LAO*, RLAO*, or BLAO*?

Peng Dai and **Judy Goldsmith**
Computer Science Dept.
University of Kentucky
773 Anderson Tower
Lexington, KY 40506-0046

Abstract

In 2003, Bhuma and Goldsmith introduced a bidirectional variant of Hansen and Zilberstein's LAO* algorithm called BLAO* for solving goal-based MDPs. BLAO* consistently ran faster than LAO* on the race-track examples used in Hansen and Zilberstein's paper. In this paper, we analyze the performance of BLAO* in comparison with both LAO* and our newly proposed algorithm, RLAO*, the *reverse* LAO* search, to understand what makes the bidirectional search work well.

Introduction

This paper provides a careful analysis of the BLAO* algorithm for Markov decision processes. BLAO* is a bidirectional variant of Hansen and Zilberstein's LAO* algorithm (Hansen & Zilberstein 2001), which solves Markov decision problems. In this paper, three similar algorithms are compared: LAO*, RLAO*, a backwards search version of LAO*, and BLAO*. The three algorithms solve the problem of goal based Markov decision processes (MDP). Given a MDP, a start state and a goal state, the goal based search problem is to find a path from the start state to the goal state that maximizes some type of gains along the path. Compared with classic algorithms such as value iteration or policy iteration (Howard 1960), graph-search based algorithms run much faster because instead of updating the value functions of the entire state space, they only update a subset of them. The use of heuristic functions helps them converge faster.

One may conjecture that BLAO* runs faster because the reverse search is itself faster than forward search. To investigate this, we code the backwards search as RLAO* and run it separately. We discover that the performance of RLAO* is particularly sensitive to the number of possible successor states in the MDPs. The larger the "outdegree" of states, the larger the "indegree" as well; with large indegree, backwards search bogs down even more than forward search.

We summarize the contributions of this paper: We disprove the above conjecture and find the real reason is that BLAO* can efficiently constrain the size of the search space during iterations of value function update. We find the performance of BLAO* is not only 10% better than LAO* on our benchmarks, as Bhuma and Goldsmith claimed (Bhuma & Goldsmith 2003), but also BLAO* is able to run 3 times faster than LAO* on some of the non-extreme cases. We also try a different implementation of BLAO* by replacing the original backwards search with our newly proposed RLAO* algorithm, and find it does not provide obvious speedup against the original BLAO*.

MDPs and previous algorithms

A Markov Decision Process (MDP) is a four-tuple (S, A, T, R). The set of states, S, tells how a system can be at a given time. We assume that systems evolve discretely rather than continuously, so we can partition a system evolution into a sequence of *stages*. Any event makes the system change from one stage t to the next stage $t+1$. For each stage t of the process, each state s has a set of applicable actions A_s^t. When an action is performed, the system changes from the current state to the next state and proceeds to stage $t+1$. T_a: $S \times S \rightarrow [0, 1]$ is the set of transition functions for each action a, which specify the probability of changing from one state to another after applying a. $R : S \longrightarrow \mathbf{R}$ is the instant reward (sometimes R can be replaced by C, which specifies the instant cost). A value function V, $V : S \longrightarrow \mathbf{R}$ associates a value of total expected reward with being in a state s. The *horizon* of a MDP is defined to be the number of stages the system will be evolved. In finite-horizon problems, we try to maximize the total expected reward associated with a course of actions of H stages. The value is defined as $V(h) = \sum_{i=0}^{H} R(s^i)$. For infinite-horizon problems, the reward is accumulated over an infinitely long path. In this case, to avoid infinite value, a discount factor $\gamma \in [0, 1]$ is generally introduced. The value function for an expected total discounted reward problem is defined as: $V(h) = \sum_{i=0}^{\infty} \gamma^i R(s^i)$.

Given a MDP, we look at the problem of finding the *policy* that maximizes total expected reward for an infinite horizon. A policy $\pi : S \rightarrow A$ tells which action to pick at any state s. Bellman (Bellman 1957) showed that the expected value of this policy can be computed using the set of *value functions* V^π. We initialize $V_0^\pi(s)$ to be $R(s)$, then:

$$V_{t+1}^\pi = R(s) + \gamma \sum_{s' \in S} \{T_{\pi(s)}(s', s) V_t^\pi(s')\}, \gamma \in [0, 1]. \quad (1)$$

The *optimal* policy is the the mapping from the state space to the set of actions, which defines the maximum expected values. Based on Equation 1, dynamic programming algorithms can be deployed to calculate the value functions.

Two basic dynamic programming based algorithms are *Value Iteration* and *Policy Iteration* (Howard 1960). For value iteration, the value functions of each state are calculated, and a policy is extracted. In each iteration, the value functions are updated according to Equation 1. Policy iteration is another dynamic programming algorithm for solving infinite horizon problems whose expected run time is smaller than value iteration for solving the same problems. The main drawback of both algorithms is that all the states in the state space are involved in each iteration of dynamic programming. There are several reasons that this is not necessary. First, some states are never reachable from the start state, so they are irrelevant in deciding the value function of the start state. Second, the value functions of some states converge faster than others, so in some iterations, we actually only need to update values of a subset of the states. Third, reaching a convergent status for every state seems to be a hard task.

Barto et al. (Barto, Bradke, & Singh 1995) proposed an algorithm named real-time dynamic programming to solve MDPs. Its main contribution is that it minimizes the search space of dynamic programming. RTDP explores possible "trials" to investigate choices of actions for each state. For each trial, the current state is initialized to the start state, and propagates towards the goal state. In each step, it updates the value function of the current state using Equation 1 and greedily picks an action based on the current policy, and changes the current state according to the transition function. Each trial stops until the goal state is reached or a certain number of steps are accomplished. So in this scenario, the states that are unreachable from the start states are ignored in the trial.

A*-based Algorithms

Another approach to speeding up dynamic programming is to decrease the number of iterations by using heuristic functions.

*A** (Hart, Nilsson, & Raphael 1968) is a basic algorithm used in graph search that combines the two evaluation functions g and h, where $g(n)$ gives the reward accumulated from the start state to the state n, and heuristic function $h(n)$ tells the estimated maximum reward of the paths from the state n to the goal (or the heuristic function). A* is optimal given that the heuristic function is admissible (Dechter & Pearl 1985).

*AO** (Nilson 1980) is an extension to the A* algorithm that applies to acyclic *AND/OR graphs* or acyclic MDPs. It finds a solution/policy that has a conditional structure which takes the form of a tree. Like other heuristic search algorithms, AO* can find a solution graph[1] without considering the entire state space. The algorithm iteratively increases the explicit graph, G'. A non-goal state can be expanded by adding to G' one of its actions and the associated successor states. A partial solution graph is defined as the best solution graph out of G'. AO* keeps expanding the best partial solution graph. In a specific expansion step, the algorithm picks an arbitrary non-goal state and adds all its successors to G'. A set Z is built which includes all the newly expanded states and their ancestors. Then the algorithm repeatedly deletes from Z a node with no descendents in Z. It updates the node's value according to

$$V(s) = min_{a \in A(s)}[R(s) + \gamma \sum_{s' \in S} T(s'|s,a)V(s')], \quad (2)$$

until Z becomes empty. The algorithm stops when a solution graph is constructed.

*LAO** (Hansen & Zilberstein 2001) is an extension to the AO* algorithm that can handle the situation that solution graphs contain loops. Thus, it can handle MDPs. Instead of updating nodes in Z in a backward topological order, it updates them all together by means of value iteration, because topological orders among them may not exist. Certain convergence tests are deployed to constrain the number of iterations in dynamic programming steps. The heuristic function used is *mean first passage* (Kirkland, Neumann, & Xu 2001), the expected number of steps needed to reach the goal state with the current knowledge. Mean first passage is admissible.

*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 expansion. In an expansion, an unexpanded "tip" state is chosen, one greedy action and all its associated successor states are included into the explicit graph. After one such expansion, the value functions of the states in Z are computed by value iteration.

The backward search is almost symmetric to the forward search, with the exception of how a state is expanded backwards. A state s which has not been expanded backwards is expanded this way: an action a together with a previous state s' that can reach s that yields the highest expected rewards is chosen as the previous action and previous state of s to be expanded. Each backward expansion only adds 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 done. 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 exceeds some predefined threshold value. If not, the optimal policy is extracted and the algorithm ends.

[1]The solution graph is the subgraph that contain all the states that are on the optimal path (the most probable path originating from the start state, applying the optimal policy, and ending at the goal state) and their descendents.

RLAO*

Our intuition of the algorithm is: if a state is far from the goal state, its successor states are probably far from the goal as well. Since we use mean first passage as the heuristic function and Equation 2 to update value functions, if we expand from the start state, the value functions in the first few iterations are far from accurate. This is because as in the first few iterations, if we have not yet reached any terminal states, when we update the value functions of the states in the explicit graph, the value functions we have used in the right hand side of Equation 2 are not true value functions, but only heuristic values. We want to design an algorithm in which at each step, we propagate much more accurate value functions towards the start state. So we think about doing the propagation in the backward manner.

We call our algorithm RLAO* because it can be seen as a reverse version of LAO* algorithm. Imagine in the graphical representation of LAO*, each state node points to several action nodes, which are the actions that are applicable at that state, and each action node points to some state nodes, which are the possible successor states of applying such action. For our RLAO* algorithm, we maintain the same graphical structure. In addition, we also keep a *reverse graph*, in which it contains the same set of vertices and edges as the original graph, but the directions of all the directed edges in the original graph are reversed. This means all the states points to the actions that lead to them, and all the actions point to the states in which they can be applied.

The algorithm is given in Figure 1: The main idea is to propagate the value functions from the goal to the start state by means of expansion. In the main function, we iteratively expand the graph. In each iteration, we pick the goal state and expand it. In the expand function, we first mark it as expanded and update the state's value function according to Equation 2 and check if all its outgoing edges in the reverse graph point to states that have been expanded in this iteration. If not, we pick one such unexpanded state, and recursively call the expand function on that state. RLAO* algorithm can also be seen as a depth first search on the reverse graph. In this case, if we look on each expansion of LAO* as moving forward one step, we can vaguely think of one expansion of RLAO* as moving backwards one step, although some MDPs are densely connected, so we cannot clearly define what is one step backward.

The convergence judgment of RLAO* is also different from the LAO* algorithm. For our algorithm, we do not require all the states in the solution graph to be expanded. In RLAO*, we cannot guarantee that all the states in the forward graph have been expanded, since we search backwards. We can only guarantee that all the states in the *reverse solution graph*, the solution graph of the reverse graph, are expanded. In reality, in most MDPs, the nature of the cyclic and densely connected states of the graph does not require states in the forward graph to be expanded before the value function of the start state converges. In our experiments, RLAO* gives the same policies as LAO* and BLAO* 100% of the time.

Reverse LAO*

```
RLAO*()
1. for every state $s$
2.   $V(s)$ = mean first passage of $s$
3. iteration = 0;
4. iteration++;
5.   expand(Goal);
6. if convergencetest($\delta_1, \delta_2$)
7.   return;
8. else goto 2;
expand(state s)
1. s.expanded = true
2. $V(s) = R(s) + \gamma \sum_{s' \in S} \{T_{\pi(s)}(s', s)V^{\pi}(s')\}$;
3. if s has any unexpanded previous state s”
4.   if s” is not the start state
5.     expand(s”);
6. return;
convergencetest($\delta_1, \delta_2$)
1. if (changes of value function of every node is less
   than $\delta_1$) and (change of $V(start) < \delta_2$)
2.   return true;
3. else return false;
```

Figure 1: Pseudocode of RLAO*

Experiments

We have tested algorithm LAO*, RLAO*, BLAO* on two types of MDPs, racetrack MDPs and randomly generated MDPs. We have all three algorithms coded in C, and run them 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.

Race track problem

To test the performance of RTDP, Barto et al. (Barto 1995) introduced a test problem named *race track*. The race track problem is a simple illustration of a car race. A car always starts at the start state and moves towards the goal. Each position in the track is represented as a square cell on the graph. At each instance of time, the car can choose to either stand still or move one cell along eight possible directions. When moving, the car has a possibility of 0.9 to succeed and 0.1 to fail, which means ending up on some other state. There are wall states in the graph. When the car hits a wall, it starts over. We compared LAO*, RLAO* and BLAO* on two instances of the race track problem. The results can be seen in Table 1.

Although LAO* runs for the fewest iterations, its running time is worse than BLAO*. The explanation is that BLAO* updates fewer states in each iteration. The running time per iteration of RLAO* is the worst of the three. We expected this, because for the reverse graph we defined in Section , the outdegree of a state is more than that of the original graph, so that in one iteration, there are exponentially more nodes

Table 1: Comparisons of LAO*, RLAO* and BLAO* on race track MDPs

Alg	Running time	# iter	Optimal?
Small problem (1849 states)			
LAO*	0.01	27	yes
RLAO*	0.03	52	yes
BLAO*	0.01	46	yes
Big problem (21371 states)			
LAO*	1.83	137	yes
RLAO*	8.02	246	yes
BLAO*	1.49	195	yes

to update. This is the main deficiency of RLAO* algorithm. We do not, however, emphasize the race track problem, because the problem is almost deterministic.

Random MDPs

We compare performances of LAO*, RLAO* and BLAO* on another class of MDPs. (See (Hansen & Zilberstein 2001) for a comparison with value and policy iteration.) We construct *random MDPs* by varying the number of actions for each state, the number of states in the MDP, and the maximum number of successor states in each action while keeping other arguments fixed. Given the maximum number of successors, we let the successor states of each state be uniformly randomly distributed over the entire state space, and define the probabilities of each transition by a set of normalized real values in $[0, 1]$.

We try each algorithm on 50 MDPs with each configuration. We find that RLAO* runs about 5% faster than LAO* and BLAO* when the state space is small (under 5000 states) and sparsely connected (each state has only two actions), which is consistent with our original intuition. For densely connected graphs with large state space, Figure 2 shows the run time when the state space changes. Figure 3 and Figure 4 show the run time and number of iterations as the number of successor states varies. Table 2 gives part of the run time when the number of actions per state varies. Figure 5 plots the run time of the three algorithms when states have 10 to 50 actions available. Note that all the algorithms gave the same policies on all the test cases that are listed below.

From Figure 2, we know that, when the number of actions of every state in MDP is 4 (we consider this to be small), the performances of the three algorithms do not show huge differences. However, BLAO* works better than the other two (roughly 10% more efficient when the number of states is around 30k). There are almost no differences between LAO* and RLAO*, since when the branching factor of the states is small, the structures of the original graph and reverse graph are quite similar. Our results are consistent with those of (Bhuma & Goldsmith 2003).

From Figure 3 and Figure 4, we discovered that, when the action number and the state space are fixed, the change

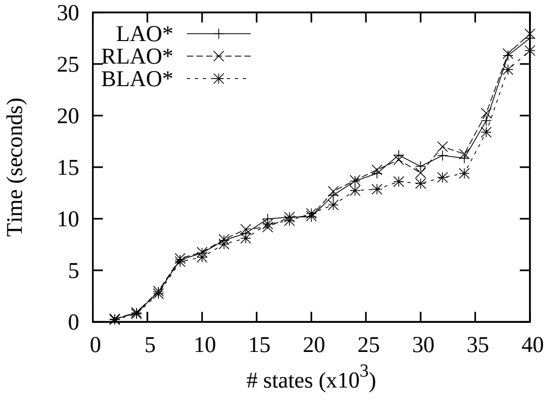

Figure 2: Run time on 4-action 5-successor state MDPs

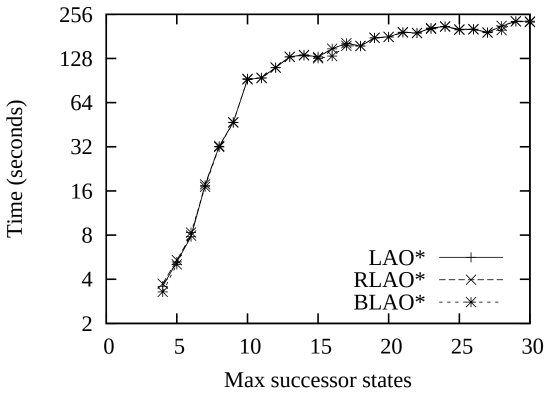

Figure 3: Run time on 10,000-state 4-action MDPs

in the number of possible successor states does not make any algorithm better than the others, since we notice that the three lines in both figures are almost overlapped. This is because, when the number of possible successor states is large, the main overhead of the algorithms is backing up states, or the process of apply Equation 2. Consider if the average number of successor states each state has is 10 and the state space is 10,000, when we recursively expand the graph, in the worst case, four step expansion will involve the update of the value functions of the entire graph. Often our optimal path contains tens of states; in this case, in each iteration the value functions of almost the entire state space are updated, so that RLAO* becomes a symmetric version of LAO*, and BLAO* reduces to LAO*.

However, when the number of actions changes, the results change drastically. As seen in Table 2 and Figure 5, when the number of actions is under 6, LAO*, RLAO* and BLAO* run in almost the same time, which confirms our previous judgments. However, when the action number is more than 10, we observe that the convergence of BLAO* becomes more than twice as fast as LAO*.[2] RLAO* is the

[2]Note that this result is not only limited to the case where the state space is 10,000. We also experiment on state spaces of up to 100,000 states, and the run times show the same scale. Because of

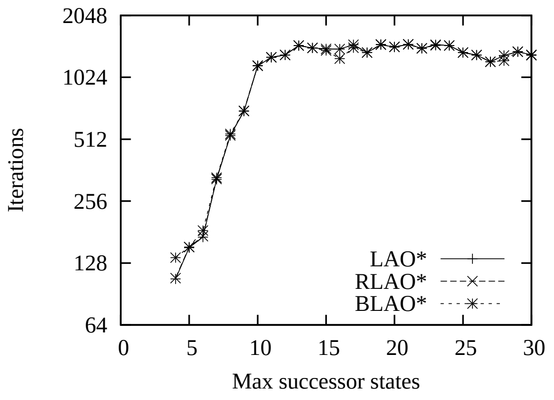

Figure 4: # of iterations on 10,000-state 4-action MDPs

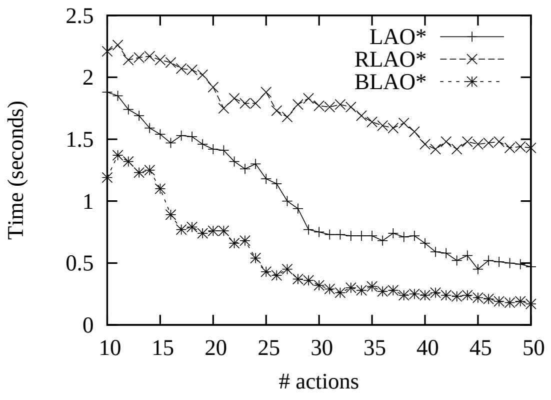

Figure 5: Run time on 10,000-state 5-successor state MDPs with

Table 2: Run time on 10,000-state 5-successor state MDPs

# actions	LAO*	RLAO*	BLAO*
2	33.670000	32.830000	32.170000
4	13.570000	13.710000	13.070000
6	4.450000	4.600000	4.190000
8	1.600000	1.880000	1.310000
10	1.880000	2.210000	1.190000
20	1.420000	1.920000	0.760000
30	0.750000	1.770000	0.320000
40	0.660000	1.460000	0.240000
50	0.470000	1.430000	0.170000

Table 3: Maximum # states updated each iteration on 10,000-state 5-successor state MDPs

# actions	LAO*	RLAO*	BLAO*
10	9064	9986	7455
15	8700	9967	7326
20	8247	9987	6135
25	8875	9994	6135
30	9103	9980	4179
35	8788	9995	4138
40	6421	9996	5879
45	5948	9972	3057

slowest, because if the action state number ratio is sufficiently large, the expanded graph includes more states. In LAO*, when the expansion reaches the goal state, almost the entire graph is involved in the forward graph, as we see in Table 3. In this case, doing a backward expansion at the same time keeps the size of the solution graph under control, but does not slow down convergence rate. We can see this from the comparisons of the number of states updated in each iteration and the number of iterations executed by LAO* and BLAO*. RLAO* is the slowest because, when the action choice becomes broader, the reverse graph becomes much denser than the original graph. This can be seen from Table 3: when the action number increases, the maximum number of states updated in each iteration does not drop as in LAO* and BLAO*, but rather remains fixed and occupies almost the entire graph. Moreover, the convergence rate of BLAO* is no worse than LAO* or RLAO*, which is shown in Table 4.

This reminds us how BLAO* is implemented. In the backward search of BLAO*, the expansion is always undertaken along the best previous action. We wonder whether we can further constrain the number of expanded states in each iteration by replacing its backwards search with RLAO*.

space constraints, we only display the case when the state space is 10,000.

We consider a variant of the original BLAO* algorithm, in which the backward expansion is not only along the best previous action, but every directed edge in the reverse graph. This means all the states that can reach the current state are expanded further. We find that the performances of these two implementations are almost the same, as shown in Table 5. We conclude that the two implementations can do equally well in constraining the number of expanded states in each iteration. So changing the reverse search does not yield a better algorithm.

Conclusion and future work

We have studied the problem of goal-based graph search and planning with Hansen and Zilberstein's LAO*, Bhuma and Goldsmith's BLAO*, and our new RLAO*. Our experiments show that BLAO* works the best of the three algorithms in racetrack problems. In randomly generated MDPs, RLAO* works the best only in sparsely connected graphs with small state spaces, and it is noticeably worse than the other two when the action number is quite large, because of the large branching factor of the reverse graph. When the number of actions per state is relatively small, BLAO* displays no advantages over the other two algorithms. Nevertheless, when the state space is fixed, as the number of

Table 4: # Iterations on 10,000-state 5-successor state MDPs

# actions	**LAO***	**RLAO***	**BLAO***
10	25	35	29
15	25	29	29
20	24	23	24
25	19	23	19
30	6	12	10
35	6	8	7
40	7	9	11
45	10	8	18

Table 5: Comparison of two BLAO* implementations

	original		**new**	
# states	time	#iter	time	#iter
2000	0.47	114	0.46	117
4000	1.94	192	1.93	194
6000	2.47	152	2.44	163
8000	7.72	336	7.78	327
10000	6.05	215	6.04	211
20000	12.74	196	12.08	196
30000	21.08	252	19.74	220

actions increases, BLAO* beats the other two.

This phenomenon is more obvious when the number of actions is large, since the backward expansion of BLAO* manages to keep the number of states expanded in each iteration under control. Based on this result, we have implemented another version of BLAO*, hoping to further control the expanded states by strengthening the backward search. However, the comparison between the two versions of BLAO* proves that the new algorithm doesn't trivialize the problem, which from a different point of view, proves the effectiveness of BLAO*.

From our experiments, we conjecture that RLAO* has only a limited usage in MDPs that have small action per state rates, while BLAO* is useful when problem spaces have large branching factors. However, our conjecture is only based on the tests of racetrack problem and artificial MDPs. In the future, we want to test our RLAO* algorithm more systematically on some more real MDP benchmarks, and we want to compare our algorithm with other relevant algorithms.

Acknowledgments We thank Lengning Liu and an anonymous referee for their comments on earlier drafts of this paper. This work is partially supported by NSF grant ITR-0325063.

References

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

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

Bhuma, V. D. K., and Goldsmith, J. 2003. Bidirectional LAO* algorithm. In *IICAI*, 980–992.

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

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *J. ACM* 32(3):505–536.

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

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

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

Kirkland, S. J.; Neumann, M.; and Xu, J. 2001. A divide and conquer approach to computing the mean first passage matrix for Markov chains via Perron complement reductions. *Journal of Numur linear algebra application* 8(5):287–295.

Nilson, N. J. 1980. *Principles of Artificial Intelligence*. Palo Alto, Ca.: Tioga Publishing Company.