as well as *representatives* (Definition 5.8) of sets of states of $\mathcal{S}$, the representatives themselves not being members of $\mathcal{S}$. We say that two different sets $\hat{\mathcal{S}}_{k'}$ and $\hat{\mathcal{S}}_{k''}$ from our sequence have a state in common if there is $s' \in \hat{\mathcal{S}}_{k'}$ and $s'' \in \hat{\mathcal{S}}_{k''}$ s.t. either $s' = s$ and $s'' = s$, where $s \in \mathcal{S}$, or both s' and s'' are representatives of the same subset of states of $\mathcal{S}$ (i.e., representatives of the same trap of $\mathcal{S}$). Thus, $\hat{\mathcal{S}}^*$ is the set of states each of which is in the intersection of infinitely many sets in $\{\hat{\mathcal{S}}_k\}_{k=0}^{\infty}$.

We now show that $\hat{\mathcal{S}}^*$ can be viewed as the state space of an SSP$_{s_0}$ MDP, and executing FRET for a sufficiently small ϵ_k starting from M eventually reduces to running state value updates on this MDP. To start with, observe that $\mathcal{S}^*$ is nonempty, because it contains $\hat{s}_0$ — either the initial state of M or a representative of a trap of M in which s_0 is located. Define an MDP $\hat{M}^*$ whose state space is $\hat{\mathcal{S}}^*$ and whose action space $\hat{\mathcal{A}}^*$, transition function $\hat{\mathcal{T}}^*$, and reward function $\hat{\mathcal{R}}^*$ are constructed as follows:

- $\hat{\mathcal{A}}^*$ contains, for every state $s \in \hat{\mathcal{S}}^*$, only those actions that, in at least one MDP $\hat{M}_k$ in the MDP sequence $\{\hat{M}_k\}_{k=0}^{\infty}$, are applicable in s and lead with a positive probability only to other states of $\hat{\mathcal{S}}^*$. That is, $\hat{\mathcal{A}}^*$ does not include any actions that lead from a state in $\hat{\mathcal{S}}^*$ to a state in $(\bigcup_{k=0}^{\infty} \hat{\mathcal{S}}_k) \setminus \hat{\mathcal{S}}^*$. Note that these excluded actions are "unnecessary" in the following sense. By the definition of $\hat{\mathcal{S}}^*$, each state in $(\bigcup_{k=0}^{\infty} \hat{\mathcal{S}}_k) \setminus \hat{\mathcal{S}}^*$ appears in only finitely many $\hat{\mathcal{S}}_k$s. Also, $|\bigcup_{k=0}^{\infty} \hat{\mathcal{S}}_k|$ is finite, because is consists only of states of $\mathcal{S}$ and representatives of subsets of $\mathcal{S}$, and there are finitely many of each of these kinds of states. Therefore, there must exist the smallest finite K s.t. for all $k \geq K$, each $\hat{\mathcal{S}}_k$ contains no states from outside of $\hat{\mathcal{S}}^*$, i.e., $\hat{\mathcal{S}}^*$ contains *all* of the states in $G_{\hat{s}_0}^{\hat{V}_k}$. Moreover, for each $k \geq K$, there must exist some n_k s.t. in all REVISE steps after the n_k-th and until FRET halts, FRET uses *only* actions that cause transitions only from states in $\hat{\mathcal{S}}^*$ to states in $\hat{\mathcal{S}}^*$. That it, for each ϵ_k s.t. $k \geq K$, the last few value function updates made by FRET over states in $\hat{\mathcal{S}}^*$ before halting at $\hat{V}_k$ do not depend on the values of states outside of $\hat{\mathcal{S}}^*$. If this was not the case for some $k \geq K$, then $G_{\hat{s}_0}^{\hat{V}_k}$ would necessarily contain a state outside $\hat{\mathcal{S}}^*$, contradicting the fact that $K-1$ was the largest number for which this happens. To sum up, for a sufficiently small ϵ_k, actions outside of $\hat{\mathcal{A}}^*$ do not play a role in the convergence of FRET.

- For every $s \in \hat{\mathcal{S}}^*$ and $a \in \hat{\mathcal{A}}^*$ s.t. a is applicable in s, the transition probabilities and rewards for using a in s are defined in the same way as in any MDP $\hat{M}_k$ where a executed in s can cause transitions only to states of $\hat{\mathcal{S}}^*$. By construction (see Algorithm 5.2), any two MDPs

$\hat{M}_{k'}$ and $\hat{M}_{k''}$ that agree on the set of positive-probability successors of state s under action a also agree on the the transition and reward functions for a at s, so the above definition of these functions uniquely determines $\hat{\mathcal{T}}^*$ and $\hat{\mathcal{R}}^*$.

$\hat{M}^*$ has no potential transient or permanent traps of the original MDP M, because, by Lemma A.6, no $G_{\hat{s}_0}^{\hat{V}_k}$ contains them. This implies that $\hat{M}^*$'s transition graph rooted at the initial state, $G_{\hat{s}_0}$, has no strongly connected components with exclusively zero-reward internal transitions, other than those involving goal states. At the same time, $\hat{M}^*$'s state space $\hat{\mathcal{S}}^*$ contains at least one goal state of M. For contradiction, suppose it did not. Then, since $\hat{M}^*$ has no strongly connected components with exclusively zero-reward internal transitions, every policy would have to accumulate an infinitely negative reward (infinite cost) starting from $\hat{s}_0$. Hence, there would have to exist some $\epsilon' > 0$ s.t. no $\hat{V}$ for $\hat{M}^*$ is ϵ-consistent at $\hat{s}_0$ and its descendants in $G_{\hat{s}_0}^{\hat{V}}$ for any $\epsilon < \epsilon'$ (i.e., Bellman backups would never converge). This contradicts the fact that, by the construction of $\hat{\mathcal{S}}^*$, there exist infinitely many value functions that are ϵ-consistent over all states in $\hat{\mathcal{S}}^*$ for an arbitrarily small ϵ. Thus, we can define $\hat{M}^*$ to have the goal set $\hat{\mathcal{G}} = \hat{\mathcal{S}}^* \cap \mathcal{G}$, where $\mathcal{G}$ is the goal set of M.

Last but not least, $\hat{M}^*$ contains no states without a proper policy. In particular, as already mentioned, it has no strongly connected components with exclusively zero-reward internal transitions, and hence no states in potential permanent traps. It also does not contain states without a proper policy that do not belong to any potential permanent trap, for the following reason. For any such state s, any s.d.M. policy $\hat{\pi}$ rooted at s must have at least one of three properties:

- It must with a positive probability lead to a potential permanent trap and stay there forever, or

- It must with a positive probability lead to a potential transient trap and stay there forever, or

- It must cause state transitions with a strictly negative reward an infinite number of times in expectation, and transitions with a positive reward only a finite number of times in expectatation. This is because such a policy involves an infinite number of steps, only a finite number of which can involve transitions with positive rewards, since otherwise an equivalent policy π in the original MDP M we would have $V_+^{\pi}(s) = \infty$, violating the second requirement of the GSSP$_{s_0}$ MDP definition (5.1). Also, only a finite number of these steps can involve zero-reward actions, since an infinite number of zero-reward steps would imply that the policy leads to a potential trap and stays there forever. Thus, an infinite expected number of steps must cause negative-reward transitions.

Now, suppose a state s without a proper policy was in $\hat{\mathcal{S}}^*$. Since $\hat{M}^*$ has no traps, no policy rooted as s can have either of the first two properties, so every such policy must satisfy the last property and hence accumulate an infinite cost starting at s. By the same reasoning that proved the existence of goal states in $\hat{\mathcal{S}}^*$, this would mean that no value function could be ϵ-consistent at s and all its descendants simultaneously for a sufficiently small ϵ, contradicting the fact that, by the construction of $\hat{\mathcal{S}}^*$, a value function ϵ-consistent at s and its descendants in $\hat{M}$'s transition graph (all of which would have to be in $\hat{\mathcal{S}}^*$) exists for an arbitrarily small $\epsilon > 0$. Therefore, $\hat{\mathcal{S}}^*$ has a complete proper policy. Moreover, this reasoning also implies that any policy that is improper w.r.t. some $s \in \hat{\mathcal{S}}^*$ has an infinitely negative expected reward at s.

To recapture the proof up to this point, we have established that for any ϵ_k s.t. $k \geq K$, FRET that starts to run on a GSSP$_{s_0}$ MDP M with an admissible monotonic heuristic ends up performing updates on an MDP $\hat{M}^* = \langle \hat{\mathcal{S}}^*, \hat{\mathcal{A}}^*, \hat{\mathcal{T}}^*, \hat{\mathcal{R}}^*, \hat{\mathcal{G}}^*, \hat{s}_0 \rangle$. $\hat{M}^*$ has a complete proper policy, and every policy for $\hat{M}^*$ that is improper w.r.t. some state s accumulates an infinitely negative expected reward starting at s. Thus, $\hat{M}^*$ satisfies the definition of an SSP$_{s_0}$ MDP (2.19). Note that running FRET on an SSP$_{s_0}$ MDP reduces to executing FIND-AND-REVISE — the ELIMINATE-TRAPS step is executed on such MDPs only once, at the end, and has no effect because SSP$_{s_0}$ MDPs have no traps. In addition, for any ϵ_k s.t. $k \geq K$, at the moment when FRET reduces to FIND-AND-REVISE running on $\hat{M}^*$, it uses a monotonic admissible value function for $\hat{M}^*$, as a consequence of Lemmas A.4, A.5, and A.1. Running FIND-AND-REVISE on an SSP$_{s_0}$ MDP starting from an admissible monotonic value function until ϵ-convergence is known to yield the optimal value function (and an policy) over all states reachable by at least one optimal s.d.M. policy from the initial state as $\epsilon \to 0$ (Theorem 32 in [11], Theorem 3 in [12]). Thus, as $\{\epsilon_k\}_{k=K}^{\infty}$ converges to 0, the sequence $\{\hat{V}_k\}_{k=K}^{\infty}$, which consists entirely of value functions for $\hat{M}^*$, converges to $\hat{V}^*$, the optimal value function for $\hat{M}^*$, over all states of $\hat{\mathcal{S}}^*$ reachable from $\hat{s}_0$ by an optimal s.d.M. policy — we denote this set as $\hat{\mathcal{S}}^{**}$

Observing that the expansion of $\hat{V}^*$, as defined in Equation 5.9, over the states in $\hat{\mathcal{S}}^{**}$ exactly equals V^*, the optimal value function for the original MDP M, over all states reachable in $\mathcal{S}$ from

s_0 by at least one optimal s.d.M. policy for M completes the proof. ■

Theorem 5.8. *For a GSSP$_{s_0}$ MDP, any policy π_{s_0} derived by FRET from the optimal value function is optimal and proper w.r.t. s_0.* ♢

Proof. The fact that π_{s_0} is proper w.r.t. s_0 follows directly from the observation that, by the construction of π_{s_0}, for every s in $G_{s_0}^{V^*}$, there exists a positive-probability trajectory from s to some $s_g \in \mathcal{G}$ under π_{s_0}. Thus, by following π_{s_0} from s_0, an agent is guaranteed to eventually reach a goal state.

To see why π_{s_0} is optimal, recall from the proof of Theorem 5.1 that a GSSP$_{s_0}$ MDP M can be converted to an equivalent SSP$_{s_0}$ MDP $\hat{M}$ where each of M's potential transient traps has been replaced by a single state. Consider converting π_{s_0} to a policy $\hat{\pi}_{s_0}$ for $\hat{M}$: for states outside of M's potential transient traps, π_{s_0} and $\hat{\pi}_{s_0}$ coincide, and at any state $\hat{s}$ of $\hat{M}$ that replaces a potential transient trap S of M, $\hat{\pi}_{s_0}$ chooses an action that π_{s_0} uses in some state of S to exit S. By construction, $\hat{\pi}_{s_0}$ is greedy w.r.t. $\hat{V}^*$, the optimal value function for $\hat{M}$. Hence, by the Optimality Principle for SSP MDPs (Theorem 2.3), $\hat{\pi}_{s_0}$ is optimal for $\hat{M}$ and $\hat{V}^{\hat{\pi}} = \hat{V}^*$ for all states reachable by $\hat{\pi}_{s_0}$ from s_0. This, in turn, means that $V^{\pi} = V^*$ for all states reachable by π_{s_0} from s_0 in M, i.e., that π_{s_0} is optimal for M. This is the case because for all states s that are reachable via π_{s_0} from s_0 and are outside of M's potential transient traps, $V^{\pi}(s) = \hat{V}^{\hat{\pi}}(s) = \hat{V}^*(s) = V^*(s)$; similarly, for any s in a given potential transient trap S of M and the state $\hat{s}$ that replaces S in $\hat{M}$, $V^{\pi}(s) = \hat{V}^{\hat{\pi}}(\hat{s}) = \hat{V}^*(\hat{s}) = V^*(s)$. ■

Theorem 5.10. *POSB* $\subset$ GSSP$_{s_0}$. ♢

Proof. By stating that $POSB$ is contained in *GSSP*$_{s_0}$ we mean that for every MDP $M_{POSB} \in$ *POSB* there exists an MDP $M_{GSSP_{s_0}} \in$ *GSSP*$_{s_0}$ whose state space contains that of M_{POSB} and whose every policy coincides with some policy for M_{POSB} over M_{POSB}'s state space and has the same expected reward at each of these states. Therefore, we prove the theorem by showing how to construct such an $M_{GSSP_{s_0}}$ for any given $M_{POSB} \in POSB$.

Suppose we are given a POSB MDP $M_{POSB} = \langle \mathcal{S}_{POSB}, \mathcal{A}_{POSB}, \mathcal{T}_{POSB}, \mathcal{R}_{POSB} \rangle$. Construct an MDP $M_{GSSP_{s_0}} = \langle \mathcal{S}_{GSSP_{s_0}}, \mathcal{A}_{GSSP_{s_0}}, \mathcal{T}_{GSSP_{s_0}}, \mathcal{R}_{GSSP_{s_0}}, \mathcal{G}_{GSSP_{s_0}}, s_0 \rangle$, where:

- The initial state s_0 is a new state, not present in $\mathcal{S}_{POSB}$.

- $\mathcal{S}_{GSSP_{s_0}} = \mathcal{S}_{POSB} \cup \{s_0\}$.

- $\mathcal{A}_{GSSP_{s_0}} = \mathcal{A}_{POSB}$.

- $\mathcal{T}_{GSSP_{s_0}}$ is s.t. $\mathcal{T}_{GSSP_{s_0}}(s, a, s') = \mathcal{T}_{POSB}(s, a, s')$ for all $s, s' \in \mathcal{S}_{POSB}$, $a \in \mathcal{A}_{POSB}$. For the initial state s_0, any action $a \in \mathcal{A}_{GSSP_{s_0}}$, and any state $s' \in \mathcal{S}_{POSB}$, $\mathcal{T}_{GSSP_{s_0}}(s_0, a, s') = 1/|\mathcal{S}_{POSB}|$. That is, the only possible transitions from s_0 are to the states of $\mathcal{S}_{POSB}$, and all of these transitions are equally likely no matter what action the agent chooses.

- $\mathcal{R}_{GSSP_{s_0}}$ is s.t. $\mathcal{R}_{GSSP_{s_0}}(s, a, s') = \mathcal{R}_{POSB}(s, a, s')$ for all $s, s' \in \mathcal{S}_{POSB}$, $a \in \mathcal{A}_{POSB}$. For the initial state s_0, any action $a \in \mathcal{A}_{GSSP_{s_0}}$, and any state $s' \in \mathcal{S}_{POSB}$, $\mathcal{R}_{GSSP_{s_0}}(s_0, a, s') = 0$, i.e., all transitions from s_0 bring no reward.

- $\mathcal{G}_{GSSP_{s_0}} \subset \mathcal{S}_{GSSP_{s_0}}$ is constructed as follows. Let G_{s_0} be the full reachability graph of $M_{GSSP_{s_0}}$ rooted at s_0. Build a DAG of SCCs of G_{s_0} and identify SCCs with no outgoing edges (i.e., leaves in the DAG) whose internal edges correspond to zero-reward actions in M_{POSB}. At least one such SCC must exist, because in a POSB MDP every state has an action with nonnegative reward but at the same time the expected sum of positive rewards of every policy in this MDP is finite at every state, as the POSB MDP definition (5.10) requires. This means that every policy from every state must eventually reach a region of the state space where it accumulates no reward, which is exactly an SCC we are looking for. The goal set $\mathcal{G}_{GSSP_{s_0}}$ of our MDP $M_{GSSP_{s_0}}$ is the set of all states in all such SCCs.

Thus, $M_{GSSP_{s_0}}$ equals M_{POSB} except for adding an initial state and designating some of M_{POSB}'s existing states as goals. By construction, every policy for $M_{GSSP_{s_0}}$ is identical to some policy for

M_{POSB} over the states in $\mathcal{S}_{POSB}$ and reaches one of these states in a single step if executed from the initial state s_0. Moreover, the expected reward of each $M_{GSSP_{s_0}}$'s policy is the same in $M_{GSSP_{s_0}}$ and M_{POSB} over the states of $\mathcal{S}_{POSB}$, because the reward functions of these MDPs coincide.

Crucially, thanks to the addition of the initial and goal states, $M_{GSSP_{s_0}}$ is a GSSP$_{s_0}$ MDP. The first requirement of the *GSSP*$_{s_0}$ definition (5.1) is satisfied by $M_{GSSP_{s_0}}$ because, as explained during the construction of $\mathcal{G}_{GSSP_{s_0}}$, every policy for $M_{GSSP_{s_0}}$ must eventually reach a zero-reward region from every state (including s_0), and each such region is a goal. The second requirement of the *GSSP*$_{s_0}$ definition is satisfied as well, for the following reason. Over the states in $\mathcal{S}_{POSB}$, every policy for $M_{GSSP_{s_0}}$ is identical to some policy for M_{POSB}, and each of the latter satisfies this requirement by the definition of POSB MDPs (5.10). Moreover all of $M_{GSSP_{s_0}}$'s policies, when executed starting at s_0, reach a state in $\mathcal{S}_{POSB}$ in one step that brings the reward of 0. Therefore, the requirement is satisfied for every policy for the state s_0 as well. To summarize, $M_{GSSP_{s_0}}$ meets all the criteria outlined at the beginning of the proof.

Note that the containment of *POSB* in *GSSP*$_{s_0}$ is strict: *GSSP*$_{s_0}$ contains MDPs some of whose states have only negative-reward actions, while *POSB* does not. ■

Theorem 5.11. *NEG* $\subset$ *GSSP*$_{s_0}$. ♢

Proof. Similar to the proof of Theorem 5.10, we show that for every MDP $M_{NEG} \in$ *NEG* there exists an MDP $M_{GSSP_{s_0}} \in$ *GSSP*$_{s_0}$ whose state space contains that of M_{NEG} and whose policies, along with their values, coincide with those for M_{NEG}.

For a given M_{NEG}, construct an $M_{GSSP_{s_0}}$ as described in Theorem 5.10's proof. Note that, as for *POSB*, the reachability graph of a NEG MDP such as M_{NEG} must contain strongly connected components whose internal edges correspond to zero-reward actions. This is the case because the NEG MDP definition (5.11) requires the existence of a policy whose expected reward is well-defined and finite for every state. From any state, this policy must eventually lead the agent to a region where no reward is accumulated, i.e., one of such zero-reward SCCs. All states in these SCCs are goals in the newly constructed $M_{GSSP_{s_0}}$.

Each policy of $M_{GSSP_{s_0}}$ coincides with exactly one policy of M_{NEG} over M_{NEG}'s state space, and therefore has the same expected reward for each each of these states. It is also easy to see that $M_{GSSP_{s_0}}$ is in fact a GSSP$_{s_0}$ MDP: by constuction, $M_{GSSP_{s_0}}$ has a policy proper w.r.t. the initial state, as explained in Theorem 5.10's proof, thereby satisfying the first requirement of the GSSP$_{s_0}$ MDP definition (5.1). The GSSP$_{s_0}$ MDP definition's second requirement is satisfied by $M_{GSSP_{s_0}}$ as well, because the reward of every action in $M_{GSSP_{s_0}}$ is nonpositive, being inherited from M_{NEG}, so the expected sum of nonnegative rewards of any policy for $M_{GSSP_{s_0}}$ is at most 0.

The reason *GSSP*$_{s_0}$ is a strict superclass of *NEG* is that *GSSP*$_{s_0}$ admits MDPs some of whose policies have values higher than 0 for some states. In NEG MDPs, this is impossible, because the reward of every action in every state is at most 0. ■

Theorem 5.13. *MAXPROB* $\subset$ *POSB* $\subset$ *GSSP*$_{s_0}$. ♢

Proof. Any MAXPROB MDP satisfies both conditions of the POSB MDP definition (5.10): the rewards of all of its action are nonnegative in all states, and the expected sum of long-term nonnegative rewards of every policy is in the $[0, 1]$ interval for every state. The containment of *POSB* in *GSSP*$_{s_0}$ has been shown in Theorem 5.10.

MAXPROB is properly contained in *POSB*, because POSB MDPs can have policies whose value at some states is higher than 1, which is impossible for MAXPROB MDPs. ■

Theorem 5.14. *For a SSPADE*$_{s_0}$ *MDP and an* $\epsilon > 0$, *if* FIND-AND-REVISE *has a systematic FIND procedure and is initialized with an admissible monotonic heuristic, it converges to a value function that is* ϵ*-consistent over the states in its greedy graph rooted at* s_0 *after a finite number of REVISE steps.* ♢

Proof. [1]As in the proof of Theorem 5.6, we first show that for any $\epsilon > 0$, FIND-AND-REVISE converges after a finite number of REVISE steps, and then demonstrate that the resulting value function is ϵ-consistent over its greedy graph $G^V_{s_0}$. For SSPADE$_{s_0}$ MDPs, as for SSP ones, it is easy to verify that Bellman backup (both full and local) preserves admissibility (Definition 2.36) and monotonicity (Definition 2.29) of value functions. Therefore, we know that the value function after every REVISE step is bounded from above by V^* at every state (due to the admissibility of the initializing value function V_0) and is at least as large as the value function after the previous REVISE step at every state (due to the monotonicity of V_0). In addition, since REVISE updates only states at which the current value function is ϵ-inconsistent, each REVISE step increases the value function by at least ϵ at some state. Therefore, since each state's value starts at $V_0(s)$ and never exceeds $V^*(s)$, FIND-AND-REVISE can apply at most $\frac{V^*(s)-V_0(s)}{\epsilon}$ REVISE steps to any single state.

The quantity $\frac{V^*(s)-V_0(s)}{\epsilon}$ is finite for some states but not others. Divide the set $\mathcal{S}$ of states of a given SSPADE$_{s_0}$ MDP into the set $\mathcal{S}_I$ of states for which there is no proper policy and the set $\mathcal{S}_P$ of states for which such a policy exists. For any $s \in \mathcal{S}_P$, $V^*(s)$ is finite, so $\frac{V^*(s)-V_0(s)}{\epsilon}$, the number of REVISE operations applied to any such state, is finite as well. However, for any $s \in \mathcal{S}_I$, $V^*(s)$ is infinite.

Nonetheless, the total number of REVISE operations FIND-AND-REVISE can perform before reaching an ϵ-consistent value function must still be finite. Suppose it was infinite. We have already established that on all $s \in \mathcal{S}_P$, the number of updates must be finite. Therefore, it would have to be infinite on some $s \in \mathcal{S}_I$. Since every update increases the value of such s by at least ϵ, the value of s would have to grow without bound. Moreover, since FIND-AND-REVISE only updates states in $G^V_{s_0}$, after any finite number of updates s would have to stay in $G^V_{s_0}$ or reenter it as some point. Due to the systematicity of the FIND procedure, s's presence in $G^V_{s_0}$ after any finite number of REVISE steps means that $V(s)$ would eventually influence $V(s_0)$. Also, since FIND-AND-REVISE updates only states reachable from s_0 via a stationary deterministic Markovian policy and there is a finite number of these policies, there must exist a minimum positive probability with which s can be reached from s_0 in $G^V_{s_0}$ for all V. This, in turn, implies that if s were present in $G^V_{s_0}$ an infinite number of times and $V(s)$ grew without bound, $V(s_0)$ would have to grow without bound as well.

[1]This proof resembles that of Theorem 2 in [12]. Although the proof of that theorem has not been published, it has been obtained by the author of this dissertation from Blai Bonet of Universidad Simón Bolívar via personal communication.

However, this contradicts the fact that for any V that FIND-AND-REVISE encounters starting from an admissible heuristic, $V(s_0) \leq V^*(s_0) < \infty$, because $s_0 \in \mathcal{S}_P$.

Now, consider the number of REVISE steps that FIND-AND-REVISE executes. For contradiction, suppose that FIND-AND-REVISE halted at a value function V whose $G^V_{s_0}$ contained states s s.t. $Res^V(s) > \epsilon$. Since FIND-AND-REVISE's FIND procedure is systematic, it will cause a REVISE step to be applied to V at such a state s, increasing $V(s)$ by at least ϵ. This contradicts our assumption that FIND-AND-REVISE halts at V. ■

Theorem 5.15. *For a SSPADE$_{s_0}$ MDP, if* FIND-AND-REVISE *has a systematic FIND procedure and is initialized with an admissible monotonic heuristic, as ϵ goes to 0 the value function and policy computed by* FIND-AND-REVISE *approaches, repsctively, the optimal value function and an optimal policy over all states reachable from s_0 by at least one optimal s.d.M. policy.* ♢

Proof. The high-level idea of the proof is to show that as ϵ gets smaller, the set of states in the greedy graph of the ϵ-consistent value function at which FIND-AND-REVISE halts "stabilizes", i.e., becomes equal to some set $\mathcal{S}^*$. Running FIND-AND-REVISE over $\mathcal{S}^*$ turns out to be equivalent to running it on an SSP$_{s_0}$ MDP whose optimal value function equals that of the original MDP over $\mathcal{S}^*$. Existing results about the behavior of FIND-AND-REVISE on SSP$_{s_0}$ MDPs imply that as $\epsilon \to 0$, the value functions and policies produced by FIND-AND-REVISE approach the optimal value function and an optimal policy over $\mathcal{S}^*$. This, in turn, indicates that $\mathcal{S}^*$ contains the set of all states reachable by at least one optimal policy from s_0, completing the proof. We now prove each of these propositions in detail.

Pick a sequence $\{\epsilon_k\}_{k=0}^{\infty}$ s.t. $\epsilon_k > 0$ for all $k \geq 0$ and $\lim_{k\to\infty} \epsilon_k = 0$. For a given SSPADE$_{s_0}$ MDP and an admissible monotonic heuristic V_0, let $\{V_k\}_{k=0}^{\infty}$ be a sequence of value functions s.t., for each ϵ_k, if FIND-AND-REVISE is initialized with V_0 and run on the given SSPADE$_{s_0}$ MDP until ϵ_k-consistency, FIND-AND-REVISE halts at V_k. Finally, let $\{\mathcal{S}_k\}_{k=0}^{\infty}$ be the sequence of subsets of the MDP's state space $\mathcal{S}$ s.t. $\mathcal{S}_k$ is the set of states in the greedy graph $G^{V_k}_{s_0}$ of V_k rooted at s_0 (Definition 2.34).

Let $\mathcal{S}^*$ be the set of states each of which appears in infinitely many sets $\mathcal{S}_k$. $\mathcal{S}^*$ is nonempty, because it contains s_0. Since each state in $\mathcal{S} \setminus \mathcal{S}^*$ appears in only finitely many $\mathcal{S}_k$'s and the number of subsets of $\mathcal{S}$ is finite, there must exist the smallest finite K s.t. for all $k \geq K$, each $\mathcal{S}_k$ contains no states from $\mathcal{S} \setminus \mathcal{S}^*$, i.e., $\mathcal{S}^*$ contains the set of states in $G_{s_0}^{V_k}$. Moreover, for each $k \geq K$, there must exist some n_k s.t. in all REVISE steps after the n_k-th and until FIND-AND-REVISE halts, FIND-AND-REVISE uses only actions that cause transitions only from a state in $\mathcal{S}^*$ to a state in $\mathcal{S}^*$. That it, for each ϵ_k s.t. $k \geq K$, the last few value function updates made by FIND-AND-REVISE over states in $\mathcal{S}^*$ before halting at V_k do not depend on the values of states outside of $\mathcal{S}^*$. If this was not the case for some $k \geq K$ then $G_{s_0}^{V_k}$ would necessarily contain a state outside $\mathcal{S}^*$, contradicting the fact that $K - 1$ was the largest number for which this happens.

Crucially, $\mathcal{S}^*$ contains no states without a proper policy. Suppose for contradiction that $\mathcal{S}^*$ contains such a state s_I. Every policy would accumulate an infinite expected cost from s_I. This would imply that for a sufficiently small ϵ, no value function can be ϵ-consistent at s_I and all of its descendants in the MDP's transition graph, all of which must be in $\mathcal{S}^*$ if s_I is. This is contrary to the fact that for any state in $\mathcal{S}^*$, including s_I and its descendents, and an arbitrarily small ϵ_k, there is a value function V_k that is ϵ_k-consistent at all of them. Thus, s_I cannot be in $\mathcal{S}^*$.

Moreover, $\mathcal{S}^*$ must contain the goal. As shown above, there is a K s.t. for all $k \geq K$, FIND-AND-REVISE's last few Bellman backups before FIND-AND-REVISE halts at V_k use only actions that cause transitions within $\mathcal{S}^*$. That is, for all $k \geq K$, any V_k-greedy policy from s_0 reaches only the states of $\mathcal{S}^*$. This means that, simlar to the hypothetical case of $\mathcal{S}^*$ containing a state with no proper policy, if $\mathcal{S}^*$ did not contain a goal then the value of every state in $\mathcal{S}^*$ would grow without bound under Bellman backup updates, and hence no value function could be ϵ-consistent over $\mathcal{S}^*$ for a sufficiently small ϵ. However, by the construction of $\mathcal{S}^*$, a monotonic V_k ϵ_k-consistent over $\mathcal{S}^*$ exists for an arbitrarily small ϵ_k, implying that at least one goal state must be in $\mathcal{S}^*$.

To sum up, so far we have established that $\mathcal{S}^*$ contains the initial state, the goal, and no states without a proper policy. In light of these facts, note that the restriction of the original SSPADE$_{s_0}$ problem to $\mathcal{S}^*$ is an SSP$_{s_0}$ problem. For each $k \geq K$, computing an ϵ_k-consistent V_k effectively involves running FIND-AND-REVISE's last few iterations on this SSP$_{s_0}$ MDP. At the time when FIND-AND-REVISE starts running on this SSP$_{s_0}$ MDP, the value function is admissible and monotonic. Running FIND-AND-REVISE on a SSP$_{s_0}$ MDP starting from an admissible monotonic value function (heuristic) until ϵ-convergence is known to yield the optimal value function (and policy) as $\epsilon \to 0$ (Theorem 32 in [11], Theorem 3 in [12]). Thus, we know that the sequence $\{V_k\}_{k=K}^{\infty}$ converges to V^* over the states in $\mathcal{S}^*$.

It remains to prove that $\mathcal{S}^*$ contains all states reachable from s_0 by at least one optimal policy for the original SSPADE$_{s_0}$ MDP. As already established, for all $k \geq K$, all actions greedy w.r.t. V_k in any state of $\mathcal{S}^*$ can only cause transitions to states of $\mathcal{S}^*$. Since $\{V_k\}_{k=K}^{\infty}$ converges to V^* over $\mathcal{S}^*$, there must exist a finite $K' \geq K$ s.t. for all $k \geq K'$ all actions greedy w.r.t. V_k in any state of $\mathcal{S}^*$ are optimal (i.e., greedy w.r.t. V^* as well). Since s_0 is also in $\mathcal{S}^*$ and all such actions can only cause transitions within $\mathcal{S}^*$, all states reachable by an optimal s.d.M. policy from s_0 must be in $\mathcal{S}^*$. ■

Theorem 5.16. *fSSPUDE*$_{s_0}$ $=$ *SSP*$_{s_0}$. ♢

Proof. To show that every fSSPUDE$_{s_0}$ MDP $M_{\text{fSSPUDE}_{s_0}}$ can be converted to an SSP$_{s_0}$ MDP, we augment the action set $\mathcal{A}$ of fSSPUDE$_{s_0}$ with a special action a' that causes a transition to a goal state from any non-goal state with probability 1 and that costs D. If applied in a goal state, it leads back to the same state and costs 0. The resulting MDP is an in stance of *SSP*$_{s_0}$, since reaching the goal with certainty is possible from every state. At the same time, the optimization criteria of fSSPUDE$_{s_0}$ and SSP clearly yield the same set of optimal policies for it.

To demonstrate that every SSP$_{s_0}$ MDP $M_{SSP_{s_0}}$ is also an fSSPUDE$_{s_0}$ MDP, for every $M_{SSP_{s_0}}$ we can construct an equivalent fSSPUDE$_{s_0}$ MDP by setting $D = \max_{s\in\mathcal{S}} V^*(s)$. The set of optimal policies of both MDPs will be the same. (Note that although the conversion procedure is impractical, since it assumes that we know V^* before solving $M_{SSP_{s_0}}$, solving $M_{SSP_{s_0}}$ by itself will yield an optimal policy for its fSSPUDE$_{s_0}$ counterpart.) ■

Theorem 5.17. *On a MAXPROB MDP M_{MP} derived from a goal-oriented MDP M, VI$_{MP}$ converges to the optimal value function P^* as its number of iterations tends to infinity.* ♢

Proof. Consider the MDP $\hat{M}_{MP}$ that results from eliminating all potential traps from M_{MP} using Algorithm 5.2. M_{MP} is also a GSSP$_{s_0}$ MDP, by Theorem 5.13, so $\hat{M}_{MP}$ is an SSP$_{s_0}$ MDP, as explained in the proof of the Optimality Principle for GSSP$_{s_0}$ MDPs (Theorem 5.1). Moreover, as explained in that proof, the expansion (Definition 5.9) of the optimal value function $\hat{V}^*$ of $\hat{M}_{MP}$ is the optimal value function of M_{MP}. The result now follows from Theorem 2.13, since to solve M_{MP}, VI$_{MP}$ runs VI on $\hat{M}_{MP}$. ■

Theorem 5.18. *For an iSSPUDE$_{s_0}$ MDP M with $P^*(s_0) > 0$, the MAXPROB-optimal derivative M^{P^*} of M is an SSP$_{s_0}$ MDP.* ♢

Proof. To prove the claim, we verify that M^{P^*} meets the SSP$_{s_0}$ MDP definition (2.19). By contruction, M^{P^*} has no dead ends, since such states of M have $P^*(s) = 0$ and are not included into M^{P^*}'s state space. Hence, M^{P^*} has a complete proper policy. Moreover, since the cost function of M^{P^*} agrees with the cost function of M over transitions within M^{P^*}'s state space, every trajectory possible in M^{P^*} is also possible in M and has the same cost as in M. By the iSSPUDE$_{s_0}$ MDP definition (5.15), each trajectory possible in M that never reaches the goal has an infinite cost. Therefore, each trajectory possible in M^{P^*} that never reaches the goal has an infinite cost too. This, in turn, implies that in M^{P^*}, every policy improper w.r.t. a state s must have incur an infinite expected cost starting from s, since executing such a policy results in an infinite goal-free trajectory with a positive probability. Thus, M^{P^*} satisfies both of Definition 2.19's requirements (inherited from Definition 2.16). ■

Theorem 5.19. *For an iSSPUDE$_{s_0}$ MDP M with $P^*(s_0) > 0$, every optimal s.d.M. policy for M's MAXPROB-optimal derivative M^{P^*} is optimal w.r.t. the goal-reaching probability in M, i.e., $P^{\pi^*}(s) = P^*(s)$ for all states s of M s.t. $P^*(s) > 0$.* ♢

Proof. By Theorem 5.18, M^{P^*} is an SSP$_{s_0}$ MDP, and by Theorem 2.4 all of its optimal policies are proper. Consider an optimal policy π^* for M^{P^*}. Since its proper, from every state of M^{P^*} all of π^*'s trajectories whose probability under π^* is positive eventually reach the goal. Therefore, in M, all of π^*'s positive-probability trajectories that visit only states with $P^*(s) > 0$ must reach the goal too. Therefore, since π^*, by the construction of M^{P^*}, uses only P^*-greedy actions of M, in M it must reach the goal with the maximum possible probability, $P^*(s)$, from every state s. ■

Theorem 5.20. *For a given tuple of MDP components $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0, \rangle$ satisfying the conditions of the SSP$_{s_0}$ MDP definition (2.19), there exists a finite penalty D_{thres} s.t. every optimal s.d.M. policy of every fSSPUDE$_{s_0}$ MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, D, s_0\rangle$ for any $D > D_{thres}$ is optimal for the MAXPROB MDP derived from this fSSPUDE$_{s_0}$ MDP as well.* ◇

Proof. The intuition behind the proof is as follows. For an fSSPUDE$_{s_0}$ MDP, one may theoretically prefer a policy with a lower goal-reaching probability P^π (Equation 5.15) to a policy with a higher one, because, despite the increased probability of hitting a state with penalty D, the expected cost of the former policy's trajectories may be significantly lower, which makes the policy less costly overall. However, for a sufficiently large penalty D, the increase in a policy's expected cost due to a higher chance of hitting a dead end cannot be offset by a lower expected cost of its trajectories. Under such a penalty, to be optimal in terms of expected cost, a policy needs to have the highest possible probability of reaching the goal, i.e., needs to be optimal under the MAXPROB criterion.

For an s.d.M. policy π for an fSSPUDE$_{s_0}$ MDP, let $\underline{V}^\pi(s)$ be the expected cost that π's trajectories originating at s incur before they reach a terminal state. Here, by a terminal state we mean either the goal or a state from which π cannot reach the goal (a "dead end under π"). Note that visiting any dead end under π makes π incur the penalty of D — this penalty is not part of a trajectory's cost as we defined it above, and does not affect $\underline{V}^\pi(s)$. For any s.d.M. policy and state, the length of all trajectories up to the point of entering a terminal state is almost surely bounded; therefore, since all actions costs are bounded as well, $\underline{V}^\pi(s)$ is finite.

Further, define two quantities, $\underline{V}^\pi_+(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \max\{0, C_t^{\pi_s} | \textit{nonterminal}\}\right]$ and $\underline{V}^\pi_-(s) =$

$\mathbb{E}\left[\sum_{t=0}^{\infty} \min\{0, C_t^{\pi_s} | \textit{nonterminal}\}\right]$. In these formulas, $C_t^{\pi_s} | \textit{nonterminal}$ are random variables for the costs incurred at the t-th step of π's trajectories provided that the trajectories have not reached a terminal state yet. Thus, $\underline{V}_+^{\pi}(s)$ is the expected sum of nonnegative costs incurred by π's trajectories before they reach a terminal state, and $\underline{V}_-^{\pi}(s)$ is the expected sum of nonpositive such costs. Both $\underline{V}_+^{\pi}(s)$ and $\underline{V}_-^{\pi}(s)$ are finite, for the same reason as $\underline{V}^{\pi}(s)$. Crucially, for each s.d.M. π and $s \in \mathcal{S}$, $\underline{V}_-^{\pi}(s) \leq \underline{V}^{\pi}(s) \leq V_+^{\pi}(s)$, i.e., $\underline{V}_-^{\pi}(s)$ and $\underline{V}_+^{\pi}(s)$ bound the expected cost of each policy's trajectories from above and below. Therefore, the quantities $\underline{V}_{min} = \min_{s \in \mathcal{S}, \pi \text{ is s.d.M.}} \underline{V}_-^{\pi}(s)$ and $\underline{V}_{max} = \max_{s \in \mathcal{S}, \pi \text{ is s.d.M.}} \underline{V}_+^{\pi}(s)$ are, respectively, a lower and an upper bound on the expected cost of trajectories of any s.d.M. policy from any state, and both are finite because the number of s.d.M. policies is finite and each $\underline{V}_-^{\pi}(s)$ and $\underline{V}_+^{\pi}(s)$ is finite.

Using the trajectory cost bounds we just derived, we can bound the value of any s.d.M. policy for an fSSPUDE$_{s_0}$ MDP under the total expected cost criterion. Recall from Equation 5.15 that every s.d.M. policy can be characterized in terms of its goal-probability value function P^{π}. The goal-probability value function gives rise to the following bounds on the expected-cost value function:

$$V^{\pi}(s) \geq \underline{V}_{min} + (1 - P^{\pi}(s))D$$
$$V^{\pi}(s) \leq \underline{V}_{max} + (1 - P^{\pi}(s))D$$

These bounds hold because the value of any policy for an fSSPUDE$_{s_0}$ MDP consists of the expected cost of its trajectories and the penalty for hitting a state with a high cost of reaching the goal, weighted by the probability of visiting such a state.

We are now ready to show that for any fSSPUDE$_{s_0}$ MDP, for any penalty D above a certain finite threshold, no MAXPROB-suboptimal s.d.M. policy can be as good in terms of expected cost as a MAXPROB-optimal one. This implies that every cost-optimal s.d.M. policy must necessarily be MAXPROB-optimal. Consider a MAXPROB-optimal s.d.M. policy π' (at least one such policy must exist, according to Theorem 5.1, since every MAXPROB MDP is also a GSSP$_{s_0}$ MDP) and a MAXPROB-suboptimal s.d.M. policy π for the same fSSPUDE$_{s_0}$ MDP. Since π is MAXPROB-suboptimal, there exists a state s' for which $P^{\pi'}(s') = P^*(s') > P^{\pi}(s)$. Suppose π was at least as good as π' in terms of expected cost. Then at s' (and possibly some other states), we would have $V^{\pi'}(s') - V^{\pi}(s') \geq 0$. In light of the above bounds, $V^{\pi}(s') \geq \underline{V}_{min} + (1 - P^{\pi}(s'))D$ and $V^{\pi'}(s') \leq \underline{V}_{max} + (1 - P^{*}(s'))D$, so we would have

$$\begin{aligned}
0 &\leq V^{\pi'}(s') - V^{\pi}(s') \\
&\leq (\underline{V}_{max} + (1 - P^{\pi}(s'))D) - (\underline{V}_{min} + (1 - P^{*}(s'))D) \\
&= (\underline{V}_{max} - \underline{V}_{min}) + (P^{\pi}(s') - P^{*}(s'))D
\end{aligned}$$

In the last line, $\underline{V}_{max} - \underline{V}_{min} \geq 0$ by the definition of $\underline{V}_{max}$ and $\underline{V}_{min}$, and $P^{\pi}(s') - P^{*}(s') < 0$ because, by our assumption, π is MAXPROB-suboptimal at s'. Now, consider a penalty threshold $D_{thres} = \frac{\underline{V}_{max} - \underline{V}_{min}}{\Delta_{min} P}$, where $\Delta_{min} P = \min_{s \in \mathcal{S}, \pi'' \text{is s.d.M.}} \{P^{*}(s) - P^{\pi''}(s) \mid P^{*}(s) - P^{\pi''}(s) > 0\}$. Unless all s.d.M. policies in the MDP are MAXPROB-optimal (in which case the theorem holds vacuously), $\Delta_{min} P > 0$, because the number of s.d.M. policies for an fSSPUDE$_{s_0}$ MDP is finite. Therefore, $D_{thres} < \infty$. For a $D > D_{thres}$,

$$\begin{aligned}
V^{\pi'}(s') - V^{\pi}(s') &\leq (\underline{V}_{max} - \underline{V}_{min}) + (P^{\pi}(s') - P^{*}(s'))D \\
&= (\underline{V}_{max} - \underline{V}_{min})(1 - \frac{P^{*}(s') - P^{\pi}(s')}{\Delta_{min} P}) \\
&< 0,
\end{aligned}$$

contradicting our assumption that π is at least as good as π' in terms of expected cost. Thus, for $D > D_{thres}$, every cost-optimal policy must be MAXPROB-optimal. ■