1-1-2013

Experimental Evaluation of Error bounds for the Stochastic Shortest Path Problem

Ibrahim Abdoulahi

Experimental evaluation of error bounds for the stochastic shortest path problem

By

Ibrahim Abdoulahi

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

Mississippi State, Mississippi

December 2013

Experimental evaluation of error bounds

for the stochastic shortest path problem

By

Ibrahim Abdoulahi

Approved:

Eric Hansen
(Major Professor)

Cindy L. Bethel
(Committee Member)

Burak Eksioglu
(Committee Member)

Edward B. Allen
(Graduate Coordinator)

Achille Messac
Dean
Bagley College of Engineering

Name: Ibrahim Abdoulahi

Date of Degree: December 14, 2013

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Eric Hansen

Title of Study: Experimental evaluation of error bounds for the stochastic shortest path problem

Pages of Study: 48

Candidate for Degree of Master of Science

A stochastic shortest path (SSP) problem is an undiscounted Markov decision process with an absorbing and zero-cost target state, where the objective is to reach the target state with minimum expected cost. This problem provides a foundation for algorithms for decision-theoretic planning and probabilistic model checking, among other applications.

This thesis describes an implementation and evaluation of recently developed error bounds for SSP problems. The bounds can be used in a test for convergence of iterative dynamic programming algorithms for solving SSP problems, as well as in action elimination procedures that can accelerate convergence by excluding provably suboptimal actions that do not need to be re-evaluated each iteration. The techniques are shown to be effective for both decision-theoretic planning and probabilistic model checking.

Key words: Stochastic shortest path problem, Error bounds, Value iteration, Convergence, Action elimination.

DEDICATION

To the beloved Madinat-Almounawara and its people.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Eric Hansen for his valuable support and time he has provided throughout the time I spent working on this thesis. I would like to thank Dr. Cindy Bethel and Dr. Burak Eksioglu for their help and for being part of my committee. I am grateful to Dave Parker for willing to share the Prism model checker source code.

I thank my friends who made my stay in Starkville more enjoyable, in and outside university: Nadeem, Khaled, Zaeem, Zadia, Peng, Anara, Proteek, Yousef, Kazi, Omo, Fan, Jinchuan. I would also like to thank Karin Lee for her friendship and valuable help in dealing with my administrative paperwork among many other things, thanks Karin!

One of the things I have most enjoyed outside school is the Boardtown Road Runners club. I am very grateful to Marcie and Mike for their friendship and for welcoming me among many others to their home every Saturday morning.

Last but not least, I would like to thank my family, especially my mother for their support and prayers. Thank you for accepting me being away from home for so many years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis implements and evaluates new error bounds for the stochastic shortest path problem that were recently derived by Hansen [18]. Experiments on decision-theoretic planning and probabilistic model checking problems show that the bounds can be used to test for convergence and eliminate suboptimal actions.

1.1 Motivation

A Markov decision process (MDP) is a widely used mathematical model for sequential decision problems under uncertainty [6]. The stochastic shortest path (SSP) problem is a Markov decision process that generalizes the deterministic shortest path problem in a graph by allowing the effect of an action in a particular state to be a probability distribution over successor states. MDPs, especially in the form of SSP problems, are commonly used in decision-theoretic planning [1, 7], probabilistic model checking [5, 20], and have many other applications.

Value iteration (VI) [6] is the baseline algorithm for solving MDPs, including SSP problems. It finds the optimal solution by iteratively improving the value of every state until all state values converge (in practice an ϵ-optimal solution is computed). Once the

value function converges, an optimal policy is extracted. But since VI only converges in the limit, it is useful to have errors bounds to approximate the solution.

Although the SSP problem has been widely used in computer science for more than two decades, no error bounds have been known until the recent work of Hansen [18]. In this thesis, the new bounds are implemented and evaluated. As we will see, the error bounds can be used to improve value iteration in two ways:

1. to test for convergence.
2. to speedup convergence by using error bounds to limit the number of actions (and states) that need to be evaluated each iteration.

1.2 Organization of the thesis

The remainder of this thesis is organized as follows. Chapter 2 reviews the stochastic shortest path problem and algorithms for solving it, especially value iteration. The new error bounds are also reviewed in Chapter 2. In Chapters 3 and 4, the error bounds are applied to decision-theoretic planning and probabilistic model checking respectively. Conclusions and future work are described in Chapter 5.

CHAPTER 2

BACKGROUND

This chapter reviews the stochastic shortest path problem, dynamic programming algorithms for solving the problem, and methods for computing error bounds.

2.1 Stochastic shortest path problem

The stochastic shortest path problem (SSP) is a generalization of the deterministic shortest path problem where each state (node) and action is associated with a probability distribution over a set of possible successor states [9]. The problem is defined by a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, c, t >$ where:

- $\mathcal{S}$ is a finite set of states numbered from 1 to n, where t is a goal or terminal state and s_1 is a start state.
- $\mathcal{A}$ is set of actions and $\mathcal{A}(s)$ denotes the actions available for state s.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$: specifies a probabilistic transition function, where $\mathcal{T}(s, a, s') = p_{s,s'}(a)$ is the probability that taking action $a \in \mathcal{A}$ in state s results in a transition to successor state s'.
- $c : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$: is a cost function that associates with each state $s \in S$ and action $a \in \mathcal{A}(s)$, the cost $c(s, a)$ of taking action a in s.

The goal state t is absorbing, $i.e., \forall\ a\ \in\ \mathcal{A}(t), p_{t,t}(a)\ =\ 1$. In addition, $c(t, a)\ =\ 0$, $\forall\, a \in \mathcal{A}(t)$.

The classic deterministic shortest path problem is a special case of the stochastic shortest path problem in which the transition function is deterministic instead of stochastic.

A policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ is a mapping from states to actions that specifies which action to take in each state. A policy π is said to be proper if the process is guaranteed to reach the goal state when policy π is followed beginning from any initial state. It is said to be improper otherwise.

Furthermore, the SSP problem is defined by the following restrictions:

- There exist at least one proper policy.
- For every improper policy π, there exist at least one state $s \in \mathcal{S}$ for which $J^{\pi}(s)$ is infinite, where $J^{\pi}(s)$ is the expected cumulative cost of following policy π starting from state s. This is defined more precisely by Equation 2.1 below.

Solving an SSP problem means finding an optimal policy, where a policy is said to be optimal if it minimizes the expected cumulative cost of the process starting from every initial state. The assumptions of the SSP problem guarantee that an optimal policy is also a proper policy.

2.2 Dynamic programming algorithms

For a given policy π and state $s \in \mathcal{S}$, the expected value of following policy π starting from state s is defined as follows:

$$J^{\pi}(s) = E\Big[\sum_{t=0}^{\infty} c(s_t, \pi(s_t))|s_0 = s\Big]. \tag{2.1}$$

The value function $J^{\pi} : \mathcal{S} \rightarrow \mathbb{R}$ for policy π is the solution of the following system of linear equations.

$$J^{\pi}(s) = \Big[c(s, \pi(s)) + \sum_{s' \in S} p_{s,s'}(\pi(s))J^{\pi}(s')\Big], s \in S. \tag{2.2}$$

Dynamic programming algorithms for solving the stochastic shortest path problem leverage the fact that the optimal value function $J^* : \mathcal{S} \rightarrow \mathbb{R}$ is the unique solution of a Bellman optimality equation defined as follows:

$$J^*(s) = \min_{a \in \mathcal{A}(s)} \Big[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J^*(s') \Big], s \in S. \tag{2.3}$$

The Bellman equations given by Equation 2.3 are non-linear. They can be solved by either value iteration or policy iteration.

2.2.1 Value iteration

The value function given by Equation 2.1 can be iteratively computed using dynamic programming [6]. The dynamic programming operator T is defined as follows:

$$TJ(s) = \min_{a \in \mathcal{A}(s)} \Big[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J(s') \Big], s \in S. \tag{2.4}$$

Value iteration (VI) [6] uses Equation 2.4 to successively approximate the optimal value function staring with an initial estimate. VI takes an SSP problem, an ϵ arbitrary small, and performs a sequence of Bellman updates for all the states until the value function converges. The value is improved each iteration and when the maximum difference between values of two consecutive iterations (Bellman error) is small enough, the algorithm is said to have converged. The following properties hold in value iteration [9]:

1. The optimal cost vector is the unique solution of Bellman's equation: $TJ^* = J^*$.
2. The value iteration method converges to the optimal cost vector J^* for an arbitrary starting vector.

Table 2.1 sketches the algorithm.

Table 2.1

Value iteration

Input: An SSP problem $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{T}, c, t >$ and a threshold parameter ϵ.
Output: A value function J for which, for every state s, $|J'(s) - J(s)| < \epsilon$.

1. Start with an initial cost vector J corresponding to a proper policy.

2. For each state s in S do,

$$J'(s) = \min_{a \in A(s)} \Big[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J(s') \Big].$$

3. Test convergence: if $|J' - J| < \epsilon$ go to step 4; else $J = J'$ and go to step 2.

4. Solution found: Extract corresponding policy:

$$\pi(s) = \arg \min_{a \in A(s)} \Big[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J(s') \Big], s \in S.$$

2.2.2 Policy iteration

Policy iteration (PI) [6] is another dynamic programming algorithm for solving the SSP problem evaluates. PI starts with an arbitrary policy π, computes its corresponding value function using Equation 2.2. Next the algorithm finds an improvement π' of π. The procedure is repeated until convergence, $i.e.\ \pi = \pi'$. In practice however, we look for an ϵ-optimal policy which corresponds to an ϵ-optimal value function.

PI has the following properties [9]:

1. A stationary policy π is optimal if and only if $J^*_\pi = J^*$.
2. The policy iteration algorithm converges to an optimal proper policy starting from an arbitrary proper policy.

The algorithm is presented in Table 2.2.

Table 2.2

Policy iteration

Input: An SSP problem $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{T}, c, t >$, initial proper policy π
Output: An optimal policy π.

1. Policy Evaluation: Find the value function J^π given π.

2. Policy Improvement: Construct a better policy $\pi^{'}$ such that for each state $s \in S$.

$$\pi'(s) = arg \min_{a \in A(s)} \Big[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) J^\pi(s') \Big].$$

3. Convergence Test : $\pi = \pi'$ go to step 4; Otherwise let $\pi = \pi'$ and go to step 2.

4. Return π.

2.3 Discounted infinite-horizon MDP and error bounds

This section describes the reduction of a discounted infinite-horizon MDP into an SSP problem as well as the associated error bounds.

2.3.1 Discounted infinite-horizon MDP

An SSP problem is an undiscounted infinite horizon MDP (discount factor $\beta = 1$). Any discounted infinite-horizon MDP ($\beta < 1$) can be converted to an equivalent stochastic shortest path problem [8]. An artificial terminal state t is created and from any non-terminal state i, a transition to t with probability $(1 - \beta)$ is added. The remaining transition probabilities are normalized by multiplication by β. Figure 2.1 illustrates this operation for $\beta = 0.9$.

The Bellman's equation of the resulting SSP problem is the same as the one of the original β-discounted problem and has all policies proper [8].

The Bellman update expression for iterations n and $n + 1$ is given by:

$$J_{n+1}(s) = \min_{a \in \mathcal{A}(s)} \Big[c(s,a) + \beta \sum_{s' \in S} p_{s,s'}(a) J_n(s') \Big], s \in S. \tag{2.5}$$

2.3.2 Error bounds

We first consider a discounted infinite-horizon MDP ($\beta < 1$). In the value iteration algorithm, we can associate with the optimal value $J^*(s)$ of each state $s \in S$ upper and lower bounds. Traditional bounds due to McQueen [26] are formulated as follows:

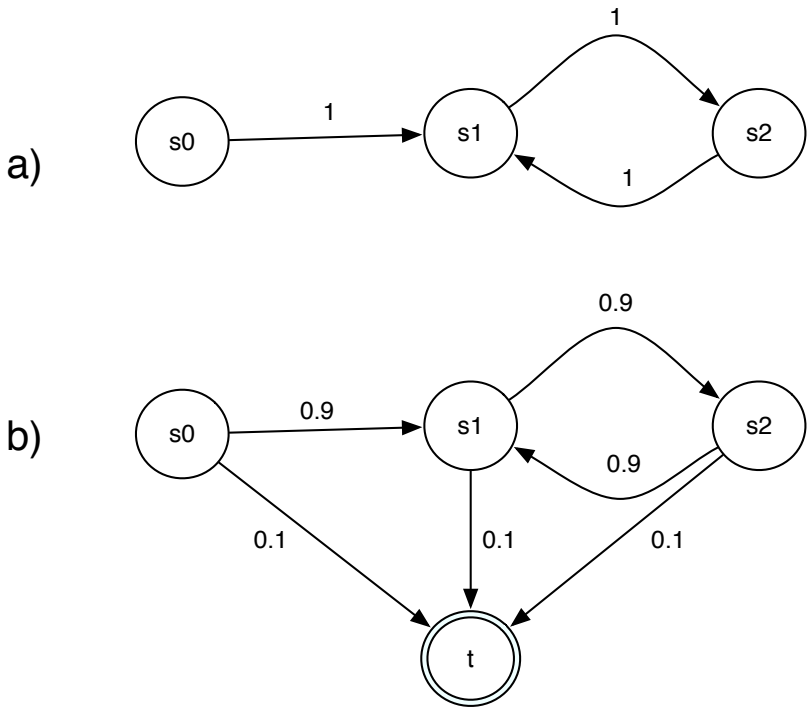

Figure 2.1

Reduction of a discounted MDP (a) to an equivalent SSP problem (b).

Let $\underline{c}_n$ and $\bar{c}_n$ be the following expressions:

$$\underline{c}_n = \min_{s \in \mathcal{S}} (J_{n+1}(s) - J_n(s)). \tag{2.6}$$

$$\bar{c}_n = \max_{s \in \mathcal{S}} (J_{n+1}(s) - J_n(s)). \tag{2.7}$$

After the nth iteration of VI algorithm, the following bounds hold for the optimal value function J^*:

$$J_n(s) + (\frac{\beta}{1-\beta})\underline{c}_n \leq J^*(s) \leq J_n(s) + (\frac{\beta}{1-\beta})\bar{c}_n, s \in S. \tag{2.8}$$

The upper and lower bounds converge monotonically [26]. The expression $\frac{\beta}{1-\beta}$ is the average number of transitions until termination [18].

2.4 Error bounds for the SSP problem with positive action costs

We now consider an SSP problem without a discount factor ($\beta = 1$). Since $\beta = 1$, $\frac{\beta}{1-\beta}$ is no longer well-defined. In this case, it is not obvious how to compute the average number of transitions to terminate as it depends on the policy. In addition, the traditional bounds are valid only if all policies are proper. Under specific conditions however, error bounds can be computed for the SSP problem even if the discount factor = 1 and not all policies are proper. These new bounds generalize McQueen's bounds. We start with some definitions:

Definition 1 (Monotone pessimistic function)

A value function (J_n) is said to be monotone pessimistic if $TJ_n(s) \leq J_n(s), \forall\ s \in S$.

The following theorem, due to Hansen [18], provides the first bounds in the undiscounted case, provided in addition that all action costs are positive.

Theorem 1 (Error bounds when $\beta = 1$)

If all action costs are positive with minimum cost $\underline{g}$, then for monotone pessimistic value functions $J_n(s)$ and $J_{n+1}(s)$, we have the following error bounds:

$$J_n(s) + (\bar{N}(s) - 1)\underline{c}_n \leq J^*(s) \leq J_n(s), \forall\, s \in S, \tag{2.9}$$

where $\bar{N}(s) = \dfrac{J_n(s)}{\underline{g}}$ is an upper bound on the average number of steps until termination for state s for any policy π for which $J^{\pi}(s) \leq J_n(s)$.

An upper bound is, in this case, the current value function. The upper and lower bounds are improved each iteration until they converge to the optimal solution.

To compute the bounds from [18], we need to compute upper bounds on the number of stages until termination $\bar{N}$(s), $\forall\ s \in S$. The value function of such initial policy is monotone pessimistic [18]. We also need the problem to have at least one proper policy. An SSP problem has by definition at least one proper policy. Once we find an initial proper policy π_0, it is evaluated, the corresponding value function J^{π_0} is an upper bound (monotone pessimistic) on the optimal value function. Value iteration is initialized using J^{π_0} and the subsequent value functions are also monotone pessimistic. The tightness of the bounds will depend on how good $\bar{N}$, the upper bound on the number of steps to terminate, is. The next section describes a way to improve the error bounds .

2.5 Improved error bounds

In this section, we explain how to use sequential space decomposition to get tighter error bounds.

2.5.1 Sequential space decomposition

A strongly connected component (SCC) of a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, is the maximum set of vertices $W \subseteq V$ such that for every pair of vertices $(u, v) \in W$, there exists a path from u to v and vice-versa. The graph G^{SCC}, obtained by contracting the states in every SCC of the original graph G into one state, is an acyclic graph and defines a topological order of G. Figure 2.2 shows an example of an SSP problem and its SCCs.

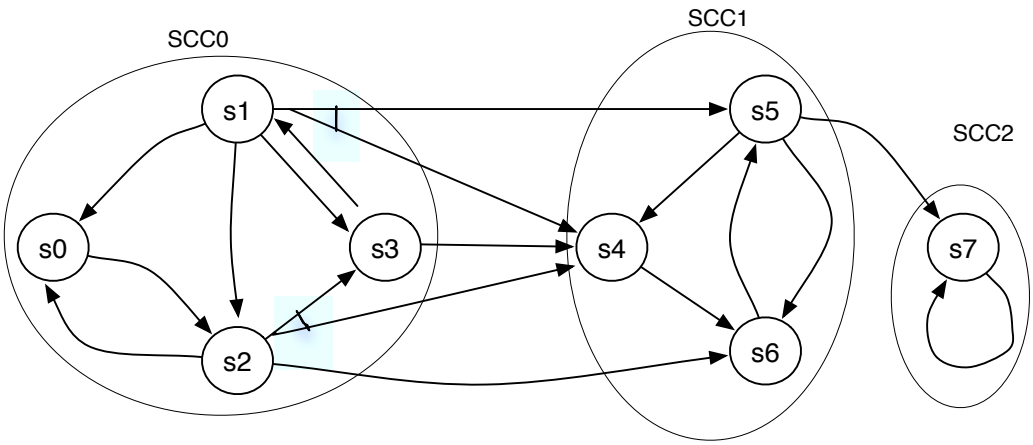

Figure 2.2

Example of sequential space decomposition.

G^{SCC} can be easily computed by depth-first traversal of G. Tarjan [27] and Kosaraju [13] algorithms are typically used to compute SCCs.

Several techniques have been developed to speedup algorithms to solve the SSP problem [14, 16, 21, 24] . These techniques take advantage of the graphical structure of the SSP problem to efficiently backup states. Topological Value Iteration (TVI) algorithm [14]

uses the topological order of the SSP problem to methodically perform backups: The SSP problem is decomposed into SCCs and each SCC is solved separately. The SCCs graph (obtained by aggregating states of each SCC into one meta-state) is acyclic and there exist an optimal backup order of the states of the SCCs graph [8]. TVI algorithm is presented in Table 2.3.

Table 2.3

Topological value iteration

Input: An MDP $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{T}, c >$, parameter ϵ: the threshold value.
Output: An ϵ-optimal value function J.

1. $\mathcal{SCC}(\mathcal{M})$
2. **For** scc in $SCCSet$ **do**
   VI(scc, ϵ)

Function $\mathcal{SCC}(\mathcal{M})$:
Input: An MDP $\mathcal{M} =< \mathcal{S}, \mathcal{A}, \mathcal{T}, c >$
$SCCSet =$ **Tarjan**$(\mathcal{M})$
return $SCCSet$.

2.5.2 Bounds and offset

The basic error bounds assume that, in order to compute an upper bound on the number of stages until termination $\bar{N}$, all actions have positive cost. It is in fact still possible to compute $\bar{N}$ if all non-terminal actions (actions not leading to a goal state) have positive costs [18]. This comes from the fact that the cost of terminal actions can be changed by a constant without changing the optimal policy [18]. The idea here is to change the cost of

terminal actions to improve the value of $\bar{N}$ which in return will improve the bounds. The following lemma [18] gives the new values of $\bar{N}$(s) $\forall\, s \in S$.

Lemma 1 (Error bounds with offset)

Let $\underline{g}$ be the smallest expected cost of any nonterminal action. If $\underline{g} > 0$, then for any monotone pessimistic value functions J, an upper bound on the mean number of stages until termination beginning from state s is given by:

$$\bar{N}(s) = \frac{\bar{J}(s) + \textit{offset}}{\underline{g}}, \tag{2.10}$$

where offset is the quantity by which the terminal costs need to be increased or decreased to make them equal to $\underline{g}$.

In order to compute the new bounds we first need to compute $\underline{g}$ the smallest cost of all non-terminal actions and the *offset*. In SCC based value iteration we start with the last component and move backward towards the start state. We also have to take into account the offset now, such that the new terminal action cost is reduced to $\underline{g}$. The way to compute a proper policy now, is to consider for each SCC, the target as the following SCC (towards the goal).

By decomposing the problem into SCCs and solving every SCC separately with specific bounds, the distance to reach the goal state (within every SCC) becomes smaller, thus we get a better initial value function. The goal here is to divide the problem into smaller problems and compute the error bounds for every subproblem instead of directly computing them on the initial problem.

2.6 Initial proper policy

This section presents two approaches to compute a monotone pessimistic value function for initializing value iteration.

2.6.1 Uniform proper policy

An initial proper policy can be obtained by choosing, for each state s, an action at random from the set of possible actions in s based on a uniform probability distribution. For an SSP problem, such policy is proper [18]. The policy is evaluated to compute the corresponding value function.

2.6.2 Proper policy using backward search from the goal

We perform backwards traversal of all nodes in the graph beginning from the goal. Initially label goal as proper, for each state s, if there is some action such that at least one successor state is already labeled proper, then we choose that action for state s and label s as proper.

At the end of the graph traversal, either all states are labeled proper (in which case the policy is proper) or at least one state is unlabeled. If there is a proper policy, it can be evaluated using a policy evaluation algorithm to get the initial value function.

This method has the advantage of checking if there is a proper policy and computes one if all states can reach the goal with probability 1.

2.6.3 Improving the initial policy

Essentially, the error bounds can be very loose at the beginning essentially due to the starting value function. So far, in order to compute an initial proper policy for a given state s, the first action in the set of possible actions leading to a successor state (that can reach the goal) is chosen without considering the probability distribution. It is not obvious how to efficiently select a good initial proper policy.

2.7 Use of error bounds

In this section, two cases where error bounds can be used are presented.

2.7.1 Testing convergence

An important parameter when using iterative algorithms, such as value and policy iterations, is a stopping criterion. In practice a state s has converged when the difference between its values from two consecutive iterations is smaller than a user-defined threshold ϵ. The algorithm has converged when all the states have converged. This threshold is arbitrary fixed and does not make much sense specially for the undiscounted problems.

We now have bounds developed in [18]. They are state-dependent and can be used to derive a test for convergence for an ϵ-optimal value function if specific conditions are met:

Theorem 2 (ϵ-optimal convergence test)

If all action costs are positive, then for monotone pessimistic value functions J and TJ and $\epsilon > 0$, TJ is ϵ-optimal if

$$-(\max_{s \in \mathcal{S}}(\bar{N}(s) - 1))\underline{c}_s \leq \epsilon, \tag{2.11}$$

where $\underline{c}_s = \min_{s \in \mathcal{S}}(TJ(s) - J(s))$ *and* $\bar{N}(s)$ *is the upper bounds on the number of transitions to terminate for state* s.

2.7.2 Action elimination

As stated previously, value iteration evaluates all the states and actions, which makes it very slow for large problems. It can be inefficient as well, since not all the states are necessarily part of the optimal policy. Many actions end up being sub-optimal and will not be part of an optimal policy. They can therefore be eliminated. Focused Topological Value Iteration (FTVI) [14] for example, combines TVI and action elimination to eliminate sub-optimal actions which results in more SCCs in the graph. FTVI uses upper and lower bounds to eliminate sub-optimal actions.

Bounds on the optimal value function can be used to detect suboptimal actions and permanently eliminate them and not be considered for future iterations. Given the upper and lower bounds defined in Section 3.2, the following test [25] detects sub-optimal actions to immediately eliminate them:

Theorem 3 (Action suboptimality test)

Let $L(s)$ *and* $U(s)$ *be respectively lower and upper bound of the optimal value function* $J^*(s)$ *for state* s *when action* a *is considered. Action* a *is sub-optimal if:*

$$c(s,a) + \sum_{s' \in S} p_{s,s'}(a) L(s') > \min_{a \in \mathcal{A}(s)} \Big[c(s,a) + \sum_{s' \in S} p_{s,s'}(a) U(s') \Big], s \in S. \tag{2.12}$$

For the bounds previously defined, $L(s)$ and $U(s)$ are given by:

$$U(s) = J_n(s), \forall\, s \in S. \tag{2.13}$$

$$L(s) = J_n(s) + (\bar{N}(s) - 1)\underline{c}_n, \forall\, s \in S. \tag{2.14}$$

A modified value iteration algorithm using upper and lower bounds on the optimal solution to eliminate actions has been successfully implemented [17, 25, 26]. Action elimination is implemented using the new error bounds and is compared to value iteration.

CHAPTER 3

APPLICATION TO DECISION-THEORETIC PLANNING

This chapter describes the experimental results of applying the error bounds to decision-theoretic planning problems.

3.1 Test problems

We consider three test problems, each having two instances. For all problems the goal can be reached with probability = 1 beginning from any other state, that is, a proper policy exists. Problems description follows next and Table 3.1 shows the size of the test problems.

3.1.1 Race track problem

The Race track problem simulates automobile racing. "A race track of any shape is drawn on graph paper, with a starting line at one end and a finish line at the other consisting of designated squares. Each square within the boundary of the track is a possible location of the car" [4] (See Figures 3.1 and 3.2 for two examples). "The car is placed on the starting line at a random position, and moves 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

difference between v' and v is -1, 0, or 1. This means that the car can maintain its speed" [4]. If the car hits the track boundary, it is returned to a random position on the starting line, its velocity is reduced to zero (that is, h' - h and v' - v are considered to be zero), and it restarts again. The objective is to control the car so that it crosses the finish line in as few moves as possible. All actions have unit cost, which means the objective is to minimize the time required to reach the finish line from the starting line.

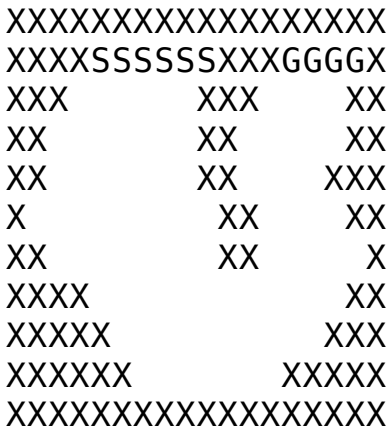

Figure 3.1

RaceTrack1: 'S' and 'G' correspond to start and goal states respectively.

3.1.2 Single-arm pendulum

The single-arm pendulum (SAP) is a two-dimensional minimum-time optimal-control problem [28]. The agent has two actions available representing positive and negative torques applied to a rotating pendulum. The agent cannot move the pendulum from the bottom to the top directly, but must rock it back and forth until it has sufficient velocity

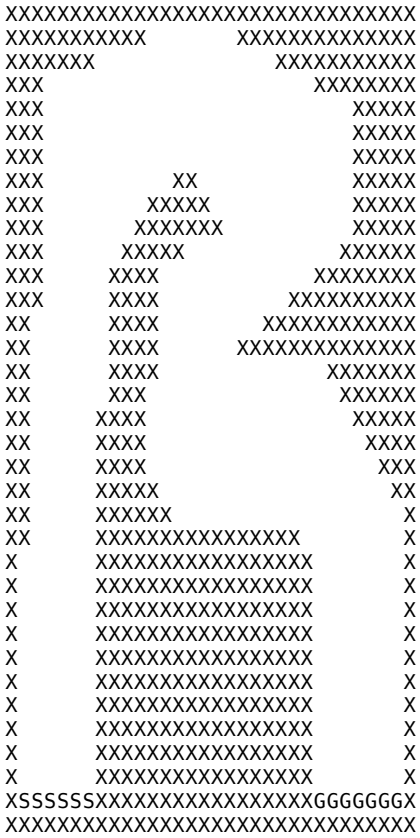

Figure 3.2

RaceTrack2.

[28]. The state space is defined by the angle of the link (θ_1) and the angular velocity of the link ($\dot{\theta_1}$). The state space is discretized. The objective is to balance the pendulum vertically at a zero degree angle and zero velocity (goal state). All actions incur unit cost.

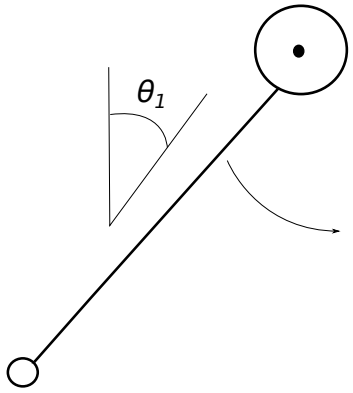

Figure 3.3

Single-arm pendulum.

3.1.3 Double-arm pendulum

The double-arm pendulum (DAP) [28] is a four-dimensional ($\theta_1, \theta_2, \dot{\theta_1}$ and $\dot{\theta_2}$) minimum-time optimal-control problem. It is similar to SAP, except that there are two arms instead of one. It is the smaller link that must be balanced vertically. It is a free-swinging link. The state space is defined by the two arms angles (θ_1, θ_2) and their angular velocities $\dot{\theta_1} \in [-10, 10]$ radians/s and $\dot{\theta_2} \in [-15, 15]$ radians/s. The agent cannot move the pendulum from the bottom to the top directly, but must swing it back and forth to generate sufficient momentum [28]. The objective is to position the pendulum vertically at zero degree angles and zero velocity. Again, actions have unit cost.

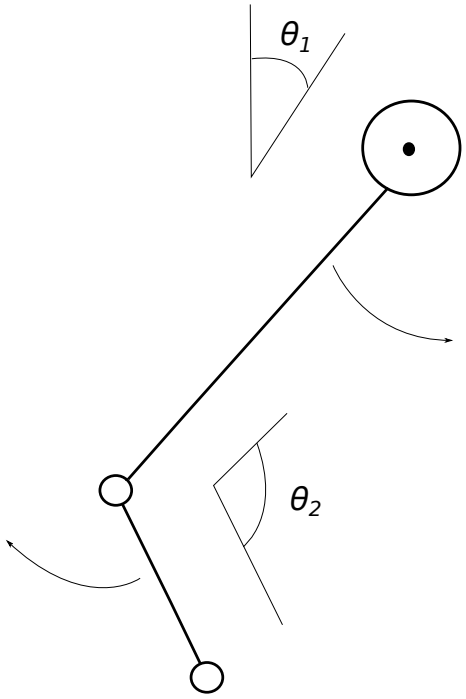

Figure 3.4

Double-arm pendulum.

Table 3.1

Test problems

Problem	Number of states	Number of actions
RaceTrack1	1,849	16,641
RaceTrack2	21,371	192,339
SAP100	10,000	20,000
SAP300	90,000	180,000
DAP10	10,000	20,000
DAP20	160,000	320,000

Experiments All test problems have unit action cost; thus $\underline{g} = 1$, moreover, there is at least one proper policy for each problem. An initial proper policy is first computed, and the value iteration algorithm is initialized using the value function obtained from the evaluation of the initial proper policy. Table 3.2 presents the starting and the optimal values of the start state for the test problems. In each iteration, error bounds are computed and convergence is tested using Theorem 2.

Table 3.2

Initial($J_0(s_0)$) and final($J^*(s_0)$) values of the start state

	RaceTrack1		RaceTrack2		SAP100		SAP300		DAP10		DAP20	
	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$
	179.15	9.16	1321.91	31.37	150.09	140.88	388.99	369.64	26.36	19.31	50.19	35.86

Tables 4.3, 4.4 and 4.5 present for each problem, the residual = $-\underline{c}_s$ and the new error bound = $-(\max_{s \in \mathcal{S}}(\bar{N}(s)-1))\underline{c}_s$ values for the start state, for each iteration until convergence. The algorithms are considered to have converged when the error bound is less than $\epsilon = 10^{-6}$.

The results presented in Tables 4.3, 4.4 and 4.5 show that the error bounds can be used to test for convergence to an ϵ-optimal solution. The bigger the problem, the longer it takes to converge. The errors are quite poor at beginning but get better as we run more iterations. The better the initial policy, the better the initial bounds and the faster the algorithm converges. As presented in Table 3.2, the initial value (for the start state) is often very high given the final value.

Table 3.3

Error bounds (Race Track)

iteration	RaceTrack1			RaceTrack2		
	res.	$\bar{N}_{max}$	error	res.	$\bar{N}_{max}$	error
0	186.505864	50.369	33837.783	1320.506367	400.278	1744874.6
4	6.523357	9.201	70.392718	52.302179	34.484	2818.9315
9	0.053002	10.503	0.538530	3.806509	31.410	124.58362
10	0.014861	9.160	0.150993	2.576267	31.392	84.089260
11	0.003179	9.201	0.032298	1.042913	31.384	34.009237
12	0.000612	9.160	0.006218	0.642675	31.381	20.949511
13	0.000112	9.160	0.001140	0.526696	31.380	17.166135
14	0.000020	9.160	0.000204	0.332708	31.379	10.842908
15	0.000004	9.160	0.000036	0.125295	31.379	4.083227
16	0.000001	9.160	0.000006	0.062574	31.379	2.039186
20	-	-	-	0.003190	31.379	0.103960
24	-	-	-	0.000225	31.379	0.007335
28	-	-	-	0.000008	31.379	0.000263
29	-	-	-	0.000004	31.379	0.000114
30	-	-	-	0.000002	31.379	0.000050
31	-	-	-	0.000001	31.379	0.000022

Table 3.4

Error bounds (Single-arm pendulum)

	SAP100			SAP300		
iteration	res.	N_{max}	error	res.	N_{max}	error
0	120.803491	149.0968	36036.3474	352.2018	387.9924	281593.11
19	17.759731	149.0968	4551.269944	38.2262	387.9924	29518.266
59	0.791498	149.0856	117.777815	24.9297	387.9924	17029.916
99	0.461258	148.9390	68.477939	16.6732	387.9924	9731.1416
139	0.299921	142.1510	42.636557	12.4160	387.9924	5258.9819
179	0.070637	141.4356	9.933969	1.064990	387.9924	418.88727
219	0.001882	140.8911	0.264038	0.838617	387.9924	329.84919
259	0.000020	140.8898	0.002830	0.737511	387.9924	290.08160
279	0.000002	140.8898	0.000338	0.697940	387.9924	274.51745
285	0.000001	140.8898	0.000133	0.685916	387.9924	269.78774
286	-	-	-	0.683466	387.9924	268.82411
316	-	-	-	0.620663	382.8564	244.12233
376	-	-	-	0.449282	370.4367	173.30112
436	-	-	-	0.076815	370.1065	29.276448
496	-	-	-	0.005737	369.7025	2.186421
556	-	-	-	0.000029	369.6475	0.011157
577	-	-	-	0.000002	369.6474	0.000925
584	-	-	-	0.000001	369.6474	0.000379

Table 3.5

Error bounds (Double-arm pendulum)

	DAP10			DAP20		
iteration	res.	N_{max}	error	res.	N_{max}	error
0	9.754700	25.066726	365.07479	19.343652	48.968765	1549.4707
5	3.130976	22.749747	111.26919	7.415049	46.879543	585.37200
10	1.267190	21.163182	36.969128	5.380304	43.536117	416.96231
15	0.568972	20.058999	15.350997	3.800557	41.513830	279.33796
20	0.256870	19.577365	6.538414	2.575460	39.806039	165.69315
25	0.118588	19.398144	2.933673	1.082850	38.045388	65.554680
30	0.041992	19.339087	1.027285	0.911191	36.902693	52.006848
35	0.014141	19.321316	0.344643	0.630935	36.319243	34.599122
40	0.004225	19.316241	0.102847	0.410741	36.051993	21.519977
45	0.001217	19.314866	0.029618	0.241949	35.938940	12.309922
50	0.000330	19.314504	0.008030	0.128658	35.893086	6.437655
55	0.000087	19.314412	0.002120	0.059487	35.874810	2.959066
60	0.000022	19.314388	0.000543	0.025545	35.867762	1.267723
65	0.000006	19.314383	0.000136	0.010446	35.865188	0.517940
72	0.000001	19.314381	0.000019	0.002770	35.864112	0.137310
73	-	-	-	0.002276	35.864051	0.112817
84	-	-	-	0.000236	35.863824	0.011707
94	-	-	-	0.000028	35.863805	0.001393
104	-	-	-	0.000003	35.863803	0.000151
109	-	-	-	0.000001	35.863803	0.000049

Action elimination is also tested on these problems. Sub-optimal actions are not considered in the state backups. Suboptimality is tested using Theorem 3 of Chapter 3. Results are presented in Table 3.6. In some cases, action elimination does not improve the runtime. This is due to the fact that every iteration, two backups per state are performed to compute state actions lower bounds. Furthermore, due to the fact that the state lower bound is quite poor at the beginning, only few actions are eliminated. Most of the actions are eliminated within a small interval towards the "middle" of the process as shown by Figures 3.5, 3.6 and 3.7. A slightly faster algorithm is obtained by switching to regular value iteration at the "beginning" and at the "end" of the process.

Table 3.6

Value iteration and Action elimination on planning problems

Problem	Value iteration(sec.)	Action elimination(sec.)	% of sub-optimal actions
RaceTrack1	0.01	0.01	59
RaceTrack2	0.19	0.19	59
SAP100	0.28	0.29	12
SAP300	6.20	6.14	6
DAP10	0.12	0.12	49
DAP20	3.90	3.86	49

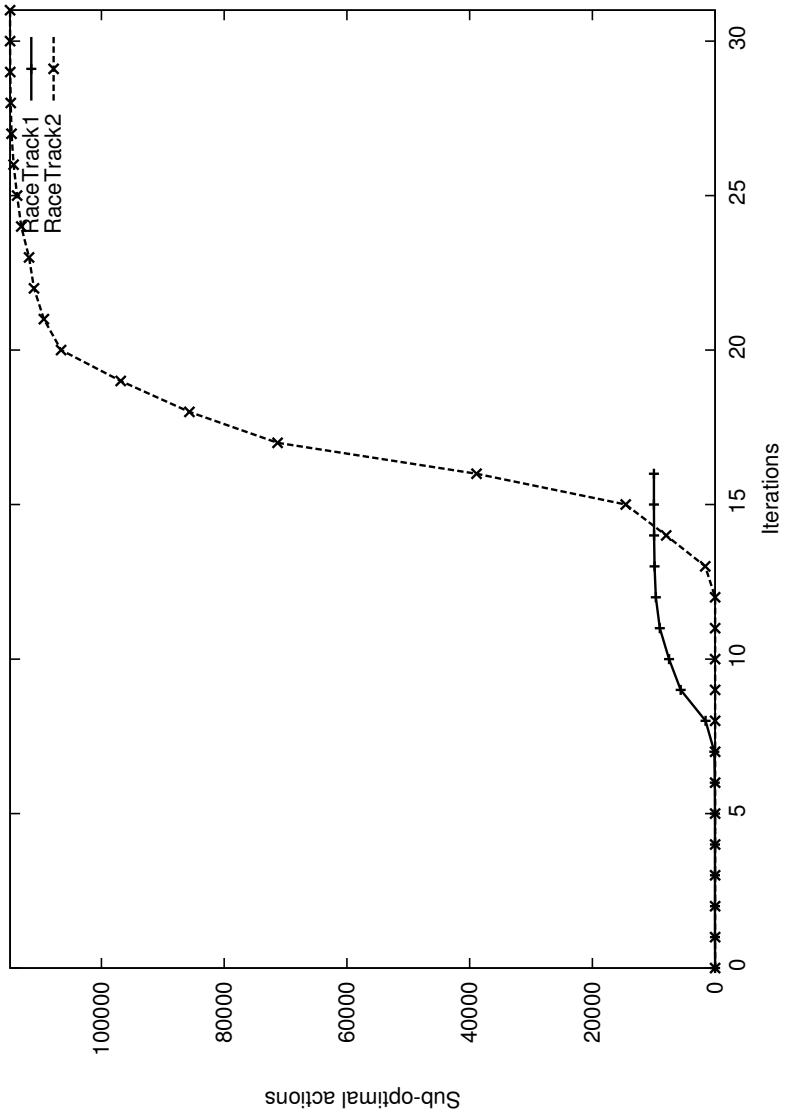

Figure 3.5

Number of sub-optimal actions: RaceTrack

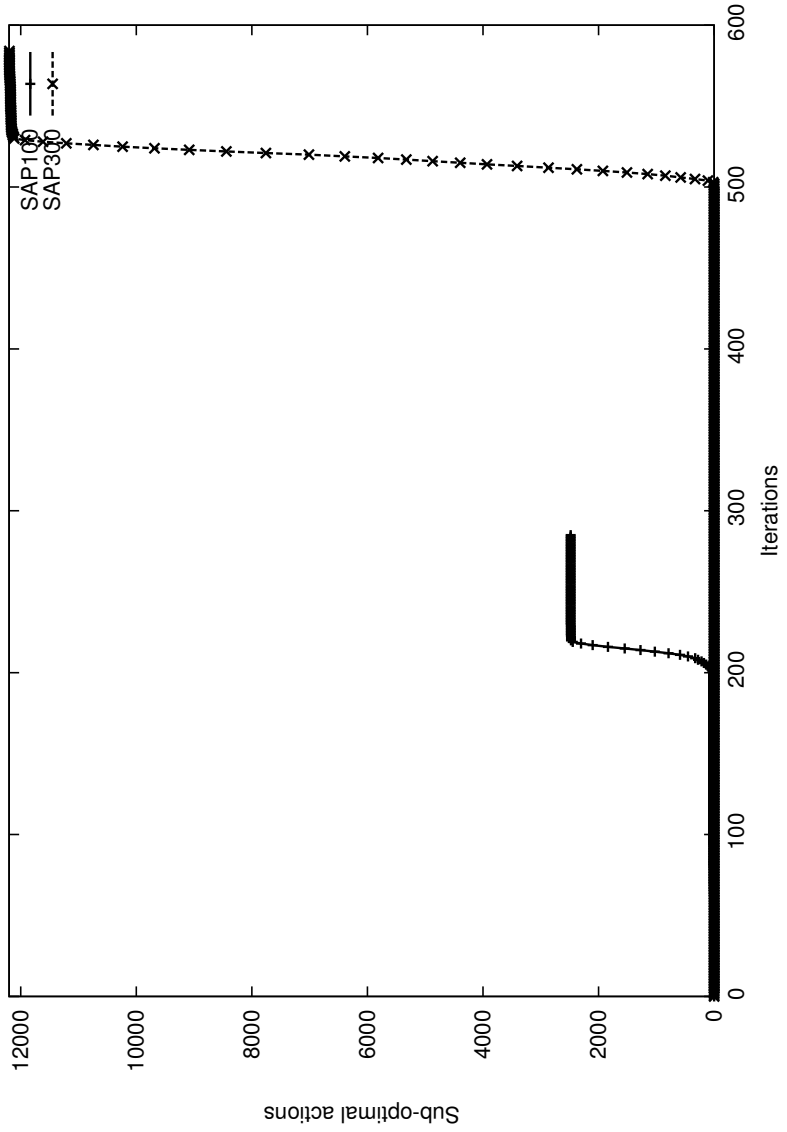

Figure 3.6

Number of sub-optimal actions: SAP

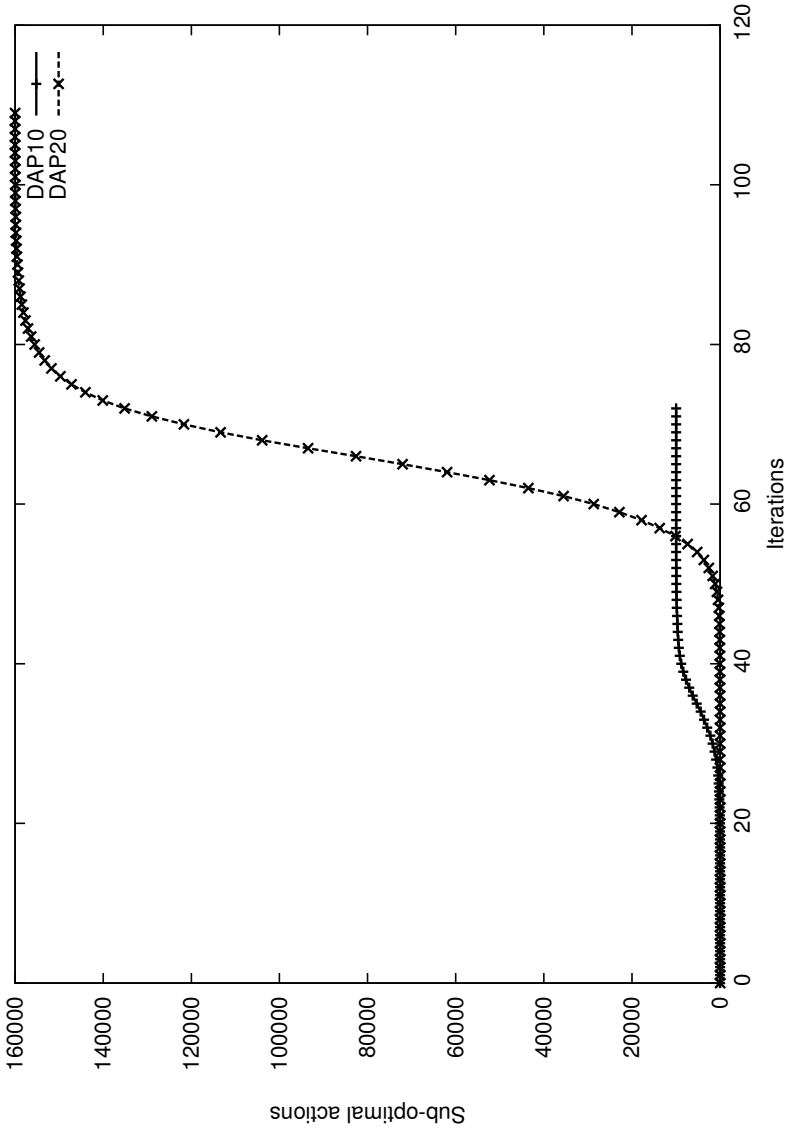

Figure 3.7

Number of sub-optimal actions: DAP

CHAPTER 4

APPLICATION TO PROBABILISTIC MODEL CHECKING

In this chapter we apply the new bounds to the minimum expected time problem in probabilistic model checking. The experimental results for both basic and SCC-based error bounds are presented.

4.1 Probabilistic model checking

Model checking is a formal verification technique used to exhaustively analyze finite-state systems [3]. A model which represents all possible execution paths of the system over time is constructed as a finite state automaton. Desired properties of the system such as "the system never ends up in an error state" or "the system always reaches a given state s_g within t time steps" are expressed in temporal logic and the model is explored to check whether or not the properties always hold. If a property is not satisfied, an error trace is generated. Figure 4.1 illustrates the principle of model checking.

It is not always possible to have absolute guarantee of correctness of a given system. There often exists some positive probability of failure that is not possible to eliminate and has to be considered in the analysis. The uncertainty often comes from the unpredictability of the system over time or from the difficulty of having a deterministic specification of the system. It becomes unrealistic to express correctness as: "the system cannot fail". Instead,

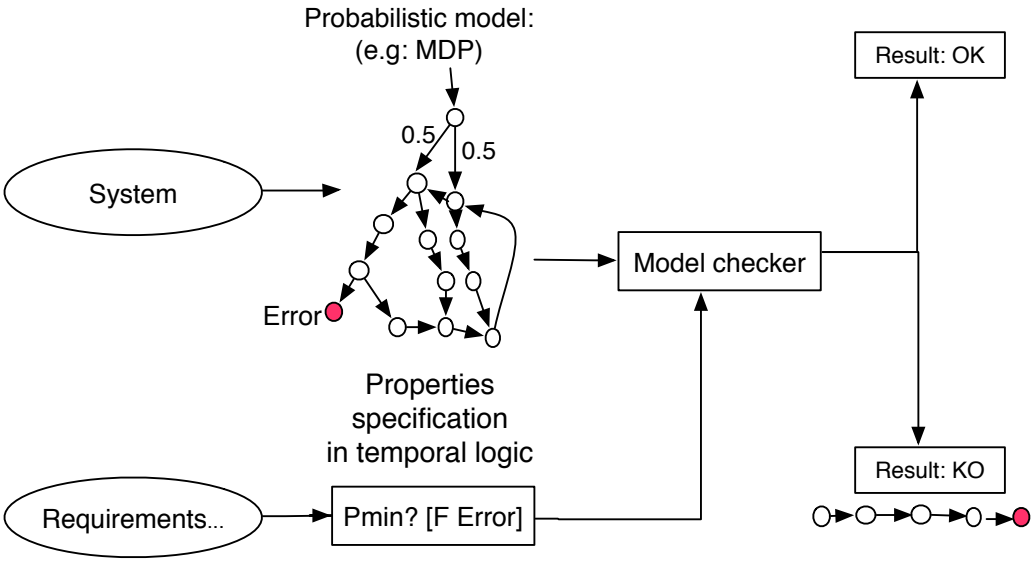

Figure 4.1

Principle of model checking.

it is expressed as: “with 99% chance the system will not fail”. Another reason to consider probabilistic behavior is when analyzing randomized algorithms that are intrinsically probabilistic. For example, distributed algorithms often include a randomization step to break “symmetry” between processes [3].

Probabilistic model checking analyzes systems with probabilistic behavior. Probabilistic finite-state systems can be modeled in a number of different ways. Markov chains model systems with a probability distribution over successor states; the next state is chosen probabilistically. Markov decision processes are used in randomized distributed systems to add nondeterminism in actions to Markov chains [16].

Probabilistic model checking considers two types of properties: qualitative and quantitative. Qualitative properties apply to events that happen with probability 0 or 1. For

example: "every philosopher will eventually get to eat (with probability 1)" or "the program will almost never be in a deadlock state (with probability 0)". These properties are verified by performing reachability analysis on the model. Depth-first search is the baseline algorithm used to verify qualitative properties.

The quantitative properties apply to the cost or probability of an event. For example: "what is the minimum expected time steps t to guarantee that all philosophers get to eat" or "with probability at least 0.99, a leader is elected within t time steps". Dynamic programming algorithms are used in quantitative analysis.

In the remainder of this chapter, we only consider systems modeled as Markov decision processes.

4.1.1 States with proper policy

In order to perform quantitative analysis on a system for the minimum expected time, we need to divide the state space into three sets: states that are guaranteed to reach the goal state with probability = 1, states that will never reach the goal state and states in between. Quantitative analysis are only performed on the latter set of states.

States S^1_{max}, for which the maximum probability of reaching the target state is 1 are computed using algorithm Prob1 [15]. Prob1 is described in Table 4.1.

Prob1's search space is initialized with the whole state space S, it then does a backward search from the target set to identify and remove trap states (states that will never reach the target set) and actions leading to trap states from the search space. It iterates this step until convergence. The algorithm runs in $\mathcal{O}(nm)$ time, where n and m are number of states

and edges in the model respectively. This often makes it the slowest part in the process of verifying a given property of the system [16].

Improved versions of Prob1 algorithm have been developed [10, 11]. These versions take advantage of the properties of the graphical structure of the model.

Table 4.1

Prob1 algorithm

Input: MDP $\mathcal{M} =< S, A, T, c >$, target set $t \in S$.
Output: $S^1_{max} = \{s \in S \mid Pr^{max}_s(reach(t)) = 1\}$.

```
1    R := S
2    do
3       R' := R
4       R := t
5       do
6          R'' := R
7          R := R'' ∪ {s ∈ S | ∃a ∈ A(s).
8          (∀s' ∈ S. (p_{s,s'}(a) > 0 → s' ∈ R')) ∧ (∃s' ∈ R''. p_{s,s'}(a) > 0)}
9       while (R ≠ R'')
10   while (R ≠ R')
11   return R.
```

Computing minimum expected time is a quantitative analysis. First, the set of states for which there is at least one path that leads to the target set is computed. These are the states with maximum probability 1 of reaching the target set. States that do not have any proper policy are not considered as they have infinite cost. This precomputation step corresponds to running the Prob1 algorithm. Value iteration can then be used to compute the expected cost, in term of time steps, for the states returned by Prob1 algorithm. The slowest part in

solving this problem is usually the Prob1 algorithm as it is quadratic in the worst case [16]. There is not trivial way of avoiding this step for the minimum expected time.

4.1.2 Minimum expected time

For problems where a proper policy exists, a probabilistic model checker is often concerned with computing the minimum expected time to reach a target state. This problem is equivalent to the stochastic shortest path problem for Markov decision processes and thus it can be solved using value iteration.

4.2 Test problems

The problems described in this section are all from the Prism model checker benchmark. All problems can be decomposed into several SCCs. Table 4.2 summarizes their properties.

4.2.1 Zeroconf(N, K)

ZeroConf is a dynamic configuration protocol for IPv4 link-local addresses [12]. The protocol describes how to configure an IP address for a new device in the network. When a new device joins the network, it first randomly selects an IP address from 65,024 possible addresses. It then waits between 0 and 2 seconds before it broadcasts four Address Resolution Protocol (ARP) packets to request the use of the selected IP address. Four ARP packets are sent to handle message loss. If the address is already used, the device using it will reply with an ARP reply packet claiming the use of the address. The new device chooses a new address and sends new ARP packets and the process is repeated. Every time

there is an address conflict, a counter C is incremented. If C = 10, the device idles for at least one minute. If the device gets no reply after sending four ARP packets, it starts using the chosen address and broadcasts two confirmation ARP messages to the network. The confirmation messages are sent at 2 second intervals. The parameter N is the size of the network before the new device joins in. The probability that the new device picks an available address is $\frac{65024 - N}{65024}$. The parameter K is the maximum number of ARP packets sent by the device; up to four packets are sent. The probability of message loss is fixed to 0.1. A detailed description of this protocol model can be found in [23]. The goal state is having the new device correctly configured. We are interested in computing the expected cost in term of time steps of reaching the goal state for different values of N and K.

4.2.2 Consensus(N, K)

The randomized consensus protocol is a distributed algorithm used to reach agreement between N asynchronous processes that communicate using read/write shared registers [2]. The processes go through as many rounds as necessary to reach agreement. Each agreement attempt involves reading the status of all other processes. If the processes disagree, a shared coin-flipping is used to decide their next preferred values. The parameter $K > 1$ is used to set a barrier on the value of a globally shared counter of the number of coin flips depending on which, a process chooses its preferred value. The algorithm terminates when all processes have the same value. The property checked is the minimum expected number of steps (time units) it takes to reach agreement.

4.2.3 Israeli and Jalfon Self-Stabilizing Protocol(N)

Consider a network of N processors organized into a ring. The network is said to be in a stable configuration when only one processor has a token. Israeli and Jalfon's self-stabilization protocol (ij) [19] when applied on a ring of N processes, always reaches a stable configuration from any initial configuration without any outside intervention within a finite number of steps. Each process in the ring has a local boolean variable q_i indicating that the token is in place i. Processes with a token are said to be active. Neighboring processes communicate using shared read/write registers. Active processes are managed by a scheduler. A scheduled process randomly passes its token to one of its neighbors. We are interested in computing the minimum expected time to reach a stable configuration.

Table 4.2

Properties of state space for test problems.

Problem	# of states	# of actions	# of transitions	# of trivial SCCs	# of n-t SCCs
Zeroconf(10, 1)	31,954	57,482	73,318	11,180	1
Zeroconf(10, 2)	89,586	164,169	207,825	29,914	1
Consensus(4, 2)	22,656	60,544	75,232	2,546	65
Consensus(4, 4)	43,136	115,840	144,352	2,546	65
ij(10)	1,023	5,120	8,960	1	9
ij(15)	32,767	245,760	430,080	1	14

4.3 Experiments

The minimum expected time problem can be modeled as a stochastic shortest path problem. A cost of 1 is associated with each action; this makes it suitable for the new error

bounds. As in the planning case, in order to use the new bounds, we have to start with an initial proper policy. It is possible to extract an initial proper policy while running the Prob1 algorithm of Table 2.3. The returned policy is then evaluated to get the corresponding initial value function. Bounds are then computed in each iteration as in the planning case.

The experiments are run using PRISM [22], a probabilistic model checker, written in C++ and Java, developed at the Universities of Birmingham and Oxford. PRISM supports several probabilistic models, including Markov decision processes, with implementations of value iteration and policy iteration. In addition to the explicit data structures used to represent the models, PRISM uses sparse matrices, binary decision diagrams (BDDs) and multi-terminal BDDs (MTBDDs). These memory-efficient data structures can handle very large models. For numerical computations, PRISM comes with three separate engines. The first engine uses explicit data structures, the second uses MTBDDs, and the third uses hybrid engine which combines MTBDDs and sparse matrices. For these experiments, only the explicit engine is used.

Properties such as the minimum expected time of reaching a set of states in the model are easily expressed in probabilistic temporal logic.

All problems described in the previous section have unit edge cost; thus $\underline{g} = 1$ and there is at least one proper policy. An initial policy is first computed and the value iteration algorithm is used. The algorithm is initialized using the value function obtained from the evaluation of the initial proper policy. Tables 5.2, 5.3 and 5.4 show the starting values (obtained from the initial value function) and the optimal values of the state with the

largest upper bound on the number of steps to terminate. Each iteration, error bounds are computed and convergence is tested using Theorem 2.

Table 4.3

Initial($J_0(s_0)$) and final($J^*(s_0)$) values of the state with the largest error bound (Zeroconf)

	Zeroconf(10, 1)		Zeroconf(10, 2)	
	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$
	27.10	25.40	30.50	28.80

Table 4.4

Initial($J_0(s_0)$) and final($J^*(s_0)$) values of the state with the largest error bound (Consensus)

	Consensus(4, 2)		Consensus(4, 4)	
	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$
	239.70	199.99	851.18	755.97

Tables 5.5, 5.6 and 5.7 show the residual, $-\underline{c}_s$, and the error bound, $-(\max_{s \in \mathcal{S}}(\bar{N}(s) - 1))\underline{c}_s$, for each test problem.

The value iteration results are presented in Tables 5.5, 5.6 and 5.7. The error bounds make it possible to test for convergence to an ϵ-optimal solution. The bigger the problem, the longer value iteration takes to converge since the values of all the states have to converge. The errors bounds are loose at beginning but get better in successive iterations. The better the initial policy, the better the initial bounds and the faster the algorithm converges.

Table 4.5

Initial($J_0(s_0)$) and final($J^*(s_0)$) values of the state with the largest error bound (Self-stabilization)

	ij(10)		ij(15)	
	$J_0(s_0)$	$J^*(s_0)$	$J_0(s_0)$	$J^*(s_0)$
	44.99	44.99	104.99	104.99

Table 4.6

Error bounds - Zeroconf (N, K)

	Zeroconf(10, 1)			Zeroconf(10, 2)		
iteration	res.	N_{max}	error	res.	N_{max}	error
1	14.12	27.10	368.53	13.48	30.50	397.66
10	1.73	25.40	42.11	2.24	28.80	62.272
20	0.64	25.40	15.42	0.65	28.80	18.070
30	0.06	25.40	1.46	0.39	28.80	10.842
40	.00006	25.40	.00146	0.01	28.80	0.2780
50	.00006	25.40	.00146	.00003	28.80	.00083
60	.00006	25.40	.00146	.00003	28.80	.00083
70	.00006	25.40	.00146	.00003	28.80	.00083
80	.00004	25.40	.00097	.00002	28.80	.00055
90	.00003	25.40	.00073	.00002	28.80	.00055
100	.00003	25.40	.00073	.00001	28.80	.00027
110	.00002	25.40	.00048	.00001	28.80	.00027
115	.000007	25.40	.00017	.00001	28.80	.00027
120	.000007	25.40	.00017	.000006	28.80	.00016
125	.000007	25.40	.00017	.000006	28.80	.00016

Table 4.7

Error bounds - Consensus (N)

iteration	Consensus(4, 2)			Consensus(4, 4)		
	res.	N_{max}	error	res.	N_{max}	error
1	25.49	239.70	6064.4	49.47	851.18	42058
100	0.91	220.34	199.59	0.95	849.77	806.33
200	0.36	208.00	74.520	0.54	840.45	453.30
300	0.15	203.33	30.349	0.50	830.14	414.57
400	0.062	201.34	12.421	0.42	820.63	344.24
500	0.025	200.54	4.988	0.35	812.70	284.09
600	0.011	200.23	2.191	0.29	806.55	233.60
700	.0046	200.09	0.915	0.24	801.13	192.03
800	.0019	200.03	0.378	0.20	796.70	159.14
900	.0008	200.01	0.159	0.16	793.20	126.75
1000	.0003	200.005	0.059	0.13	790.15	102.58
1100	.0001	200.001	0.019	0.11	787.66	86.532
1200	.00006	199.999	0.011	0.09	785.69	70.622
1300	.00002	199.998	.00397	0.07	783.97	54.807
1400	.00001	199.998	.01989	0.06	782.57	46.894
2400	.000009	199.998	.00179	0.01	776.96	7.759
3400	.000009	199.998	.00179	.0015	776.12	1.162
4400	.000009	199.998	.00179	.00024	775.99	0.185
5400	.000009	199.998	.00179	.00003	775.97	0.023
6134	.000009	199.998	.00179	.000009	775.97	0.006
12000	.000009	199.998	.00179	.000009	775.97	0.006
24000	.000009	199.998	.00179	.000009	775.97	0.006
24472	.000009	199.998	.00179	.000009	775.97	0.006

Table 4.8

Error bounds - Self-stabilization (N)

iteration	ij(10)			ij(15)		
	res.	N_{max}	error	res.	N_{max}	error
1	0.000009	44.99	0.0003	0.000009	104.99	0.0009
2	0.000000	44.99	0.0000	0.000000	104.99	0.0000

Another way of improving the bounds is using sequential space decomposition to decompose the problem into smaller stochastic shortest path problems. The problem is decomposed into strongly connected components (SCCs) which are sequentially solved starting from the goal. Every SCC is a goal for the SCCs that can immediately reach it. This makes the number of steps to terminate smaller. An upper bound on the average number of steps to finish is computed using the offset defined in Section 3.3. The best case is when the problem decomposes into equal-size strongly connected components. Instead of backing up all the states, each SCC is separately solved which means fewer states evaluated each iteration and faster convergence when SCCs are solved in a topological order.

In our test problems we have several trivial SCCs (one state per SCC) and only few non-trivial SCCs for Zeroconf and Consensus problems. When backing up states in topological order, trivial SCCs need only one iteration to converge. Table 4.9 shows the number of iterations needed by value iteration and SCC-based value iteration on the largest SCC, to converge.

Adding action elimination to value iteration often makes it slower for model checking problems. This is due to the fact that these problems have very few actions per state (2-3 actions in average) and at the end the number of backups saved from eliminating sub-optimal actions (less than 10%) is not enough to compensate the overhead created by action elimination.

Table 4.9

Value iteration vs. SCC-based value iteration

Problem	Value iteration	# SCC-based VI on the largest SCC
Zeroconf(10, 1)	113	113
Zeroconf(10, 2)	119	118
Zeroconf(15, 3)	125	105
Consensus(4, 2)	1,426	852
Consensus(4, 4)	6,134	4,884
Consensus(4, 8)	24,458	22,357
ij(10)	1	1
ij(12)	1	1
ij(15)	1	1

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis we have implemented and evaluated new error bounds for the stochastic shortest path problem both for decision-theoretic planning and probabilistic model checking. We have shown that it is possible to use these bounds in test for convergence and in action elimination. We have also shown that better error bounds can be computed by using sequential space decomposition.

One the drawbacks of the value iteration algorithm is that all the states are evaluated each iteration. These error bounds can be used in branch and bound algorithm where both suboptimal action and non-relevant states are pruned for the state space. This is a topic for future research.

REFERENCES

[1] D. Aberdeen, S. Thiébaux, and L. Zhang, "Decision-theoretic military operations planning," *Proc. ICAPS*, 2004, vol. 14, pp. 402–411.

[2] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *Journal of algorithms*, vol. 11, no. 3, 1990, pp. 441–461.

[3] C. Baier, J.-P. Katoen, et al., *Principles of model checking*, vol. 26202649, MIT press Cambridge, 2008.

[4] A. G. Barto, S. J. Bradtke, S. P. Singh, T. T. R. Yee, V. Gullapalli, and B. Pinette, "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 72, 1995, pp. 81–138.

[5] D. Beauquier, "On probabilistic timed automata," *Theoretical Computer Science*, vol. 292, no. 1, 2003, pp. 65–84.

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

[7] E. Benazera, R. Brafman, N. Meuleau, E. A. Hansen, et al., "Planning with continuous resources in stochastic domains," *International Joint Conference on Artificial Intelligence*. LAWRENCE ERLBAUM ASSOCIATES LTD, 2005, vol. 19, p. 1244.

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

[9] 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.

[10] K. Chatterjee and M. Henzinger, "Faster and Dynamic Algorithms for Maximal End-Component Decomposition and Related Graph Problems in Probabilistic Verification," *SODA*, 2011, pp. 1318–1336.

[11] K. Chatterjee and J. Lacki, "Faster Algorithms for Markov Decision Processes with Low Treewidth," *CoRR*, vol. abs/1304.0084, 2013.

[12] S. Cheshire, B. Adoba, and E. Guttman, "Dynamic configuration of IPv4 link-local addresses (draft August 2002)," *Zeroconf Working Group of the Internet Engineering Task Force (www. zeroconf. org)*, 2002.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd edition, McGraw-Hill Higher Education, 2001.

[14] P. Dai, M. Mausam, D. S. Weld, and J. Goldsmith, "Topological value iteration algorithms," *Journal of Artificial Intelligence Research*, vol. 42, no. 1, 2011, pp. 181–209.

[15] L. De Alfaro, "Computing minimum and maximum reachability times in probabilistic systems," *CONCUR99 Concurrency Theory*, 1999, pp. 781–781.

[16] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, M. Ujma, et al., *Incremental runtime verification of probabilistic systems*, Tech. Rep., Tech. Rep. RR-12-05, Department of Computer Science, University of Oxford, 2012.

[17] R. C. Grinold, "Technical Note: Elimination of Suboptimal Actions in Markov Decision Problems," *Operations Research*, vol. 21, no. 3, 1973, pp. 848–851.

[18] E. A. Hansen, "Contraction Properties of the Dynamic Programming Operator for the Stochastic Shortest Path Problem and Related Error Bounds," *Submitted for journal publication*, 2013.

[19] A. Israeli and M. Jalfon, "Token management schemes and random walks yield self-stabilizing mutual exclusion," *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 1990, PODC '90, pp. 119–131, ACM.

[20] H. E. Jensen, "Model checking probabilistic real time systems," *Proc. 7th Nordic Workshop on Programming Theory*. Citeseer, 1996, pp. 247–261.

[21] A. Kolobov, M. Mausam, D. S. Weld, and H. Geffner, "Heuristic search for generalized stochastic shortest path MDPs," *Twenty-First International Conference on Automated Planning and Scheduling*, 2011.

[22] 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.

[23] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, "Performance analysis of probabilistic timed automata using digital clocks," *Formal Methods in System Design*, vol. 29, no. 1, 2006, pp. 33–78.

[24] 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.

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

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

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

[28] 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.