1-1-2017

Improved Heuristic Search Algorithms for Decision-Theoretic Planning

Ibrahim Abdoulahi

Improved heuristic search algorithms for decision-theoretic planning

By

Ibrahim Abdoulahi

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2017

Improved heuristic search algorithms for

decision-theoretic planning

By

Ibrahim Abdoulahi

Approved:

Eric Hansen
(Major Professor)

Christopher Archibald
(Committee Member)

Ioana Banicescu
(Committee Member)

Donna Reese
(Committee Member)

T. J. Jankun-Kelly
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering

Name: Ibrahim Abdoulahi

Date of Degree: December 8, 2017

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Eric Hansen

Title of Study: Improved heuristic search algorithms for decision-theoretic planning

Pages of Study: 106

Candidate for Degree of Doctor of Philosophy

A large class of practical planning problems that require reasoning about uncertain outcomes, as well as tradeoffs among competing goals, can be modeled as Markov decision processes (MDPs). This model has been studied for over 60 years, and has many applications that range from stochastic inventory control and supply-chain planning, to probabilistic model checking and robotic control. Standard dynamic programming algorithms solve these problems for the entire state space. A more efficient heuristic search approach focuses computation on solving these problems for the relevant part of the state space only, given a start state, and using heuristics to identify irrelevant parts of the state space that can be safely ignored. This dissertation considers the heuristic search approach to this class of problems, and makes three contributions that advance this approach.

The first contribution is a novel algorithm for solving MDPs that integrates the standard value iteration algorithm with branch-and-bound search. Called branch-and-bound value iteration, the new algorithm has several advantages over existing algorithms. The sec-

ond contribution is the integration of recently-developed suboptimality bounds in heuristic search algorithm for MDPs, making it possible for iterative algorithms for solving these planning problems to detect convergence to a bounded-suboptimal solution. The third contribution is the evaluation and analysis of some techniques that are widely-used by state-of-the-art planning algorithms, the identification of some weaknesses of these techniques, and the development of a more efficient implementation of one of these techniques – a solved-labeling procedure that speeds converge by leveraging a decomposition of the state-space graph of a planning problem into strongly-connected components. The new algorithms and techniques introduced in this dissertation are experimentally evaluated on a range of widely-used planning benchmarks.

Key words: Markov Decision Process, Planning under Uncertainty, Value Iteration, Heuristic Search, Suboptimality Bounds, Action Elimination

DEDICATION

"What does it matter to the one who has smelt the dust of Ahmad's grave

That they should never smell the finest perfumes?

Calamities have been poured on me

Had they been poured on days they would have turned into eternal nights."

– "Umm abiha", God bless them both and give them peace

ACKNOWLEDGEMENTS

I am very grateful to my advisor, Dr. Eric Hansen, for support and continuous supervision throughout the course of this research. I would like to thank Dr. Christopher Archibald, Dr. Ioana Banicescu and Dr. Donna Reese for being part of my committee and their valuable feedback on my work.

I am grateful to my current and former labmates: Arindam Khaled, Jinchuan Shi, Peng Zhao, Ibrahim K. Asif, Sanyam Satia, Mohammed Safayet Arefin, Kazi Siddiqui, Proteek Roy and Anara Kozhokanova.

I am very grateful to my friends and roommates Olanrewaju Raji and Hasan Toprak, whose friendship and moral support have been invaluable over the past few years.

I express my gratitude to my friends who made my stay in Starkville more enjoyable (in alphabetical order): Bilal Abdul Azeez, Abu Salah, Satam Alotibi, Hussein Alsabbahi, Nouman "Al-shazawi", Youssef Hammi, Ismael Khan, Naseer Kutchy, Afzaal Nadeem, Muhammad Nadeem, Muhammad Riaz, Ahmad Salem, Rani Sullivan, Mike and Marcie White, and many others to whom I apologize for not citing their names here.

I am very grateful to Arsalan Adil and Omar Shubailat. Your friendship has been a source of spiritual growth.

I would like to thank the local Muslim community for their friendship and support, because of them Starkville felt almost like home.

Last but not least, I am especially grateful to my family, for their patience during my prolonged absence over the past several years. Their moral support and prayers have meant a lot to me.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The objective of this work is to develop new algorithms, and improve existing algorithms, for decision-theoretic planning problems that are modeled as Markov decision processes. This chapter motivates the research and outlines the contributions of this dissertation.

1.1 Motivation

Planning is a widely-studied area of artificial intelligence that deals with choosing and applying a sequence of actions in order to achieve a goal state, while also optimizing some measure of the quality of the plan, such as the cost or time to achieve the goal. Classical planning assumes that all actions have deterministic outcomes. Under this assumption, plans can be found using algorithms for deterministic shortest-path problems, including the heuristic search algorithm A* [46], and its variants [47]. However, purely deterministic planning is not adequate for many real-word problems where the agent does not have full control of the outcomes of its actions. Planning under uncertainty relaxes the deterministic assumption by allowing actions with stochastic outcomes. A problem of planning under uncertainty is typically formalized as a Markov Decision Process (MDP) [4, 32]. An MDP

models action uncertainty by describing the effects of actions as probability distributions over possible successor states. The MDP model is described in detail in Chapter 2.

Problems of decision-theoretic planning are commonly modeled by a special class of MDPs called stochastic shortest path problems [35]. A stochastic shortest path problem (SSP) is an undiscounted Markov decision process with an absorbing and zero-cost goal state where the objective is to reach the goal state with minimum expected cost. An SSP problem can also be viewed as a generalization of the classic shortest path problem that allows actions to have stochastic outcomes. The SSP framework has been widely used in decision-theoretic planning applications, including robot navigation [16], probabilistic model checking [25], supply-chain management [57], and many others.

An SSP problem is solved by finding a conditional plan, or policy, that minimizes the expected cost of reaching a goal state, starting from any start state. Two well-known dynamic programming (DP) algorithms for solving SSP problems are value iteration (VI) [4] and policy iteration (PI) [32]. Both algorithms solve a problem for the entire state space. However, many planning problems are vulnerable to *Bellman's curse of dimensionality*, that is, the size of the state space grows exponentially with the number of variables used to describe the problem domain. Thus, computing a solution for all states can be infeasible for large problems.

When the start state is given, computing a solution for the entire state space is often unnecessary. Heuristic search algorithms take advantage of this fact to focus computation, which means they do not have to evaluate the entire state space in order to find an optimal policy for the start state. For example, well-known heuristic search algorithms such as A*

and AO* [46] solve a problem only for parts of the state space that need to be considered to find an optimal path from the start state to the goal state. In this work, we consider heuristic search algorithms for SSP problems that focus computation in a similar way. The difference is that the A* algorithm finds simple sequential plans and the AO* algorithm finds conditional plans that include branches, while the heuristic search algorithms we consider in this dissertation find conditional plans that include cycles as well as branches.

Two important strategies for using heuristic search in solving SSP problems have been developed. The first, introduced by Barto et al. [2], is called real-time dynamic programming (RTDP). The second, introduced by Hansen et al., is called LAO* [29]. The two algorithms are similar in that both perform backups only for states that are reachable from the start state by choosing actions greedily based on the current cost-to-go function. Both are guaranteed to converge *in the limit* to the optimal cost-to-go function, without necessarily evaluating the entire state space, provided that the initial cost-to-go function is an underestimate of the optimal cost-to-go function, that is, it is an *admissible* heuristic. The difference between the two approaches is that RTDP uses trial-based exploration, originally developed for solving deterministic shortest path problems using the Learning Real-Time A* algorithm [36], while LAO* uses a systematic search approach that generalizes the AO* algorithm for AND/OR graph search.

This dissertation adopts the systematic search approach introduced by LAO* because it has the advantage that it computes a Bellman residual each iteration for the subset of reachable states under a greedy policy, which can be used to monitor the quality of the current policy each iteration. It computes a Bellman residual as a result of performing,

each iteration, a complete depth-first traversal of the set of states reachable from the start state by following a greedy policy. The systematic approach introduced by LAO* has also been shown to converge much faster than the trial-based approach for RTDP.

1.2 Contributions

This dissertation makes the following contributions.

1.2.1 Branch-and-Bound Value Iteration

The value iteration algorithm updates the cost-to-go function for the entire state space, each iteration. Therefore, VI tends to be very slow for solving real-world problems because the state space is very large. However, many actions happen to be suboptimal, that is, they would never be part of an optimal policy. Furthermore, for a given start state s_0, updating parts of the state space that are unreachable from s_0 is unnecessary. With respect to state s_0, not updating unreachable states, does not prevent convergence to an optimal policy. Given a start state, the convergence of value iteration can be accelerated by pruning irrelevant states and actions. We describe a *Branch-and-Bound Value Iteration* algorithm the generalizes the classic action elimination procedure. The algorithm uses bounds on the optimal cost-to-go function to speed up the convergence of value iteration by eliminating unreachable states as well as suboptimal actions. We use state counters to minimize the overhead of state elimination. In addition to state and action elimination, the algorithm uses the bounds to efficiently detect convergence to an ϵ-optimal policy. A solution is ϵ-optimal when the difference between the upper and lower bounds is less than $\epsilon > 0$.

Because it is based on value iteration, the branch-and-abound algorithm preserves the nice properties of value iteration, in addition of other unique features. The algorithm is described and empirically evaluated in Chapter 3.

1.2.2 Integration of Suboptimality Bounds in Heuristic Search

The bounds used by the branch-and-bound value iteration algorithm described in Chapter 3 are for discounted infinite-horizon MDPs. No similar bounds have been available for undiscounted MDPs, included SSP problems. As a result, heuristic search algorithms for SSP problems, as well as the value iteration algorithm on which they are based, have lacked an efficient test for convergence to an ϵ-optimal policy. In practice, these algorithms most often compute an ϵ-consistent solution, which does not provide genuine suboptimality bounds. Computing meaningful suboptimality bounds requires separately computing both upper and lower bounds on the current cost-to-go function, which incurs substantial extra overhead – especially since useful initial upper bounds are rarely available, and their computation incurs additional overhead.

The bounds can be used in any algorithm for solving SSP problems that computes a Bellman residual, including value iteration, and also systematic heuristic search algorithms that compute a Bellman residual, such as LAO*. Computing the bounds incurs no extra overhead besides the overhead for computing the Bellman residual, which is already computed by these algorithms.

Chapter 4 describes an integration, and an experimental evaluation of these new bounds in a variation of the LAO* algorithm, called *Focused Value Iteration*, which uses the same

search strategy as LAO*. The integration of these bounds allows heuristic search algorithms for SSP problems to find bounded-suboptimal solutions of any degree of approximation.

1.2.3 Improved Solved-Labeling in Heuristic Search

Solved-labeling, a technique used in the classic AO* algorithm, has been implemented in heuristic search algorithms for probabilistic planning, including Labeled RTDP [12], HDP [10] and LDFS [14], to speed up convergence by labeling states as "solved", allowing the algorithm to focus computation on the parts of the state space that are still unsolvable. We consider and analyze these algorithms to identify their strengths and weaknesses. We focus on HDP because it elegantly generalizes solved-labeling by using Tarjan's algorithm to decompose the policy graph into strongly connected components. However, HDP keeps a stack that creates unnecessary overhead in Tarjan's algorithm. Furthermore, HDP uses a Find-and-Revise framework that prevents it from computing a Bellman residual each iteration, thus it cannot use the practical suboptimality bounds of Chapter 4 to monitor the quality of the policy. We introduce a version of Focused Value Iteration that uses an improved and more efficient implementation of solved-labeling that addresses both weaknesses. The Labeled Focused Value Iteration algorithm is described and evaluated in Chapter 5.

1.3 Guide to the dissertation

Chapters 3, 4, and 5 describe the contributions of this work, as just summarized. Chapter 6 provides some additional perspective and discusses future work. An appendix describes some of the problems used for testing.

CHAPTER 2

BACKGROUND

This chapter reviews relevant background for the following chapters. It is organized as follows. Section 2.1 motivates the use of the Markov decision process framework in decision-theoretic planning. Section 2.2 reviews a special class of Markov decision processes: the stochastic shortest path problem. Section 2.3 surveys computational methods for solving stochastic shortest path problems. Section 2.5 presents the PPDDL language, which is widely-used to describe probabilistic planning problems.

2.1 Markov Decision Processes

Problems of decision-theoretic planning often share the following characteristics: they evolve through a sequence of "states". The transition between the "states" occurs following an action taken by the agent at different time steps. At any time, the state of the problem is completely observable. The actions can be stochastic, that is, the agent does not have full control of the outcomes of an action, which, when executed, can result in the system transitioning into several possible successor states with a given probability distribution. Executing an action comes with a certain cost (or reward) and the objective is to minimize (or maximize) the expected accumulated cost (or reward) over time. We assume that the objective is always to minimize the expected total cost. Problems with these features

are typically modeled as fully observable Markov decision processes (MDPs) [5, 35, 50]. Formally, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, c)$ where:

- $\mathcal{S}$ is the state space, which is a finite set of all possible states of the problem;
- $\mathcal{A}$ is a finite set of actions, and $\mathcal{A}(s)$ is the set of actions applicable in state s;
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ defines the the transition function, where $\mathcal{T}(s, a, s') = p_{s,s'}(a)$ is the probability of the agent transitioning to successor state s' after taking action a in state s; and
- $c : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the cost per step, where $c(s, a)$ denotes the immediate cost incurred when the agent takes action a in state s.

By assumption, the system under study is *stage-invariant*, that is, action costs and transition functions do not change over time.

The objective is to find a *policy*, that is, a mapping from states to actions, $\mu : \mathcal{S} \to \mathcal{A}$, that minimizes expected total cost. The policy μ is a particular course of action to be adopted by the agent, where in each state $s \in \mathcal{S}$, a feasible action $\mu(s) \in \mathcal{A}(s)$ is taken. The expected total cost for a policy μ is defined as

$$J_\mu(s_0) = E\left[\sum_{t=0}^{\infty} \beta^t \cdot c(s_t, \mu(s_t))|s_0 = s\right], \tag{2.1}$$

where E is the expectation operator, s_t is the state at step t, and s_0 is the start state. The *discount factor* β is a positive real number with $0 < \beta \leq 1$. When $\beta < 1$, it means that future costs matter less than the present ones and the problem is said to be discounted; it is undiscounted otherwise. The discount factor is sometimes interpreted as an interest rate, a probability of living another step, or, more practically, as a trick to bound the infinite sum of Equation (2.1) and ensure the existence of a stationary policy for state s_0.

2.2 Stochastic Shortest Path Problems

A stochastic shortest path (SSP) problem [8] is an undiscounted MDP defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, c, \mathcal{G})$. The parameters $\mathcal{S}, \mathcal{A}$ and $\mathcal{T}$ are defined as in the regular MDP problem. The set $\mathcal{G} \subseteq \mathcal{S}$ of goal states is absorbing and zero-cost. For every state $s \in \mathcal{G}$, no action takes the agent outside of $\mathcal{G}$ and for every action a and state $s \in \mathcal{G}$, $c(s, a) = 0$. The objective is to find a course of action that reaches the goal set with probability one, and has minimum expected total cost. Such problems are called SSP problems because they can be viewed as a generalization of the classic deterministic shortest path problem in which actions can have stochastic outcomes. For an SSP problem, an optimal policy must be proper.

Definition 2.2.1 (Proper policy)**.** *A policy $\mu : \mathcal{S} \to \mathcal{A}$ is said to be proper if the agent is guaranteed to reach the goal state in a finite expected number of steps when policy μ is followed, beginning from any start state. It is said to be improper otherwise.*

For an SSP problem, the cost-to-go function for a policy μ can be defined as follows:

Definition 2.2.2 (Cost-to-go function)**.** *A cost-to-go function $J_\mu(s)$ gives the expected cost of reaching the goal set from any state $s \in \mathcal{S}$, by following policy μ. It is given by:*

$$J_\mu(s) = \begin{cases} 0 & \text{if } s \in \mathcal{G}, \\ c(s, \mu(s)) + \sum_{s' \in \mathcal{S}} p_{s,s'}(\mu(s)) J_\mu(s') & \text{otherwise}. \end{cases} \tag{2.2}$$

For a proper policy μ, $J_\mu(s)$ is finite for every state $s \in \mathcal{S}$. Furthermore, the following assumptions are standard for SSP problems:

(a) There exist at least one proper policy, and

(b) For every improper policy μ, the corresponding cost $J_\mu(s)$ is infinite for at least one state s.

These assumptions ensure the existence of at least one proper policy, and, for every improper policy μ, at least one state $s \in \mathcal{S}$ for which the expected cumulative cost J_μ, of following μ starting from state s is infinite. Note that they generalize the familiar assumptions for a deterministic shortest path problem: (a) there exists a path from each state to the goal state, and (b) any cycle in a path has infinite cost.

The optimal cost-to-go function $J^*(s)$, for each $s \in \mathcal{S}$, is the unique solution of the Bellman optimality equation defined as follows:

$$J^*(s) = \begin{cases} 0 & \text{if } s \in \mathcal{G} \\ \min\limits_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum\limits_{s' \in \mathcal{S}} p_{s,s'}(a) J^*(s') \right] & \text{otherwise.} \end{cases} \tag{2.3}$$

The greedy policy μ^* with respect to J^* is an optimal policy, and is given by:

$$\mu^*(s) = \operatorname*{argmin}_{a \in \mathcal{A}(s)} \left\{ c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J^*(s') \right\}, \quad s \in \mathcal{S}. \tag{2.4}$$

Similarly, for any stationary policy μ, the costs $J_\mu(s), s \in \mathcal{S}$, are the unique solution to the equation

$$J_\mu(s) = c(s, \mu(s)) + \sum_{s' \in \mathcal{S}} p_{s,s'}(\mu(s)) J_\mu(s'), \quad s \in \mathcal{S}, \tag{2.5}$$

and given an arbitrary initial cost-to-go function $J_0(s), s \in \mathcal{S}$, the sequence (J_k) generated by the update

$$J_k(s) = c(s, \mu(s)) + \sum_{s' \in \mathcal{S}} p_{s,s'}(\mu(s)) J_{k-1}(s'), \quad s \in \mathcal{S}, \tag{2.6}$$

converges to $J_\mu(s)$, for each $s \in \mathcal{S}$.

Definition 2.2.3 (ϵ-optimal cost-to-go function)**.** *The cost-to-go function J is ϵ-optimal if $|J(s) - J^*(s)| \leq \epsilon$ for all states $s \in \mathcal{S}$.*

Definition 2.2.4 (ϵ-optimal policy)**.** *The policy μ^k is ϵ-optimal if $|J_{\mu^k}(s) - J^*(s)| \leq \epsilon$ for all states $s \in \mathcal{S}$.*

A policy μ^* is optimal if $J_{\mu^*} \leq J_\mu$ for all states $s \in \mathcal{S}$ and policies μ, that is, μ^* minimizes the expected cost-to-go of reaching a goal set starting from any start state.

2.3 Value Iteration

This section reviews value iteration, which is the most widely-used dynamic programming algorithm for solving SSP problems.

Let $(J_k), k \in \mathbb{N}$, be a sequence generated by the dynamic programming update

$$J_k(s) = \min_{a \in \mathcal{A}(s)} \left[c(s, a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J_{k-1}(s') \right], \quad s \in \mathcal{S}, \tag{2.7}$$

starting with an arbitrary initial cost-to-go function $J_0(s), s \in \mathcal{S}$. It has been proved that $(J_k), k \in \mathbb{N}$, converges to the optimal cost $J^*(s)$ for each state $s \in \mathcal{S}$.

Value iteration (VI) is an iterative algorithm that uses Equation (2.7) to compute the optimal cost-to-go function by successive approximations. Starting with an initial cost-to-go function $J_0 : \mathcal{S} \to \mathbb{R}$, VI performs a sequence of Bellman updates or *backups* for all states that iteratively improve the cost-to-go function. When the maximum difference between values of two consecutive iterations (Bellman residual) is equal to zero, the algorithm is said to have converged. VI is guaranteed to converge, in the limit, to the optimal cost-to-go function J^*, starting with an arbitrary initial cost-to-go function, given that the assumptions of the SSP problem hold [8]. In practice however, VI is stopped when the Bellman residual is sufficiently small.

A single Bellman backup for a state $s \in \mathcal{S}$ runs in worst-case $O(|\mathcal{A}(s)||\mathcal{S}|)$ time, since it requires iterating over all actions and all successor states. A single iteration requires $|\mathcal{S}|$ backups, so an iteration has quadratic complexity in the number of states. But since most systems are sparse, the actual complexity of an iteration is typically linear in the number of states. The pseudocode of VI is given in Algorithm 1.

Algorithm 1: Value Iteration

Input: Set of states $\mathcal{S}$, real number ϵ, initial cost-to-go function J_0
Output: An ϵ-consistent cost-to-go function J_k

Algorithm `VI()`
  $k \leftarrow 0$
  while *($\overline{c}_k > \epsilon$ or timeout)* **do**
    $k \leftarrow k + 1$
    $\overline{c}_k \leftarrow 0$
    for $s \in \mathcal{S}$ **do**
      $J_k(s) = \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J_{k-1}(s') \right]$
      $\overline{c}_k = \max\left(\overline{c}_{k-1}, |J_k(s) - J_{k-1}(s)|\right)$

Definition 2.3.1 (Bellman residual)**.** *Let* $(J_k), k \in \mathbb{N}$ *be the sequence defined by Equation (2.7). The Bellman residual,* $\bar{c}_k$*, is equal to* $\max_{s \in \mathcal{S}} |J_k(s) - J_{k-1}(s)|$.

Definition 2.3.2 (ϵ-consistent policy)**.** *The policy* μ *is said to be* ϵ*-consistent when the Bellman residual is smaller than* ϵ*.*

Gauss-Seidel Updates

In iteration k, the dynamic programming update in the pseudocode of Algorithm 1 uses the cost-to-go values from iteration $k-1$ to compute the new values. These updates are called Jacobi updates. The convergence of VI can often be accelerated by replacing Jacobi updates with Gauss-Seidel updates defined as

$$J_k(s) = \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum_{s'=1}^{s-1} p_{s,s'}(a) J_k(s') + \sum_{s'=s}^{n} p_{s,s'}(a) J_{k-1}(s') \right], \tag{2.8}$$

where $J_k(s')$ is used instead of $J_{k-1}(s')$ whenever $J_k(s')$ is already available. The amount of speedup achieved by using Gauss-Seidel updates is problem-dependent. It also depends on the quality of the ordering of states. Gauss-Seidel updates, with a good ordering of states can often accelerate the convergence of value iteration. Gauss-Seidel updates also save memory because they only require one copy of the cost-to-go function vector to be kept in memory, instead of two.

VI eventually converges no matter what order states are updated in each iteration. However, its performance can be significantly improved by choosing an intelligent order-

ing of backups. Several variations of VI that exploit the structure of the problem to find a good ordering of backups have been developed [20, 21, 24, 35, 44].

2.4 Heuristic Search

This section reviews heuristic search algorithms that improve on value iteration by solving the problem for a given start state, without necessarily considering all states.

The complexity of VI is proportional to the state space size since it computes a best action for every state. But considering the entire state space is unnecessary when all that is needed is a policy for a particular start state. When a start state is given, any state that cannot be reached from the start state can be ignored, which can drastically reduce the amount of computation needed for solving a given SSP problem. Heuristic search algorithms use the start state and a heuristic function to focus only on relevant states for computing an optimal policy for the start state. The set of relevant states can be arbitrarily smaller than the entire state space, thus saving on computational resources.

Definition 2.4.1 (Heuristic function)**.** *A heuristic function h is a cost-to-go function estimate that is used to initialize the cost-to-go function of the states the first time they are visited.*

Definition 2.4.2 (Admissible heuristic function)**.** *A heuristic function h is admissible if $h(s) \leq J^*(s)\ \forall s \in \mathcal{S}$, that is, it is a lower bound on the optimal cost-to-go function. The heuristic said to be* inadmissible *otherwise.*

Definition 2.4.3 (Monotone cost-to-go function)**.** *A cost-to-go function J is monotone if it never decreases when updated, that is, it verifies the Equation*

$$J(s) \leq \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J(s') \right], \forall s \in \mathcal{S}. \tag{2.9}$$

The heuristic search algorithms we consider solve a restricted form of the SSP problem where the objective is to solve the problem for a given start state only. Whereas value iteration finds a *complete policy*, that is, a mapping from every state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}(s)$, heuristic search algorithms find a *partial policy* μ that only needs to be defined for a subset of states that includes all states that are reachable from the start state s_0 by following the policy μ.

To express this restriction, we qualify the meaning of some key concepts.

Definition 2.4.4 (Proper policy relative to a start state)**.** *A policy μ is* proper relative to a start state s_0 *if it ensures that a goal state is reached from the start state with probability one.*

Definition 2.4.5 (Optimal policy relative to a start state)**.** *A policy μ is* optimal relative to a start state s_0 *if $J_\mu(s_0) \leq J_{\mu'}(s_0)$, for all possible policies μ'.*

Definition 2.4.6 (ϵ-optimal policy relative to a start state)**.** *A policy μ is* ϵ-optimal relative to a start state s_0 *if $J_\mu(s_0) - \epsilon \leq J_{\mu'}(s_0)$, for all possible policies μ'.*

Note that there may exist a policy that is proper relative to a given start state even if there is not a proper policy *per se*, that is, a policy that is proper relative to all possible start

states. Similarly, a policy may be optimal (or ϵ-optimal) relative to a start state without being optimal (or ϵ-optimal) for all states.

Definition 2.4.7 (ϵ-consistent policy relative to a start state)**.** *The policy μ is said to be ϵ-consistent relative to a start state when the Bellman residual, computed over the states visited by policy μ, is smaller than ϵ.*

In the rest of this dissertation, we often write simply ϵ-consistent instead of ϵ-consistent relative to a start state, and rely on the context to indicate the meaning.

Definition 2.4.8 (SSP problem relative to a start state)**.** *An* SSP problem relative to a start state *is a problem for which the objective is to find a policy that is optimal (or ϵ-optimal) relative to the start state. The problem is well-defined under the following two assumptions that relax the two assumptions of an SSP problem per se: (i) there is a proper policy relative to the start state, and (ii) every improper policy relative to the start state has positive infinite cost for the start state.*

For an SSP problem relative to a start state, heuristic search algorithms can find an optimal (or ϵ-optimal) policy without evaluating the entire state space, allowing them to be more efficient than standard value iteration.

The rest of this section describes the RTDP and LAO* algorithms, which introduced the heuristic search approach for SSP problems.

2.4.1 RTDP

Real-Time Dynamic Programming (RTDP) [2] is an algorithm for time-constrained planning. RTDP generalizes the Learning Real-Time A* (LRTA*) algorithm [36], a heuris-

tic search algorithm for deterministic planning. RTDP quickly finds a reasonably good policy and improves it until it is optimal or time runs out. RTDP operates by executing the current greedy policy along a sample trajectory through the state space, beginning from the start state. RTDP only updates the cost-to-go function for states that are reachable from the start state by following the current greedy policy. Each trajectory, called a *trial*, consists of repeatedly selecting a greedy best action for the current state s, performing a Bellman backup on the value of s, and transitioning to a successor of s under the greedy policy until a goal state is reached. Assuming the initial cost-to-go function is an admissible heuristic, RTDP asymptotically converges to the optimal cost-to-go function (relative to the start state) if the initial cost-to-go function is a lower-bound function, that is, if it is an admissible heuristic, and the goal state is reachable from every state. In practice, the convergence of RTDP is slow because unlikely trajectories tend to be ignored by the greedy simulated exploration. In addition to the slow convergence, RTDP lacks a termination test Algorithm 2 gives the pseudocode for RTDP.

Many algorithms have been developed that adopt the RTDP approach [12, 22, 52, 54].

2.4.2 LAO*

The LAO* algorithm [29] solves the same class of problems as RTDP, but adopts an approach that generalizes the classic heuristic search algorithm AO* [46]. LAO* uses systematic search to find an ϵ-consistent policy for the start state. LAO* gradually constructs a greedy policy graph rooted at the start state, following the best action for each state. In each iteration k, states in the fringe of the greedy policy graph are expanded until no

Algorithm 2: RTDP algorithm

Input: SSP problem with start state s_0
Output: policy for start state

Algorithm `RTDP` (s_0)
  repeat
    $s \leftarrow s_0$
    while *s is not a goal state* **do**
      foreach *action a* **do**
        $Q(s,a) \leftarrow c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J(s')$
      Select a best action $a^* \leftarrow \operatorname{argmin}_{a \in \mathcal{A}(s)} Q(s,a)$
      Update cost-to-go function $J(s) \leftarrow Q(s,a^*)$
      Sample next state s' with probability $p_{s,s'}(a^*)$ and set $s \leftarrow s'$
  until *timeout*

unexpanded states remain in the fringe, in which case the policy is said to be *closed*. The expansion step is typically guided by a heuristic function. LAO* performs a depth-first traversal of states rooted at the start state, following the greedy policy. When a state s is first visited, a backup is performed and the best action is identified. Each successor state s' is pushed onto the (implicit) stack used to organize the depth-first traversal, if the state s' has not already been visited in iteration k. The variable s'.UPDATE (line 14 of Algorithm 4) indicates whether state s' has been visited yet in iteration k. The non-goal fringe states are expanded and initialized by their heuristic value. Once the expansion step is done, the depth-first traversal stack contains all states in the current greedy graph. States on the stack are then updated in postorder backup. The process is repeated until the cost-to-go function is ϵ-consistent. The pseudocode of LAO* is given by Algorithm 3.

Algorithm 3: LAO* algorithm

Input: SSP problem with start state s_0
Output: ϵ-consistent policy for start state

Algorithm LAO*(s_0)
  // Assume s.STATUS $= unvisited$ and s.UPDATE $= 0$ for all $s \in \mathcal{S}$
  $k \leftarrow 0$ // global variable k is iteration count
  s_0.STATUS $\leftarrow$ *open*
  repeat
    repeat
      $k \leftarrow k + 1$ // begin new iteration
      $\bar{c}_k \leftarrow$ LAO*rec(s_0)
    until $\bar{c}_k \neq nil$ // nil indicates open policy
  until $\bar{c}_k \leq \epsilon$ // test for ϵ-consistent policy

Algorithm 4: Recursive function called in LAO*

Function LAO*rec(s)
  if *s is a goal state* **then**
    return 0
  if s.STATUS $=$ *open* **then**
    s.STATUS $\leftarrow$ *closed*
    foreach $a \in \mathcal{A}(s)$ *and* $s' \in Succ(s, a)$ **do**
      if s'.STATUS$=$ *unvisited* **then**
        $J_k(s') \leftarrow h(s')$ // h is admissible
    return nil // residual is undefined for open policy
  $\mu^k(s) \leftarrow \operatorname{argmin}_{a \in \mathcal{A}(s)} [c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J_{k-1}(s')]$ // update policy
  $J_k(s) \leftarrow \min_{a \in \mathcal{A}(s)} [c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J_{k-1}(s')]$ // pre-order backup
  $residual \leftarrow J_k(s) - J_{k-1}(s)$ // residual is local variable
  foreach $s' \in Succ(s, \mu^k(s))$ **do**
    if s'.UPDATE $< k$ **then** // s' not already updated this iteration
      s'.UPDATE $\leftarrow k$
      $r \leftarrow$ LAO*rec (s') // r is best residual of descendent
      if *($r = nil$)* **then**
        $residual \leftarrow nil$
      else if *($residual \neq nil$) and ($r > residual$)* **then**
        $residual \leftarrow r$
    $J_k(s) \leftarrow \min_{a \in \mathcal{A}(s)} [c(s,a) + \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J_{k-1}(s')]$
    return $residual$

Before a goal state is reached, the choice of the best policy is only based on the heuristic values. Depending on the quality of the heuristic, LAO* may expand a significant part of the state space before finding at least one goal state.

Many subsequent heuristic search algorithms have adopted the approach of LAO* [9, 10, 14, 15, 19].

Both RTDP and LAO* converge asymptotically to an optimal policy for the start state without necessarily evaluating the entire state space. In practice, these algorithms are terminated after a finite number of iterations. While LAO* is terminated when the Bellman residual is smaller than some threshold ϵ, RTDP does not have a test for termination.

We adopt the systematic search approach of LAO* because it has the advantage that it computes a Bellman residual each iteration for the subset of reachable states, which can be used to monitor the quality of the current policy each iteration. Systematic search also has the advantage of performing complete depth-first traversals of the set of states reachable following the greedy policy, which are used in a technique for speeding up the convergence of algorithms we describe later.

2.5 Factored Representation for Planning Problems

The formal definition of an SSP problem says little about the actual description of a problem domain and its instances. A practical question is therefore how to effectively describe an actual instance of a problem. The most direct approach is to explicitly enumerate all states and actions, as well as the transition function, but this representation can take a huge amount of space for large problems.

Fortunately, many problems have a lot structure that can be leveraged to represent the problem more compactly. In a *factored* representation, states and actions are encoded using domain variables. The state space is not explicitly generated in this case. The representation only gives rules for generating the state space. Formally, a factored SSP problem is defined by the tuple $(\mathcal{X}, \mathcal{S}, \mathcal{A}, \mathcal{T}, c, \mathcal{G})$ where:

- $\mathcal{X} = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_n\}$ is a set of state or domain variables with possible values in sets $dom(\mathcal{X}_1), \ldots, dom(\mathcal{X}_n)$ respectively. The domain variables are typically binary, that is, $dom(\mathcal{X}_i) = \{True, False\}$, but not necessarily.
- $\mathcal{S} = (dom(\mathcal{X}_1) \times dom(\mathcal{X}_2) \times \ldots \times dom(\mathcal{X}_n))$ is a set of all possible states;
- $\mathcal{A}$ is a finite set of all actions applicable in $\mathcal{S}$;
- $\mathcal{T} : (dom(\mathcal{X}_1) \times \ldots \times dom(\mathcal{X}_n)) \times \mathcal{A} \times (dom(\mathcal{X}_1) \times \ldots \times dom(\mathcal{X}_n)) \rightarrow [0, 1]$ defines the the transition function;
- $c : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the cost per step;
- $\mathcal{G}$ is the goal set, a set of states satisfying the the goal encoding (set of literals over variables $\mathcal{X}_i$).

Different research communities use different languages for factored representations. The AI planning community, for example, uses PDDL (Planning Domain Definition Language) [43] and RDDL (Relational Dynamic Influence Diagram Language) [51]. In probabilistic model checking, a similar language, called the PRISM language [38], is used.

For the test problems in this dissertation, we use an extension of PDDL called Probabilistic PDDL (PPDDL) [59] that allows modeling of probabilistic planning problems with rewards/costs. PPDDL is the language used to encode the International Probabilistic Planning Competition (IPPC) benchmarks.

PDDL (as well as PPDDL) domains are usually described in two files: the first file defines the domain dynamics (state variables, actions), and the second file defines an in-

stance of the problem. A problem domain consists of domain constants, a set of predicates, a set of functions and a set of actions. Predicates and functions are used to encode domain variables. A state is a particular assignment of the domain variables. Actions represent the set of state transitions. An action has three components: *parameters*, *preconditions* (which predicates have to be true in a state for the action to be applicable in that state) and *effects* (what becomes true about the state, as a result of the action execution). Actions can be probabilistic, that is, applying an action can result in more than one possible successor state. An instance is given by the description of the start and goal states.

Example of Problem Description in PPDDL

Listing 2.1 gives the description of the *Blocksworld* problem domain from the International Probabilistic Planning Competition of 2008. For this domain, a state can have a maximum of six actions. Listing 2.2 gives the description of instance $p01$ of this domain.

```
1 (define (domain blocks domain)
2   (:requirements :probabilistic effects :conditional effects :equality :typing :rewards↩
        )
3   (:types block)
4   (:predicates (holding ?b   block) (emptyhand) (on table ?b   block) (on ?b1 ?b2   ↩
        block) (clear ?b   block))
5
6   (:action pick up
7     :parameters (?b1 ?b2   block)
8     :precondition (and (emptyhand) (clear ?b1) (on ?b1 ?b2))
9     :effect (and (decrease reward 1) (probabilistic 3/4 (and (holding ?b1) (clear ?b2) ↩
          (not (emptyhand)) (not (on ?b1 ?b2))) 1/4 (and (clear ?b2) (on table ?b1) (not ↩
          (on ?b1 ?b2))))))
```

```
10
11  (:action pick up from table
12    :parameters (?b   block)
13    :precondition (and (emptyhand) (clear ?b) (on table ?b))
14    :effect (and (decrease reward 1) (probabilistic 3/4 (and (holding ?b) (not (↩
          emptyhand)) (not (on table ?b))))))
15
16  (:action put on block
17    :parameters (?b1 ?b2   block)
18    :precondition (and (holding ?b1) (clear ?b1) (clear ?b2) (not (= ?b1 ?b2)))
19    :effect (and (decrease reward 1) (probabilistic 3/4 (and (on ?b1 ?b2) (emptyhand) (↩
          clear ?b1) (not (holding ?b1)) (not (clear ?b2))) 1/4 (and (on table ?b1) (↩
          emptyhand) (clear ?b1) (not (holding ?b1))))))
20
21  (:action put down
22    :parameters (?b   block)
23    :precondition (and (holding ?b) (clear ?b))
24    :effect (and (on table ?b) (emptyhand) (clear ?b) (not (holding ?b))))
25
26  (:action pick tower
27    :parameters (?b1 ?b2 ?b3   block)
28    :precondition (and (emptyhand) (clear ?b1) (on ?b1 ?b2) (on ?b2 ?b3))
29    :effect (probabilistic 1/10 (and (holding ?b2) (clear ?b3) (not (emptyhand)) (not (↩
          on ?b2 ?b3)))))
30
31  (:action put tower on block
32    :parameters (?b1 ?b2 ?b3   block)
33    :precondition (and (holding ?b2) (on ?b1 ?b2) (clear ?b3) (not (= ?b1 ?b3)))
34    :effect (probabilistic 1/10 (and (on ?b2 ?b3) (emptyhand) (not (holding ?b2)) (not ↩
          (clear ?b3))) 9/10 (and (on table ?b2) (emptyhand) (not (holding ?b2)))))\\
35
36  (:action put tower down
37    :parameters (?b1 ?b2   block)
```

```
38     :precondition (and (holding ?b2) (on ?b1 ?b2))
39     :effect (and (on table ?b2) (emptyhand) (not (holding ?b2)))))
```

Listing 2.1: Blocksworld domain in PPDDL

```
1 (define (problem blocks p01)
2   (:domain blocks domain)
3   (:objects b1 b2 b3 b4 b5   block)
4   (:init (emptyhand) (on table b1) (on table b2) (on b3 b5) (on b4 b1) (on table b5) (↩
        clear b2) (clear b3) (clear b4))
5   (:goal (and (emptyhand) (on b1 b3) (on b2 b4) (on table b3) (on b4 b1) (on b5 b2) (↩
        clear b5))))
```

Listing 2.2: Blocksworld problem instance $p01$

Figure 2.1 shows the start state and the goal state from Listing 2.2. A policy for this problem instance is the sequence of moves that transforms the start state into the goal state. Figure 2.2 shows the execution of an optimal policy computed for this problem instance. The optimal policy graph has branches. Tor example, action *pick-up (b_1 ,b_4)* has two possible successors states. There are also loops in the graph. For example, action *pick-up-from-table (b_1)* loops back to the same state in addition to another successor state.

Although many of the test problems solved in this dissertation are represented compactly in PPDDL, the algorithms described in this dissertation do not leverage this representation to improve scalability. Other algorithms do, however, including Symbolic value

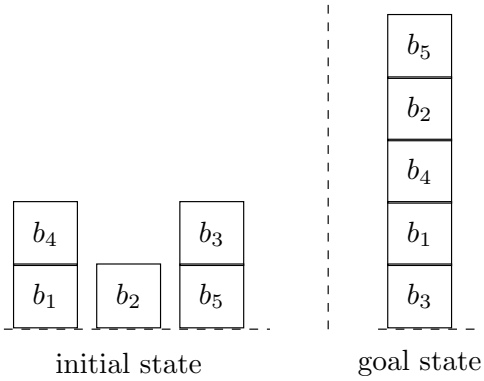

Figure 2.1: Start and goal states of the Blocksworld problem instance $p01$.

iteration [31], Symbolic LAO* [23], Symbolic Bounded RTDP [22], and symbolic algorithms for probabilistic model checking [37]. Most of these algorithms use Algebraic Decision Diagrams(ADDs) [1, 17], as memory-efficient data structures. They are called symbolic algorithms because they leverage problem structure to perform backups on sets of states, instead of individual states.

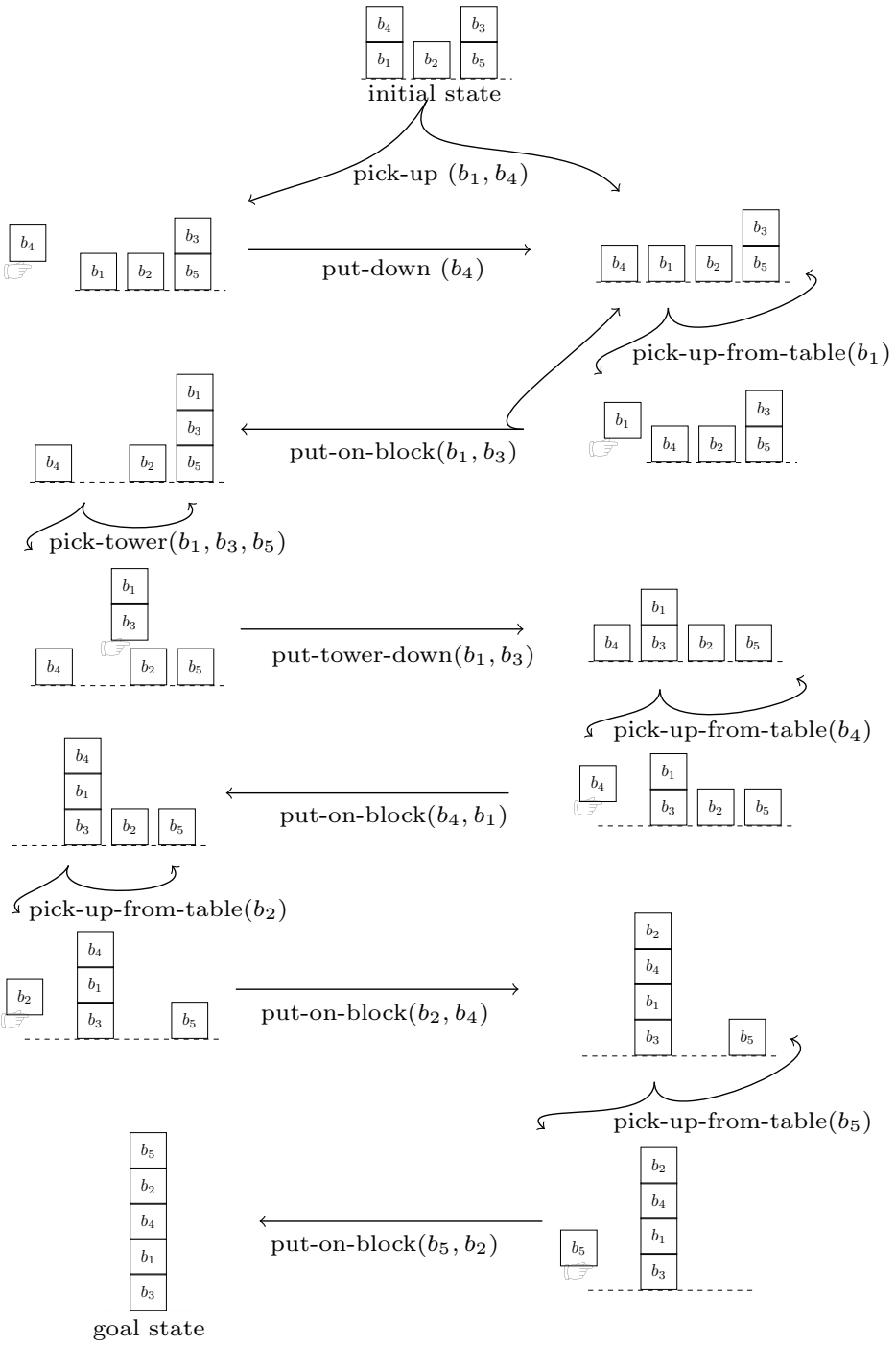

Figure 2.2: Execution of optimal policy for Blocksworld problem instance $p01$.

CHAPTER 3

BRANCH-AND-BOUND VALUE ITERATION

In this chapter we describe an approach to solving Markov decision processes that integrates branch-and-bound search with the value iteration algorithm. The approach generalizes the classic action elimination technique where bounds on the optimal cost-to-go function are used not only to eliminate sub-optimal actions, but to eliminate states that are unreachable from the start state under an optimal policy [40, 48]. The approach accelerates the convergence of value iteration by focusing computation on the relevant parts of the problem. In addition to speeding up the convergence of value iteration, our algorithm can detect when all but one action has been eliminated for each state, thus allowing early termination. We use discounted infinite-horizon MDP problems to experimentally evaluate its performance. The results show a significant advantage of the branch-and-bound algorithm over regular value iteration with and without action elimination.

The chapter is organized as follows. Section 3.1 reviews the discounted infinite-horizon Markov decision problem and how to solve it. Section 3.2 reviews how to derive bounds used in the action elimination procedure. Section 3.3 gives the details of the branch-and-bound value iteration algorithm. Section 3.4 describes the test problems and presents the experimental results.

3.1 Background

A discounted infinite-horizon Markov decision process is an MDP where future costs are discounted. The discounted factor is β, with $0 < \beta < 1$. Let $\mathcal{S} = \{1, 2, \ldots, n\}$ be the finite set of states of the MDP. Let $\mathcal{A}(s)$ be the set of actions possible in state s. Let $c(s, a)$ be the immediate cost of taking $a \in \mathcal{A}(s)$ in state s. The expected total discounted cost $J_\mu(s)$, incurred by following a policy μ starting from a start state s_0 is given by

$$\lim_{N\to\infty} J_\mu(s_0) = E\Big[\sum_{t=0}^{N} \beta^t \cdot c(s_t, \mu(s_t))|s_0 = s\Big], \;\; s \in \mathcal{S}. \tag{3.1}$$

The optimal cost-to-go function $J^*(s), s \in \mathcal{S}$ satisfies the condition,

$$J^*(s) = J_{\mu^*}(s) \leq J_\mu(s) \;\; \forall \mu \in M, \tag{3.2}$$

where $M = \{\mu|\mu(s) \in \mathcal{A}(s), s \in \mathcal{S}\}$ is the set of deterministic stationary policies.

An optimal policy can be found using the value iteration algorithm. Starting from an arbitrary cost-to-go vector J_0, value iteration generates a sequence $(J_k), k \in \mathbb{N}$, of increasingly better estimates of the the optimal cost-to-go function by repeatedly performing the dynamic programming update,

$$J_k(s) = \min_{a\in\mathcal{A}(s)} \Big[c(s, a) + \beta \cdot \sum_{s'\in\mathcal{S}} p_{s,s'}(a) J_{k-1}(s')\Big], \;\; s \in \mathcal{S}. \tag{3.3}$$

The dynamic programming update of Equation (3.3) is called a Jacobi dynamic programming update. Jacobi updates use a copy of the cost vector, J_{k-1}, from the previous iteration

to update the values in the current iteration. One of the ways of accelerating value iteration is to replace the Jacobi update with the Gauss-Seidel dynamic programming update given by

$$J_k(s) = \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \beta \cdot \sum_{s'=1}^{s-1} p_{s,s'}(a) J_k(s') + \beta \cdot \sum_{s'=s}^{n} p_{s,s'}(a) J_{k-1}(s') \right]. \tag{3.4}$$

With Gauss-Seidel updates, the most up-to-date cost-to-go estimate of successor states $s' \in \mathcal{S}$ is used, when updating the cost-to-go estimate of state s. In addition, there is no need to store the cost vector from the previous iteration. The speedup can be arbitrary large, provided we have a good state ordering. Gauss-Seidel updates are more widely-used than Jacobi updates in practice.

For both Jacobi and Gauss-Seidel updates, value iteration is guaranteed to converge to the unique fixed point J^* that solves the Bellman equation,

$$J^*(s) = \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \beta \cdot \sum_{s' \in \mathcal{S}} p_{s,s'}(a) J^*(s') \right], \quad s \in \mathcal{S}. \tag{3.5}$$

A greedy policy with respect to J^* is guaranteed to be optimal.

Because value iteration finds an optimal policy for the entire state space, it considers every action in all states and does not take advantage of knowledge of start state. Several approaches, including action elimination, have been used to speed up its convergence. Action elimination uses upper and lower bounds on the optimal cost-to-go function to prune provably sub-optimal actions. Suboptimality bounds for discounted MDPs are discussed in the next section, as well as how they are used for action elimination.

3.2 Suboptimality Bounds for Discounted Infinite-Horizon MDPs

Consider the sequence $(J_k), k \in \mathbb{N}$, of improved estimates of the optimal cost-to-go function J^*, generated by the value iteration algorithm, using the Jacobi dynamic programming updates of Equation (3.3). The following upper and lower bounds on the optimal cost-to-go function J^*, are due to McQueen and Porteus [42, 48, 49]:

$$J_k(s) + \left(\frac{\beta}{1-\beta}\right) \cdot \underline{c}_k \leq J^*(s) \leq J_k(s) + \left(\frac{\beta}{1-\beta}\right) \cdot \overline{c}_k, \;\; s \in \mathcal{S}, \tag{3.6}$$

where the *residuals* $\underline{c}_k$ and $\overline{c}_k$ are given by

$$\underline{c}_k = \min_{s \in \mathcal{S}} \left(J_k(s) - J_{k-1}(s)\right) \tag{3.7}$$

and

$$\overline{c}_k = \max_{s \in \mathcal{S}} \left(J_k(s) - J_{k-1}(s)\right). \tag{3.8}$$

The bounds of Equation (3.6) can be used in value iteration to detect convergence to an ϵ-optimal policy. A greedy policy μ^k with respect to J_{k-1} is ϵ-optimal if the following inequality holds:

$$\left(\frac{\beta}{1-\beta}\right) \cdot (\overline{c}_k - \underline{c}_k) \leq \epsilon. \tag{3.9}$$

The bounds can also be used in a test for action suboptimality originally proposed by McQueen [41]. An action $a \in \mathcal{A}(s)$ is sub-optimal, and can be eliminated, if

$$c(s,a) + \beta \sum_{s' \in S} p_{s,s'}(a) \underline{J}_{k-1}(s') > \overline{J}_k(s), \tag{3.10}$$

where $\overline{J}_k$ and $\underline{J}_{k-1}$ are upper and lower bounds on the optimal cost-to-go function, respectively. The inequality of Equation (3.10) means that if the lower bound on the cost-to-go function of state s with respect to action a is strictly larger than the upper bound on the optimal cost-to-go function, then this action cannot be an optimal action for state s. Provably sub-optimal actions can be eliminated without affecting the correctness of the optimal solution computed by value iteration.

Using Equation (3.6), the action elimination test for action $a \in \mathcal{A}(s)$ can be written as follows:

$$c(s,a) + \beta \sum_{s' \in S} p_{s,s'}(a) \underline{J}_{k-1}(s') > \left(\frac{\beta}{1-\beta} \right) \cdot (\overline{c}_k - \underline{c}_k). \tag{3.11}$$

Because the test for suboptimality of Equation (3.11) depends on the residuals $\overline{c}_k$ and $\underline{c}_k$, which are available only at the end of iteration k, action elimination is delayed to the end of iteration k. The drawback of delaying action elimination is that the values $c(s,a)+\beta \sum_{j \in S} p_{s,s'}(a) \underline{J}_{k-1}(s')$ from the previous iteration need to be stored or recalculated every iteration.

A solution was proposed by Hastings and Mello [30]. The values of the residuals $\overline{c}_{k-1}$ and $\underline{c}_{k-1}$, from iteration $k-1$, are related to their values in iteration k as follows:

$$(\overline{c}_k - \underline{c}_k) \leq \beta \cdot \left(\overline{c}_{k-1} - \underline{c}_{k-1} \right) \tag{3.12}$$

The residuals $\overline{c}_{k-1}$ and $\underline{c}_{k-1}$ can therefore be used in iteration k to do action elimination. The action elimination test for action $a \in \mathcal{A}(s)$ becomes

$$c(s,a) + \beta \sum_{j \in S} p_{s,s'}(a) \underline{J}_{k-1}(s') > \left(\frac{\beta^2}{1-\beta} \right) \cdot (\overline{c}_{k-1} - \underline{c}_{k-1}). \tag{3.13}$$

The action elimination test of Equation (3.13) is based on the bounds of Equation (3.6), which assume that value iteration uses Jacobi updates. Under Gauss-Seidel updates, the following suboptimality bounds hold [48, 49]:

$$J_k(s) + \left(\frac{\beta}{1-\beta} \right) \cdot \min\{0, \underline{c}_k\} \leq J^*(s) \leq J_k(s) + \left(\frac{\beta}{1-\beta} \right) \cdot \max\{0, \overline{c}_k\}. \ s \in \mathcal{S}. \tag{3.14}$$

Given the inequality of Equation (3.12), we have the following action elimination test for Gauss-Seidel updates. An action a for state s is suboptimal if:

$$c(s,a) + \beta \sum_{s'=1}^{s-1} p_{s,s'}(a) J_k(s') + \beta \cdot \sum_{s'=s}^{n} p_{s,s'}(a) J_{k-1}(s') > J_k(s') + \left(\frac{\beta^2}{1-\beta} \right) \cdot \left(\max\{0, \overline{c}_{k-1}\} - \min\{\underline{c}_{k-1}, 0\} \right). \tag{3.15}$$

The bounds of Equation (3.14) are the ones we use in the branch-and-bound value iteration algorithm, described in the next section.

3.3 Branch-and-Bound Value Iteration

This section shows how the bounds of the previous section are used in branch-and-bound value iteration. We assume that we are solving an MDP for a given start state.

The algorithm we propose is a generalization of the classic action elimination method. In action elimination, upper and lower bounds on the optimal cost-to-go function are used to identify and eliminate suboptimal actions, which speeds up convergence because the eliminated actions do not need to be evaluated in subsequent iterations.

The starting point of the new algorithm is the observation that as a result of action elimination, parts of the state space may become unreachable from the start state because actions leading to them have been pruned. When states are unreachable from the start state, it is unnecessary to update their cost-to-go estimates in subsequent iterations because, with respect to the start state, updating unreachable parts of the state space has no effect on the optimal policy. By ignoring unreachable states we speed up the convergence of value iteration because only a limited part of the state space may need to be evaluated in order to find an optimal solution for the start state.

The convergence of value iteration is therefore accelerated not only by eliminating suboptimal actions, but also by eliminating states that are unreachable from the start state. In addition to the action elimination test of Equation (3.15), we introduce a test for state elimination that is be performed after a successful action elimination test.

In order to make the state elimination step as efficient as possible, the branch-and-bound value iteration algorithm begins with a pre-processing step. We first identify the set of states that are initially reachable from the start state by performing a breadth-first traversal of the graph of the MDP. During the graph traversal, every reachable state is assigned a counter which is equal to the number of incoming transitions for that state. The start state is assigned a counter of $+\infty$ because it should never be eliminated, regardless of whether

or not it has any incoming transitions. The counters allow state elimination with minimum overhead because they have a much lower overhead than repeated graph traversals. Furthermore, they allow value iteration to update states in any order, instead of the order in which states are visited during a breadth-first traversal. Each iteration of the algorithm, state counters are decremented every time actions leading to those states are eliminated. Whenever an action a is eliminated in state s, the counter is decremented for all successor states of s. A state becomes unreachable from the start state as soon as its counter is decremented to zero. The procedure for pruning states is recursive because when a state is eliminated, all of its actions are also pruned resulting in potentially more unreachable states. Algorithms 5 and 6 show the counter initialization and state elimination procedures, respectively.

Algorithm 5: Initialize-State-Counters

Input: $\mathcal{G}$, graph of the MDP
Output: $\mathcal{G}$, with state counters initialized

Algorithm `Initialize-State-Counters(`s_0`)`

  foreach state $s \in S - \{s_0\}$ of $\mathcal{G}$, set s.COUNTER equal to the number of transitions into state s that are reachable from the start state s_0, as follows:

    (a) for all $s \in \mathcal{S}$, set s.COUNTER $\leftarrow 0$.

    (b) perform a breadth-first traversal of graph $\mathcal{G}$ beginning from the state of the graph corresponding to the start state.

    (c) for each state-action pair that is visited, increment the counter of each successor state by 1.

  Set s_0.COUNTER $\leftarrow +\infty$, to ensure the start state is never eliminated.

Algorithm 6: Decrement-Counters-and-State-Elimination

Input: State s, action a, and iteration k

Output: Updated counters for state s and its successor states, after iteration k

```
1 Algorithm Decrement-Counters-and-State-Elimination(s, a, k)
2     foreach state s' for which p_{s,s'}(a) > 0 do
3         s'.COUNTER ← s'.COUNTER − 1
4         if s'.COUNTER == 0 then
5             S_k ← S_k − {s'} // eliminate state s'
6             foreach a' ∈ A_k(s'), recursively call
7             Decrement-Counters-and-State-Elimination(s', a', k)
```

After the initialization step, the branch-and-bound value iteration algorithm works just like regular value iteration except for the following differences. Each iteration, the algorithm

- only updates the states reachable from the start state instead of all states,
- calls the action and state elimination procedures to check for sub-optimal actions and unreachable states, and
- tests for early convergence, that is, it checks if every unpruned state has a unique action.

Algorithm 7 gives the pseudocode of the branch-and-bound value iteration algorithm.

The branch-and-bound value iteration algorithm has the following advantages. First, it improves the cost-to-go function by the same amount as regular value iteration in each iteration, assuming that both algorithms update states in the same order and begin with the same cost-to-go function estimate. It follows that it does not take more iterations to converge than regular value iteration. Second, both algorithms use the residuals $\overline{c}_k$ and $\underline{c}_k$ to test for convergence. Regular value iteration computes $\overline{c}_k$ and $\underline{c}_k$ for all states $\mathcal{S}$ af-

Algorithm 7: Branch-and-bound value iteration algorithm

Input: MDP problem, discount factor β and threshold $\epsilon > 0$
Output: An ϵ-optimal policy for the start state s_0

1 **Algorithm** `BBVI()`
2 **Initialization**:
3 $Initialize\text{-}State\text{-}counters(s_0)$ *// Invoke function defined by Algorithm 5.*
4 Set $k \leftarrow 0$, $\mathcal{S}_0 \leftarrow \{s \in \mathcal{S} |\ s.\text{COUNTER} > 0\}$, and $\mathcal{A}_0(s) \leftarrow \mathcal{A}(s)$, $\forall s \in \mathcal{S}_0$.
5 Initialize policy μ_0.
6 **Iteration** k: Set $k \leftarrow k+1$, $\mathcal{S}_k \leftarrow \mathcal{S}_{k-1}$, and $\mathcal{A}_k(s) \leftarrow \mathcal{A}_{k-1}(s)$.
7 **foreach** *state* $s \in \mathcal{S}_k$ **do**
8 Evaluate best action from previous iteration:
9 $a \leftarrow \mu^k(s) \leftarrow \mu^{k-1}(s)$
10 $J_k(s) \leftarrow c(s,a) + \beta \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s')$
11 **foreach** *of the remaining actions* $a \in \mathcal{A}_k(s) - \{\mu^{k-1}(s)\}$ **do**
12 – Evaluate and test for improvement:
13 $Q_k^a(s) \leftarrow c(s,a) + \beta \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s')$.
14 **If** $Q_k^a(s) < J_k(s)$ **then** set $J_k(s) \leftarrow Q_k^a(s)$ and $\mu^k(s) \leftarrow a$.
15 – Perform action and state elimination tests:
16 **If** $Q_k^a(s) > J_k(s) + \dfrac{\beta^2}{(1-\beta)} \cdot (\max\{0, \overline{c}_{k-1}\} - \min\{0, \underline{c}_{k-1}\})$,
17 **then** let $\mathcal{A}_k(s) \leftarrow \mathcal{A}_k(s) - \{a\}$, which eliminates action a, and
18 invoke $Decrement\text{-}Counters\text{-}and\text{-}State\text{-}Elimination(s,a,k)$.
19 **Test for convergence to unique optimal policy**:
20 **If** $|\mathcal{A}_k(s)| = 1$, for all $s \in \mathcal{S}_k$, then stop with optimal policy.
21 **Test for convergence to ϵ-optimal policy**:
22 **If** $\dfrac{\beta}{(1-\beta)} \cdot (\max\{0, \overline{c}_k\} - \min\{0, \underline{c}_k\}) \leq \epsilon$, **then** stop with ϵ-optimal policy.
If not converged, go to line 6.

ter iteration k, whereas branch-and-bound value iteration computes those values on the set of unpruned states $\mathcal{S}_k \subseteq \mathcal{S}$ after iteration k. It follows that the bounds computed for branch-and-bound value iteration are potentially tighter. Therefore, branch-and-bound value iteration can converge in fewer iterations than regular value iteration. Third, in addition to testing for convergence to an ϵ-optimal policy using the bounds, branch-and-bound value iteration also tests for convergence to an optimal policy (when all but one action has been eliminated for each state in $\mathcal{S}_k$). Using either convergence test, branch-and-bound value iteration can converge in fewer iterations than standard value iteration because each convergence test only considers the states in $\mathcal{S}_k$.

Ordering of Backups

The performance of value iteration with Gauss-Seidel updates is very dependent on the order of Bellman updates, each iteration. Without a good ordering of backups, Gauss-Seidel updates may have no advantage over Jacobi updates. When a start state is given, a good ordering can be determined by performing a breadth-first traversal of states reachable from the start state, and ordering states backwards from the goal, in the reverse order in which they are visited. By arranging states in this way, states values are propagated backwards from the goal in each iteration, which improves the rate of convergence.

Since the preprocessing step for initializing the counters performs a breadth-first traversal of reachable states, it can be used to determine a good ordering of backups.

3.4 Experimental Evaluation

This section presents an experimental evaluation of the branch-and-bound value iteration algorithm using the classic automobile replacement MDP and two different inventory control MDPs from the Operations Research community. All the test problems are discounted MDPs. We compare the branch-and-bound algorithm to both regular value iteration and value iteration with action elimination. To ensure a fair comparison, both branch-and-bound value iteration and regular value iteration update states in the same order each iteration.

3.4.1 Automobile Replacement

Howard's automobile replacement problem [32] is a discounted MDP, with 40 states, where each state has 41 actions. A stage in the problem is a period of three months. The state of the system is the age of the car in units of three months. A car of age 40 is considered to be in the worst condition possible and so ages no more. In each state at any stage, the possible actions are to replace the car with one of age k, where $0 < k < 39$, or to keep it for at least three more months. During any stage if the car suffers a serious breakdown a transition to state 40 is made, otherwise the car ages by 1 unit. If the car is replaced with one of age 0 or 39, or the car is kept when the system is in state 39 or 40, there is only one possible transition. In all other cases, there are two possible transitions. The parameters that define the problem are the cost of buying a car of age s, the trade-in value of a car of age s, the expected one stage operating cost for a car of age s, and the probability that a car of age s, will suffer a serious breakdown during a stage. A discount

	B&B VI	Action elimination	VI
Runtime (in CPU seconds)	0.002	0.003	0.006
Num. Iterations	58	58	101
Num. pruned actions	1,663	1,640	-
Num. pruned states	23	-	-
Num. backups	2,215	2,520	4,284

Table 3.1: Results for the auto replacement problem: 40 states, 41 actions/state, $\epsilon = 10^{-6}$.

factor $\beta = 0.97$ is used. The objective is to compute the best policy (keep or replace that car), for a given car of age k. This test problem dates back to the early 1960's, and is used as a test example in almost all papers that discuss action elimination.

Tables 3.1 shows the result of the comparison. The branch-and-bound value iteration algorithm performs better than the other two algorithms. It pruned about half of the state space, in addition to pruning suboptimal actions. Most of the improvement comes from action elimination. We are also able to detect early convergence when each unpruned state has only one action left. This test problem is too small to draw meaningful conclusions in terms of runtime. The next test problem is big enough to draw such conclusions.

3.4.2 Inventory Control

The inventory control problem [55] models the dynamics of a retailer who manages an inventory of a single item facing stochastic demand. The maximum capacity of the inventory is $M \in \mathbb{N}$. The stock is reviewed daily and the retailer must decide about the quantity to be ordered for the following day. The ordered quantity is received in the morning and the inventory is filled up. During the day, some random demand is realized.

We assume that the demand for the day is known to be non-negative, and that it is modeled as a sequence of independent and identically distributed integer-valued random variables. In particular we assume that the demand follows a *Poisson distribution* of parameter $\lambda \in \mathbb{N}$.

The sequence of events every day is as follows. At the beginning of the day, the inventory on hand is observed (X_t) and an order is placed (A_t), which will arrive the next morning. Next, the stochastic demand is realized and satisfied with on hand inventory. If the demand exceeds the the inventory, then we have an out-of-stock situation. Unmet demand is lost. The next morning, the order placed at the beginning of the day arrives and afterwards stochastic demand continues to occur, up until the beginning of the next ordering. The sequence of events is illustrated in Figure 3.1.

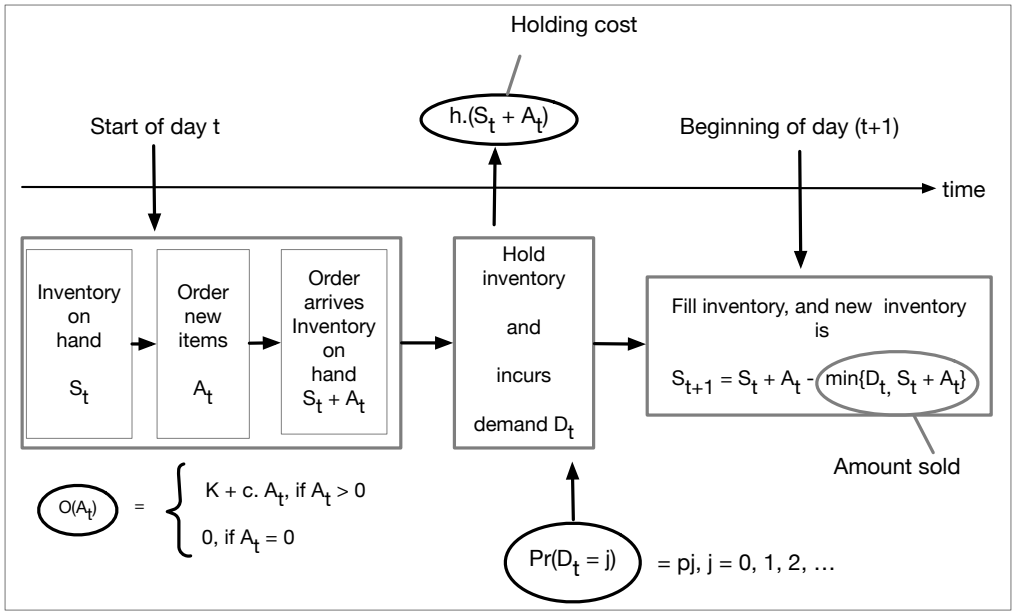

Figure 3.1: Inventory control problem.

The goal of the manager is to keep the inventory so as to maximize the value of the expected total future income (or minimize the overall cost).

We use the MDP framework to model this problem as follows. A state is given by the inventory on hand at the beginning of day t, X_t. An action is A_t, the number of items ordered at the end of day t. Let D_t be the demand occurring between the morning and the evening of the $(t)^{th}$ day. Given quantities X_t, A_t and D_t, the size of the inventory on the $(t+1)^{th}$ day, that is the next state X_{t+1}, is given by:

$$X_{t+1} = \max(\min(X_t + A_t, M) - D_{t+1}, 0), \quad t = 0, 1, 2, \ldots \tag{3.16}$$

The min operator ensures that X_{t+1} will not exceed the maximum capacity M, and the max operator enforces the fact that unmet demand is lost.

Because the demand is stochastic, we actually have a probability distribution over the set of possible next states $\{X_{t+1}\}$ given current state X_t, demand D_t, and action A_t.

The problem has the following cost components, on day t:

- A cost associated with ordering A_t items equal to $K + c \cdot A_t$, where the cost K is a fixed cost that is incurred when ordering a nonzero number of items, and the quantity c is the cost per ordered item. Both K and c are positive,
- A holding cost per unit of inventory, denoted by h. The cost of storing an inventory of size X_t is therefore $h \cdot X_t$.
- The manager is paid the amount p per sold item. We assume $p > h$, in order to give an incentive to order new items.

The overall cost on the $(t+1)^{th}$ day, for a given action A_t, in state X_t is given by:

$$c(X_t, A_t) = K + c \cdot \max\{\min(X_t + A_t, M) - X_t, 0\} + h \cdot X_t \\ - p \cdot \max\{\min(X_t + A_t, M) - X_{t+1}, 0\}, t = 0, 1, 2, \ldots \quad (3.17)$$

Since we do not have a goal state, we use a discount factor $0 < \beta < 1$, to ensure the Bellman operator is a contraction mapping.

Formally, the MDP is defined by the following parameters.

- The set of possible states is $\mathcal{S}$ and is given by $\mathcal{S} = \{0, 1 \ldots, M\}$,
- The set of possible actions is $\mathcal{A}$ and is given by $\mathcal{A} = \{0, 1 \ldots, M\}$,
- The cost function given by Equation (3.17) is defined for every possible successor state X_{t+1}. The cost of an action is averaged over possible successor states.

The stochasticity of the MDP comes from the uncertain demand and the nondeterminism from the fact the more than one action is possible.

The parameter of the Poisson distribution, λ can be seen as the average demand per day. We only consider demand values that are positive and less than the maximum inventory capacity. In order to limit the number of possible demand values, we limit ourselves to demand values in the interval $\left(\max\left(0, \lambda - \frac{|M|}{4}\right), \min\left(\lambda + \frac{|M|}{4}, M\right)\right)$. This means that the probability of the demand being outside the interval is equal to zero. We use a discount factor $\beta = 0.95$.

Tables 3.2 shows the performance of the algorithms on the inventory control problem. The first column of the table gives state space sizes as well as the parameters used to generate the problem instances. The branch-and-bound value iteration algorithm is over

an order of magnitude faster than regular value iteration, and about two times faster than value iteration with action elimination. For this domain, action elimination is especially effective because each state s can have up to s possible actions, most of which are suboptimal. This is the best case for action elimination. Since action and state elimination are correlated, only a small portion of the state space ends up to be relevant for the optimal policy. Note that the difference in the number of pruned actions is exactly equal to the number of pruned states. This is because for each state that is not pruned, all but one action is removed by action elimination. The branch-and-bound takes fewer iterations than the other algorithms because early convergence is detected, which means the algorithm can be terminated before convergence to an ϵ-optimal cost-to-go function. A hyphen in the table means that the metric does not apply to the algorithm.

It is worth noting that the optimal policy computed for the inventory control problems can be compactly characterized by a pair (s, S), where S is called the *target stock*, and s the *minimum refill*. Such policy is called the (s, S) $policy$. The parameter s means that if the inventory size drops to or below s, the optimal action is to order a quantity that would raise the inventory to the target stock S. No order should be placed as long as the inventory size is greater than the minimum refill s. The values of S and s depend on the cost structure of the problem, and the start state. The branch-and-bound value iteration algorithm is effective for this domain because any action greater than S is suboptimal, and any state greater than s is irrelevant, with respect to the start state.

Problem instance	Metrics	B&B VI	Action elimination	VI
$Inv90$	Runtime (in CPU seconds)	1.73	3.60	22.56
$\|S\|= 300$	Num. Iterations	115	116	287
$\lambda = 90$	Num. pruned actions	90,575	90,300	-
$(K, c, h, p) = (5, 2, 2, 3)$	Num. pruned states	275	-	-
	Num. backups	11,166	35,636	86,976
$Inv130$	Runtime	5.43	11.33	81.37
$\|S\|= 500$	Num. Iterations	153	154	310
$\lambda = 130$	Num. pruned actions	250,976	250,500	-
$(K, c, h, p) = (5, 2, 2, 3)$	Num. pruned states	476	-	-
	Num. backups	17,600	78,312	156,122
$Inv140$	Runtime	15.98	27.62	275.82
$\|S\|= 750$	Num. Iterations	132	134	285
$\lambda = 140$	Num. pruned actions	563,873	563,250	
$(K, c, h, p) = (5, 2, 2, 3)$	Num. pruned states	623	-	-
	Num. backups	33,294	102,272	215,072

Table 3.2: Performance on the inventory control problem, $\epsilon = 10^{-6}$. The parameters of a problem are: λ is the average demand, K is the cost of placing an order, c is the cost per ordered item, h is the cost of storage per item, and p is the amount paid to the manager per sold item.

3.4.3 Supply-chain Optimization

We next consider a more complex inventory management problem, used as a test problem by [29]. Consider the process of manufacturing lamps. The production depends on the supply of parts such as lamp shade, bulb, basement, and cable. It is desired to keep optimal levels of inventory at various stages of the chain of production depending on supply and demand. The demand for the lamp is stochastic. The problem is to determine how many units of the product to produce, what price to sell the product for, how many units of the part to order, and what inventory to keep on hand. The cost of parts goes down if they are ordered in advance. Each order has a guaranteed delivery time. The longer the period between ordering and delivery, the lower the cost. This reflects the fact that advance notice

allows the part manufacturer to increase its own efficiency. The problem instances used in the experiments are described in Table 3.3. The discount factor is $\beta = 0.95$. The objective is to keep levels of inventory, that optimize long-term total cost.

Table 3.3 shows the results for this domain. The results are consistent with the ones of the previous domain. However, this domain has fewer actions per state than the inventory control domain. Nevertheless, branch-and-bound value iteration is consistently faster than regular value iteration with and without action elimination. The branch-and-bound algorithm does not always find a unique optimal policy for this problem. This is due to either the presence of more than one best action for some states, or not enough pruning before convergence.

Problem instance	Metrics	B&B VI	Action elimination	VI
	Runtime (in CPU seconds)	1.94	4.18	4.30
DL8	Num. Iterations	293	293	293
$\|\mathcal{S}\|= 6,720$	Num. pruned actions	386,283	379,010	-
$\|\mathcal{A}\|= 144$	Num. pruned states	6,612	-	-
	Num. backups	17,707,527	20,338,679	113,998,620
	Runtime	1.75	3.21	4.75
DL9	Num. Iterations	250	252	252
$\|\mathcal{S}\|= 18,000$	Num. pruned actions	318,219	298,407	-
$\|\mathcal{A}\|= 27$	Num. pruned states	17,196	-	-
	Num. backups	12,019,907	19,586,598	81,081,372
	Runtime	1.78	4.12	5.42
DL16	Num. Iterations	296	297	297
$\|\mathcal{S}\|= 14,080$	Num. pruned actions	378,944	364,453	-
$\|\mathcal{A}\|= 48$	Num. pruned states	13,613	-	-
	Num. backups	13,748,442	22,622,743	113,464,520

Table 3.3: Results for the supply-chain problem, $\epsilon = 10^{-6}$.

3.5 Summary

In this chapter we described a generalization of the action elimination technique for discounted infinite-horizon MDPs, where unreachable states are pruned in addition to sub-optimal actions. We showed how to implement the state elimination technique with minimum overhead. The experimental evaluation showed a clear advantage of the branch-and-bound algorithm over regular value iteration with and without action elimination. The efficient state pruning mitigates the usual overhead for action elimination.

The new algorithm works best for problems with a lot of actions per state, and for which most of the actions are suboptimal. At the beginning of the search, not much pruning happens because the bounds are not tight enough. It might be worth delaying the action elimination procedure for the first few iterations until the bounds become useful. The use of state counters greatly reduces the overhead of state elimination. The drawback of initializing state counters is that the initialization requires visiting all reachable states from the start state. In the worst case, all states are reachable. However, the initialization step can be used to compute a good ordering of backups, which is important in speeding up value iteration based algorithms that use Gauss-Seidel backups.

The branch-and-bound value iteration algorithm described in this chapter used suboptimality bounds for discounted MDPs. In Chapter 4 we review bounds that can be used for undiscounted MDPs.

CHAPTER 4

INTEGRATION OF SUBOPTIMALITY BOUNDS IN HEURISTIC SEARCH

In this chapter, we introduce Focused Value Iteration (FVI), an efficient heuristic search algorithm for SSP problems. FVI is a simplification of the LAO* algorithm [29], that performs almost as well as LAO*, and is easier to analyze. It is introduced in one of our papers [28]. We also show how to integrate recently-derived suboptimality bounds in FVI, in order to test for convergence to an ϵ-optimal policy. The new bounds are due to Hansen [26]. The contribution of this chapter is to show how to integrate these bounds in heuristic search algorithms such as FVI, and to evaluate their performance. As we will see, the convergence of the bounds often follows a different pattern when they are used in a heuristic search algorithm than when they are used in value iteration.

This chapter is structured as follows. Section 4.1 describes the FVI algorithm. Section 4.2 describes efficient suboptimality bounds that can be used in heuristic search algorithms for SSP problems such as FVI to (a) test for convergence to an ϵ-optimal policy and (b) to detect when the greedy policy is proper. Section 4.3 shows how to integrate the suboptimality bounds in FVI. Section 4.4 experimentally evaluates the performance of the new bounds. Section 4.5 compares FVI to the branch-and-bound value iteration algorithm introduced in Chapter 3.

4.1 Focused Value Iteration

Focused Value Iteration (FVI) is a depth-first traversal based heuristic search algorithm. Although it is a new algorithm that has not previously been described, it is better to view it as a simplification of previously-described algorithms. In fact, it can be viewed as the simplest possible heuristic search algorithm for this class of problems.

FVI updates a cost-to-go function over a sequence of iterations, just like value iteration. But in each iteration k, FVI only updates the cost-to-go function for the subset of states reachable from the start state s_0 under the current best policy μ^k, denoted by $\mathcal{S}^{\mu^k}(s_0)$. Like other heuristic search algorithms, FVI assumes that the cost-to-go function is a lower-bound function, that is, it is an admissible heuristic.

In iteration k, FVI performs a depth-first traversal of the states in $\mathcal{S}^{\mu^k}(s_0)$. When a state $s \in \mathcal{S}^{\mu^k}(s_0)$ is first visited, a backup is performed and the best policy $\mu^k(s)$ is identified, then each successor state s' of state s, under the policy μ^k, is pushed onto the (implicit) stack used to organize the depth-first traversal, if the state s' has not already been visited in iteration k. The variable s'.UPDATE indicates whether state s' has been visited yet in iteration k. At the end of the traversal, a greedy policy μ^k has been found, and the cost-to-go function has been updated for all states in $\mathcal{S}^{\mu^k}(s_0)$.

The pseudocode for FVI is shown in Algorithm 8. FVI performs a pre-order and a post-order backup each iteration for each state $s \in \mathcal{S}^{\mu^k}(s_0)$ (lines 12 and 20 of the pseudocode, respectively). The pre-order backup, performed when the state is first visited, identifies the best action for the state. The post-order backup, which is performed when backtracking from the state, further improves the cost-to-go function. The post-order backup tends to

Algorithm 8: Focused Value Iteration algorithm

Input: SSP problem with state space $\mathcal{S}$, start state s_0
Output: ϵ-consistent policy for start state

Algorithm `FocusedVI(`s_0`)`
  Set $k \leftarrow 0$ *// global variable k is iteration count.*
  Repeat
    Set $k \leftarrow k + 1$
    s_0.UPDATE $\leftarrow k$
    $\overline{c}_k \leftarrow$ `FocusedVIrec(`s_0`)`
  until $\overline{c}_k \leq \epsilon$ *// test for ϵ-consistency policy*

Function `FocusedVIrec(`s`)`
  If *s is a goal state* **then**
    return 0
  $\mu^k(s) \leftarrow \underset{a \in \mathcal{A}(s)}{arg\min} \left[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s') \right]$ *// policy update*
  $J_k(s) \leftarrow \underset{a \in \mathcal{A}(s)}{\min} \left[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s') \right]$ *// update cost-to-go function*
  $residual \leftarrow J_k(s) - J_{k-1}(s)$
  foreach $s' \in Succ(s, \mu^k(s))$ **do**
    If s'.UPDATE $< k$ **then** *// j not already updated this iteration*
      s'.UPDATE $\leftarrow k$
      $r \leftarrow$ `FocusedVIrec(`s'`)` *// best residual of decedent*
      If $(r > residual)$ **then**
        $residual \leftarrow r$
  $J_k(s) \leftarrow \underset{a \in \mathcal{A}(s)}{\min} \left[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s') \right]$ *// postorder backup*
  return $residual$

improve the cost-to-go function more than the pre-order backup because it is performed after the successors of the state have been backed up, and updated values are propagated backwards from the goal. But the post-order backup only updates the cost-to-go function, it does not change the current best policy. The greedy policy is selected when states are first visited by the depth-first traversal to ensure that the set of states $\mathcal{S}^{\mu^k}(s_0)$ is exactly the set of states visited by following the greedy policy μ^k starting from s_0. The post-order backup is not used to compute the Bellman residual over the set $\mathcal{S}^{\mu^k}(s_0)$, denoted by $\overline{c}_k$, either. The residual is computed based on the pre-order backup. This is necessary to ensure that the residual is computed for all states visited by the policy μ^k.

FVI adopts the same search strategy as LAO*, with the following difference: LAO* gradually expands an *open* policy over a succession of iterations until it is *closed*, whereas FVI evaluates the best closed policy each iteration. A policy is said to be closed if it specifies an action for every state that is reachable from the start state under the policy; otherwise, it is said to be open.

The depth-first traversal approach used by FVI was first used by the LAO* algorithm, and has been adopted by several other heuristic search algorithms, including Labeled RTDP (LRTDP) [13], Heuristic Dynamic Programming (HDP) [11], and Learning Depth-First Search (LDFS) [15]. However, these algorithms do not perform complete depth-first traversals, they "bail out " as soon as they visit a state for which the residual exceeds a given threshold ϵ. These algorithms are considered, in detail, in Chapter 5.

4.2 Suboptimality Bounds for SSP Problems

In this section we review the convergence issues of algorithms for SSP problems. We also review different ways for detecting convergence found in the literature. We finally describe new easy-to-compute bounds that can be used to bound the suboptimality of solutions found by those algorithms. For an extensive discussion on the new bounds, we refer to [26]. The contribution of this chapter is the experimental evaluation of the performance of the bounds and their integration in heuristic search algorithms for SSP problems. We use the FVI algorithm to evaluate the performance of the new suboptimality bounds. The results of this chapter have been published in [27, 28].

Value iteration and related heuristic search algorithms converge only in the *limit*, and thus may require an infinite number of iterations to converge to the optimal solution. In practice, it is important to detect convergence to an ϵ-optimal solution after a finite number of iterations. However, these algorithms have lacked an efficient way of detecting convergence to an ϵ-optimal solution for SSP problems because they have lacked easy-to-compute suboptimality bounds. Three approaches to testing for convergence have been proposed.

The first consists in using bounds originally derived for value iteration to test for convergence; these bounds are due to Bertsekas [7]. However, these bounds are very expensive to compute because they require solving a system of linear equations of size equal to the number of reachable states. As a consequence, this test for convergence has not been used by heuristic search algorithms, or by value iteration.

The second approach is to compute both lower and upper bounds, and test whether the gap between them is less than the threshold $\epsilon > 0$. This approach has been implemented in variations of RTDP [45, 52, 54] and LAO* [57], and it has the advantage of providing meaningful suboptimality bounds. In addition, if the initial upper-bound is computed by evaluating a proper policy, it gives a monotone upper bound. A greedy policy with respect to a monotone upper-bound function is guaranteed to be proper. However the drawback of this approach is that it computes both lower and upper-bound functions separately, which is equivalent to solving the problem twice. It must also compute an initial monotone upper-bound function, which is more difficult than computing an initial lower-bound function, because it requires an expensive policy evaluation step that is performed for the entire state space.

The third approach, used most often, is to test for convergence to an ϵ-*consistent* cost-to-go function, that is, to test whether the Bellman residual is less than $\epsilon > 0$. This test for convergence is computationally much more efficient than the first two approaches. However, it has the drawback that it does not provide a genuine suboptimality bound. It does not even guarantee that a policy found by this approach will reach the goal state with probability one, that is, it does not even guarantee that a proper policy has been found.

4.2.1 Bertsekas Bounds

For SSP problems, Bertsekas [6] derives suboptimality bounds for value iteration that take the following form when the cost-to-go function J_k is a lower-bound function. For every state $s \in \mathcal{S}$,

$$J_k(s) \leq J^*(s) \leq J_{\mu^k}(s) \leq J_k(s) + (N_{\mu^k}(s) - 1) \cdot \overline{c}_k, \tag{4.1}$$

where μ^k is a greedy policy with respect to J_{k-1}, $N_{\mu^k}(s)$ is the expected number of steps to reach a goal state starting from state s and following the greedy policy μ^k, and

$$\overline{c}_k = \max_{s \in \mathcal{S}} \left\{ J_k(s) - J_{k-1}(s) \right\}, \tag{4.2}$$

is the Bellman residual.

Unfortunately, the overhead for computing this upper bound is prohibitive because computing $N_{\mu^k}(s)$ requires solving the following system of $|\mathcal{S}|$ linear equations in $|\mathcal{S}|$ unknowns,

$$N_{\mu^k}(s) \, = \, 1 + \sum_{s' \in \mathcal{S}} p_{s,s'}(\mu^k(s)) N_{\mu^k}(s'). \; s \in S, \tag{4.3}$$

which takes $\mathcal{O}(|\mathcal{S}|^3)$ time to evaluate, each time the policy μ^k changes.

4.2.2 Positive-cost Bounds

Because they are computationally expensive, the Bertsekas bounds are not used in practice. Hansen and Abdoulahi [27] describe practical suboptimality bounds for SSP problems with positive action costs. These bounds are bounds on the Bertsekas bounds, but have the advantage that they can be computed efficiently.

Theorem 4.2.1. *For an SSP problem where all actions taken in a non-goal state have positive cost, and* $\underline{g} = \min_{s \in \mathcal{S}, a \in \mathcal{A}(s)} c(s,a)$ *denotes the smallest action cost, if* $\overline{c}_k < \underline{g}$ *then:*

1. *a greedy policy* μ^k *with respect to* J_{k-1} *is proper, and*
2. *for each state* $s \in \mathcal{S}$*, we have the following upper bound, where* $J_{\mu^k}(s) \leq \overline{J}_{\mu^k}(s)$*:*

$$\overline{J}_{\mu^k}(s) = \frac{(J_k(s) - \overline{c}_k) \cdot \underline{g}}{(\underline{g} - \overline{c}_k)}. \tag{4.4}$$

A formal proof is given by [27]. By plugging the new upper bound in Equation (4.1), we get the following equation:

$$J_k(s) \leq J^*(s) \leq J_{\mu^k}(s) \leq \frac{(J_k(s) - \overline{c}_k) \cdot \underline{g}}{(\underline{g} - \overline{c}_k)}. \tag{4.5}$$

The derivation of Equation (4.5) is based on the insight that when all action costs are positive, with minimum cost $\underline{g} > 0$, an upper bound $\overline{N}_{\mu^k}(s)$ on $N_{\mu^k}(s)$ is related to an upper bound $\overline{J}_{\mu^k}(s)$ on $J_{\mu^k}(s)$ by the formula:

$$\overline{N}_{\mu^k}(s) = \frac{\overline{J}_{\mu^k}(s)}{\underline{g}}. \tag{4.6}$$

This formula simply states that an upper bound $\overline{N}_{\mu^k}(s)$ on the expected number of steps until termination when following a policy μ^k starting from state s is given by an upper bound $\overline{J}_{\mu^k}(s)$ on the cost-to-go $J_{\mu^k}(s)$ divided by the smallest action cost $\underline{g}$. Given this formula, the bound of Equation (4.5) is derived by substituting $\overline{N}_{\mu^k}(s)$ for $N_{\mu^k}(s)$ in the bound of Equation (4.1) to obtain the upper bound,

$$\overline{J}_{\mu^k}(s) = J_k(s) + (\overline{N}_{\mu^k}(s) - 1) \cdot \overline{c}_k, \tag{4.7}$$

and then substituting $\dfrac{\overline{J}_{\mu^k}(s)}{\underline{g}}$ for $\overline{N}_{\mu^k}(s)$ based on Equation (4.6), and solving for $\overline{J}_{\mu^k}(s)$, which gives

$$\overline{J}_{\mu^k}(s) = \frac{(J_k(s) - \overline{c}_k) \cdot \underline{g}}{(\underline{g} - \overline{c}_k)}. \tag{4.8}$$

The upper bound given by Equation (4.8) is easy to compute because it depends only on the cost-to-go function J_k and the residual $\overline{c}_k$, which are always available for both value iteration and related heuristic search algorithms, without requiring any overhead to compute them. In fact, the new bounds are just as easy to compute as the classic bounds for discounted infinite-horizon MDPs, used in Chapter 3. Even though these bounds are bounds on the Bertsekas bounds, they quickly become as tight as the expensive Bertsekas bounds, as we will see in the experimental evaluation below.

4.2.3 Generalized Bounds

The positive cost bounds perform well in practice and become quickly as tight as the expensive Bertsekas as experimentally shown in [27] and below. But the approach has two limitations. First, it is only applicable if all action costs are positive[1]. Second, the bounds are tightest if action costs are uniform, or nearly uniform. For problems with non-uniform action costs, the quality of the suboptimality bounds decreases in proportion to the difference between the smallest action cost and the average action cost.

Hansen and Abdoulahi [28] show how to compute suboptimality bounds that are equally good whether action costs are uniform or not, and are available regardless of the cost structure of the problem; in particular, the bounds do not depend on all action costs being positive. The insight is to compute the quantity $\overline{N}_{\mu^k}$ independently of J_k, and thus in a way that does not depend on the cost structure of the problem.

Consider a *steps-to-go function* $N_k(s)$ that estimates the number of steps required to reach a goal state from state s. In addition to updating the cost-to-go function $J_k(s)$ for state s, the algorithm performs the following update,

$$N_k(s) = 1 + \sum_{s' \in \mathcal{S}} p_{s,s'}(\mu^k(s)) N_{k-1}(s'), \tag{4.9}$$

and also computes the following residual after each iteration

$$\overline{n}_k = \max_{s \in \mathcal{S}} \left(N_k(s) - N_{k-1}(s) \right). \tag{4.10}$$

[1] Similar bounds that can be used when all costs are negative have been derived in [26].

When all action costs are equal to 1, it is easy to see that $J_k(s) = N_k(s)$, for $s = 1, \ldots, n$, and $\overline{c}_k = \overline{n}_k$. In that case, there is no reason to compute these additional values. But when action costs are not uniform, or when they are not all positive, the additional values $N_k(s)$ and $\overline{n}_k$ can differ greatly from the values $J_k(s)$ and $\overline{c}_k$, and they provide a way to compute bounds of the same quality as those available when action costs are uniform and positive.

The following theorem assumes that the algorithm also computes a steps-to-go function.

Theorem 4.2.2. *For any SSP problem, consider a lower-bound function J_k that is updated by value iteration, where $\overline{c}_k$ is the residual defined by Equation* (4.2)*. Consider also a steps-to-go function N_k that is updated each iteration, where $\overline{n}_k$ is the residual defined by Equation* (4.10)*. If $\overline{n}_k < 1$ then:*

1. *a greedy policy μ^k with respect to J_{k-1} is proper, and*
2. *for each state $s \in \mathcal{S}$, we have the following upper bound on $J_{\mu^k}(s)$, where $J_{\mu^k}(s) \leq \overline{J}_{\mu^k}(s)$:*

   (a) If $0 \leq \overline{n}_k < 1$, then

   $$\overline{J}_{\mu^k}(s) = J_k(s) + \left(\frac{N_k(s) - \overline{n}_k}{1 - \overline{n}_k} - 1 \right) \cdot \overline{c}_k. \tag{4.11}$$

   (b) If $\overline{n}_k \leq 0$, then

   $$\overline{J}_{\mu^k}(s) = J_k(s) + (N_k(s) - 1) \cdot \overline{c}_k. \tag{4.12}$$

Computing the steps-to-go function N_{μ^k} for a policy μ^k can be viewed as a positive-cost SSP problem where the smallest action cost is 1, and thus the greedy policy μ^k must be proper when $\overline{n}_k < 1$, by Theorem 4.2.1.

Applying the bounds of Equation (4.1) to the problem of computing N_{μ^k}, we have:

$$N_{\mu^k}(s) \leq N_k(s) + (N_{\mu^k}(s) - 1) \cdot \overline{n}_k. \tag{4.13}$$

If the policy μ^k is proper, there must be an upper bound $\overline{N}_{\mu^k}(s)$, with $N_{\mu^k}(s) \leq \overline{N}_{\mu^k}(s)$, that is the solution of the linear equation:

$$\overline{N}_{\mu^k}(s) = N_k(s) + (\overline{N}_{\mu^k}(s) - 1) \cdot \overline{n}_k. \tag{4.14}$$

Solving for $\overline{N}_{\mu^k}(s)$, we get

$$\overline{N}_{\mu^k}(s) = \frac{N_k(s) - \overline{n}_k}{1 - \overline{n}_k}. \tag{4.15}$$

Substituting the value of $\overline{N}_{\mu^k}(s)$ from (4.15) into (4.7), we get the bound of Equation (4.11).

For an SSP problem with positive action costs that are not uniform, the upper bounds of Theorem 4.2.2 are tighter than the upper bounds of Theorem 4.2.1. Moreover, Theorem 4.2.2 can be used to compute upper bounds for any SSP problem, even if action costs are zero or negative. It only requires the slight extra overhead of updating the steps-to-go function in each iteration.

The bounds we have described so far apply to value iteration because the Bellman residual on which they depend is computed over the entire state space. Here we show why they also apply to heuristic search algorithms that only compute a Bellman residual for reachable states under the current policy.

Corollary 4.2.2.1. *The suboptimality bounds of Theorems 4.2.1 and 4.2.2 apply to heuristic search algorithms that compute a Bellman residual on the states visited by the current policy, without necessarily considering the entire state space.*

Proof. In each iteration $k \in \mathbb{N}$, heuristic search algorithms compute the Bellman residual on the set $\mathcal{S}^{\mu^k}(s_0)$ of states visited following the greedy policy μ^k. Every state $s \in \mathcal{S}^{\mu^k}(s_0)$ has a single action $a = \mu^k(s)$, and the successor states of s, following μ^k, are also in the set $\mathcal{S}^{\mu^k}(s_0)$ The probability distribution over the successor states following policy μ^k is the same as in the original MDP. The set $\mathcal{S}^{\mu^k}(s_0)$ is sub-MDP over which the Bellman residual is computed. Therefore, the bounds are valid for the sub-MDP given by $\mathcal{S}^{\mu^k}(s_0)$. □

4.3 Integration of Suboptimality Bounds in Focused Value Iteration

This section shows how to integrate the suboptimality bounds of Section 4.2 in FVI.

The original FVI algorithm described in Section 4.1 computes an ϵ-consistent policy. This means that it does not have meaningful bounds that can be used to detect an ϵ-optimal policy. In this section we describe how to integrate the new easy-to-compute bounds in FVI, not only to compute an ϵ-optimal policy, but to detect when a policy is proper as well.

Algorithm 9 shows how the bounds are integrated in FVI. Let $\underline{g}$ be the smallest action cost over the set of states $\mathcal{S}^{\mu^k}(s_0)$. The suboptimality bounds are used in FVI as follows. For each iteration k, the Bellman residual $\overline{c}_k$ is computed. If $\overline{c}_k$ is smaller than $\underline{g}$, the greedy policy is proper (line 7 of the pseudocode). If the greedy policy is proper, then an upper bound on the cost-to-go function is computed (line 8 of the pseudocode) using the Equation (4.4). The algorithm has converged to an ϵ-optimal policy when the differ-

Algorithm 9: Focused Value Iteration algorithm with suboptimality bounds

Input: SSP problem with state space $\mathcal{S}$, start state s_0
Output: ϵ-optimal policy for start state

Algorithm `FocusedVI(`s_0`)`
  Set $k \leftarrow 0$ *// global variable k is iteration count.*
  Repeat
    Set $k \leftarrow k + 1$
    s_0.UPDATE $\leftarrow k$
    $\overline{c}_k \leftarrow$ `FocusedVIrec(`s_0`)`
    If $\overline{c}_k < \underline{g}$ **then** *// test for proper policy*
      $\overline{J}_k(s_0) = (J_k(s_0) - \overline{c}_k) \cdot \underline{g}/(\underline{g} - \overline{c}_k)$ *// derived upper bound*
    else
      $\overline{J}_k(s_0) \leftarrow \infty$ *// trivial upper bound*
  until $(\overline{J}_k(s_0) - J_k(s_0)) \leq \epsilon$ *// test for ϵ-optimality policy*

Function `FocusedVIrec(`s`)`
  If s *is a goal state* **then**
    return 0
  $\mu^k(s) \leftarrow arg\min_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s')\right]$ *// policy update*
  $J_k(s) \leftarrow \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s')\right]$ *// update cost-to-go function*
  $residual \leftarrow J_k(s) - J_{k-1}(s)$
  foreach $s' \in Succ(s, \mu^k(s))$ **do**
    If s'.UPDATE $< k$ **then** *// j not already updated this iteration*
      s'.UPDATE $\leftarrow k$
      $r \leftarrow$ `FocusedVIrec(`s'`)` *// best residual of decedent*
      If $(r > residual)$ **then**
        $residual \leftarrow r$
  $J_k(s) \leftarrow \min_{a \in \mathcal{A}(s)} \left[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s')\right]$ *// postorder backup*
  return $residual$

ence between upper and lower bounds is smaller than the threshold $\epsilon > 0$ (line 11 of the pseudocode).

Test for Action Elimination

In Chapter 3, we showed how to use suboptimality bounds for discounted MDPs in an action elimination test. The new bounds introduced in this chapter can be used in an action elimination test for SSP problems as follows. Let

$$Q_k^a(s) = c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J_{k-1}(s') \tag{4.16}$$

be the lower bound on the cost-to-go function of taking action a in state s. Let $\overline{J}_{\mu^k}(s)$ be the upper bound on the cost-to-go cost of state s, given by Equation (4.4). An action a is suboptimal in state s, and therefore can be eliminated, if

$$Q_k^a(s) > \overline{J}_{\mu^k}(s). \tag{4.17}$$

4.4 Experimental Evaluation of the Bounds

In this section, we empirically evaluate the performance of the new suboptimality bounds when they are used in the FVI algorithm. FVI computes a Bellman residual each iteration. The experiments show how the bounds for the start state s_0 improve over successive iterations of the algorithm. The threshold for termination with an ϵ-optimal policy is $\epsilon = 10^{-6}$.

Figure 4.1 shows the performance of the positive-cost bounds for two classic SSP test problems. Both problems have unit action costs. The racetrack problem with 21,371 states and 9 actions per state, has an optimal policy for the start state that visits only 2,262 states. An ϵ-optimal policy is found at iteration 42. The mountain car problem with 10,000 states and two actions per state has an optimal policy for the start state that visits 6,643 states. An ϵ-optimal policy is found at iteration 55. A detailed description of these problems is given in the appendix.

Three different bounds are shown in Figure 4.1. The top line is the positive-cost upper bound $\overline{J}_{\mu^k}(s_0)$, computed by our new approach. The middle line is the upper bound $J_{\mu^k}(s_0)$ computed by evaluating the current greedy policy μ^k. This bound is not practical because it requires solving a system of linear equations of size equal to the number of states visited by the current policy. It is only included for the purpose of comparison. It shows that although $\overline{J}_{\mu^k}(s_0)$ is only guaranteed to be an upper bound on $J_{\mu^k}(s_0)$, it quickly tightens to be almost equal to $J_{\mu^k}(s_0)$. The bottom line is the lower bound $J_k(s_0)$ that corresponds to the cost-to-go function computed by value iteration.

We can observe that the upper bounds do not converge monotonically. This is due to the fact that the Bellman residual is only computed for states visited by the current greedy policy for a heuristic search algorithm. When the greedy policy changes, the Bellman residual is not guaranteed to decrease monotonically.

The Bellman residual is less likely to decrease monotonically when the size of the set of states in the greedy policy is small and changes often during the search. The more states the algorithm visits each iteration, the more regular the residual is likely to be. For

example, Figure 4.2 shows the bounds for instances of Double pendulum and Elevator problems. Both problems have unit action costs. The double pendulum with $160,000$ states and 2 actions per state, has an optimal policy that visits $146,153$ states, which is most of the state space. The elevator problem with $14,976$ and 5 actions per state, has an optimal policy that visits only 12 states. For this problem instance, the greedy policy is small and changes often; it is not even proper each iteration. The graph of the elevator problem (right) is discontinued for some iterations because the greedy policy is not proper for those iterations, thus the bounds are not available. The bounds for double pendulum are smoother than the ones for elevator because most of the state space is visited for the double pendulum each iteration.

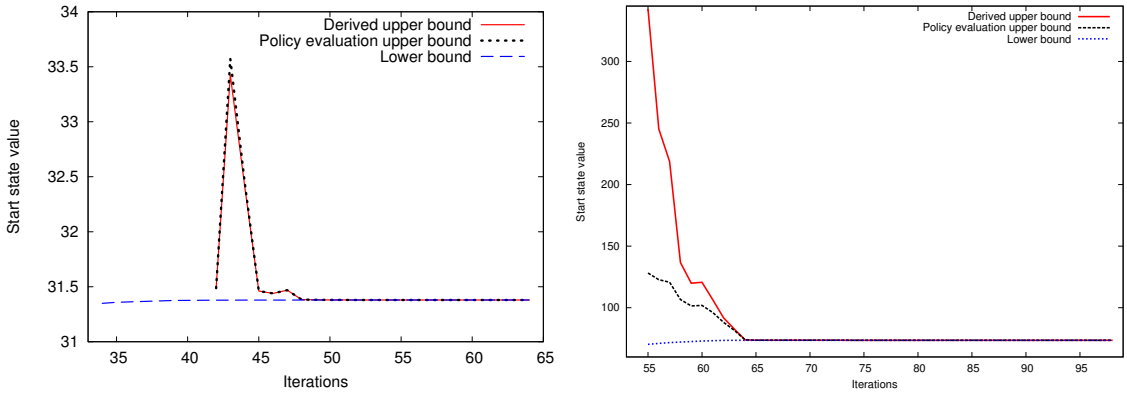

Figure 4.1: Quality of bounds in FVI: Racetrack (left) and Mountain car (right)

We also tested the performance of the generalized bounds on two test problems with non-uniform action costs. The test problems are taken from the ICAPS Planning Competitions benchmark. For these problems, action costs are positive but not uniform. The

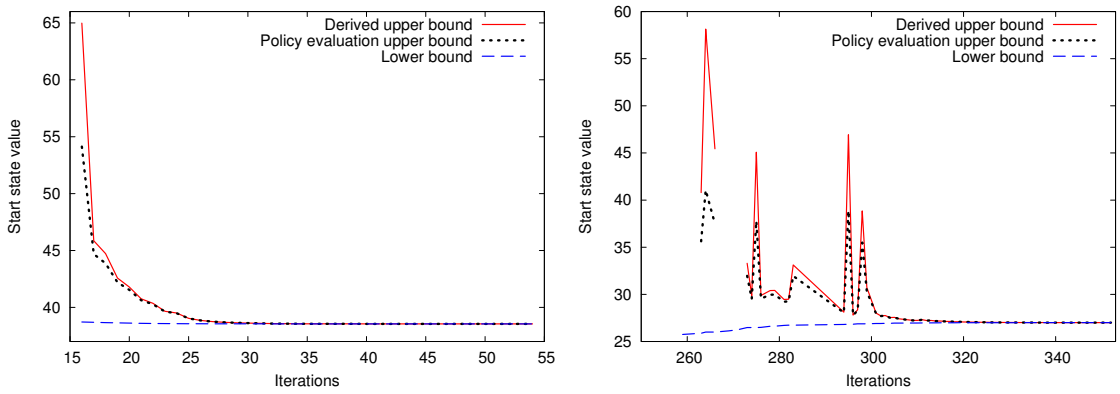

Figure 4.2: Quality of bounds in FVI: Double pendulum (left) and Elevator (right)

Tireworld problem has one action with a cost of 100, while the other actions have unit cost. The Zeno Travel problem has action costs of 1, 10, and 25.

Figure 4.3 shows the bounds for the start state s_0, computed using FVI. The top line is the positive-cost bound, the middle and bottom lines are the new general bound and the lower bound, respectively. The new general bounds are tighter than the positive-cost bounds, on problems with non-unit action costs. For these experiments, the overhead for computing the steps-to-go function for the new bounds is less than 1% of the overall running time of FVI.

However, the positive-cost bounds of Theorem 4.2.1 still perform very well for these test problems, even though action costs are not uniform. It takes only a few more iterations for the positive-cost bounds to reach the same point as the new bounds. The two bounds are identical for problems with unit action costs.

The most important advantage of the bounds of Theorem 4.2.2 is that they do not require all action costs to be positive, which means they apply to a broad range of SSP problems for which the positive-cost bounds of Theorem 4.2.1 cannot be used.

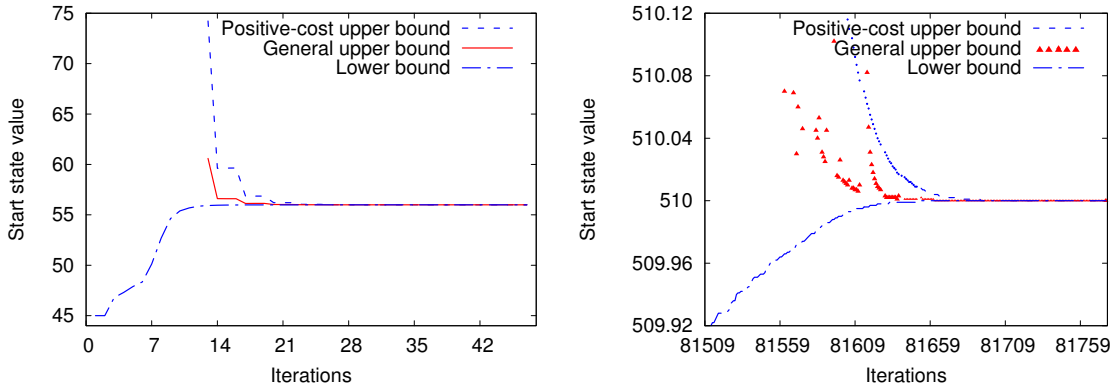

Figure 4.3: General bounds for Tireworld problem (left), and Zeno Travel problem (right)

4.5 Experimental Comparison to Branch-and-Bound Value Iteration

The branch-and-bound value iteration (BBVI) algorithm of Chapter 3 has the property that it has no more iterations than regular VI. FVI on the other hand, has no bound on the number of iterations before termination. The number of iterations of FVI depends on the start state. We compare BBVI to FVI on four different planning problems. The objective of the comparison is to evaluate the tradeoff between fewer but expensive iterations of BBVI and faster but potentially more iterations of FVI. The preprocessing step of BBVI is used to find a good order of backups. The depth-first traversal in FVI naturally provides a good ordering of backups. Table 4.1 shows the results of the comparison. FVI has a clear advantage over BBVI, which is expected. The performance of the two algorithms

Problem	Problem Characteristics			Bounds Performance (BBVI)		Runtime in CPU seconds	
	$\|S\|$	actions/state	$\|policy\|$	pruned states	pruned actions	BBVI	FVI
bigger	51,943	9	**9,037**	39,291	428,061	33.81	6.53
blocksworld	103,121	6	**30**	46,127	602259	137.49	13.61
dap20	160,000	2	**146,158**	13,842	173,841	89.176	70.53
mcar500	250,000	2	**98,263**	162,090	410,581	282.22	91.47

Table 4.1: Comparison between BBVI and FVI . Running times in CPU seconds until ϵ-optimality with $\epsilon = 10^{-6}$

.

is closest when most of the state space is visited each iteration. BBVI would have an advantage for "dense" MDPs, that is, problems with a lot of actions per state, and when the greedy policy visits most of the state space. FVI on the other hand, would work best for "sparse" MDPs, where only a fraction of the state space is relevant to computing the solution. The performance of FVI also depends on the quality of the heuristic used to guide the search.

4.6 Summary

In this chapter we reviewed and experimentally evaluated efficient suboptimality bounds for SSP problems. We showed how to integrate the bounds in Focused Value Iteration, an efficient heuristic search algorithm for SSP problems. The bounds can be used in any heuristic search algorithm that computes a Bellman residual, in a test for convergence to an ϵ-optimal policy, as well as in testing if the greedy policy is proper. The bounds can, for example, be used in the LAO* algorithm. They cannot be used, however, in RTDP because it does not even have a way to test for convergence to an ϵ-consistent policy. Using the new bounds in test for convergence to an ϵ-optimal policy, is as efficient as testing for convergence to an ϵ-consistent policy. That is, they incur no extra overhead. We also

compared Focused Value Iteration to the branch-and-bound value iteration algorithm introduced in Chapter 3. Focused Value Iteration is compared to other widely-used heuristic search algorithms in the next chapter.

CHAPTER 5

IMPROVED SOLVED-LABELING IN HEURISTIC SEARCH

In Chapter 4, we described Focused Value Iteration (FVI), an efficient heuristic search algorithm for SSP problems that finds a policy of any desired degree of approximation. In this chapter we consider widely-used heuristic search algorithms such as HDP and LDFS. These algorithms have two features in common: (a) they all adopt the Find-and-Revise framework, and (b) they use solved-labeling to speed up convergence. We describe and analyze these algorithms to understand the factors that influence their performance. We identify the strengths and weaknesses of the approaches they adopt, and address those weaknesses. The resulting algorithm is FVI enhanced with solved-labeling. The chapter is organized as follows. Section 5.1 reviews state space decompostion, an often used approach for speeding up value iteration and related heuristic search algorithms. Section 5.2 describes and analyzes heuristic search algorithms for SSP problems that adopt the approaches of solved-labeling and Find-and-Revise. In Section 5.3, we compare these algorithms to FVI, and identify their strengths and weaknesses. Section 5.4 shows how to address the weaknesses identified. Section 5.5 describes an improved solved-labeling heuristic search algorithm that is based on FVI. Section 5.6 presents an experimental evaluation of the new algorithm.

5.1 State Space Decomposition

Several variations of VI and related heuristic algorithms have used the graphical structure of the SSP problem to speed up convergence. We first review Tarjan's algorithm because it is used to decompose the state space into strongly connected components (SCCs).

Tarjan's algorithm [56] is a depth-first based algorithm that detects SCCs in a directed graph. Starting from a given node s, Tarjan's algorithm performs a depth-first traversal of nodes reachable from s. When visited for the first time, a node is marked with variables IDX and LOW, where IDX denotes the order in which it is visited, and LOW the minimum visit order of its descendent nodes. Once a node is visited, the depth-first is recursively called on its successor nodes. A node s is a root of an SCC if s.IDX and s.LOW are equal. The depth-first traversal uses a stack to keep track of the nodes visited. Figure 5.1 shows the SCC decomposition of a policy graph. The SCCs are marked by dashed rectangles.

Tarjan's algorithm has linear complexity in the size of the graph. Because it is extensively used in state decomposition, several improvements have been proposed to mitigate the overhead of using a stack. Algorithm 10 is an improved version of the Tarjan's original algorithm. In this version, the root node of each SCC is never pushed into the depth-first stack in order to save time and space [39].

State space decomposition has been used in several algorithms for MDPs, to speedup convergence. Topological Value Iteration (TVI) [20] improves on value iteration by first decomposing the state space into SCCs. The SCCs are then sorted in topological order. Value iteration is performed on each SCC, in reverse order starting from the goal SCC. Once an SCC has converged, its states are not updated in subsequent iterations. Bonet and

Algorithm 10: Tarjan's algorithm

Input: Graph
Output: SCC decomposition of the graph

Algorithm `Tarjan(`s`)`
  // visit node s
  s.IDX $\leftarrow$ $index$
  s.LOW $\leftarrow$ $index$
  $index \leftarrow index + 1$
  // visit all successor nodes of s
  for $s' \in Succ(s)$ **do**
    // node s' not visited yet
    if s'.IDX is undefined **then**
      `Tarjan(`s'`)`
      s.LOW $\leftarrow \min(s.\text{LOW}, s'.\text{LOW})$
    else if $s' \in stack$ **then**
      s.LOW $\leftarrow \min(s.\text{LOW}, s'.\text{IDX})$
  // check if node s is root of an SCC
  if (s.LOW == s.IDX) **then**
    while $stack \neq \emptyset \wedge TOP(stack).\text{IDX} \geq s.\text{IDX}$ **do**
      POP($stack$) and report
  else
    PUSH($stack, s$)

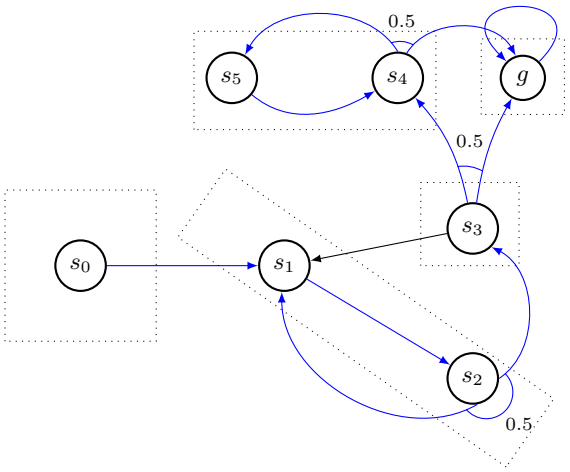

Figure 5.1: Policy graph decomposition into SCCs using Tarjan's algorithm.

Geffner developed several heuristic search algorithms that use the state space decomposition to speed up convergence [11, 12, 15].

5.2 Review of Algorithms

We begin by a description of the solved-labeling procedure and the Find-and-Revise framework.

5.2.1 Solved-Labeling

The idea of labeling states as solved in order to accelerate convergence was originally implemented in the AO* algorithm for AND/OR graphs [46]. The technique was generalized for SSP problems by Bonet and Geffner, and is included in the following heuristic search algorithms: LRTDP [12], HDP [11], and LDFS [14]. A state s is labeled "solved" if:

- s is a goal state, and
- every state reachable from s following a greedy policy is labeled "solved".

In an acyclic graph, a state is labeled "solved" if the best action is found for the state, and if every successor state of the state is labeled "solved". In a cyclic graph, it is a much more complicated condition to check, and requires the decomposition of the greedy policy graph into strongly connected components. The state in an SCC can be labeled as "solved" if all successor states of the SCC are labeled "solved", and if an ϵ-consistent policy is found for the states in the SCC.

Solved-labeling speeds up convergence because solved states do not need to be updated in subsequent iterations of the algorithm. RTDP, for example, lacks a stopping criterion

and as it runs longer, it tends to revisit states that have already converged, wasting resources. Labeled RTDP uses solved-labeling to speed up RTDP by labeling states that have converged as "solved". Labeled RTDP uses trials like RTDP, however. In Labeled RTDP, a trial is terminated when a "solved" state is reached. At the end of a trial, Labeled RTDP performs a depth-first traversal following the greedy action from every state in the trial. All states in the trial, for which the Bellman residual is smaller then ϵ, are labeled "solved". Labeled RTDP has converged when the start state has been labeled "solved".

The rest of the chapter focuses on the HDP algorithm because it generalizes solved-labeling in an elegant and more principled way than Labeled RTDP. HDP uses Tarjan's algorithm to decompose the greedy policy graph into strongly connected components. Strongly connected components are then labeled "solved" backwards from the goal, whenever they have converged.

HDP is similar to FVI in the sense that, each iteration k, both algorithms perform a systematic depth-first traversal of states reachable from the start state, following the greedy policy μ^k. But HDP uses Tarjan's algorithm, an extension of depth-first search that decomposes the greedy policy graph into strongly connected components. A strongly connected component is labeled "solved" when all its states are ϵ-consistent. Algorithm 11 gives the pseudocode of HDP.

Because solved-labeling relies on the decomposition of the greedy graph into strongly connected components, its performance is problem-dependent. For example, in a problem with a single strongly connected component, the start state is reachable from every other state, and it is not possible to label any state as "solved" before the start state is labeled

Algorithm 11: HDP [11]

Input: SSP problem with start state s_0
Output: ϵ-consistent policy for start state

```
1  Algorithm HDP(s_0)
2      while s_0 is not labeled as SOLVED do
3          index ← 0
4          DFS(s_0)
5          reset IDX to ∞ for visited states
6          clean stack and visited

7      Function DFS(s)
8          if s.SOLVED ∨ s.GOAL then
9              s.SOLVED ← true

10         if s.RESIDUAL > ε then
11             BACKUP(s)
12             return true // "bail" out to end dfs

13         visited.PUSH(s)
14         stack.PUSH(s)
15         s.IDX ← s.LOW ← index
16         index ← index + 1

17         flag ← false
18         for s' ∈ Succ(s, μ^k(s)) do
19             if (s'.IDX == ∞) then
20                 flag ← flag ∨ DFS(s')
21                 s.LOW ← min{s.LOW, s'.LOW}
22             else if s' ∈ stack then
23                 // state already visited
24                 s.LOW ← min{s.LOW, s'.IDX}

25         if flag then
26             BACKUP(s)
27             return true
28         else if (s.IDX == s.LOW) then
29             Remove states of SCC and label them as "SOLVED"
30         return flag
```

"solved". Furthermore, solved-labeling for SSP problems is potentially more expensive than solved-labeling for acyclic AND/OR graphs because a policy for an SSP problem can have cycles whereas a policy is always acyclic for an acyclic AND/OR graph.

5.2.2 Find-and-Revise

Find-and-Revise is a widely-used framework in heuristic search algorithms for SSP problems, including HDP. It works as follows. Let $G_{s_0}^J$ be the graph induced by the greedy policy, with respect to the cost-to-go function J, rooted at the start state s_0. In each iteration of the algorithm, the *Find* step searches the graph $G_{s_0}^J$, for an ϵ-inconsistent state, that is, a state for which the Bellman residual is larger than ϵ. The *Revise* step performs a Bellman backup to improve the value of the state. The Bellman backup has the potential of changing the current greedy graph. The two steps are repeated until the graph $G_{s_0}^J$ has no ϵ-inconsistent state, in which case, an ϵ-consistent policy has been found. Algorithm 12 gives the pseudocode of the Find-and-Revise framework.

Algorithm 12: Find-and-Revise framework

Input: SSP problem with start state s_0
Output: ϵ-consistent policy for start state
while *the greedy graph* $G_{s_0}^J$ *contains an* ϵ*-inconsistent state* **do**
    Find a state $s \in G_{s_0}^J$ for which the Bellman residual $\overline{c}_J(s) > \epsilon$
    Revise $J(s)$ by performing a Bellman backup
        $J(s) \leftarrow \min_{a \in \mathcal{A}(s)} \left\{ c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J(s') \right\}$
return *greedy policy with respect to J*

5.3 Limitations and Analysis

In this section, we compare Focused Value Iteration to heuristic algorithms that use both Find-and-Revise and Solved-labeling.

5.3.1 Comparison of Algorithms

In order to understand the benefits and drawbacks of the solved-labeling and Find-and-Revise approaches, we compare FVI to the above-mentioned heuristic search algorithms using the implementation and test problems by Bonet and Geffner [15]. A description of the test problems can be found in the appendix. With the exception of LAO*, all the other algorithms use the *Find-and-Revise* framework, where a depth-first traversal of the reachable states is terminated as soon as a state for which the residual is larger than ϵ is encountered. FVI and LAO* perform full depth-first traversals of the reachable states, each iteration. The algorithms considered include HDP, LRTDP, LDFS, and LDFS+. Table 5.1 gives the results of the comparison. These results are consistent with the ones reported in [15]. They show that FVI performs as well or better than the other algorithms.

Problem	Characteristics		Runtime in CPU seconds						
	$\|S\|$	|policy|	VI	LRTDP	HDP	LDFS	LDFS+	LAO*	FVI
big	22,534	4,321	1.31	1.44	0.69	0.51	0.21	0.26	0.30
bigger	51,943	9,037	4.33	3.13	2.40	1.93	0.67	1.16	0.63
square-3	42,085	790	1.76	0.06	0.03	0.02	0.04	0.06	0.07
square-4	383,970	1,000	46.57	0.08	0.05	0.04	1.76	1.09	1.37
ring-5	94,396	12,374	5.47	4.37	2.22	1.90	0.70	1.38	1.53
ring-6	352,135	37,437	35.64	48.86	16.75	16.16	4.39	6.01	6.35
wet-160	25,600	1,364	1.85	7.13	70.29	50.72	4.60	0.06	0.06
wet-200	40,000	749	2.15	3.61	24.62	17.18	1.93	0.03	0.03
nav-18	262,143	2,494	90.32	55.83	2421.97	3034.58	2.07	1.67	1.68
nav-20	1,048,575	1,861	407.65	85.28	1946.06	1892.55	3.24	2.06	2.26

Table 5.1: Algorithm running times in CPU seconds until ϵ-consistency with $\epsilon = 10^{-8}$.

5.3.2 Results of Analysis

We identify two limitations. The first limitation is the Find-and-Revise framework. Find-and-Revise prevents the algorithms performing as many post-order backups, which are potentially more beneficial than pre-order backups because they are performed after the descendants of a state have been updated. Using Find-and-Revise also prevents the algorithms from performing a complete depth-first traversal until the last iteration. Therefore, they do not compute a residual until they have converged. It follows that these algorithms cannot use the suboptimality bounds to monitor the progress of the search and dynamically decide when to terminate. Because they cannot use these bounds, Find-and-Revise based algorithms terminate when the cost-to-go value of the start state is ϵ-consistent.

The second limitation is the overhead for using a stack in Tarjan's algorithm. Algorithms that use solved-labeling keep track of the decomposition of the greedy policy graph into strongly connected components, using an explicit stack. It turns out that it is possible to get rid of the stack in Tarjan's algorithm. This can potentially speed up the algorithms, if many states are pushed and removed from the stack each iteration.

It is worth noting that if both solved-labeling and Find-and-Revise are removed from HDP, it essentially becomes Focused Value Iteration. In conclusion, Find-and-Revise should not be used, and the Solved-labeling needs improvements.

We propose to keep the good features of HDP, and address its limitations, by adding labeling to Focused Value Iteration. The *Labeled Focused Value Iteration* algorithm will have convergence guarantees, will detect when a policy is proper, and will use an efficient version of Tarjan's algorithm.

5.4 Improved Solved-Labeling

Labeled Focused Value Iteration will implement an improved labeling scheme that

– removes the overhead of keeping an explicit stack in Trajan's algorithm, and

– uses Solved-labeling to speed up convergence.

5.4.1 Pre-order and Post-order Backups

Pre-order backups are performed the first time a state is visited, to choose the greedy policy. Post-order backups are performed after the successor states of a state have been visited. Therefore, post-order backups tend to bring more improvement to the cost-to-go function, backward from the goal state. The solved-labeling algorithms discussed above do not fully take advantage of the post-order backs because the Find-and-Revise often bails out of the depth-first traversal. We propose to get rid of the Find-and-Revise and perform a complete depth-first traversal of the greedy policy graph, each iteration.

5.4.2 Tarjan's Graph Decomposition Algorithm without a Stack

Tarjan's algorithm uses an explicit stack to store the strongly connected components it finds. The stack is used by existing solved-labeling algorithms to label ϵ-consistent components as "solved". The stack is however, is not required, and we propose to replace the stack with a local depth-first traversal. Every time we find the root of a strongly connected component that is ready to be labeled "solved", a depth-first traversal, from the root state, is performed to label all states of the component as "solved".

5.4.3 Strongly Connected Components Backups

Tarjan's algorithm can detect a strongly connected component for which there is no exit under the current policy, but there is an exit under another policy that takes the agent to a state outside the closed strongly connected component. In this case, it is possible to accelerate convergence while preserving the admissibility of the cost-to-go function, as shown by the following theorem.

Theorem 5.4.1. *Consider a strongly connected component from which there is no exit under the current policy. For each state s, and for each action $a \in \mathcal{A}(s)$ that leads to a state that is outside the SCC, determine how much taking this action would increase the value of the state. (It cannot decrease the value of the state because the action was not initially selected, which means another action has a cost at least as low.) Let $\delta(s, a)$ denote this increase in value. Now find the state s and action a with the smallest $\delta(s, a)$ among all states in the strongly connected component, and all actions that lead to a state outside the strongly connected component. Let $p > 0$ denote that probability of making a transition to a state outside the strongly connected component as a result of taking this action. Then increasing the value of every state in the strongly connected component by the amount,*

$$\frac{\delta(s, a)}{1 - p} \tag{5.1}$$

preserves the admissibility of the cost-to-go function.

Proof. First, note that under the assumption that any state from which the goal state is not reached with probability one has infinite cost, a policy that does not exit the strongly connected component is suboptimal.

Second, note that changing the current policy to select action a in state s increases the value of every state by the same amount by which the value of state s is increased, since every state must exit the strongly connected component by passing through state s, and this amount must be equal to Equation (5.1).

Third, note that the state-action pair (s, a) chosen by the algorithm described above increases the cost-to-go for each state by less than any other policy that could be selected, and thus the increase must have preserved the admissibility of the cost-to-go function. □

In order to illustrate how strongly connected components backups can speedup convergence of FVI, we consider the SSP problem of Figure 5.2. We assume the initial cost-to-go estimate is equal to zero for each state. FVI's depth-first traversal starting from s_0 will visit states s_0, s_1 and s_2 because in state s_2, the best thing to do is to take the action that goes back to s_1. Each iteration, the cost-to-go value of states s_1 and s_2 increases by 2 (assuming both pre-order and post-order backups are performed). The action leading to the goal will be selected when $(1,000 + 0.9 \cdot J(s_2)) < (1 + J(s_1))$, that is, after $5,000$ iterations.

When Theorem 5.1 is applied to this problem, the cost-to-go values of states s_1 and s_2, are increased in a single iteration by $\frac{1,000 + 0.9 \cdot 0}{0.1}$, that is, by $10,000$.

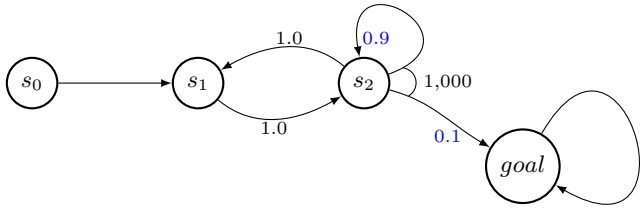

Figure 5.2: Example of SSP problem with a non-deadend SCC

5.5 Labeled Focused Value Iteration

Labeled Focused Value Iteration works just like Focused Value Iteration except for labeling. In each iteration k, Labeled Focused Value Iteration performs a depth-first traversal of the states in the policy graph $S^{\mu^k}(s_0)$, beginning from s_0. When a state $s \in S^{\mu^k}(s_0)$ is first visited, a pre-order backup is performed and the best action $\mu^k(s)$ is identified. The pre-order backup is also used to compute the Bellman residual. Each successor state $s' \in Succ(s, \mu^k(s))$ is pushed onto the (implicit) stack used to organize the depth-first traversal, if the state has not already been visited in iteration k. At the end of the traversal, a greedy policy μ^k has been found, and the cost-to-go function has been updated for all states in $S^{\mu^k}(s_0)$. When we backtrack from a state, a post-order backup is performed. However, the post-order backup only updates the cost-to-go function, not the best action.

Tarjan's algorithm uses the depth-first traversal of Labeled Focused Value Iteration to decompose the set of states in the policy graph into strongly connected components. The strongly connected components are identified backwards from the goal state. When the root of a strongly connected component is reached, the states in the strongly connected component are labeled as "solved" if the Bellman residual of every state in the connected component is smaller than ϵ, and a better action has not been identified after the post-order

backup. States in a strongly connected component are labeled as "solved" by performing a local depth-first traversal from the root, following the best policy.

5.6 Experimental Evaluation

In this section, we experimentally show the benefit of the improvements we proposed in Section 5.4.

5.6.1 Removing Find-and-Revise

In order to show the advantage of performing complete depth-first traversals each iteration instead of Find-and-Revise, we compare HDP with S-LFVI, where S-LFVI does everything the same way as HDP except the Find-and-Revise. S-LFVI performs complete depth-first traversals, instead. The results are shown in Table 5.2. What this comparison shows is that it is almost always better to perform complete depth-first traversals than "bailing out". The difference is runtime is very significant, especially for the navigation domain, where the presence of self-loop actions dramatically slow down Find-and-Revise.

We also compare these two algorithms to FVI and Labeled Focused Value Iteration (LFVI). LFVI does not use a stack for Tarjan's algorithm. When we look at the performance FVI and S-LFVI, we see that the advantage of solved-labeling is not clear on these test examples. On possible explanation is that there is not much labeling to mitigate the overhead of maintaining Tarjan's variables. Looking at the performance of S-LFVI and LFVI, we do not notice a significant difference in runtime either. The reason is that not many states are pushed on the stack for these test problems, this is suggested the fact that the final policy is relatively small for these problems.

Problem	Characteristics			Runtime in CPU seconds			
	$\|S\|$	policy size	num. Sccs in μ^*	HDP	FVI	S-LFVI	LFVI
small	9,394	**1,159**	16	0.08	0.10	0.10	0.10
big	22,534	**4,321**	26	0.78	0.34	0.33	0.33
bigger	51,943	**9,037**	52	2.76	1.20	1.11	1.09
square-3	42,085	**790**	12	0.03	0.08	0.07	0.07
square-4	383,970	**1,000**	15	0.06	1.47	1.45	1.45
ring-4	33,243	**3,973**	19	0.37	0.40	0.32	0.31
ring-5	94,396	**12,374**	26	2.48	1.71	1.57	1.53
ring-6	352,135	**37,437**	29	18.69	7.24	6.98	7.16
wet-160	25,600	**1,364**	139	88.89	0.07	0.06	0.05
wet-180	32,400	**275**	34	1.69	0.01	0.00	0.00
wet-200	40,000	**749**	96	31.26	0.03	0.03	0.02
nav-16	65,535	**192**	64	99.63	0.07	0.07	0.07
nav-18	262,143	**2,494**	1035	2581.84	1.63	1.66	1.68
nav-20	1,048,575	**1,861**	742	2142.06	2.23	2.81	2.16

Table 5.2: Effect of complete depth-first traversals. Running times in CPU seconds until ϵ-consistency with $\epsilon = 10^{-8}$

.

5.6.2 Removing the Stack from Tarjan's algorithm

We compare FVI, S-LFVI, and LFVI using a grid problem where most of the state space is visited each iteration. For this problem, the stack of Tarjan's algorithm is heavily used. The final policy visits over 98% of the state space. Table 5.3 shows the results. The results show the advantage of not using an explicit stack during the depth-first traversal. LFVI is about 25% faster than S-LFVI. It is worth noting that FVI remains competitive in this experiment and the previous one. The speedup of solved-labeling in S-LFVI does not mitigate the overhead of using an explicit stack in Tarjan's algorithm. Of the two improvements of getting rid of Find-and-Revise and removing the stack from Tarjan's algorithm, the former seems to have more benefit.

Problem	Characteristics			Runtime in CPU seconds		
	$\|S\|$	policy size	num. Sccs in μ^*	FVI	S-LFVI	LFVI
grid-160	25,600	**25,295**	4	3.25	3.25	2.49
grid-240	57,600	**57,600**	4	10.50	10.88	8.13
grid-300	90,000	**89,381**	5	17.47	18.25	13.43
grid-400	160,000	**159,314**	5	43.65	45.22	33.95
grid-500	250,000	**249,018**	4	85.39	89.27	67.23

Table 5.3: Effect of using a stack in Tarjan's algorithm. Running times in CPU seconds until ϵ-consistency with $\epsilon = 10^{-8}$

.

5.6.3 Strongly Connected Components Backups

We experimentally tested the speedup technique of Theorem 5.4.1. Figure 5.3 shows that this technique can substantially reduce the number of iterations it takes for a heuristic search algorithm to converge. The speedup helps, in a way, in finding a proper policy faster. However, the speedup is problem dependent, it is very modest on other problems we tested it on.

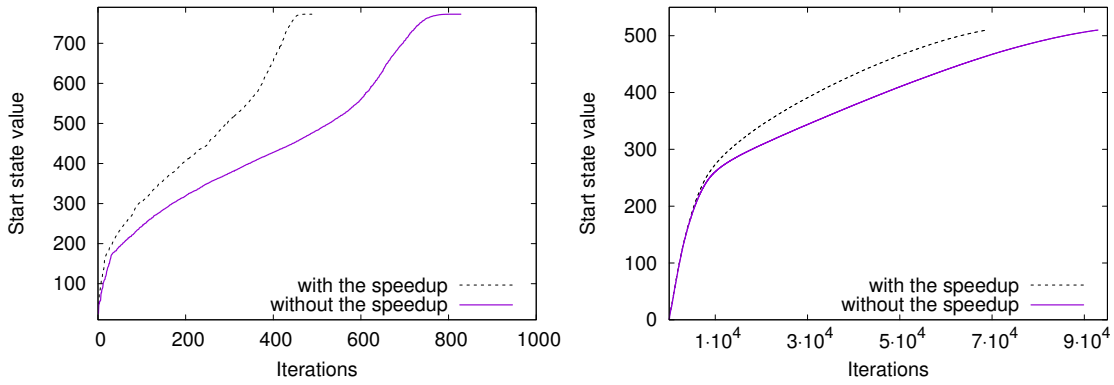

Figure 5.3: Speedup on instances of the Blocks World (left), and Zeno Travel (right) problems.

5.6.4 Suboptimality Bounds

When a complete depth-first traversal is performed each iteration, it becomes possible to compute a Bellman residual, and the new suboptimality bounds can be used to monitor the quality of the solution. In addition, suboptimality bounds can be used to label a state "solved" when the Bellman residual is small enough, for the policy of the start state to be ϵ-optimal, instead of just ϵ-consistent. Each iteration, the algorithm computes the value $residualBound$, using the upper bound on the cost-to-go function of the start state. The quantity $residualBound$ is roughly equal to ϵ divided by the upper bound on the optimal cost-to-go value of the start state. The way the value of $residualBound$ is computed guarantees that when the start state is labeled as "solved", its policy is ϵ-optimal. Algorithm 14 gives the pseudocode of Labeled Focused Value Iteration.

Algorithm 13: Driver for Labeled focused value iteration with suboptimality bounds

```
1  Input: SSP problem, start state s_0
2  Output: ε-optimal value function for the start state s_0
3  LFVI(s_0)
4  begin
5      index ← residualBound ← k ← 0
6      repeat
7          k ← k + 1; c̄_k ← −∞
8          c̄_k ← LFVIrec(s_0, residualBound, k)
9          if (c̄_k < g) then
10             s_0.UPVALUE ← (s_0.VALUE − c̄_k) / (g − c̄_k)
11             residualBound ← ε / (s_0.UPVALUE − g + ε)
12         else
13             residualBound ← 0
14     until (s_0.SOLVED)
```

Algorithm 14: Recursive function for Labeled focused value iteration

```
1  LFVIrec(s, residualBound, k)
2  begin
       // base cases
3      if (s is goal) then s.STATUS ← SOLVED
4
5      if (s.SOLVED) then return 0
6
       // pre-order backup, to select best action
7      RESIDUAL ← BACKUP(s)
8      c̄_k ← max{c̄_k, RESIDUAL}
9      μ^k(s) ← greedy_action
       // set Tarjan variables
10     s.LOW ← index
11     s.IDX ← index
12     s.VISITED ← k
13     ++index
14
15     foreach s' ∈ Succ(s, μ^k(s)) do
16         if (s'.VISITED < k) then
               // s' not visited yet
17             LFVIrec(s', residualBound, k)
18             s.LOW ← min{s.LOW, s'.LOW}
19         else if (s'.VISITED == k) then
               // s' on the stack
20             s.LOW ← min{s.LOW, s'.IDX}
21
       // post-order backup with local residuals
22     RESIDUAL ← BACKUP(s)
23     maxResidual ← max{maxResidual, RESIDUAL}
24
       // If root of an SCC, label its states, if possible
25     if (s.LOW == s.IDX ) then
26         if (maxResidual < residualBound) then
               // local DFS from the root of the SCC
27             LABEL_SOLVED(s)
28     return c̄_k
```

5.7 Summary

In this chapter, we showed how to improve existing heuristic search algorithms that use solved-labeling to accelerate convergence. We analyzed these algorithms to understand what makes them less efficient. We identified two main limitations. The Find-and-Revise approach, and the unnecessary overhead of using a stack in Tarjan's algorithm. The Find-and-Revise seems to be the major drawback, as suggested by the experimental results. In addition to slowing down these algorithms, Find-and-Revise does not compute a Bellman residual each iteration, and an algorithm using it cannot dynamically monitor the quality of the solution it computes. We described an algorithm that uses solved-labeling with an efficient Tarjan's algorithm, and performs complete depth-first traversals instead of Find-and-Revise. From the experimental results, we learned that solved-labeling helps only when we take out the stack in Tarjan's algorithm. The Labeled Focused Value Iteration is about 20% faster than Focused Value Iteration. Focused Value Iteration remains competitive because it has virtually no overhead, and is easy to analyze.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the results of the dissertation and identifies directions for future work.

6.1 Summary of Contributions

This dissertation makes three main contributions.

Branch-and-Bound Value Iteration

In Chapter 3, we described a branch-and-bound value iteration algorithm for MDPs. It generalizes the action elimination procedure by pruning parts of the state space that become unreachable from that start state, as the result of action elimination. The algorithm uses suboptimality bounds to speed up the convergence of value iteration by eliminating unreachable states as well as suboptimal actions. State elimination has been implemented using state counters, to minimize overhead. The algorithm preserves the convergence guarantees of value iteration. Experimental results showed that this algorithm is up to an order of magnitude faster than regular value iteration. The branch-and-bound approach will works best for problems for which, only a fraction of the state space needs to be

evaluated to compute an optimal policy for the start state. In the worst case however, it will have to update the entire state space each iteration, just like regular value iteration.

Integration of Suboptimality Bounds in Heuristic Search

In Chapter 4, we reviewed and experimentally evaluated efficient suboptimality bounds for SSP problems. The bounds can be used in value iteration to address the lack of an efficient test for convergence to an ϵ-optimal policy. The test for convergence to an ϵ-optimal policy using the new bounds is as practical as the test for convergence to an ϵ-consistent policy. We also showed how to integrate the new bounds in heuristic search algorithms that compute a Bellman residual each iteration. Using the bounds in these algorithms adds no extra overhead. Experimental results showed that the new bounds become quickly as tight as otherwise prohibitively expensive suboptimality bounds that require an exact evaluation of a proper policy. In addition, the suboptimality bounds can be used to detect whether a greedy policy with respect to a lower bound cost-to-go function is proper, that is, whether the goal state is reached with probability one.

Improved Solved-Labeling in Heuristic Search

In Chapter 5, we analyzed heuristic search algorithms for SSP problems that use Solved-labeling to speed up convergence by labeling states as "solved". We identified features that make these algorithms less efficient. We described Labeled Focused Value Iteration, an algorithm that addresses the weaknesses of current Solved-labeling based algorithms, and uses the suboptimality bounds of Chapter 4. Experimental evaluation showed the advantage of the new algorithm.

6.2 Directions for Future Work

We enumerate a few natural extensions of this work, as future work.

6.2.1 Detecting Unsolvability

The heuristic search algorithms for the SSP problem, discussed in this dissertation assume the existence of a proper policy. However, problems where the goal state cannot be reached with probability one also arise in probabilistic planning. For example, many domains from the first two International Probabilistic Planning Competitions (IPPC) have *deadends*, which are states from which a goal state cannot be reached at all, as well as additional states from which a goal state can be reached, but with probability less than one. For example, for the SSP problem of Figure 6.1, algorithms for SSP problems are not able to recognize that it is unsolvable, that is, the goal state G cannot be reached with probability one from the start state s_0.

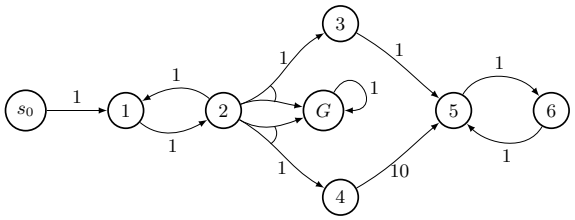

Figure 6.1: Example of an unsolvable SSP problem.

One extension of Labeled Focused Value Iteration is to recognize when an SSP problem is unsolvable, that is, the start state does not have a proper policy.

6.2.2 Maximizing the Probability of Reaching a Goal State

When a goal state cannot be reached with probability one, the question of finding a policy that maximizes the probability of reaching the goal becomes important. We consider a generalization of the SPP problem where the objective is to find a policy that maximizes the probability of reaching a goal state. The problem of finding a policy that maximizes the probability of reaching a goal state is called the MAXPROB problem.

While value iteration and related heuristic search algorithms are guaranteed to converge to the optimal cost-to-go function in the classical SSP problem definition, this is not the case for MAXPROB. The MAXPROB problem is complicated by the presence of non-goal zero-cost cycles, where the agent can stay indefinitely without incurring cost, thus avoiding the goal state. In this case, The Bellman optimality equation may have multiple suboptimal fixed-point solutions. Furthermore, not all policies that are greedy with respect to the optimal cost-to-go function are proper even though all optimal polices are greedy with respect to the optimal cost-to-go function. For example, in Figure A.4, the policy cycling on states s_0, s_1, s_2 and s_3 is greedy but not proper.

An even more interesting objective is to compute the maximum probability of reaching the goal state while minimizing the associated expected cost.

The state-of-the-art heuristic search algorithm for the MAXPROB is FRET [34]. FRET ignores action costs in non-goal states and gets a reward of 1 only when it hits the goal state, which means it does not have a way of discriminating among policies based on the expected cost to reach the goal state.

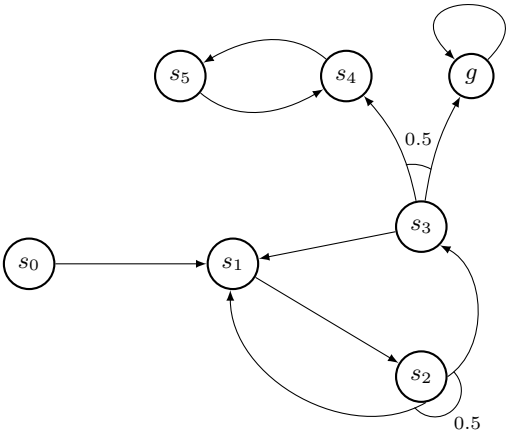

Figure 6.2: An SSP problem where the goal is not reached with probability one.

A research direction is to extend Labeled Focused Value Iteration to improve on the state-of-the-art FRET algorithm in solving the MAXPROB problem.

REFERENCES

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebric decision diagrams and their applications," *Formal methods in system design*, vol. 10, no. 2-3, 1997, pp. 171–206.

[2] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 72, no. 1, 1995, pp. 81–138.

[3] A. G. Barto, S. J. Bradtke, and S. P. Singh, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 72, no. 1, 1995, pp. 81–138.

[4] R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, USA, 1957.

[5] D. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models*, Prentice-Hall, Englewood, USA, 1987.

[6] D. Bertsekas, *Dynamic Programming and Optimal Control, Vol. 1*, 3rd edition, Athena Scientific, Belmont, MA, 2005.

[7] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, 2nd edition, Athena Scientific, 2000.

[8] D. P. Bertsekas and J. N. Tsitsiklis, "An analysis of stochastic shortest path problems," *Mathematics of Operations Research*, vol. 16, no. 3, 1991, pp. 580–595.

[9] V. D. K. Bhuma, "Bidirectional LAO* algorithm (a faster approach to solve goal-directed MDPs)," *University of Kentucky Master's Theses. 225*, 2004.

[10] B. Bonet and H. Geffner, "Faster heuristic search algorithms for planning with uncertainty and full feedback," *IJCAI*, 2003, pp. 1233–1238.

[11] B. Bonet and H. Geffner, "Faster heuristic search algorithms for planning with uncertainty and full feedback," *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI-03)*. 2003, pp. 1233–1238, Morgan Kaufmann.

[12] B. Bonet and H. Geffner, "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming.," *ICAPS*, 2003, vol. 3, pp. 12–21.

[13] B. Bonet and H. Geffner, "Labeled RTDP: Improving the convergence of real-time dynamic programming," *Proc. of the 13th Int. Conf. on Automated Planning and Scheduling (ICAPS-03)*. 2003, pp. 12–21, AAAI Press.

[14] B. Bonet and H. Geffner, "Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs.," *ICAPS*, 2006, vol. 6, pp. 142–151.

[15] B. Bonet and H. Geffner, "Learning Depth-First Search: A Unified Approach to Heuristic Search in Deterministic and Non-Deterministic Settings, and Its Application to MDPs," *Proc. of the 16th Int. Conf. on Automated Planning and Scheduling (ICAPS-06)*. 2006, pp. 142–151, AAAI Press.

[16] A. J. Briggs, C. Detweiler, D. Scharstein, and A. Vandenberg-Rodes, "Expected shortest paths for landmark-based robot navigation," *The International Journal of Robotics Research*, vol. 23, no. 7-8, 2004, pp. 717–728.

[17] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, 1992, pp. 293–318.

[18] D. Bryce and O. Buffet, "6th international planning competition: Uncertainty part," *Proceedings of the 6th International Planning Competition (IPC-08)*, 2008.

[19] P. Dai and J. Goldsmith, "LAO*, RLAO*, or BLAO*," *AAAI Workshop on Heuristic Search*, 2006, pp. 59–64.

[20] P. Dai and J. Goldsmith, "Topological Value Iteration Algorithm for Markov Decision Processes.," *IJCAI*, 2007, pp. 1860–1865.

[21] P. Dai and E. A. Hansen, "Prioritizing Bellman Backups without a Priority Queue.," *ICAPS*, 2007, pp. 113–119.

[22] K. V. Delgado, C. Fang, S. Sanner, L. N. De Barros, et al., "Symbolic Bounded Real-Time Dynamic Programming.," *SBIA*. Springer, 2010, pp. 193–202.

[23] Z. Feng and E. Hansen, "Symbolic LAO* search for factored markov decision processes," *Proceedings of the AIPS-02 Workshop on Planning via Model Checking*, 2002, pp. 49–53.

[24] D. Ferguson and A. Stentz, "Focussed processing of MDPs for path planning," *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*. IEEE, 2004, pp. 310–317.

[25] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," *Formal Methods for Eternal Networked Software Systems*, Springer, 2011, pp. 53–113.

[26] E. A. Hansen, "Error bounds for stochastic shortest path problems," *Mathematical Methods of Operations Research*, vol. 86, no. 1, 2017, pp. 1–27.

[27] E. A. Hansen and I. Abdoulahi, "Efficient bounds in heuristic search algorithms for stochastic shortest path problems," *Proceedings of the 29th AAAI Conference on Artificial Intelligence(AAAI-15)*. 2015, AAAI Press, Austin, Texas.

[28] E. A. Hansen and I. Abdoulahi, "General error bounds in heuristic search algorithms for stochastic shortest path problems," *Proceedings of the 30th AAAI Conference on Artificial Intelligence(AAAI-16)*. 2016, AAAI Press, Phoenix, Arizona.

[29] E. A. Hansen and S. Zilberstein, "LAO*: A heuristic search algorithm that finds solutions with loops," *Artificial Intelligence*, vol. 129, no. 1, 2001, pp. 35–62.

[30] N. Hastings and J. Mello, "Tests for suboptimal actions in discounted Markov programming," *Management Science*, vol. 19, no. 9, 1973, pp. 1019–1022.

[31] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "SPUDD: Stochastic planning using decision diagrams," *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 279–288.

[32] R. Howard, *Dynamic programming and Markov processes*, MIT Press, 1960.

[33] IPPC-04, "IPC 2004 Benchmark,", `https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume24/younes05a-html/`, [Online; accessed 14-Aug-2017].

[34] A. Kolobov, Mausam, D. Weld, and H. Geffner, "Heuristic Search for Generalized Stochastic Shortest Path MDPs," *ICAPS*, 2011.

[35] A. Kolobov and M. Mausam, "Planning with Markov decision processes: An AI perspective," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, 2012, pp. 1–210.

[36] R. E. Korf, "Real-time heuristic search," *Artificial intelligence*, vol. 42, no. 2-3, 1990, pp. 189–211.

[37] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: A hybrid approach," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 2, 2004, pp. 128–142.

[38] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-time Systems," *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, G. Gopalakrishnan and S. Qadeer, eds. 2011, vol. 6806 of *LNCS*, pp. 585–591, Springer.

[39] M. Kwiatkowska, D. Parker, and H. Qu, "Incremental quantitative verification for Markov decision processes," *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 359–370.

[40] J. MacQueen, "A Test for Suboptimal Actions in Markovian Decision Problems," *Operations Research*, vol. 15, no. 3, 1967, pp. 559–561.

[41] J. MacQueen, "A Test for Suboptimal Actions in Markovian Decision Problems," *Operations Research*, vol. 15, no. 3, 1967, pp. 559–561.

[42] J. B. MacQueen, *A modified dynamic programming method for Markovian decision problems*, Tech. Rep., DTIC Document, 1965.

[43] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL - The Planning Domain Definition Language," *Technical Report CVC TR98003/DCS TR1165*, 1998.

[44] H. B. McMahan and G. J. Gordon, "Fast Exact Planning in Markov Decision Processes.," *ICAPS*, 2005, pp. 151–160.

[45] H. B. McMahan, M. Likhachev, and G. J. Gordon, "Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees," *Proceedings of the 22nd international conference on Machine learning*. ACM, 2005, pp. 569–576.

[46] N. J. Nilsson, *Principles of artificial intelligence*, Morgan Kaufmann, 2014.

[47] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*, Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.

[48] E. L. Porteus, "Some bounds for discounted sequential decision processes," *Management Science*, vol. 18, no. 1, 1971, pp. 7–11.

[49] E. L. Porteus, "Bounds and Transformations for Discounted Finite Markov Decision Chains," *Oper. Res.*, vol. 23, no. 4, 1975, pp. 761–784.

[50] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st edition, John Wiley & Sons, Inc., New York, NY, USA, 1994.

[51] S. Sanner, "Relational Dynamic influence Diagram Language (RDDL): Language description," *Unpublished MS. Australian National University*, 2010, p. 32.

[52] S. Sanner, R. Goetschalckx, K. Driessens, G. Shani, et al., "Bayesian Real-Time Dynamic Programming.," *IJCAI*. Citeseer, 2009, pp. 1784–1789.

[53] SkyAI, "Tutorial - Example - Mountain Car", `http://skyai.org/wiki/?SkyAI`, [Online; accessed 3-Oct-2017].

[54] T. Smith and R. Simmons, "Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic," *AAAI*, 2006, pp. 1227–1232.

[55] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, 2010, pp. 1–103.

[56] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, 1972, pp. 146–160.

[57] H. Warnquist, J. Kvarnström, and P. Doherty, "Iterative Bounding LAO*.," *ECAI*, 2010, pp. 341–346.

[58] D. Wingate and K. Seppi, "Efficient Value Iteration using Partitioned Models," *Proceedings of the International Conference on Machine Learning and Applications*, 2003, pp. 53–59.

[59] H. L. Younes and M. L. Littman, "PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects," *Techn. Rep. CMU-CS-04-162*, 2004.

APPENDIX A

TEST PROBLEMS

A.1 SSP Test Problems from IPPC Competitions

In this section we describe the test domains we use for the experimental evaluation. We present six domains from the ICAPS International Probabilistic Planning Competition 2008 [18]. The reason we use the 2008 domains in that subsequent competitions focused on finite horizon problems. The BlocksWorld and Zeno Travel domains have a proper policy all instances. These domains are solvable by any regular algorithm for SSP problems. The ExplodingBlocksWorld and TireWorld have a mix of solvable and unsolvable instances (unsolvable meaning, the dead end state are unavoidable from the start state under all policies).

A.1.1 Zenotravel

The Zenotravel domain is based on the deterministic Zenotravel used in the IPC-3. It involves three types of objects: people, cites and airplanes. An instance consists of using airplanes to move a given a number of people from their initial locations to their destinations using a fleet of airplanes.

The actions available are: start-boarding, complete-boarding, start-debarking complete-debarking, start-refueling, start-flying, complete-flying, start-zooming and complete-zooming. Zooming is flying at a faster speed and requires more fuel than regular flying. The start-X actions are deterministic and the complete-X actions succeed with probability p. The value of p is $1/2, 1/4, 1/7, 1/25$ and $1/15$ for complete-boarding, complete-debarking, complete-refueling, complete-flying and complete-zooming, respectively. The cost of all actions is 1 except for actions flying and zooming that have

costs 10 and 25 respectively. The initial state is given by the location of the planes, the initial fuel level (possible levels are $1/4, 1/2, 3/4$ and $full$) and the location of all people. The goal state is given by the destinations of the planes and all people. All polices for this domain are proper. Even though the space becomes quickly large (value iteration would only solve the first two instances), the final policy visits less this domain takes The final policy is very simple and visits less than 0.01% of the state space.

A.1.2 Boxworld

The Boxworld domain is a probabilistic version of the deterministic logistics domain. Manipulated objects are: cites, boxes, trucks and planes. There are six possible actions: load-box-on-truck-in-city, unload-box-from-truck-in-city, load-box-on-plane-in-city, unload-box-from-plane-in-city, drive-truck and fly-plane. All actions have unit costs expect drive-truck and fly-plane, with cost 5 and 25, respectively. The drive-truck action is probabilistic. The start configuration is given by a graph of cities, the initial locations of boxes. The goal configuration specifies the final location of each box. The objective is to move from the start configuration to the goal configuration, with minimum expected costs.

A.1.3 Blocksworld

The Blocksworld domain we use here is also a probabilistic version of the deterministic Blocks World. The domain involves two types of objets: blocks and tables. An instance has a number of blocks and a table. The objective is to find a plan that transforms a given initial state (configuration of the blocks) into a goal state. The actions of the domain are: pick-up$(\mathrm{b}_1, \mathrm{b}_2)$ to remove block b_1 on top of b_2 and put it on the table,

$\text{pick-up-from-table(b)}$ to pick block b from table, $\text{put-on-block}(\text{b}_1, \text{b}_2)$ to put block b_1 on top of b_2, put-down(b) to put down block b on the table, $\text{pick-tower}(\text{b}_1, \text{b}_2, \text{b}_)$ to pick remove blocks b_1 and b_2 from b_3 at once, $\text{put-tower-on-block}(\text{b}_1, \text{b}_2, \text{b}_3)$ put tower (b_1, b_2) on top of block b_3, and $\text{put-tower-down}(\text{b}_1, \text{b}_2)$ to put tower (b_1, b_2) on the table. For a full description of the actions (with their preconditions and effects), we refer to the domain description given in Section 2.3 of Chapter 2. The actions have stochastic outcomes. The success probabilities are $3/4, 3/4, 3/4, 1, 1/10, 1/10$ and 1, respectively. All actions have unit cost for this domain. There is always a proper policy for a given instance of this domain. An example of a policy that is proper (but not necessarily optimal) is:

1. From the initial configuration, put each block onto the table
2. Construct the goal configuration bottom up, by placing each block into its place.

Figure A.1 shows an example of an initial and goal configurations for the Blocks World domain.

Figure A.1: initial (left) and goal (right) configurations

A.1.4 Exploding Blocksworld

The Exploding Blocksworld is very similar to the Blocksworld domain. As in the Blocksworld domain, blocks and tables are manipulated. The difference between the two

domains is that, in the Exploding Blocksworld domain, every time actions $\text{put-down}(\text{b})$ and $\text{put-on-block}(\text{b}, \text{b'})$ are executed, the block b has probabilities $2/5$ and $1/10$ of exploding and destroying the block or the table below it, respectively. Nothing can be placed on a destroyed block or table. It is therefore possible to end up in a "dead-end" configuration, where it is not possible to construct a goal configuration from a given initial configuration. Not all instances of this domain have a proper policy due to the presence of such unavoidable dead-ends. All actions have unit cost. In contrast with the first two domains, the instances in this domain are not always solvable by algorithms for regular SSP problems since they assume the existence of at least one proper policy.

A.1.5 Tire World

The Tire World domain is a probabilistic domain with $locations$ as the only type of object manipulated. A car has to travel from an initial location to a goal location. The available actions are: $\text{move-car}(\text{c}_1, \text{c}_2)$ to move the car from location c_1 to c_2, $\text{load-tire}(\text{c})$ to load a spare tire in location c, and $\text{change-tire}()$ to change the tire. The stochasticity in this domain comes from action $\text{move-car}(\text{c}_1, \text{c}_2)$ where, every time the car moves from location c_1 to c_2, it gets a flat tire with probability $1/2$. Once the tire is flat, the car cannot move until the tire is changed (execute action $\text{change-tire}()$). The car can carry at most one spare tire. Not all locations have available spare tires. If the does not have a spare tire, the action $\text{change-tire}()$ is disabled. It is therefore possible to be in a "dead-end" state if the car does not have a spare tire and gets a flat tire. All actions in this domain have unit

cost. As in the previous domain, not all instances of this domain have a proper policy. Figure A.2 shows an example of this domain [33].

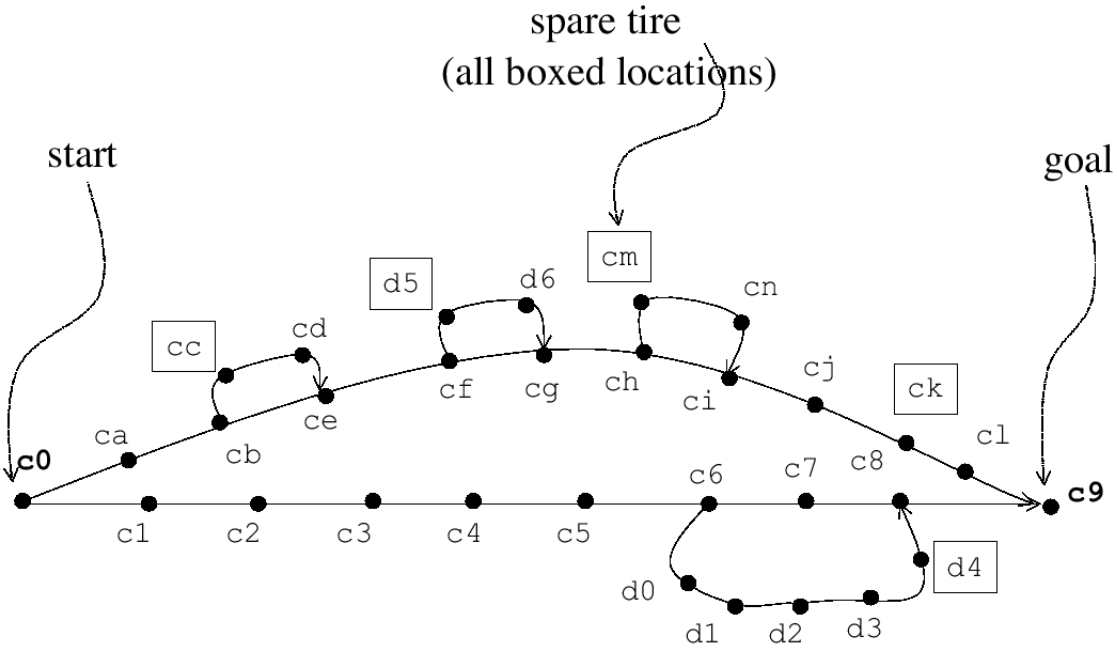

Figure A.2: Tire World map

A.2 Other SSP Test Problems

In this section we describe other classic test problems that are used in our experiments.

A.2.1 Racetrack

The Racetrack problem [3] is an undiscounted MDP that simulates automobile racing. A car is placed on the starting line at a random position, and moves are made in which the car attempts to move down the track toward the finish line. Acceleration and deceleration are simulated as follows. If in the previous move the car moved h squares horizontally

and v squares vertically, then the present move can be h' squares vertically and v' squares horizontally, where the difference between h' and h is -1, 0, or 1, and the same differences between v' and v. If the car hits the track boundary, it is moved back to a random position on the starting line, reducing its velocity to zero (i.e., $h' - h$ and $v' - v$ are equal to zero), and start again. The objective is to learn to control the car so that it crosses the finish line in as few moves as possible. Figure A.3 shows the tracks for two instances of the racetrack problem that are used in our experimental evaluation. The green cells are the start states and the red ones are the goal states.

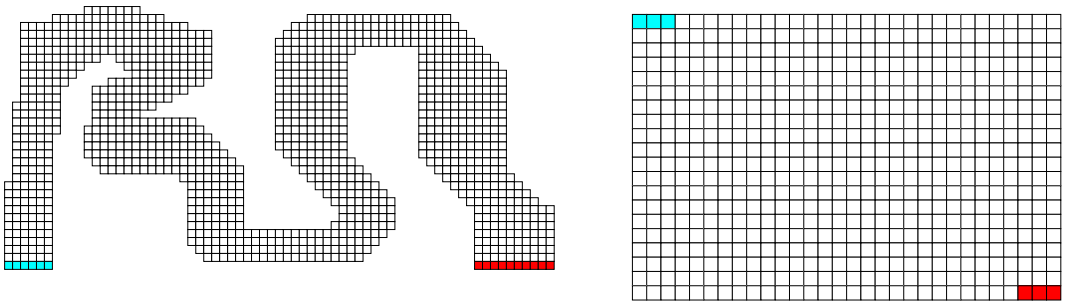

Figure A.3: Racetracks *bigger* (left) and *square-3* (right)

A.2.2 Double-arm pendulum

The double-arm pendulum (DAP) [58] is a pendulum with another pendulum attached to it. The state space is described by four variables: angles θ_1, θ_2 and the angular velocities $\dot{\theta_1} \in [-10, 10]$ and $\dot{\theta_2} \in [-15, 15]$ radians/s. The agent has two actions available representing positive and negative torques applied to a rotating pendulum. Rewards are zero

everywhere but in the balanced point. The objective is to balance the pendulum vertically at a zero degree angles and zero velocities.

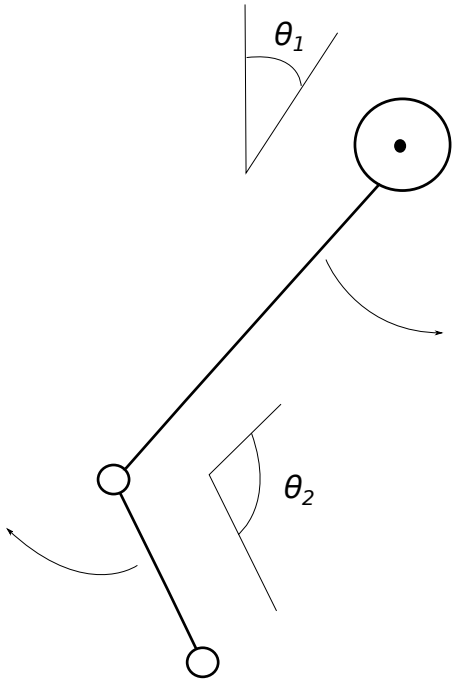

Figure A.4: Double-arm pendulum

A.2.3 Wet floor

This domain is the classic navigation grid problem, where some cells are wet and thus slippery. Every cell in the grid is wet with some probability p. A dry cell has four deterministic actions. In a wet cell, an action can have up to four successor states. The objective is to compute a contingency plan that minimizes the expected number of steps to reach the goal state.

A.2.4 Mountain car

The mountain car domain consists of an underpowered car that must drive up a steep mountain to reach a goal point, starting from the bottom of the valley (see Figure A.5 for illustration [53]). Because the mountain is very steep, the car cannot directly reach the goal, it has to go back and forth to gain enough momentum to get to the goal. The objective is to learn the right sequence of accelerations and decelerations (optimal policy) to reach the goal with minimum number of steps.

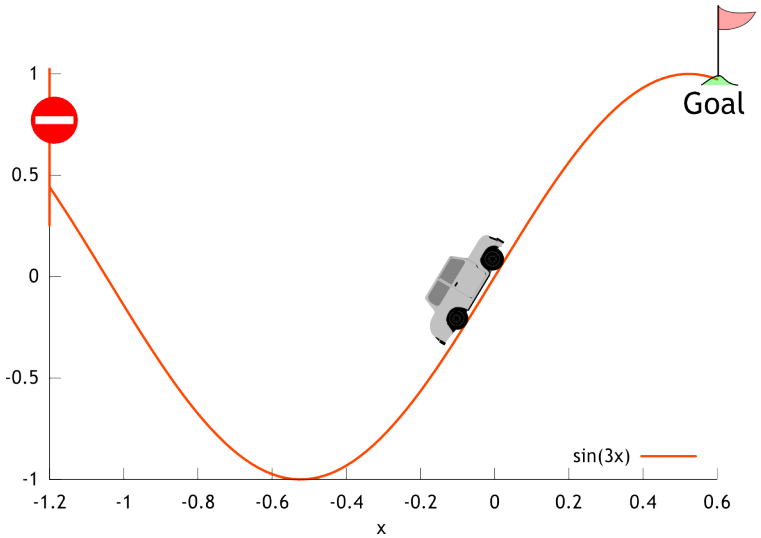

Figure A.5: Example of Mountain car problem