Generating Heuristic Functions for Stochastic Shortest Path Problems by Targeted Aggregation of States

Chao-Wen Perng

Generating Heuristic Functions for Stochastic Shortest Path Problems by Targeted Aggregation of States

Chao-Wen Perng

Supervisors: Joerg Hoffmann, Bruno Scherrer

June 2, 2011

Abstract

In a reinforcement learning problem, the agent learns to achieve a goal from interacting with the environment. Markov decision process is a way to formulate such problems. We are interested in solving the stochastic shortest path (SSP) problems, which is a special case of Markov Decision Processes in which one seeks to calculate a policy that minimizes the average cost to reach a goal state.

Real Time Dynamic Programming (RTDP) is a recent heuristic search algorithm that can solve SSP, and we will use the Labeled Real Time Dynamic Programming (LRTDP), an alternative version of RTDP with a labeling scheme that speeds up its convergence. Its behavior depends on the proximity of the heuristic. We seek to find such a heuristic by aggregating the states into macrostates and use the value function obtained from the abstracted problem as the heuristic for the original problem. We designed an aggregation algorithm to obtain an aggregation of good quality. Experiments are run on several benchmarks to see the performance time of the algorithm and how the heuristic obtained improves the performance of LRTDP.

Contents

Chapter 1

Introduction

The reinforcement learning problem is a goal-directed learning from interaction—learning what to do to maximize a numerical reward signal [1]. The learner and decision-maker is called the agent, and everything outside the agent which it interacts with, is called the environment. The agent is not told which actions to take and must discover which actions yield the most reward by trying them (trial-and-error search). Every time the agent selects an action, the environment responds and provides the agent with an immediate reward and a new situation (i.e. state). Figure 1.1 shows the interaction between the agent and the environment at each discrete time steps t. Since the actions selected affect the next situation, all subsequent rewards are also affected. The goal is to obtain a policy, a function that maps each state to an action, which maximizes the total amount of rewards received.

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem [1]. Given a problem, we need to formulate it in a way that only the most important aspects—such as the sense of cause and effect, the sense of uncertainty and nondeterminism, and the existence of explicit goals—are captured and represented in a simplest possible form. A

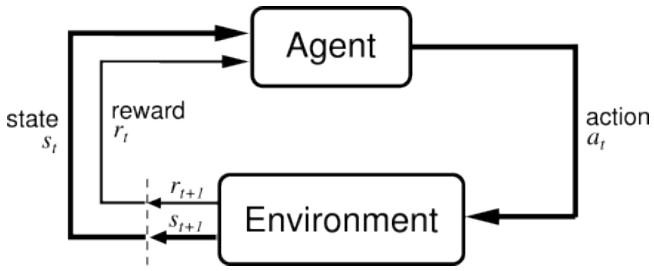

Figure 1.1: The agent-environment interaction. At t, the agent receives some representation of the environment's state, s_t, and select an action a_t. One time step later it receives the numerical reward r_{t+1} and enter a new state s_{t+1}. [1]

formal framework defines the interaction between an agent and its environment in terms of states, actions, and rewards. The agent makes its decisions as a function of the environment's states.

The state signal should not be expected to inform the agent of everything about the environment. For example, in the case of playing blackjack, we should not be expected to know what the next card in the deck is. Ideally, we would like a state signal that summarizes past sensations compactly that all relevant information is retained. A state signal that succeeds in retaining all relevant information is said to have the Markov property. For example, a checkers position–the current configuration of all the pieces on the board–would serve as a Markov state because it summarizes everything important about the complete sequence of positions that led to it [1]. All that really matters for the future of the game is retained and it is all the agent needs in order to make a decision. In other words, the resulting state after an action only depends on the current state and action, regardless of the previous states.

A reinforcement learning task that satisfies the Markov property is called a Markov decision process (MDP). When the state and action spaces are finite, it is a finite MDP. The stochastic shortest path (SSP) is a special case of Markov Decision Processes in which one seeks to calculate a policy that minimizes the average cost to reach a goal state. Using the general theory of MDPs, we can analyze the stochastic shortest-path problem. Each node is a state and each arc between two states is an action. The costs of the arcs are then the absolute value of the negative rewards the agent receives when entering a non-terminal state. At each node, there is a probability distribution over all possible successor nodes, thus the path traversed is random. We are interested in finding the policy from a particular initial state.

The value function is a function of states that estimates how good it is for the agent to be in a given state. A heuristic is an approximate value function—a lower bound of the average cost. Heuristic search methods solve a given problem by repeatedly generating a possible solution (i.e. a path from the initial state) until this possible solution is a real solution (by comparing the state reached with the set of goal states). The behavior, however, depends on the proximity of the heuristic. It provides a rule of thumb that probably leads to a real solution, and thus reduces the number of possible solution tested.

Real Time Dynamic Programming (RTDP) is a recent heuristic search algorithm that can solve SSP, and we will use the Labeled Real Time Dynamic Programming (LRTDP), an alternative version of RTDP with a labeling scheme that speeds up its convergence, proposed by Bonet and Geffner [2].

Since the behavior of heuristic search algorithm depends on the heuristic function, finding a suitable heuristic will then improve the performance of LRTDP. The objective of the internship is to find such a heuristic. In order to do so, we apply the techniques from Munos and Moore [3]:

1. First we aggregate the states into macro-states.

2. Estimate the related error caused by the aggregation.

3. Refine the macro-states that have greater errors.

4. Run a simple method called value iteration (see Chapter 2) on the set of resulting macro-states and obtain the value function for the macro-states. We expect that the time to solve the abstracted problem is trivial since the state space is smaller than the original problem.

5. Use the value function obtained as the heuristic and run LRTDP.

We wish that, at the end, the total time of obtaining the heuristic and running LRTDP will be smaller than running LRTDP without a heuristic (heuristic is 0).

Chapter 2

Background

2.1 Definitions

Each MDP is composed of a discrete and finite state space S, a set of actions A, transition probabilities T, and rewards R. Among the states, we have an initial state s_0 and a set of goal states G. The reward received by being in state s is $R(s)$ and the transition probability of reaching state s' after applying action $a \in A$ is $T(s, a, s')$.

A policy defines the agent's behavior at a given time. It is denoted π. $\pi(s)$ gives the suggested action to be performed when being in state s. The policy is the core of a reinforcement learning agent.

A reward function $R(s)$ defines what are the good and bad events for the agent. It maps each perceived state of the environment to a single number, a reward, indicating the intrinsic desirability of that state [1]. A common approach is to give a reward of -1 for entering a state that is not the goal, which encourages the agent to reach the goal as quickly as possible. This will be how we define our reward functions in our testing benchmarks. Let s_t be the state at time step t, then the goal at time step t is to maximize the expected return

$$R_t = R(s_{t+1}) + R(s_{t+2}) + R(s_{t+3}) ... + R(s_T)$$

where T is the final time step. T is some finite numbers in the cases such as traversing a path and reaching a certain goal.

In the cases where the agent-environment interaction goes on continually without limit, $T = \infty$. For example, this could an application to a robot with a long life span. We call these continuing tasks. The total reward could then be infinite. To avoid this, the concept of discounting is introduced. The total reward would be the expected discounted return

$$R_t = R(s_{t+1}) + \gamma R(s_{t+2}) + \gamma^2 R(s_{t+3}) + ...$$

where γ is a parameter ranges between 0 and 1, called the discount factor. It indicates the present value of future rewards [1]. If it is 0, it means that we are only concerned about the immediate reward received.

Comparing to a reward function, which indicates what is good in an immediate sense, a value function specifies what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state [1]. It indicates the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states. Thus, a state could have a low immediate reward but still have a high value because it is followed by other states that yield high rewards. We denote $V^{\pi}(s)$ as the value function obtained by following policy π. For MDPs, it is defined as

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\}$$

where $E_{\pi}\{\}$ denotes the expected value given that the agent follows policy π, and t is any time step [1]. The value of the terminal state is always 0.

Value functions satisfy particular recursive relationships:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\} \\ &= E_{\pi}\{r_{t+1} + \sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s\} \\ &= R(s) + \sum_{s'} T(s,\pi(s),s')\left[\gamma E_{\pi}\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s'\}\right] \\ &= R(s) + \gamma\sum_{s'} T(s,\pi(s),s')V^{\pi}(s') \end{aligned}$$

This is called the Bellman equation for V^{π}, which expresses a relationship between the value of a state and the values of its successor states. The value function V^{π} is the unique solution to its Bellman equation [1].

A policy π is defined to be better than or equal to another policy π' if for all states $s \in S$, $V^{\pi}(s) \geq V^{\pi'}(s)$. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy, denoted by π^*. The corresponding value function, called the optimal value function, is defined as

$$V^*(s) = max_{\pi}V^{\pi}(s)$$

The Bellman equation for V* is

$$V^*(s) = R(s) + \gamma max_a \sum_{s'} T(s,a,s')V^*(s') \tag{2.1}$$

For finite MDPs, equation 2.1 has a unique solution independent of the policy. In solving a reinforcement learning problem, the value function is what we are most concerned. In most cases, equation 2.1 can be solved only with

extreme computational cost. Instead, the value function is estimated. The most important component of almost all reinforcement learning algorithms is a method for efficiently estimating values [1].

Once one has V^*, it is relatively easy to determine an optimal policy. For each state, there will be one or more actions at which the maximum is obtained in the Bellman equation. Any policy that chooses these actions is an optimal policy. We say that such policy is *greedy*.

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP [1].

2.2 Value Iteration

When computing the value function of the macro-states, we use a traditional dynamic programming method which is called value iteration. The fundamental idea is the Bellman equation (equation 2.1). Value iteration starts with an initial guess in the right-hand side of the equation and thus obtains a new guess on the left-hand side. Let $V_k(s)$ be the value of s in the kth iteration, then

$$V_{k+1}(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s')V_k(s')$$

The iteration continues for all states until it converges with a sufficiently small residual. Below is the pseudo code for value iteration.

Algorithm 1 $valueIteration(S, A, T, R)$

Initialize $V(s) = 0$ for all $s \in S$
repeat
  $\Delta = 0$
  for $s \in S$ **do**
    $v = V(s)$
    $V(s) = R(s) + max_a \gamma \sum_{s'} T(s, a, s')V(s')$
    $\Delta = max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$
for $s \in S$ **do**
  $\pi(s) =$arg$max_a \gamma \sum_{s'} T(s, a, s')V(s')$
end for
return (V, π)

2.3 Real-Time Dynamic Programming (RTDP)

RTDP is a simple DP algorithm that involves a sequence of trials or runs, each starting in the initial state s_0 and ending in a goal state [2]. Each trial simulates

the greedy policy and at the same time updates the values $V(s)$ using

$$V(s) := R(s) + \gamma max_a \sum_{s'} T(s, a, s')V(s')$$

over the states s that are visited. Thus, it is in fact an asynchronous value iteration algorithm that in each iteration, it selects a single state visited by the simulation for update.

In RTDP, it is often not necessary to update all states, which means the algorithm is capable of solving problems with a large state space. On the other hand, the quality of the heuristic functions is crucial for performance. Initially the values of the value function V are given by an heuristic function h, and corresponding entry of $V(s)$ will be updated throughout the simulation [2]. The convergence of RTDP is asymptotic, and the number of trials before convergence is not bounded. This is because low probability transitions are taken eventually, but after a large number of steps.

2.4 LRTDP

To deal with the convergence problem of RTDP, Bonet and Geffner [2] proposed a strategy. They keep track of the states over which the value function has converged and avoid visiting those states again. In order to do so, a labeling procedure is introduced, called CHECKSOLVED. Assume that a set of states are reachable from a state s with the greedy policy, referred as the greedy envelope, and refer the graph made up of these states as the greedy graph. Then, when invoked in a state s, CHECKSOLVED searches the greedy graph rooted at s for a state s' with a residual greater than ϵ. If there is no such state, the state s is labeled as SOLVED and CHECKSOLVED(s) returns true. This labeled version of the RTDP is called labeled RTDP (LRTDP). Initially only the goal states are solved, and the trials terminate when a solved state is reached. They will then invoke the CHECKSOLVED procedure in reverse order, until the procedure returns false on some state. LRTDP terminates when the initial state s_0 is labeled as SOLVED.

2.5 Obtaining a Heuristic from Aggregation

In our algorithm, we first need to aggregate the states s into macro-states. Suppose we have n states $s_1, s_2, ..., s_n$ and we group them into 3 macro-states $\hat{s}_1 = \{s_1, s_2, ..., s_{\lfloor \frac{n}{3} \rfloor}\}$, $\hat{s}_2 = \{s_{\lfloor \frac{n}{3} \rfloor + 1}, ..., s_{\lfloor \frac{2n}{3} \rfloor}\}$, and $\hat{s}_3 = \{s_{\lfloor \frac{2n}{3} \rfloor + 1}, ..., s_n\}$. The transition probability from $\hat{s}$ to $\hat{s}'$ by performing action a is then

$$\hat{T}(\hat{s}, a, \hat{s}') = \frac{1}{|\hat{s}|} \sum_{(s,s') \in \hat{s} \times \hat{s}'} T(s, a, s')$$

and the reward function is

$$\hat{R}(\hat{s}) = \frac{1}{|\hat{s}|} \sum_{s} R(s)$$

These give us a full description of a problem defined on the macro-states, and we can run value iteration to obtain the value function $\hat{V}(\hat{s})$. The values of $\hat{V}(\hat{s})$ are used as an approximation (heuristic) for the value function $V(s)$ of the original problem. Simply put, the approximation value function of state s, denoted $\hat{V}(s)$, is $\hat{V}(\hat{s})$ where $s \in \hat{s}$.

Since there are several states in a macro-state and they usually do not behave the same, there will a difference between the approximation and the real value function. We would like to reduce this difference, thus the notion of local interpolation error and approximation error is introduced.

2.6 Local Interpolation Error and Approximation Error

The quality of the heuristic (approximated value function) depends on the aggregation. In order to develop a good aggregation, we need to consider the errors induced.

The Bellman equation can be written as $V(s) = BV(s)$ where B is called the Bellman operator. Let B be the real Bellman operator of the original problem, $\hat{B}$ be the approximate Bellman operator of the abstract problem with aggregated states, $V(s)$ be the real value function and $\hat{V}(s)$ be the approximate value function. Then

- The local interpolation error $E_{int}(s)$ is the immediate local error when applying the approximate Bellman operator on the real value function $V(s)$.

$$E_{int}(s) = |\hat{B}V(s) - BV(s)|$$

- The approximation error measures the difference between the approximate value function and the true value function.

$$E_{app}(s) = |\hat{V}(s) - V(s)|$$

In Munos and Moore [3], it is shown that bounds on the approximation error can be expressed in terms of the local interpolation error. Given a bound of the local interpolation error, the solution of the Bellman equation below gives an upper bound of the approximation error.

$$\overline{E_{app}}(s) = \overline{E_{int}}(s) + \gamma max_a \sum_{s'} \hat{T}(s, a, s') \overline{E_{int}}(s')$$

Thus, the bound of the approximation error can be computed using value iteration.

By using the inequality that $|V(s)| \leq \frac{R_{max}}{1-\gamma}$, the bound of the local interpolation error inside a macro-state can be defined as

$$\overline{E_{int}}(\hat{s}) = max_a \overline{\Delta R}(\hat{s}) + \frac{\gamma R_{max}}{1-\gamma} \sum_{\hat{s}' \in \hat{S}} \overline{\Delta T}(\hat{s}, \hat{s}')$$

where

$$\begin{cases} \overline{\Delta R}(\hat{s}) = max_{(s,s') \in \hat{s} \times \hat{s}} |R(s) - R(s')| \\ \overline{\Delta T}(\hat{s}, \hat{s}') = max_{(s,s'',a) \in \hat{s} \times \hat{s} \times A} \sum_{s' \in \hat{s}'} |T(s,a,s') - T(s'',a,s')| \end{cases}$$

2.7 Influence Iteration

Munos and Moore [3] introduced the notion of influence of a Markov Chain as a way to measure the extent to which a state contributes to the value of another state. If $V(s)$ satisfies the Bellman equation, then the influence of a state s on another state s' is

$$I(s|s') = \frac{\partial V(s')}{\partial R(s)}$$

In the case of local interpolation error and approximation error, this would be how much a state's local interpolation error contributes to the approximation error of another state. When we refine a macro-state with a higher influence value, it is likely that the approximation error of other macro-states will decrease more. More precisely, assume that we are concerned with the influence on the inital state $\hat{s}_0$, then there is the relationship

$$\Delta E_{app}(\hat{s}_0) \propto I(\hat{s}|\hat{s}_0) \cdot \Delta E_{int}(\hat{s})$$

When we refine a macro-state $\hat{s}$ and its interpolation error is decreased, then the approximation error of the initial state $\hat{s}_0$ will decrease in proportional to $I(\hat{s}|\hat{s}_0)$ and $\Delta E_{int}(\hat{s})$. Therefore, it is possible that refining a macro-state decreases its interpolation error a lot, but does not change the approximation error of the other macro-states much (because it has a low influence). It is important to take into account the influence value so that when we select a macro-state to split, it improves the quality of the aggregation more. In our algorithm, we will choose the macro-states with higher $I(\hat{s}|\hat{s}_0) \cdot \Delta E_{int}(\hat{s})$ for splitting.

The influence of a state s on a set of states Ω is defined as $I(s|\Omega) = \sum_{s' \in \Omega} I(s|s')$. It can be calculated by a fixed-point equation:

$$I(s|\Omega) = \gamma \sum_{s'' \in \Omega} T(s'', a, s) I(s''|\Omega) + \begin{cases} 1 : s \in \Omega \\ 0 : s \notin \Omega \end{cases}$$

This is not a Bellman equation, but the computation is similar. For simplicity, we use $I(s)$ instead of $I(s|\Omega)$ in the pseudo code below.

Algorithm 2 $influenceIteration(S, \pi, T, \Omega)$

Initialize $I(s) = 0$ for all $s \in S$
repeat
  $\Delta = 0$
  for $s \in S$ **do**
    $i = I(s)$
    $I(s) = \gamma \sum_{s''} T(s'', \pi(s''), s) I(s'')$
    if $s \in \Omega$ **then**
      $I(s) += 1$
    end if
    $\Delta = max(\Delta, |i - I(s)|)$
  end for
until $\Delta < \theta$
return I

Chapter 3

Algorithms

The aggregation algorithm is structured as the following:

1. First, we choose a simple initial partition which groups the states that have the same reward values into the same macro-state—the immediate reward partition [4].
2. Calculate the bound of the interpolation error E_{int} of each macro-state.
3. Run value iteration on the macro-states using the interpolation error as rewards. The value function obtained will be the bound of the approximation error $\overline{E_{app}}$, and the policy π_{err} obtained is the policy that maximizes the approximation error.
4. Run influence iteration on the set of the macro-states. The subset of states Ω can be set to the initial macro-state or the whole state space $\hat{S}$.
5. Calculate the splitting criterion $\Delta Eapp(\hat{s})$ of each macro-state. It is defined as $I(\hat{s}) \cdot \Delta E_{int}$, where ΔE_{int} is the change of interpolation error after splitting the macro-state. ΔE_{int} depends on how we want to split the macro-state.
6. Choose a portion of the macro-states with the largest $\Delta Eapp(\hat{s})$ and split them into 2 sub macro-states of equal size.
7. Repeat steps 2 to 6 until the terminating condition is reached.
8. Run value iteration on the resulting macro-states and obtain the approximating value function $\hat{V}^{\hat{\pi}}$.
9. Run LRTDP using $\hat{V}^{\hat{\pi}}$ as heuristic.

There are several parameters in this algorithm, which are:

1. The initial partition, which we chose to be immediate reward partition

2. The discount factor γ
3. The set Ω for the influence iteration
4. The way to split a macro-state, we have set up 2 options
5. The amount of macro-states to be split in each iteration
6. The terminating condition of the aggregation process, such as the number of resulting macro-states, value of the approximation error, etc

Experiments need to be run in order to decide which options are better.

3.1 High Level Pseudo Code

Algorithm 3 is the high level pseudo code for the aggregation process:

- Line 1: Do the immediate reward partition. The detailed pseudo code is in the next section.
- In line 2, we search the maximum absolute reward value R_{max} for calculating the interpolation error. In our benchmarks, this will be 1.
- The aggregation iterations start in line 3.
- In line 4-6, we calculate the local interpolation error. Since we use the immediate reward partition as the initial partition, $\overline{\Delta R}(\hat{s})$ would be 0. We still keep it in the pseudo code in case the initial partition is changed.
- In line 7 we compute the approximation error using value iteration. We use the maximum approximation error among the macro-states as a possible terminating condition for the aggregation process (line 9).
- If we choose to terminate the process according to the approximation error, then we check the maximum approximation error among the states (normalized by R_{max}) in line 8-10.
- In line 11 we run the influence iteration from Chapter 2.
- In order to split a macro-state into two, an array $a(s)$ is defined on the member states (see line 14 of the algorithm). There are many possible choices. The values can even be random, but to optimize the aggregation, it is better to design it so that it can show the behavior of the member states. The 2 options we set up are:
    1. Option 1: $a(s) = max_a \sum_{s' \notin \hat{s}} T(s, a, s')$, the states with larger value are more likely to transit to other macro-states, which means they are "further" from the other member states.
    2. Option 2: $a(s) = max_a \sum_{s'} T(s, a, s')\hat{V}^{\pi}(\hat{s}') + R(s)$, the states with higher values are the states that are more preferable to the agent.

Algorithm 3 highlevel pseudo code

$(\hat{S}, A, \hat{T}, \hat{R}) = immediateRewardPartition(S, A, T, R)$
$R_{max} = max_{s \in S} |R(s)|$
loop
  for $\hat{s} \in \hat{S}$ **do**
    $E_{int}(\hat{s}) = \overline{\Delta R}(\hat{s}) + \frac{\gamma R_{max}}{1-\gamma} \sum_{\hat{s}' \in \hat{S}} \overline{\Delta T}(\hat{s}, \hat{s}')$
  end for
  $(\overline{E_{app}}, \pi_{err}) = ValueIteration(\hat{S}, A, \hat{T}, E_{int})$
  if $(max_{\hat{s} \in \hat{S}} \overline{E_{app}(\hat{s})})/R_{max} < \theta$ **then**
    break //terminating condition reached
  end if
  $I(\hat{s}) = influenceIteration(\hat{S}, \pi_{err}, \hat{T}, \Omega)$ with $\Omega := \{\hat{s}_0\}$ or $\Omega := \hat{S}$
  for $\hat{s} \in \hat{S}$ **do**
    for $s \in \hat{s}$ **do**
      $a(s) = max_a \sum_{s' \notin \hat{s}} T(s, a, s')$
      //Or $a(s) = max_a \sum_{s'} T(s, a, s') \hat{V}^{\pi}(\hat{s}') + R(s)$
      //In this case, need to compute $(\hat{V}^{\pi}, \hat{\pi}) = ValueIteration(\hat{S}, A, \hat{T}, \hat{R})$
    end for
    Sort $\hat{s} = \{s_1, ...s_n\}$ according to $a(s)$ and split $\hat{s}$ into 2 subsets
    $\hat{s}_1 = \{s_1, ..., s_{n/2}\}$ and $\hat{s}_2 = \{s_{n/2+1}, ..., s_n\}$
    $E_{int}(\hat{s}_1) = \overline{\Delta R}(\hat{s}_1) + \frac{\gamma R_{max}}{1-\gamma} \sum_{\hat{s}' \in \hat{S}} \overline{\Delta T}(\hat{s}_1, \hat{s}')$
    $E_{int}(\hat{s}_2) = \overline{\Delta R}(\hat{s}_2) + \frac{\gamma R_{max}}{1-\gamma} \sum_{\hat{s}' \in \hat{S}} \overline{\Delta T}(\hat{s}_2, \hat{s}')$
    $\Delta E_{int} = max(|E_{int}(\hat{s}_1) - E_{int}(\hat{s})|, |(E_{int}\hat{s}_2) - E_{int}(\hat{s})|)$
    $\Delta E_{app}(\hat{s}) = I(\hat{s}) * \Delta E_{int}$
  end for
  Sort all $\hat{s}$ by $\Delta E_{app}(\hat{s})$ with descending order $//\hat{S} = \{\hat{s}_1, ...\hat{s}_n\}$
  for $i = 0 : |\hat{S}|/f$ **do**
    $(\hat{S}, A, \hat{T}, \hat{R}) = split(\hat{S}, \hat{s}_i, A, \hat{T}, \hat{R})$
  end for
  //other possible terminating conditions can be checked here,
  //such as number of iterations or number of macro-states
end loop
$(\hat{V}^{\hat{\pi}}, \hat{\pi}) = ValueIteration(\hat{S}, A, \hat{T}, \hat{R})$
$(LRTDP(S, A, T, R, \hat{V}^{\hat{\pi}}))$

- After sorting the member states, we split it into 2 sub macro-states. This is not the actually splitting. We do it just for calculating the possible change in its interpolation error.
- From line 20 to line 23, we calculate the splitting criterion for each macro-state. First calculate the interpolation errors of the 2 sub states, then calculate the differences from the original interpolation error. The greater difference of the 2 is timed by the influence value of the original macro-state to use as the splitting criterion.
- Then, the macro-states are sorted by their splitting criteria (line 25).
- In line 26-28, we choose a portion of the macro-states with higher splitting criteria to do the actual splitting.
- If other terminating conditions are proposed, they can be inserted into line 29.
- In line 31 the aggregation iteration ends.
- Line 32 runs the value iteration to obtain the value function of the macro-states and in line 33 it is used as the heuristic for LRTDP.

3.2 Sub-routines

Here we include the pseudo code for the immediate reward partition and the splitting.

For the immediate reward partition (Algorithm 4), first we initialize everything (line 1). Second, for each state s, we check if there is any macro-states $\hat{s} \in \hat{S}$ that has the same reward value as s. If so, add s into $\hat{s}$ (line 3-4); otherwise, add a new macro-state $\{s\}$ into $\hat{S}$ and set the reward $\hat{R}(\hat{s}) = R(s)$ (line 6-8).

The easiest way to define the transition probability function of the macro-states is to add up all the corresponding transition probability and then normalize it with the number of member states insides that macro-state, which do it in line 11 to 17, after all the states are added into a macro-state.

The set of actions A will remain the same after aggregation.

Algorithm 4 immediateRewardPartition(S,A,T,R)

1: Initialize $\hat{S}, \hat{T}, \hat{R}$ to be empty
2: **for** $s \in S$ **do**
3: **if** $\exists \hat{s} \in \hat{S}$ *s.t.* $\hat{R}(\hat{s}) = R(s)$ **then**
4: $\hat{s}.add(s)$
5: **else**
6: $\hat{s} = \{s\}$
7: $\hat{S}.add(\hat{s})$
8: $\hat{R}(\hat{s}) = R(s)$
9: **end if**
10: **end for**
11: **for** $s \in S$ **do**
12: **for** $a \in A$ **do**
13: **for** $s' \in S$ **do**
14: $\hat{T}(\hat{s}, a, \hat{s}') = \frac{1}{|\hat{s}|} \sum_{(s,s') \in \hat{s} \times \hat{s}'} T(s, a, s')$
15: **end for**
16: **end for**
17: **end for**
18: **return** $(\hat{S}, A, \hat{T}, \hat{R})$

For splitting a macro-state (Algorithm 5), since we already have the sub-states when calculating the splitting criterion, so in line 1 we do not mention the details. To do the actual splitting, the original macro-state is removed from the state space and the 2 sub-states are added in (line 2-4). Since we did the immediate reward partition, the reward of the sub-states will be the same as the original macro-state (line5-6). After splitting the macro-state, we need to recalculate the corresponding transition probabilities (line 7-16).

Algorithm 5 $split(\hat{S}, \hat{s}, A, \hat{T}, \hat{R})$

Split $\hat{s}$ into $\hat{s}_1$ and $\hat{s}_2$
Remove $\hat{s}$ from $\hat{S}$,$\hat{T}$,$\hat{R}$
$\hat{S}.add(\hat{s}_1)$
$\hat{S}.add(\hat{s}_2)$
$\hat{R}(\hat{s}_1) = \hat{R}(\hat{s})$
$\hat{R}(\hat{s}_2) = \hat{R}(\hat{s})$
for $\hat{T}(\hat{s}', a, \hat{s}'') \in \hat{T}$ **do**
  if $(\hat{s}' = \hat{s}$ && $\hat{T}(\hat{s}', a, \hat{s}'') > 0)$ **then**
    $\hat{T}(\hat{s}_1, a, \hat{s}'') = \frac{1}{|\hat{s}_1|} \sum_{(s,s') \in \hat{s}_1 \times \hat{s}''} T(s, a, s')$
    $\hat{T}(\hat{s}_2, a, \hat{s}'') = \frac{1}{|\hat{s}_2|} \sum_{(s,s') \in \hat{s}_2 \times \hat{s}''} T(s, a, s')$
  end if
  if $(\hat{s}'' == \hat{s}$ && $\hat{T}(\hat{s}', a, \hat{s}'') > 0)$ **then**
    $\hat{T}(\hat{s}', a, \hat{s}_1) = \frac{1}{|\hat{s}'|} \sum_{(s,s') \in \hat{s}' \times \hat{s}_1} T(s, a, s')$
    $\hat{T}(\hat{s}', a, \hat{s}_2) = \frac{1}{|\hat{s}'|} \sum_{(s,s') \in \hat{s}' \times \hat{s}_2} T(s, a, s')$
  end if
end for
return $(\hat{S}, A, \hat{T}, \hat{R})$

3.3 Step-by-step Walkthrough

To make the concept of the algorithm clear, in this section we show a step-by-step walkthrough of the aggregation process with a single iteration, performed on a simple example problem.

The problem is defined as follows (see Figure 3.1):

- There are a total of 4 states. The task is to start from the initial state (s_0) and reach the goal (s_3).
- The action set is {moveRight, moveLeft}.
- If the agent is in s_0, then takiing action "moveLeft" will result in staying in s_0 with transition probability 1.
- If the agent is in s_3, then takiing action "moveRight" will result in staying in s_3 with transition probability 1.
- Otherwise, taking an action will bring the agent one state to the right or left with probability 0.8. The agent will stay in the same state with probability 0.2.
- $R(s_3) = 0$.
- $R(s) = -1$ for all $s \neq s_3$.
- The discount factor is set to 0.9.

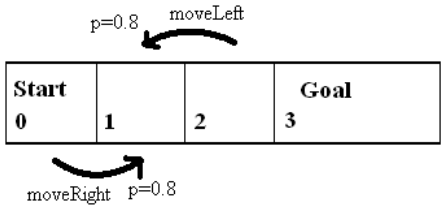

Figure 3.1: the simple example

Now let us start the process:

1. The immediate reward partition groups the states into $\hat{s}_0 = \{s_0, s_1, s_2\}$ (with reward -1) and $\hat{s}_1 = \{s_3\}$ (with reward 0).

   $\hat{T}(\hat{s}_0, moveRight, \hat{s}_0) = \frac{1}{3}[T(s_0, moveRight, s_0) + T(s_0, moveRight, s_1) + T(s_1, moveRight, s_1)$
   $+T(s_1, moveRight, s_2) + T(s_2, moveRight, s_2)] = \frac{2.2}{3}$
   $\hat{T}(\hat{s}_0, moveRight, \hat{s}_1) = \frac{1}{3}[T(s_2, moveRight, s_3)] = \frac{0.8}{3}$
   $\hat{T}(\hat{s}_0, moveLeft, \hat{s}_0) = 1$
   $\hat{T}(\hat{s}_0, moveLeft, \hat{s}_1) = 0$
   $\hat{T}(\hat{s}_1, moveRight, \hat{s}_0) = 0$
   $\hat{T}(\hat{s}_1, moveRight, \hat{s}_1) = 1$
   $\hat{T}(\hat{s}_1, moveLeft, \hat{s}_0) = T(s_3, moveLeft, s_2) = 0.8$
   $\hat{T}(\hat{s}_1, moveLeft, \hat{s}_1) = T(s_3, moveLeft, s_3) = 0.2$
   $\hat{R}(\hat{s}_0) = -1$
   $\hat{R}(\hat{s}_1) = 0$

   We can see that the transition probabilities are normalized so that for each macro-state $\hat{s}$ and each action a, $\sum_{\hat{s}'} \hat{T}(\hat{s}, a, \hat{s}') = 1$.

2. $R_{max} = 1$.

3. Calculate the interpolation error.

$$\begin{aligned} E_{int}(\hat{s}_0) &= 0 + \frac{0.9 \times 1}{1 - 0.9} \left[\overline{\Delta T}(\hat{s}_0, \hat{s}_0) + \overline{\Delta T}(\hat{s}_0, \hat{s}_1)\right] \\ &= 9 \left[\overline{\Delta T}(\hat{s}_0, \hat{s}_0) + \overline{\Delta T}(\hat{s}_0, \hat{s}_1)\right] \\ &= 9(2 + 0.8) \\ &= 25.2 \end{aligned}$$

$$\begin{aligned} E_{int}(\hat{s}_1) &= 0 + \frac{0.9 \times 1}{1 - 0.9} \left[\overline{\Delta T}(\hat{s}_1, \hat{s}_0) + \overline{\Delta T}(\hat{s}_1, \hat{s}_1)\right] \\ &= 0 \end{aligned}$$

4. Run value iteration on $(\hat{S}, A, \hat{T}, E_{int})$. We get

$$\overline{E_{app}}(\hat{s}_0) = 251.148, \pi_{err}(\hat{s}_0) = moveLeft$$

$$\overline{E_{app}}(\hat{s}_1) = 220.501, \pi_{err}(\hat{s}_1) = moveLeft$$

Here the policy π_{err} is to take the action that brings the agent to a state with a greater approximation error. Since $\hat{s_0}$ has a greater approximation error, the policy suggests the agent to always move left.

5. By running the influence iteration with $\Omega = \{\hat{s}_0\}$, we get

$$I(\hat{s}_0) = 9.11371$$

$$I(\hat{s}_1) = 0$$

To calculate them on our own, we have

$$\begin{aligned} I(\hat{s}_1) &= \gamma[T(\hat{s}_0, moveLeft, \hat{s}_1)I(\hat{s}_0) + T(\hat{s}_1, moveLeft, \hat{s}_1)I(\hat{s}_1)] \\ &= 0.9[0 + 0.8I(\hat{s}_1)] \end{aligned}$$

for each iteration. Since the initial $I(\hat{s}_1)$ is 0, it will remain 0. For $I(\hat{s}_0)$ we have

$$\begin{aligned} I(\hat{s}_0) &= 1 + \gamma[T(\hat{s}_0, moveLeft, \hat{s}_0)I(\hat{s}_1) + T(\hat{s}_1, moveLeft, \hat{s}_0)I(\hat{s}_1)] \\ &= 1 + 0.9[I(\hat{s}_0) + 0.8I(\hat{s}_1)] \\ &= 1 + 0.9I(\hat{s}_0) \end{aligned}$$

Starting from 0, the iterations will have values $0, 1, 1+0.9, 1+0.9+0.9^2, \ldots$. This is a geometric series and the solution is $\frac{1}{1-0.9} = 10$. The influence iteration terminates with a residual of 0.1 (doesn't really go to infinity), so there is a slight difference.

The same idea applies to the approximation error, where we have $E_{app}(\hat{s}_0) = 25.2 + 0.9E_{app}(\hat{s}_0)$ with solution $\frac{25.2}{1-0.9} = 252$.

6. By using option 1 for $a(s)$, the states in $\hat{s}_0$ is ordered into $\{s_0, s_1, s_2\}$, and the 2 suggested splitting substates are $\hat{s}'_0 = \{s_0\}$ and $\hat{s}''_0 = \{s_1, s_2\}$. After computing their corresponding interpolation error, the splitting criterion $\Delta E_{app}(\hat{s}_0) = I(\hat{s}_0) * \Delta E_{int}(\hat{s}_0)$ is 229.665.

7. Since $\hat{s}_1$ cannot be split and has $\Delta E_{app}(\hat{s}_1) = 0$, $\hat{s}_0$ has the greatest splitting criterion value. The algorithm will choose to split $\hat{s}_0$.

8. The resulting macro-states will then be $\hat{s}_0 = \{s_0\}$, $\hat{s}_1 = \{s_1, s_2\}$, and $\hat{s}_2 = \{s_3\}$. The transition probabilities are also recalculated.

9. Now we run value iteration on $(\hat{S}, A, \hat{T}, \hat{R})$ and obtain $\hat{V}(\hat{s}_0) = 3.61005$, $\hat{V}(\hat{s}_1) = 2.43002$, and $\hat{V}(\hat{s}_2) = 0$.

10. Run LRTDP with the heuristic function $h(s_0) = 3.61005$, $h(s_1) = 2.43002$, $h(s_2) = 2.43002$, and $h(s_3) = 0$. The real value function for s_0 obtained from LRTDP is 3.77004313 and $\pi(s_0) = moveRight$.

The values of interpolation error and approximation error are large because we use $\gamma = 0.9$. It results in timing the difference in transition probabilities by 9 for the interpolation error and timing interpolation error by 10 for the approximation error. If we use $\gamma = 0.1$, then the values will become much smaller.

Chapter 4

Experiments

To evaluate the aggregation algorithm, we ran several experiments to test which parameters work the best. All the experiments are run on a Windows machine using Cygwin to simulate the Linux environment. The time is measured in seconds.

There are five different kinds of benchmarks used for the experiments:

- Racetrack: a typical kind of benchmark used for SSP. The task is to start from the initial position and reach the goal at the other end of the track. 14 tracks are available, which differ in size and shape. There are 9 actions that specify the direction of movement, and each action has 2 possible outcomes with the probabilities of 0.9 and 0.1.
- Puzzle: the traditional tile-sliding puzzle. The probability of success for each action is set to 0.8 (stay in the same state if fails).
- Rectangle: a simple path traversing problem. The agent can either move left, right, or forward. The goal is to reach the other end, so always move forward would be the best option. The probability of success for each action is set to 0.8.
- Tree: moving along a perfect binary tree. The task is to start from the root node and reach one of the leaves. The actions are moving to the left child and moving to the right child. There exist some random-selected nodes which are noisy. For these nodes, there is a certain probability that the resulting is going to the right child when the action chosen is moving to the left child. In addition, the probability of success for the actions is set to 0.9. When the action fails, it will move one step backwards, going back to the parent node.
- Wet: traversing a "wet" square grid. The initial and goal coordinates are randomly chosen, not necessarily be the corners. The actions available are the movements in the 4 directions. For each coordinate, a random

value called "water" is assigned to it. The next states and the transition probabilities depend on this value.

In all the benchmarks, the reward of any non-terminal state is -1, and 0 for a terminal state.

4.1 γ Comparison

The discount factor γ is designed for continuing tasks. In the benchmarks we use, the time steps are finite, and we actually don't need to use it. Equation 2.1 then becomes

$$V^*(s) = R(s) + max_a \sum_{s'} T(s, a, s')V^*(s')$$

and the update rule for LRTDP becomes

$$V(s) := R(s) + max_a \sum_{s'} T(s, a, s')V(s')$$

We can also say that in this case, $\gamma = 1$.

In contrast, when we run value iteration for the approximation errors, there is no certain goal to reach (no terminal state) and the time steps will be infinite. Hence, here we need to use γ to make the values for the approximation errors converge. Since this is not related to the original problem, γ turns out to be a free parameter. Its value only affects the aggregation process, not the value function for the original problem.

To begin with, we first run some experiments to determine which value of γ works the best. A 2×3 puzzle (state space size 360), a 60×10 rectangle (state space size 600), and a small ring racetrack in Figure 4.1 (state space size 429) are taken for testing. The aggregation algorithm is run for 65 iterations using $\gamma = 0.1, 0.3, 0.5, 0.7, 0.9$. At the end of each iteration, value iteration is called for calculating the heuristic and then LRTDP is called using the heuristic calculated. The other parameters are kept constant: $\Omega = \{\hat{s_0}\}$, split using option 1, splitting fraction (amount of macro-states to be split in each iteration) is 10% of the total number of macro-states.

Figure 4.2 and Figure 4.3 are the results obtained from the 2×3 puzzle. We can see that the time needed for the aggregation grows linearly with the number of iterations, and the amount of time for LRTDP decreases as number of iterations increases. It is actually hard to tell from this benchmark which value of γ is better. There is not much difference in aggregation time. When $\gamma = 0.9$, the time spent between 45 iterations and 65 iterations is much larger than the others, but the other γs perform similarly. In terms of LRTDP time, since all of them are less than 0.04 seconds, it is also hard to decide which has a better influence. $\gamma = 0.3$ and $\gamma = 0.5$ seemed to be more stable.

As for the rectangle benchmark, from Figure 4.4, we can see that $\gamma = 0.9$ has the smallest aggregation time, followed by $\gamma = 0.7$ and $\gamma = 0.1$. Unlike

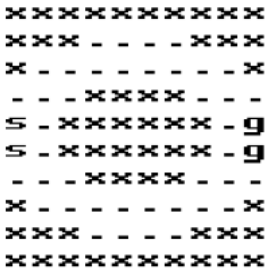

Figure 4.1: a small ring racetrack. The s's are the initial points, and the g's are the goals. The dots are the points that the agent is allowed to enter, while the x's are not.

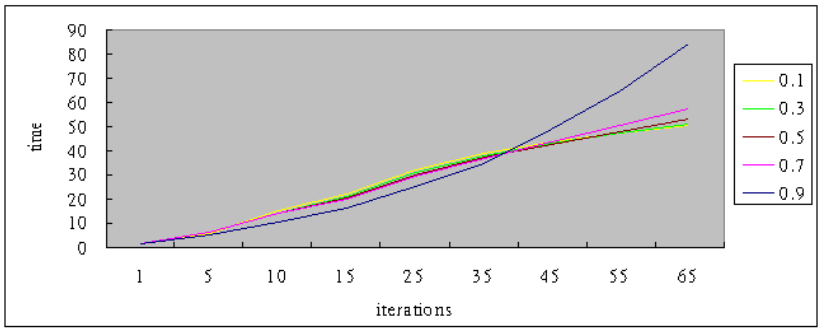

Figure 4.2: time needed for the aggregation (plus heuristic calculation) for puzzle benchmark. Each data line is the result of a different value of γ.

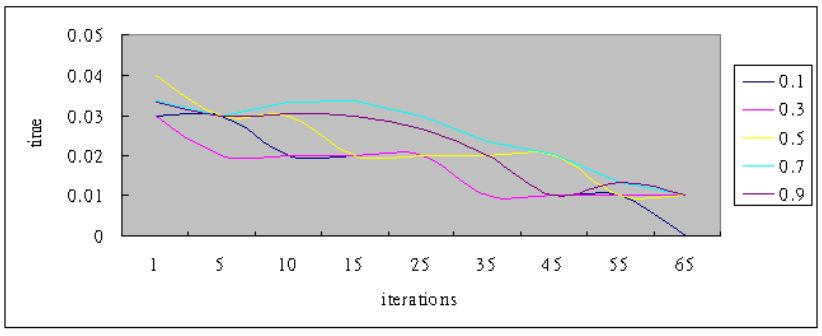

Figure 4.3: time needed for LRTDP for puzzle benchmark. Each data line is the result of a different value of γ.

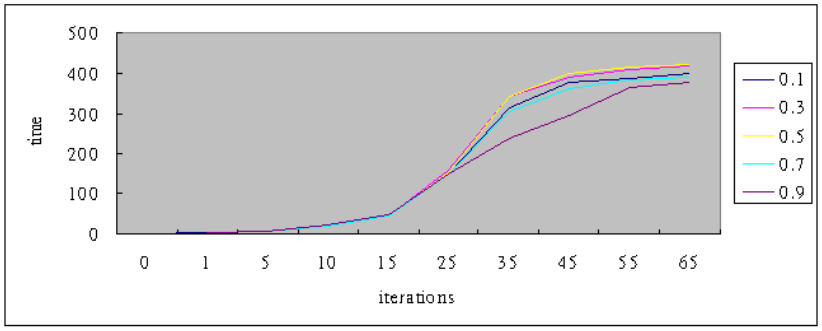

Figure 4.4: γ comparison test result for rectangle: aggregation time

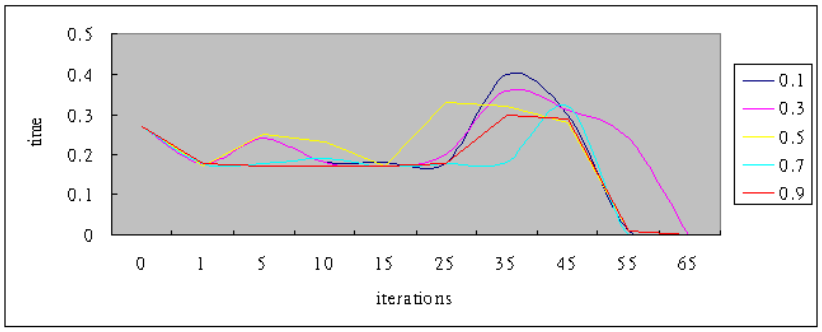

Figure 4.5: γ comparison test result for rectangle: LRTDP time

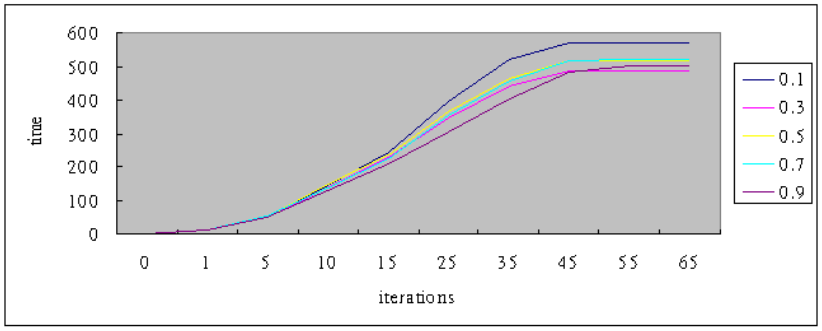

Figure 4.6: γ comparison test result for racetrack: aggregation time

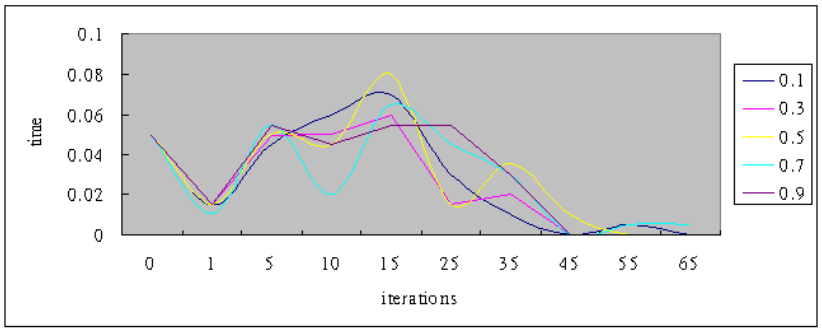

Figure 4.7: γ comparison test result for racetrack: LRTDP time

the puzzle brenchmark, the time for LRTDP does not always improve when the number of iteration increases. In Figure 4.5, we can see that for $\gamma = 0.1$ and 0.3, LRTDP required the most time after 35 iterations, and LRTDP required more time between 25 and 45 iterations for $\gamma = 0.5$. Among all 5 γs, $\gamma = 0.7$ and $\gamma = 0.9$ perform a bit better than the others.

For the small ring racetrack benchmark, $\gamma = 0.1$ requires the most aggregation time, and $\gamma = 0.9$ the least (see Figure 4.6). The influence on LRTDP time is the worst among the 3 benchmarks. Especially for $\gamma = 0.5$ and 0.7, the time really goes up and down (Figure 4.7). In comparison, $\gamma = 0.3$ and $\gamma = 0.9$ are more stable.

To conclude, $\gamma = 0.9$ works the best in 2 of the 3 benchmarks, which means that the impact of future actions is important. Hence, all the following experiments will use $\gamma = 0.9$. Notice that for the puzzle benchmark, $\gamma = 0.9$ also has the lowest aggregation for the first 40 iterations. We do not know why it suddenly becomes slower thereafter. It would be interesting to explore it.

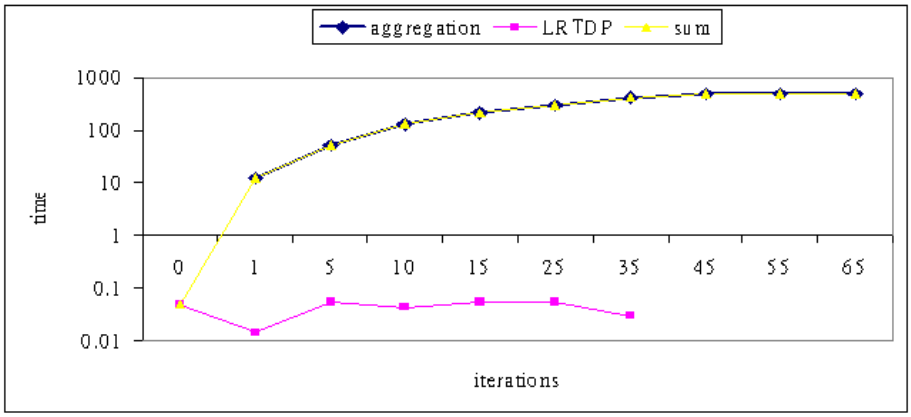

Figure 4.8: performance time for small ring racetrack in log scale. The values that are 0 are not shown, such as the LRTDP time after 35 iterations. The values for aggregation time and sum are very close that they overlap with each other.

4.2 Performance on the Example Problems

After selecting $\gamma = 0.9$, the performances on all example problems are tested. To show the overall performance, the aggregation time, LRTDP time, and the sum of the two are plotted together for comparison.

4.2.1 Racetrack

Figure 4.8 shows the result for the racetrack benchmark. Since we use a log scale in the graph, the data points that are 0 are not shown. Even though there are 14 tracks, the state space size for the other tracks are far too large for testing. An incomplete test on a square racetrack benchmark with 2477 states was run for 15 iterations. It took more than 3 hours for the 15 iterations (see Table 4.1), thus the experiment was halted. An interesting point to notice is that it appears that in both ring racetrack and square racetrack, the first aggregation iteration helps the LRTDP performance a lot, even though the performance becomes worse afterwards. The sum of aggregation time and LRTDP time is the overall performance, and from Figure 4.8 we can see that LRTDP time is negligible comparing to aggregation time, so the graph of the aggregation time overlaps with the graph of the sum. Therefore, in this benchmark the amount of time needed is the lowest when there is no aggregation.

4.2.2 Rectangle

For the 60×10 rectangle used in the previous section, LRTDP time goes down during the first 15 aggregation iterations, and goes up between 15 and 45 iter-

iterations	aggregation time	LRTDP time
0	0	0.16
1	659.187	0.0209961
5	2913.162	0.0700684
10	6143.87	0.0498047
15	111458.6	0.0615234

Table 4.1: aggregation time for 10x10 square racetrack with 2477 states

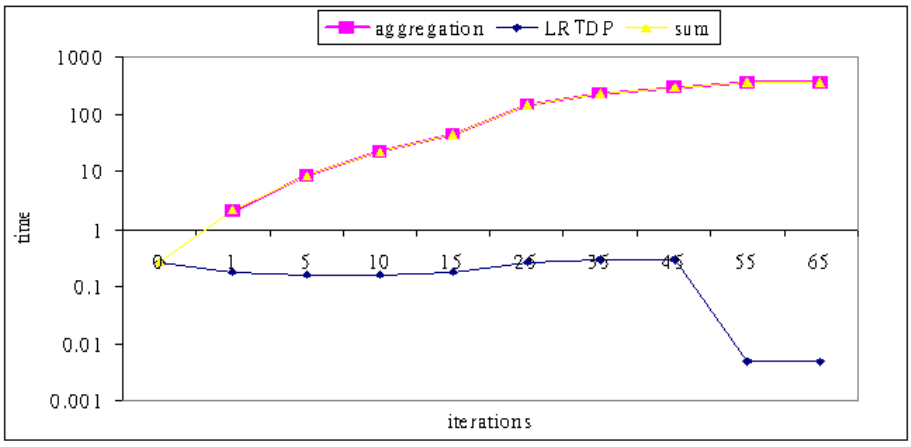

Figure 4.9: 60×10 rectangle benchmark result in log scale. The values for aggregation time and sum are very close that they overlap with each other.

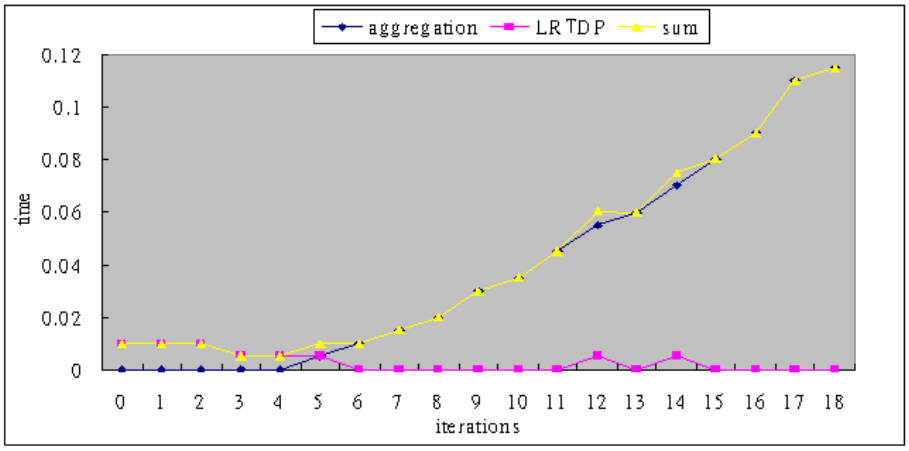

Figure 4.10: 10×3 rectangle result.

ations. After 55 iterations, it goes down again quickly, and becomes very close to 0 after 55 iterations. The aggregation time increases vastly between 15 and 55 iterations, and remains about the same after 55 iterations. Like in the small ring racetrack, LRTDP time is negligible comparing to aggregation time, so here again the graphs of aggregation time and the sum overlap with each other. The best overall performance, thus, is when there is no aggregation (See Figure 4.9), where LRTDP takes 0.27 seconds.

Seeing that the aggregation time needed for 600 states is quite a lot, another experiment was run with a 10×3 rectangle. With only 30 states, the algorithm runs within 0.2 seconds, and it takes only 18 iterations for the algorithm to completely refine the macro-states (every macro-state contains only one state). The result is shown in Figure 4.10, where we can see that the best performance time takes place after 3 iterations. Without heuristic, LRTDP took 0.01 seconds, and with the heuristic obtained from the aggregation after 3 iterations, the total time was 0.005 seconds.

Having tested the rectangle benchmark with 600 states and 30 states, a 25×8 rectangle (200 state) was then tested, to see what happens if the state size is between the previous 2 cases. The result obtained is very similar to the result from the 60×10 rectangle. Without aggregation has the best performance and the graph looks similar to Figure 4.9.

4.2.3 Puzzle

In the 2×3 puzzle, we can see that LRTDP time generally decreased as the number of aggregation iterations increased (Figure 4.11). Even though the improvement in LRTDP performance is obvious, the aggregation time was much larger than LRTDP time, thus it is still better to run LRTDP without aggregation.

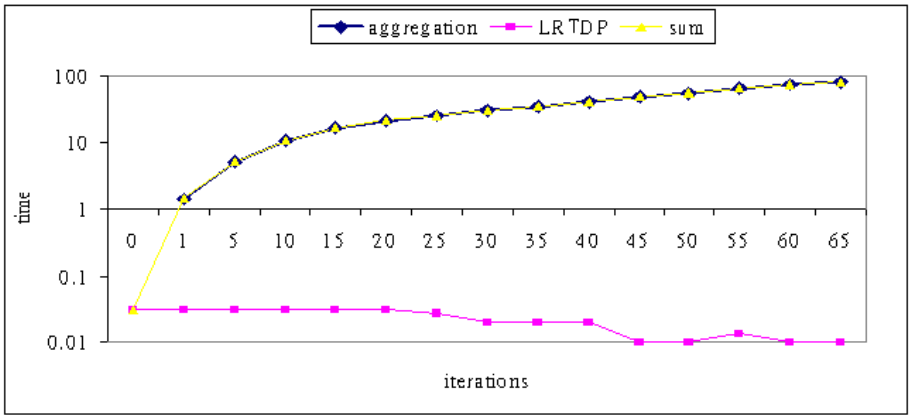

Figure 4.11: 2×3 puzzle result in log scale. The values for aggregation time and sum are very close that they overlap with each other.

To see precisely how the heuristic helped the LRTDP performance, Table 4.2 shows the LRTDP time, the number of LRTDP trials, the number of states visited by the LRTDP trials, the number of macro-states, and the maximum approximation error (among the macro-states) with respect to the number of aggregation iterations. It shows that as the number of iterations increased, the number of LRTDP trials and the states visited by the trials eventually decreased. After 65 iterations, the heuristic obtained allows LRTDP to terminate with only 1 trial and visit only 14 states, thus the time required was a minimum. In fact, this minimum was achieved with 45 iterations (278 macro-states), where number of trials needed was 36 and only 197 states were visited, much smaller than the values from 40 iterations. We can also see that the maximum approximation error first increased in the beginning, and then started to decrease after 20 iterations. The error remains the same for the last 15 iterations. We will discuss this some more in section 3 of this chapter.

4.2.4 Tree

Due to the problem encountered in square racetrack, the experiment on the tree benchmark was chose to run with a small binary tree of depth 7, with 127 states. In this case, 65 iterations only took less than 9 seconds (in Figure 4.12 we shown only 50 iterations, because the macro-states are fully refined at that point). However, the time for LRTDP without heuristic was 0.011 seconds, still much smaller than the aggregation time. With this small state space, simply running value iteration on the original problem took the same amount of time as LRTDP. On the other hand, the heuristics generated from the aggregation algorithm did help improve the performance of LRTDP. The time goes to 0 after one iteration. The detailed data is shown in Table 4.3. Even though the aggregation was not fully refined until 50 iterations, the number of trials went to 1 starting from 30

iterations	LRTDP time	LRTDP trials	states visited	macro-states	max E_{app}
0	0.03	70	360	-	-
1	0.03	78	360	3	252
5	0.03	61	356	7	699.614
10	0.03	56	350	14	1129.36
15	0.0299988	60	360	26	1531.12
20	0.0299988	65	358	45	1606.37
25	0.026665	56	357	76	1232.59
30	0.0200005	65	350	126	971.999
35	0.0200005	50	331	196	755.999
40	0.0200005	57	331	246	683.999
45	0.01001	36	197	278	611.999
50	0.01001	35	201	290	540
55	0.013336	29	166	297	540
60	0.0100021	31	162	307	540
65	0.01001	1	14	317	540

Table 4.2: 2×3 puzzle performance

iterations, and states visited remained 7 from 10 iterations. Also, the maximum approximation error started going down on the 15^{th} iteration, but there is no significance on the other data during that iteration.

4.2.5 Wet

The wet benchmark is a remarkable case. This is a problem in which value iteration perform far better than LRTDP. Here we take a 8×8 grid, 64 states. Without the aggregation, value iteration takes around 0.03 seconds, while LRTDP takes 36.521 seconds. This is caused by the complexity of the movements. When running LRTDP trials, it is hard to reach the goal, thus resulting with 205017 trials. Value iteration terminates in 130 iterations.

Since there are only 64 states, the aggregation algorithm runs fast. 65 iterations runs in 2.334 seconds, and we can see clearly how it improves the performance of LRTDP (Figure 4.3). To take a closer look, Table 4.3 shows the LRTDP time and the number of LRTDP trials with respect to the number of aggregation iterations. After the 24th iteration, the number of macro-states becomes 64 and the heuristic is the exact value function, which is why the LRTDP time approaches 0. According to the definition, the iterpolation error should be 0 when all macro-states contain one single member state, and thus the approximation should also be 0. Noted that in our pseudo code, the approximation error is calculated before the aggregation, so in Table 4.3, the approximation error becomes 0 on the 25th iteration. For this test case, the total time is minimum when the number of iterations is 25, where the number of macro-states is 64. This actually indicates that we are running a value iteration on the original problem, then taking the solution as the heuristic for LRTDP.

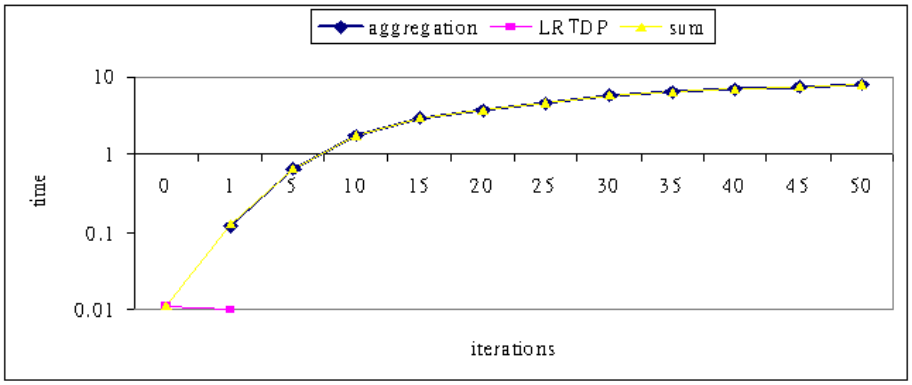

Figure 4.12: depth 7 tree benchmark result in log scale. The aggregation time and the sum overlap with each other. Since LRTDP time becomes 0 after the first iteration, the remaining data points are not shown in the graph.

iterations	LRTDP time	LRTDP trials	states visited	macro-states	max E_{app}
0	0.011	34	81	-	-
1	0.01	35	26	3	293.881
5	0	26	10	7	237.566467
10	0	25	7	14	320.091217
15	0	13	7	26	367.945496
20	0	14	7	45	319.908081
25	0	14	7	76	259.61145
30	0	1	7	98	156.798889
35	0	1	7	107	128.044159
40	0	1	7	113	109.96096
45	0	1	7	122	100.92157
50	0	1	7	127	0

Table 4.3: depth 7 tree result. The LRTDP times were shown 0 under a 10^{-11} precision

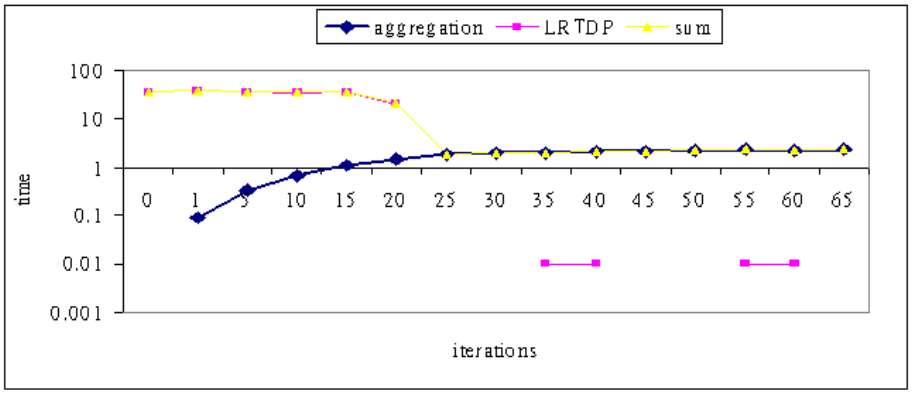

Figure 4.13: time needed for aggregation and LRTDP in log scale. The data points of LRTDP on iteration 25, 30, 45, 50, and 65 are not shown in the graph because they are zero.

iterations	LRTDP time	LRTDP trials	macro-states	max E_{app}
0	37.521	205017	-	-
1	38.5750008	159277	3	239.236
5	36.7630005	136626	7	525.3011
10	34.9799957	136795	14	617.5667
15	35.9789963	202245	26	572.6253
20	19.5579987	81126	45	433.2602
21	15.958	73684	50	435.147
22	8.8249969	56995	56	280.4222
23	2.35700073	7394	61	226.1238
24	0.010009766	17	64	180.532
25	0	17	64	0

Table 4.4: wet benchmark test-LRTDP performance.

iterations	agg time	LRTDP time	LRTDP trials	macro-states	max E_{app}
0	0	382.195	480249	-	-
1	0.61	123.978996	252959	3	261.998
5	2.24295044	85.7430115	136261	7	612.879944
10	4.06594849	85.5130005	134180	14	804.340698
15	6.79998779	84.0009766	119900	26	796.016602
20	9.95397949	72.6149902	102946	45	933.471863
25	12.5570068	52.1129902	92127	76	785.229431
30	15.3109741	38.0450439	55626	126	525.952881
31	16.112999	23.9640007	33412	139	519.212585
32	16.8440018	20.4000015	30415	139	354.237701
33	17.5950012	4.05500031	6264	166	259.950043
34	18.2759976	0.011001586	15	169	152.792999

Table 4.5: wet benchmark with a 13×13 grid.

Seeing that the aggregation algorithm works well with this benchmark, we tried another 13×13 grid, with 169 states. For this state space size, the aggregation still runs pretty fast. The result is in table 4.5. The states are fully refined after 34 iterations. Without heuristic, LRTDP runs for 382.195 seconds and needs 480249 trials. After the first iteration, these numbers are cut down to half. They keep on decreasing until at the end, the number of macro-states becomes 169 and LRTDP time approaches 0. We can see that LRTDP time and number of trials drop dramatically in the last 2 iterations. Even though in this example the best overall time also comes with fully refined macro-states, it still clearly demonstrates how the aggregation improves LRTDP performance.

4.3 Terminating Condition

Ideally, we would like to look for a time when the overall performance is minimum in order to terminate the aggregation process. In many cases the aggregation time is larger than LRTDP time without heuristic, so without aggregation is often the best choice. To focus on how the heuristic improves LRTDP performance, here we ignore the aggregation time need and just look at LRTDP time.

According to the test results from the previous section, LRTDP time often becomes 0 before 65 iterations. Therefore, it is important to decide the terminal condition for the aggregation procedure so that it is only run for a sufficient number of iterations. We have considered options:

1. A fixed number of iterations
2. A fixed value for the maximum approximation error
3. Number of macro-states (possibly be a fraction of total states in the original problem)

By simply looking at the graphs, the number of iterations needed for LRTDP time to approach 0 or for best overall performance really depends on the type of benchmark and its state space size. For example, the 60×10 rectangle needed 55 iterations (Figure 4.9) and the 10×3 rectangle needed only 3 (Figure 4.10). However, the depth 7 tree with 127 states needed only 5 iterations to obtain the lowest LRTDP time (Figure 4.12), while the 64 states wet benchmark needed 25 (Figure 4.13). On the other hand, the best choice for the small ring racetrack might actually be running only 1 aggregation iteration (Figure 4.8). Hence, it is not practically to set a fixed number of iteration for terminating.

In terms of the number of macro-states, when LRTDP time reaches a minimum, the 10×3 rectangle had 4 (30%) macro-states after 3 iteration, the depth 7 tree had 7 (25.926%) macro-states after 5 iterations, the 2×3 puzzle has 278 (77.222%) macro-states after 45 iterations, the small ring racetrack has 3 (0.699%) macro-states after 1 iteration (or 412—96.037%— macro-states after 45 iterations), and the 64 states wet benchmark had 64 (100%) macro-states after 25 iterations.

If we look at the maximum approximation error, setting a fixed value is still not a good idea. First of all, the value of the error really differs between the benchmarks. The 2×3 puzzle had a maximum approximation error up to 1606.37 (Table 4.2), but the depth 7 tree had a maximum of 367.945496 (Table 4.3). Secondly, the approximation starts with a low value and will first increase then decrease, setting a certain value might result in terminating too early.

A possibility is to test if the approximation error is less than the approximation error in the first iteration. We can see in Table 4.3 that in the first iteration, max E_{app} is 293.881, and it became less than this value on the 30^{th} iteration, where the number of LRTDP trials became 1. This is also applicable for the 64 states wet benchmark (Table 4.4), in which on the 23^{th} iteration, max E_{app} is less than the value in the first iteration, and LRTDP time and number of trials were much less than the values without aggregation. For the 169 states benchmark, it lets the process terminate with 34 iterations (see Figure 4.5). This is not the most precise choice, since the depth 7 tree can be terminated with 5 iterations instead of 30, but it makes a reasonable guess. However, there are also exceptions, so it might be useful if we also test if the change in approximation error is sufficiently small. For example, in the puzzle benchmark, the initial approximation error is 252, but from iteration 50 to iteration 65 it remains 540. If we only compare it with the initial approximation error, the aggregation will keep going. Testing the change in approximation error will allow the process to terminate after 50 iterations.

4.4 Aggregation Time Analysis

Table 4.6 shows the time needed for each step of the aggregation during the first iteration for 5 benchmarks. Table 4.7 is the time for the 10^{th} iteration. We can see that the calculation of the local interpolation error took most of the time, followed by the calculation of the splitting criterion. However, the

calculation of the splitting criterion also requires calculating the interpolation error, which means that most of the time is in fact spent by the calculation of the local interpolation error. This is reasonable because there are several nested loops inside the calculation, which is why when the number of states increases, the computation time increases very fast.

benchmark	2×3 puzzle	ring racetrack	60×10 rectangle	depth 7 tree	8×8 wet
E_{int} calculation	0.201	7.39	1.142	0.05	0.04
E_{app} computation	0	0	0	0	0
influence iteration	0	0	0	0	0
splitting criterion	0.626	5.368	0.831	0.04	0.03
split	0.01	0.03	0.01	0.01	0.01
heuristic computation	0	0	0	0	0

Table 4.6: time for each function during the first aggregation iteration

benchmark	2×3 puzzle	ring racetrack	60×10 rectangle	depth 7 tree	8×8 wet
E_{int} calculation	0.781	7	0.892	0.07	0.04
E_{app} computation	0	0.01	0	0	0
influence iteration	0	0	0	0	0
splitting criterion	0.421	3.746	1.022	0.03	0.021
split	0.01	0.03	0.02	0.011	0
heuristic computation	0.04	0.01	0	0	0

Table 4.7: time for each function during the 10^{th} aggregation iteration

Chapter 5

Conclusion

It can be seen from the test results in the previous chapter that in many cases the time needed for the aggregation is much larger than the time for LRTDP. Especially for the 60×10 rectangle and the small ring racetrack benchmarks, where the state space size is 600 and 429, the amount of time is tremendous. For a smaller state space, the aggregation performance was quite good. For the 10×3 rectangle and the 64 states wet benchmark, the aggregation indeed improved the overall performance.

There is no perfect terminal condition for the algorithm. The performance really varies between the benchmarks. Since every benchmark has a different size, we cannot find a general rule in terms of number of iterations.

Looking at the number of macro-states, the fractions for the 10×3 rectangle and the depth 7 tree are similar, but if we apply it to the 64 states wet benchmark, which means letting it terminate after having around 19 macro-states, then it would terminate between 10 to 15 iterations. This is not a good terminating point at all, because the LRTDP time is still greater than 30 seconds. Thus, it is hard to decide the terminating condition using the number of macro-states.

The behaviors of the approximation error are also different, and we can only make a guess using the initial maximum approximation error and the change in the approximation error. To make the guess more precise, combining the 3 options might be useful, but further experiments are needed.

From the experiments on 3 benchmarks, we chose γ to be 0.9. Due to time constraint, we did not examine the other parameters (such as the set Ω or the splitting option) in the algorithm. Experiments should be done so that better performance could be achieved.

To make the aggregation algorithm more practical, it is essential to improve its calculation time. In section 4.4 we see that most of the time is spent on calculating the interpolation error. Therefore, in the future, a method for approximating the interpolation error should be proposed in order to reduce the amount of time needed for the aggregation.

Bibliography

[1] R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, 1998.

[2] B. Bonet and H. Geffner, "Labeled RTDP: Improving the convergence of real-time dynamic programming," In *Proceedings of ICAPS*, pp.12-31, 2003.

[3] R. Munos and A. Moore, "Rates of convergence for variable resolution schemes in optimal control," *International Conference on Machine Learning*, 2000.

[4] T. Dean and R. Givan, "Model Minimization in Markov Decision Processes," In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp.106-111, 1997.