Multi-threaded BLAO* Algorithm

Peng Dai and **Judy Goldsmith**
Computer Science Dept.
University of Kentucky
773 Anderson Tower
Lexington, KY 40506-0046
{pdai2, goldsmit}@cs.uky.edu

Abstract

We present a heuristic search algorithm for solving goal based Markov decision processes (MDPs) named Multi-threaded BLAO* (MBLAO*). Hansen and Zilberstein proposed a heuristic search MDP solver named LAO* (Hansen & Zilberstein 2001). Bhuma and Goldsmith extended LAO* to the bidirectional case (Bhuma & Goldsmith 2003) and named their solver BLAO*. Recent experiments on BLAO* (Dai & Goldsmith 2006) discovered that BLAO* outperforms LAO* by restricting the number of Bellman backups. MBLAO* is based on this observation. MBLAO* further restricts the number of backups by searching backward from the goal state, and also from some middle states (states along the most probable path from the start state to the goal state). Our experiments show that MBLAO* is more efficient than BLAO* and other state-of-the-art heuristic search MDP planners.

Introduction

Given a set of states, a set of actions, a start state and a set of goal states, the classical AI planning problem is to find a policy, a sequence of actions that originates from the start state and reaches any goal state. *Decision theoretic planning* (Boutilier, Dean, & Hanks 1999) is an attractive extension of the classical AI planning paradigm, because it allows one to model problems in which actions have uncertain and cyclic effects. Uncertainty is because one event can lead to different outcomes, and the occurrences of these outcomes are unpredictable but probabilistic, though they are constrained by some form of predefined statistics. The systems are cyclic because an event might leave a state unchanged or return to a visited state.

Markov decision processes (MDP) are a formalism AI researchers have been using for representing decision theoretic planning problems. *Value iteration* (Bellman 1957) and *policy iteration* (Howard 1960) are two fundamental dynamic programming algorithms for solving MDPs. However, these algorithms are sometimes inefficient, because they spend too much time backing up states that may be useless or redundant. Recently, researchers have been working on more efficient solvers. One technique uses reachability information and heuristic functions to avoid unnecessary expansions, such as RTDP (Barto, Bradke, & Singh 1995), LAO* (Hansen & Zilberstein 2001), LRTDP (Bonet & Geffner 2003b) and HDP (Bonet & Geffner 2003a). Another uses approximation methods to simplify the problems by living with suboptimal results, such as the approaches in (Guestrin *et al.* 2003; Poupart *et al.* 2002; Patrascu *et al.* 2002). A third technique aggregates groups of states of an MDP by features, represents them as factored MDPs and solves the factored MDPs by making use of mathematical tools such as *Algebraic decision diagrams* (ADDs) and *Binary decision diagrams* (BDDs) (Bahar *et al.* 1993). Often the graphical structure of factored MDPs are exponentially simpler, but the strategies of factored MDP solvers are often trickier. SPUDD (Hoey *et al.* 1999), sLAO* (Feng & Hansen 2002), sRTDP (Feng, Hansen, & Zilberstein 2003) are examples. We can also use prioritization to decrease the number of inefficient backups. Focused dynamic programming (Ferguson & Stentz 2004), prioritized policy iteration (McMahan & Gordon 2005), state group prioritization (Wingate & Seppi 2005) and topological value iteration (Dai & Goldsmith 2007) are recent examples. Note that our categorization is by no means complete.

Background

In this section, we define Markov decision processes and discuss some extant MDP solvers.

MDPs and Two Basic Dynamic Programming Solvers

An MDP is a four-tuple (S, A, T, C). S is the set of states that describe a system at different times. We consider the system developing along a sequence of discrete time slots. In each time slot, only one event takes effect. We call these time slots *stages*. At any stage t, each state s has an associated set of applicable actions A_s^t. The effect of applying any action is to make the system change from the current state s to the next state s' at stage $t+1$. The transition function for each action, T_a: $S \times S \rightarrow [0,1]$, tells the probability of the system changing to state s' after applying a in state s. $C : S \rightarrow \mathbf{R}$ is the instant cost. In addition to the four components of an MDP, a *value function* V, $V : S \rightarrow \mathbf{R}$, is used to denote the best (maximum, if C is nonpositive) total expected cost from being in a state s. The *horizon* of a MDP is the total number of stages the system evolves. In problems where the horizon is a finite number H, our aim is to

Copyright © 2007, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

define a mapping from reachable states to actions that maximizes the expected cumulative reward, more concretely, the value of $f(s) = \sum_{i=0}^{H} C(s^i)$. For infinite-horizon problems, the reward is accumulated over an infinitely long path. To emphasize the relative importance of instant rewards, we introduce a discount factor $\gamma \in [0, 1]$ for future rewards. In this case, our goal is to maximize $f(s) = \sum_{i=0}^{\infty} \gamma^i C(s^i)$.

Given an MDP, a policy $\pi : S \rightarrow A$ is a mapping from states to actions. An optimal policy guides the agent to pick an action at stage that minimizes the expected cost or value. Bellman (Bellman 1957) showed that the expected value of a specific policy π can be computed using the set of value functions V^π. For finite-horizon MDPs, $V_0^\pi(s)$ is defined to be $C(s)$, and we define V_{t+1}^π according to V_t^π:

$$V_{t+1}^\pi(s) = C(s) + \sum_{s' \in S} \{T_{\pi(s)}(s'|s)V_t^\pi(s')\}. \quad (1)$$

The optimal value function is the maximum value function over all possible policies:

$$V_{t+1}^*(s) = max_{a \in A(s)}[C(s) + \sum_{s' \in S} \{T_{\pi(s)}(s'|s)V_t^*(s')]. \quad (2)$$

For infinite-horizon MDPs, the optimal value function is defined with the discount factor:

$$V^*(s) = max_{a \in A(s)}[C(s) + \gamma \sum_{s' \in S} T_a(s'|s)V^*(s')], \gamma \in [0, 1]. \quad (3)$$

Equation 2 and 3 are named *Bellman equations*. Based on Bellman equations, we can use dynamic programming techniques to compute the exact value of optimal value functions. An optimal policy is easily extracted by choosing an action for each state that contributes to its value function. The process of finding the optimal value function and optimal policy is called solving an MDP.

Value iteration is a dynamic programming algorithm that solves MDPs. Its basic idea is to iteratively update the value functions of every state until the optimal value functions are derived, and we say they *converge*. In each iteration, the value functions are updated according to Equation 3. We call one such update a *Bellman backup*. The *Bellman residual* of a state s is defined to be the difference between the value functions of s in two consecutive iterations. The *Bellman error* is the maximum Bellman residual of the state space. When this Bellman error is less than some threshold value, we conclude that the value functions converge. Policy iteration (Howard 1960) is another approach to solve infinite-horizon MDPs. This algorithm consists of two interleaved steps: policy evaluation and policy improvement. The algorithm stops when in some policy improvement phase, no changes are made. These algorithms both suffer from efficiency problems. Both of them converge in time polynomial in the number of states and $1/(1 - \gamma)$ (Littman, Dean, & Kaelbling 1995), so for realistic problems when the state spaces are large, these algorithms are sometimes slow.

The main drawback of the two algorithm is that, in each iteration, the value functions of every single state are updated, which is highly unnecessary. Firstly, the value functions of each state are initialized by instant cost functions, and sometimes this initialization is too conservative. So the backups in the early iterations are less helpful. Secondly, in each iteration of dynamic programming, all the states are backed up. This could be extremely unnecessary, since different states converge with different rates, when only a few states have not converged, we may only need to back up a subset of the state space in the next iteration.

Other solvers

In this section, we discuss several MDP solvers that fall into our first category. Because our new solver MBLAO* is also in this category, our experiments compare MBLAO* with other solvers in this group. This section helps put our work in the context of recent MDP planning.

Barto et al. (Barto, Bradke, & Singh 1995) proposed an online MDP solver, *real time dynamic programming*. This algorithm assumes that initially it knows nothing about the system except the start state and the goal states. It simulates the evolution of the system by a series of trials. Each trial starts from the start state and ends at a goal state. In each step of the trial, one greedy action is selected based on the current knowledge of value functions and the state is changed stochastically. During the trial, all the visited states are backed up once. The algorithm succeeds when a certain number of trials are finished.

LAO* (Hansen & Zilberstein 2001) is a solver that uses heuristic functions. Its basic idea is to expand an explicit graph G iteratively based on the best-first strategy. Heuristic functions are used to guide which state is expanded next. Every time a new state is expanded, the values of its ancestor states are updated iteratively, using value iteration. In Hansen and Zilberstein's algorithm, they use the *mean first passage* heuristic. LAO* converges faster than RTDP since it expands states instead of actions.

The advantage of RTDP is that it can find a good suboptimal policy pretty fast, but its convergence is slow. Bonet and Geffner extended RTDP to labeled RTDP (LRTDP) (Bonet & Geffner 2003b), and the convergence of LRTDP is much faster. In their approach, they define a state s as *solved* if the Bellman residuals of s and all the states that are reachable through the optimal policy from s are small enough. Once a state is solved, we regard its value function as converged, so it is treated as a "tip state" in the graph. LRTDP terminates when the start state is solved.

HDP is another state-of-the-art algorithm by Bonet and Geffner (Bonet & Geffner 2003a). HDP uses a similar labeling technique to LRTDP, and also finds the connected components in the solution graph of the MDP. HDP labels a component solved when all the states in that component have been labeled. HDP expands and updates states in a depth-first fashion rooted at the start states. All the states belonging to the solved components are regarded as tip states. Their experiments show that HDP dominates LAO* and LRTDP on most of the racetrack MDP benchmarks when the heuristic function h_{min} (Bonet & Geffner 2003b) is used.

*BLAO** (Bhuma & Goldsmith 2003; Bhuma 2004) extends the LAO* algorithm by searching from the start state and the goal state in parallel. In detail, BLAO* has two searches: forward search and backward search. Initially, the value functions of the state space are assigned by heuristic functions. Both searches start concurrently in each iteration. The forward search is almost the same as that of LAO*. It keeps adding unexpanded states into the explicit graph by means of expansions. In an expansion step, an unexpanded "tip" state is chosen, one greedy action and all its associated successor states are added into the explicit graph. After one such expansion, the new value functions of the states in G that are ancestors of the newly expanded state are computed by value iteration.

The backward search is different from the forward search. It originates from the goal state and expands toward the start state. A state s which has not been expanded backwards is expanded along the *best predecessor state*. For one state s' to become the best predecessor state of another state s, it has to fulfill two requirements. First, s' must belong to the set of state L, where the current best action of each state in L has s as a successor state. Second, s' must be the one that has the highest probability to reach s from all the states in L. So the backward search trial is a path instead of a tree. Each backward expansion step adds at most one more node to the explicit graph. The update of value functions after each expansion is the same as the forward search.

Each forward (backward) search terminates when the search loops back to an expanded state, or reaches the goal (start) state or a nonterminal leaf state. After each iteration, a convergence test is called. The convergence test checks whether this iteration expands any states, or the highest difference between value functions of the current iteration and last iteration of each state in G exceeds some predefined threshold value. If not, the optimal policy is extracted and the algorithm ends.

The common advantage of the above algorithms is using start state information to constrain the set of states expanded. The states that are unreachable from the start state are never expanded nor backed up. The algorithms also make use of heuristic functions to guide the search to promising branches.

Multi-thread BLAO*

In our previous experiments on the performance of LAO* and BLAO* (Dai & Goldsmith 2006), we discovered that BLAO* outperforms LAO* by about 10% in racetrack domains. More promisingly, in randomly generated MDPs with large action numbers, BLAO* sometimes runs three times as fast as LAO*. This performance gain is not the result of constraining the number of expanded states. For some problems, BLAO* expands a superset of the states expanded by LAO*, since the backward search may introduce more states to the explicit graph. However, BLAO* is superior to LAO* in that it performs fewer backups. In order to see why, let's take a closer high-level look at the algorithms. In MDP heuristic search planners, the heuristic functions we often use are generated in a backwards manner. For this reason, the heuristic values of states near the goal are often more accurate than those of states near the start state. If we only search forward, as long as the search has not reached the portion that is near the goal, when we do Bellman backups, the values used on the right hand side of Bellman equations are crude heuristic values, and therefore are not guaranteed to be accurate. So the backups performed during these steps are mostly less useful. In BLAO*, by doing backward searches from time to time, we can propagate "more accurate" values by improving the heuristic values of states that are far away from the goal. In this case, the backups performed during the earlier steps of BLAO* make more sense. And sometimes these value propagations turn out to be very helpful.

MBLAO*

```
MBLAO*(int $n$)
1. for every state $s$
2.   $V(s)$ = heuristic value
3.   $\pi(s)$ = an arbitrary action
4. iteration = 0;
5. iteration++;
6. for every state $s$
7.   $s.expanded = false$
8. initialize the following $n+1$ threads
9.   Forward_search(Start);
10.   for $i \leftarrow 1$ to $n$
11.     pick one state $s$ along the optimal path
12.     Backward_search($s$);
13. if Convergence_test($\delta$)
14.   return $V$ and $\pi$;
15. else goto 5;
Forward_search(state $s$)
1. $s.expanded = true$;
2. $a = \pi(s)$
3. while $a$ has any unexpanded successor state $s'$
4.   $G = G \cup s'$
5.   $A = A \cup$ ancestor states of states of $s' \cup s'$
6. for every state in $A$
7.   perform value iteration on them
8. pick an unexpanded tip state $s \in G$
9. Forward_search($s$)
10. return;
Backward_search(state $s$)
1. $s.expanded = true$;
2. $V(s) = max_a\{C(s) + \gamma \sum_{s' \in S}(T_a(s', s)V(s'))\}$;
3. $\pi(s) = argmax_a\{\sum_{s' \in S}(T_a(s', s)V(s'))\}$;
4. $s' \leftarrow$ best predecessor state of $s$
5.   if $s'$ is not the start state or has not been expanded
6.     Backward_search($s'$);
7. return;
Convergence_test($\delta$)
1.  return (changes of value function of every node is
less than $\delta$)
```

Figure 1: Pseudocode of MBLAO*

The intuition behind MBLAO* is based on the above observation. We wondered: can we further decrease the number of backups? We tried changing the strategy of BLAO* (Dai & Goldsmith 2006). The backward search of BLAO* is only along the best predecessor states. We hoped to further constrain the number of backups by broadening the width of the backward search by searching along all predecessor states, but the experimental results showed that enlarging the branching factor of the backward search is not useful.

In MBLAO*, we try something new. The idea is to concurrently start several threads in the explicit graph expansion step. One of them is the same as the forward search in BLAO*, and the rest of them are the backward searches, but have different starting points. We define the *optimal path* of an MDP to be the most probable path from the start state to the goal state, if in every step we follow one optimal policy and choose one successor state with the highest probability of the transition function. The starting points of these backward searches are selected from states belonging to the current optimal path (not including the start state). The underlying idea is the following: One backward search from the goal could help propagate more accurate values from the goal, but not from other sources. This could be complemented by backward searches from other places. Since the planning is mostly interested in the states on the optimal path, value propagations from middle points on this path should be helpful. Also note that the optimal path may change from iteration to iteration, so the sources and trajectories of the backward searches also change. The pseudocode of MBLAO* is shown in Figure 1. The input number n gives the number of backward search threads of our algorithm. As long as the value functions are not close enough to the optimal, as determined by the convergence_test() subroutine, MBLAO* initializes $n + 1$ threads in parallel. One of them is the forward search originating from the start state, and the rest are backward ones. The value functions are updated by value iteration in the forward search, since the forward search graph may contain cycles. The backward search only do one backup at each step.

Experiments

We compare multi-thread BLAO*'s performance with value iteration (VI), LAO*, BLAO*, LRTDP and HDP. All the algorithms are coded in C, and run on the same processor Intel Pentium 4 1.50GHz with 1G main memory and a cache size of 256kB. The operating system is Linux version 2.6.15 and the compiler is gcc version 3.3.4.

Our experiments use some popular domains from the literature: racetrack (Barto, Bradke, & Singh 1995), mountain car (MCar) , single-arm pendulum (SAP) and double-arm pendulum (DAP) (Wingate & Seppi 2005). Racetrack MDPs are simulations of race cars on different tracks. We choose two racetrack domains. One of them is a small track with 1849 states, and the other has 21371 states. Mountain car is an optimal control problem, whose aim is to make the car reach the destination with enough momentum within minimum time. SAP and DAP are similar domains to MCar, whose differences from MCar is that the goal states[1] in SAP and DAP are reachable from the entire state space. All algorithms except VI in our list are initial-state driven algorithms, but in MCar and SAP and DAP domains, we do not have any assumptions about initial states. When we run algorithms on these instances, we randomly pick 10 states from the state space as initial states, and average the statistics over them.

In MBLAO*, our goal is to decrease the number of Bellman backups. In our first group of experiments, we measure both the overall computational cost and the number of backups performed by each approach. We run all the algorithms listed on different instances of domains. Due to the constraints of this paper, we only pick the statistics for one or two instances from each domain, with the size of the state space ranging from 1849 to 160,000. Also, we only pick the results of MBLAO* with 11 threads, that is, one forward search thread and 10 backward ones. We list the size of the solution states[2] of each domain and number of backups performed by each algorithm in Table 2. Note that for instances not shown in the table, the data we have got are similar to their domain representatives. From the backup data, we find that MBLAO* performs fewer backups than LAO* and other heuristic search solvers, including its ancestor, BLAO*. This efficiency helps MBLAO* run faster than other solvers on some domains, as is clearly shown in Table 1. MBLAO* is not suitable for all domains, since we notice the save ratio on racetrack domains is not that appealing. However, it provides relatively large savings on MCar and SAP domains.

In our second group of experiments, we tune the number of backward searches, and analyze how the number of backward searches influence the number of backups MBLAO* ultimately performs. We choose the MCar (300 × 300) domain in particular. As we mentioned earlier, MCar problems are not initial-state driven. So we randomly pick states from the state space as start states. For every different start state, we call this particular MDP an instance. On each instance, we compare the number of backups performed by LAO* and MBLAO* with various number of threads. We try 100 different instances, and randomly pick the statistics from six of them. The number of backups are plotted into the six figures in Table 3. After scrutiny of the figures, we come up with the following conclusions.

- It is not the case that the more threads, the better the performance. This is because, when the number of threads is too large, the backward search threads themselves produce a lot of waste, since the backups performed in backward searches could become less critical. Furthermore, too many backward searches distract from the forward search.

- For different instances, the ratios of the number of backups to the number of threads have no uniform pattern. Neither do they have the same "optimal" thread number, in which the number of backups performed by running MBLAO* is minimal. Particularly, 7, 4, 6, 3, 30 and 5 are the optimal thread number of the instances we have chosen respectively.

[1] In MCar, SAP and DAP, states that have positive instant reward are goal states. Each SAP or DAP problem has one goal state, but an MCar problem has several, because the destination can be reached with various speeds.

[2] the number of states that might be reached by simulating the optimal policy from the start state

Domains	# of solution states	VI	LAO*	BLAO*	MBLAO*	LRTDP	HDP
Racetrack(small)	76	0.06	0.01	0.01	0.00	0.02	0.78
DAP(10^4)	9252	1.27	0.74	0.73	0.52	1.01	70.83
Racetrack(big)	2250	2.08	1.73	1.37	0.80	20.06	10.49
MCar(300×300)	2660	6.45	1.21	1.01	0.65	8.78	1.17
SAP(300×300)	49514	48.51	4.36	3.64	3.11	N/A	N/A
MCar (400×400)	24094	N/A	0.78	0.57	0.30	0.55	1.57

Table 1: Convergence time performed for different algorithms on different domains ($\delta = 10^{-6}$)

Domains	# of solution states	VI	LAO*	BLAO*	MBLAO*	LRTDP	HDP
Racetrack (small)	76	29584	3195	2986	1877	5166	5781
DAP (10^4)	9252	721091	250959	250105	232922	353487	217959
Racetrack (big)	2250	854840	325988	305916	283577	2728517	577160
MCar(300×300)	2660	1102981	91015	86120	43225	453156	71640
SAP(300×300)	49514	48690019	3015618	2764891	1828943	N/A	N/A
MCar (400×400)	24094	N/A	457594	401740	352719	821740	400039

Table 2: # of backups performed by different algorithms on different domains ($\epsilon = 10^{-6}$)

- As the length of the optimal path increases, the optimal thread number does not necessarily increase. At the beginning, we tried to find out the pattern for how many threads are optimal for each problem, but failed. However, the optimal thread number never exceeds 10% of the length of the optimal path in our experiments.

Another interesting discovery is that although MBLAO* sometimes increases the number of state expansions compared to LAO*, the increase rate never exceeds 1%. In some cases, MBLAO* even decreases the expansions, since better heuristic values help make the search more focused. Due to space constraints, we do not discuss this issue in detail.

Conclusion

In this paper, we have discussed an MDP solver named MBLAO*. MBLAO* is an extension to LAO* and a generalization to BLAO*. Unlike most other state-of-the-art heuristic search MDP solvers such as LAO*, LRTDP, and HDP, which mainly focus on applying heuristic search and reachability analysis strategies to constrain the number of expanded nodes, it also restricts the number of Bellman backups by performing timely backward searches. We notice that although the problem of inefficient backups has been addressed by previous approaches, the decrease in the number of expanded states achieved by traditional heuristic search strategies does not necessarily lead to the economic use of Bellman backups. This is because for a large proportion of time, the backups performed in the forward search use inaccurate heuristic functions. This situation can be alleviated by backward searches that help propagate more accurate value functions, and therefore improve heuristics.

Our previous approach in this category is BLAO*, which is able to reduce the number of backups to some degree, but not notably. For this reason, we design MBLAO*, which performs several backward searches for each forward one. Our experimental results show that by integrating a number of backward searches and value propagation, the decreases in the number of backups for our test domains are more distinct. We believe that reducing Bellman backups could become an interesting research topic in heuristic search planning algorithms. Different from the typical way of minimizing the number of expanded states, good strategies could be deployed to make Bellman backups more fruitful, and therefore decrease the overhead in backup computations. These two approaches can complement each other. It's this synthesis that makes MBLAO* so efficient.

Acknowledgment

We thank David Wingate and Kevin Seppi for publishing their programs online. David Wingate's timely answering of our questions is highly appreciated.

References

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1993. Algebraic decision diagrams and their applications. In *Proc. of the 1993 IEEE/ACM international conference on Computer-aided design (ICCAD-03)*, 188–191. Los Alamitos, CA, USA: IEEE Computer Society Press.

Barto, A.; Bradke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *J. of Artificial Intelligence* 72:81–138.

Bellman, R. 1957. *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bhuma, K., and Goldsmith, J. 2003. Bidirectional LAO* algorithm. In *Proc. of Indian International Conferences on Artificial Intelligence (IICAI)*, 980–992.

Bhuma, K. 2004. Bidirectional LAO* algorithm (a faster approach to solve goal-directed MDPs). Master's thesis, University of Kentucky, Lexington.

Bonet, B., and Geffner, H. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proc. of 18th International Joint Conf. on Artificial Intelligence (IJCAI-03)*, 1233–1238. Morgan Kaufmann.

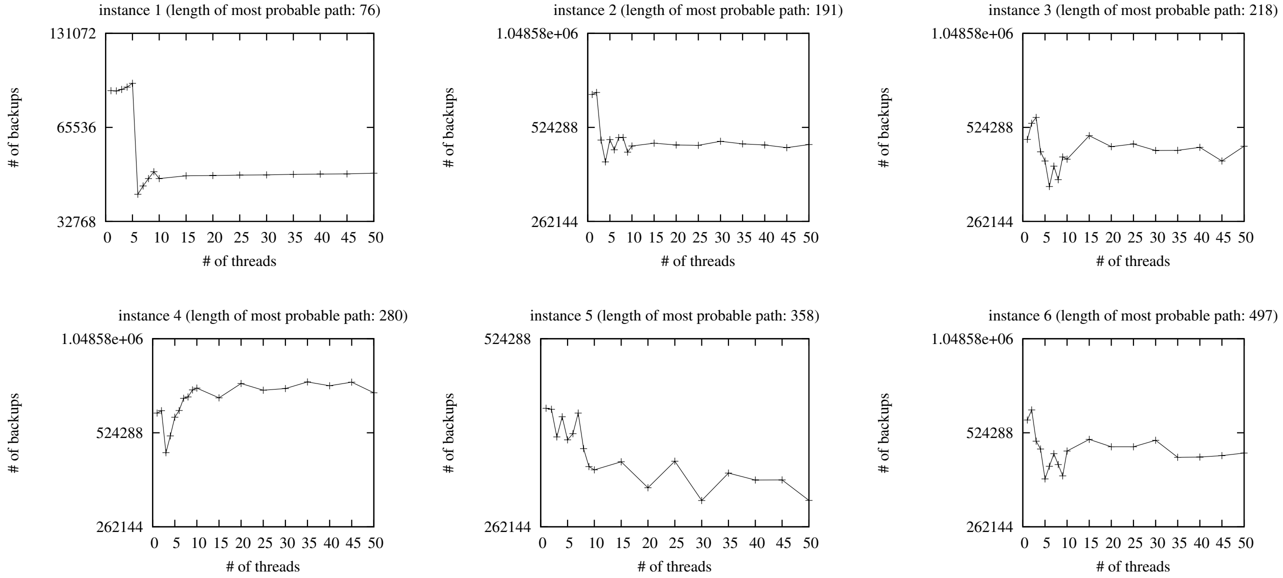

Table 3: Running time of different thread number of MBLAO* on MCar(300 × 300) instances

Bonet, B., and Geffner, H. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. 13th International Conf. on Automated Planning and Scheduling (ICAPS-03)*, 12–21.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *J. of Artificial Intelligence Research* 11:1–94.

Dai, P., and Goldsmith, J. 2006. LAO*, RLAO*, or BLAO*? In *AAAI Workshop on heuristic search*, 59–64.

Dai, P., and Goldsmith, J. 2007. Topological value iteration algorithm for Markov decision processes. In *Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 1860–1865.

Feng, Z., and Hansen, E. A. 2002. Symbolic heuristic search for factored Markov decision processes. In *Proc. of the 17th National Conference on Artificial Intelligence (AAAI-05)*.

Feng, Z.; Hansen, E. A.; and Zilberstein, S. 2003. Symbolic generalization for on-line planning. In *Proc. of the 19th Conference in Uncertainty in Artificial Intelligence (UAI-03)*, 209–216.

Ferguson, D., and Stentz, A. 2004. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Carnegie Mellon University, Pittsburgh, PA.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient solution algorithms for factored MDPs. *J. of Artificial Intelligence Research* 19:399–468.

Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence J.* 129:35–62.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proc. of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-95)*, 279–288.

Howard, R. 1960. *Dynamic Programming and Markov Processes*. Cambridge, Massachusetts: MIT Press.

Littman, M. L.; Dean, T.; and Kaelbling, L. P. 1995. On the complexity of solving Markov decision problems. In *Proc. of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, 394–402.

McMahan, H. B., and Gordon, G. J. 2005. Fast exact planning in Markov decision processes. In *Proc. of the 19th International Joint Conference on Planning and Scheduling (ICAPS-05)*.

Patrascu, R.; Poupart, P.; Schuurmans, D.; Boutilier, C.; and Guestrin, C. 2002. Greedy linear value-approximation for factored Markov decision processes. In *Proc. of the 17th National Conference on Artificial Intelligence (AAAI-02)*, 285–291.

Poupart, P.; Boutilier, C.; Patrascu, R.; and Schuurmans, D. 2002. Piecewise linear value function approximation for factored MDPs. In *Proc. of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 292–299.

Wingate, D., and Seppi, K. D. 2005. Prioritization methods for accelerating MDP solvers. *J. of Machine Learning Research* 6:851–881.