Incremental Search Algorithms for Real-Time Decision Making

Joseph C. Pemberton and Richard E. Korf *

pemberto@cs.ucla.edu *korf@cs.ucla.edu*

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024

Abstract

We propose incremental, real-time search as a general approach to real-time decision making. We model real-time decision making as incremental tree search with a limited number of node expansions between decisions. We show that the decision policy of moving toward the best frontier node is not optimal, but nevertheless performs nearly as well as an expected-value-based decision policy. We also show that the real-time constraint causes difficulties for traditional best-first search algorithms. We then present a new approach that uses a separate heuristic function for choosing where to explore and which decision to make. Empirical results for random trees show that our new algorithm outperforms the traditional best-first search approach to real-time decision making, and that depth-first branch-and-bound performs nearly as well as the more complicated best-first variation.

Introduction and Overview

We are interested in the general problem of how to make real-time decisions. One example of this class of problems is air-traffic control. If a natural disaster such as an earthquake or severe winter storm forces an airport to close, then the air-traffic controllers must quickly decide where to divert the incoming airplanes. Once the most critical plane has been re-directed, the second most critical plane can be handled, etc., until all planes have landed safely. Another example is factory scheduling when the objective is to keep a bottleneck resource busy. In this case, the amount of time available to decide which job should be processed next is limited by the time required for the bottleneck resource to process the current job. Once a new job is scheduled, the time until the new job finishes processing can be used to decide on the next job. In general, this class of problems requires the problem solver to incrementally generate a sequence of time-limited decisions which consist of three sub-parts: simulating the effect of future actions (*e.g.*, what are the consequences of diverting plane P to airport A, what is the cost of processing job J next), deciding when to stop simulating and make a decision (*i.e.*, what the decision deadline is), and making a decision based on the available information and results of the simulations (*e.g.*, where a plane should land, what job to process next).

We first present an abstract real-time decision-making problem based on searching a random tree with limited computation time. We then address the questions of how to make decisions and how to simulate future actions. We next describe real-time adaptations of traditional search algorithms and identify two pathological behaviors of the best-first approach. We then present a new best-first algorithm that is an improvement over the traditional best-first approach. We also present results that compare the performance of our new algorithm with the traditional search algorithms for several random-tree problems. The last two sections discuss related work and conclusions.

A Real-Time Decision Problem

For this paper, we have focussed on an abstract real-time decision-making problem. The problem space is a uniform-depth search tree with constant branching factor. Each node of the tree corresponds to a decision point, and the arcs emanating from a node correspond to the choices (operators or actions) available for the decision at that node. Each edge has an associated random value chosen independently and uniformly from the continuous range $[0, 1]$. This value corresponds to the cost or penalty that the problem solver will incur if it chooses to execute the action associated with that edge. The *node_cost(x)* is the sum of the edge costs along the path from the root to node x. For this set of assumptions, the node costs along all paths from the root are monotonic nondecreasing. This random-tree problem can be characterized as *solution rich* since every leaf node of the tree is a solution to the problem. The objective of the problem solver is to find a lowest-cost leaf node given the computational constraints.

The real-time constraint is modeled as a constant number of node generations allowed per decision. A node generation is defined as the set of operations required to create and evaluate a node in the search tree,

*This research was supported by NSF Grant #IRI-9119825, and a grant from Rockwell International.

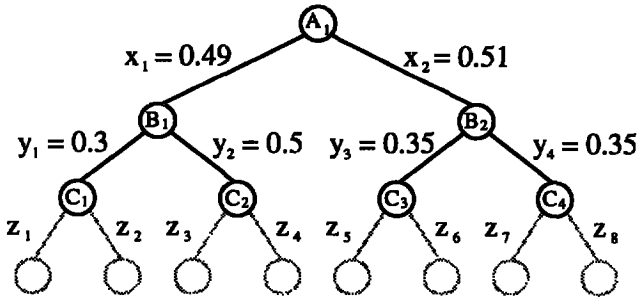

Figure 1: Example tree $T(b = 2, s = 2, u = 1)$. Which node is a better decision, B_1 or B_2?

whereas a node expansion consists of generating each child of the node. The node generation constraint is equivalent to having a deadline for each decision. We assume that the problem solver simulates the effect of future actions by expanding nodes (exploration), until the time available runs out, after which it chooses one of the children of the current root node as the new root node (decision making). This process is repeated, incrementally generating a complete solution. We have assumed that there is a deadline for each decision that determines when the problem solver should stop expanding nodes. The remaining questions are where to explore and how to make decisions. The main advantage of this model is that the branching factor, depth, and edge-cost distributions are under experimental control, making it a good problem for analysis and experimentation.

How should we make decisions?

When making decisions based on a partially explored problem space, the objective is to choose a child of the current root node that will minimize the cost of the resulting solution path. Since the complete path is typically not available before the choice must be made, the decision maker must estimate the expected cost of the complete path. We adopt the shorthand $T(b, s, u)$ to refer to an incremental search tree with branching factor b, explored search depth s, and unexplored depth u. Consider the tree ($T(b = 2, s = 2, u = 1)$) in figure 1. The first two levels have been explored, and the last level of the tree (the gray nodes and edges) has not been explored. At this point, the decision maker must choose between nodes B_1 and B_2. We assume that once the choice is made, then the problem solver will be able to expand the remaining nodes and complete the search path optimally. The reader is encouraged to stop and answer the following question: Should we move to node B_1 or node B_2?

Perhaps the most obvious answer is to move toward node B_1 because it is the first step toward the lowest-cost frontier node ($node\_cost(C_1) = .49 + .3 = .79$). This decision strategy, which is called minimin decision making in (Korf 1990), has been employed by others for single-agent search (*e.g.*, (Russell & Wefald 1991)), and is also a special case of the two-player minimax decision rule (Shannon 1950).

In fact, the optimal decision is to move to node B_2 because it has a lower expected total path cost. The expected minimum cost of a path ($E(MCP)$) through node B_1 or B_2 can be calculated given the edge costs in the explored tree and the edge cost distribution for the unexplored edges. For the edge costs in figure 1 we get $E(MCP(B_1)) = 1.066$, whereas $E(MCP(B_2)) = 1.06$. Thus node B_2 is the best decision because it has the lowest expected total path cost. Intuitively, a move to node B_1 relies on either z_1 or z_2 having a low cost, whereas a move to B_2 has four chances (z_5, z_6, z_7 and z_8) for a low final edge cost.

Note that this distribution-based decision strategy is only optimal for this particular case where the next round of exploration will expose the remainder of the problem-space tree. In general, an optimal incremental decision strategy will need to know the expected MCP distribution for the unexplored part of the tree given the current decision strategy, exploration strategy and computation bounds. This distribution is difficult to obtain in general. In addition, the general equation for the expected minimum complete path cost of a node will have a separate integral step for each frontier node in the subtree below it, thus the computational complexity of the distribution-based decision method is exponential in the depth of the search tree.

The question that remains is what is the cost in terms of solution quality if we use the minimin decision strategy instead of the $E(MCP)$ decision strategy? To answer this question, we implemented the $E(MCP)$ equations for the search tree in figure 1, and three other trees created from $T(b = 2, s = 2, u = 1)$ by incrementing either b, s, or u (see figure 2). We performed the following experiment on all four trees. For each trial, random values were assigned to each edge in the tree, and then the minimin and $E(MCP)$ decisions were calculated based on the edge costs in the explored part of the tree. The complete solution path costs were then calculated for both decision methods.

The results in table 1 show the percentage of the trials that the minimin and $E(MCP)$ decisions are equal to the first step on the optimal path, the solution-cost error with respect to the optimal path cost, and the percentage of the trials that each decision algorithm produced a lower-cost solution than the other (wins). In addition, the last column shows the percentage of the trials that the $E(MCP)$ method and the minimin method produced the same decision. The data is averaged over 1 million trials. Note that our implementation of the $E(MCP)$ for $T(b = 2, s = 3, u = 1)$ requires about 4500 lines of MAPLE[1] generated C-code.

These results show that although distribution-based decisions are slightly better than minimin decisions on

[1]MAPLE is an interactive computer algebra package developed at the University of Waterloo.

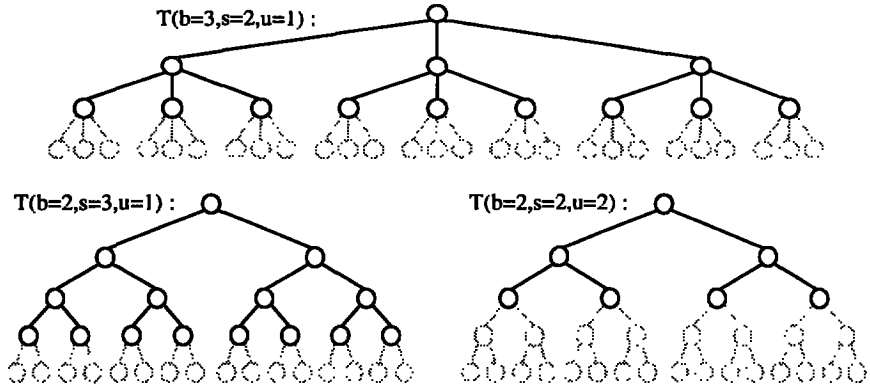

Figure 2: Three additional search trees

Tree			$E(MCP)$ decisions			Minimin decisions			Same
b	s	u	optimal	error	wins	optimal	error	wins	Decision
2	2	1	84.5%	3.1%	1.7%	84.2%	3.2%	1.5%	96.8%
3	2	1	81.9%	4.0%	2.5%	81.6%	4.2%	2.2%	95.3%
2	3	1	87.1%	1.8%	1.9%	86.8%	1.9%	1.6%	96.4%
2	2	2	80.8%	4.0%	2.2%	80.4%	4.1%	1.9%	95.9%

Table 1: Results comparing distribution-based decisions with minimin decisions.

the average, the average amount of the difference is very small and the difference only occurs less than 5% of the time. Since more code will be required for larger search trees, the $E(MCP)$ decision strategy is not practical. Fortunately, minimin decision making is a reasonable strategy, at least for small, uniform decision trees. Since there is not an appreciable change in the results for increased branching factor, explored search depth, or unexplored depth, we expect minimin to also perform well on larger trees.

What nodes should be expanded?

Exploration is the process of expanding nodes in the problem-space tree in order to simulate and evaluate the effect of future actions to support the decision-making process. Decision making consists of evaluating the set of nodes expanded by the exploration process, and deciding which child of the current decision node should become the next root node. The best exploration strategy will depend on the decision strategy being used and vice versa.

In general, the objective of an exploration policy is to expand the set of nodes that, in conjunction with the decision policy, results in the lowest expected path cost. For this paper, we have considered best-first exploration methods that use either a *node-cost* heuristic or an *expected-cost* heuristic to order the node expansions. We have also considered a depth-first exploration method. The *node-cost* and *expected-cost* heuristic functions were also used to evaluate frontier nodes in support of minimin decision making. These heuristic-based exploration and decision methods will be further discussed in the context of the specific algorithms presented in the next section.

Real-Time Search Algorithms

We have considered real-time search algorithms based on two standard approaches to tree search: depth-first branch-and-bound and best-first search. All algorithms presented are assumed to have sufficient memory to store the explored portion of the problem space.

Depth-first branch-and-bound (DFBnB) is a special case of general branch-and-bound that operates as follows. An initial path is generated in a depth-first manner until a goal node is discovered. The cost bound is set to the cost of the initial solution, and the remaining solutions are explored depth-first. If the cost of a partial solution equals or exceeds the current bound, then that partial solution is pruned since the complete solution cost cannot be less. This assumes that the heuristic cost of a child is at least as great as the cost of the parent (*i.e.*, the node costs are monotonic nondecreasing). The cost bound is updated whenever a new goal node is found with lower cost. Search continues until all paths are either explored to a goal or pruned. At this point, the path associated with the current bound is an optimal solution path.

The obvious way to apply DFBnB to the real-time search problem is to use a depth cutoff. If the cost of traversing a node that has already been generated is small compared with the cost of generating a new node, then the cost of performing DFBnB for an increasing

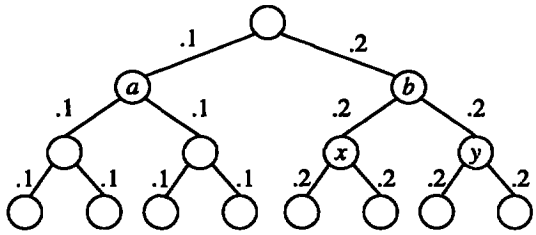

Figure 3: Example of best-first "swap pathology".

series of cutoff depths will be similar to the cost of performing DFBnB once using the last depth cutoff value. Thus we can iteratively increase the cutoff depth until time expires (iterative deepening), and then base the move decision on the backed-up values of the last completed iteration. One advantage of iterative deepening is that the backed-up values from the previous iteration can be stored along with the explored tree and used to order the search in the next iteration, thereby greatly improving the pruning efficiency over static ordering (*i.e.*, ordering based on static node costs). In some sense, this is the best-case situation for DFBnB.

One drawback of DFBnB is that it generates nodes that have a greater cost than the minimum-cost node at the cutoff depth. This means that DFBnB will expand more new nodes than a best-first exploration to the same search depth. Another drawback is that DFBnB is less flexible than best-first methods because its decision quality only improves when there is enough available computation to complete another iteration.

Traditional best-first search (BFS) expands nodes in increasing order of cost, always expanding next a frontier node on a lowest-cost path. This typically involves maintaining a heap of nodes to be expanded. The obvious way to apply BFS to a real-time search problem is to explore the problem space using a heuristic function to order the exploration, and when the available computation time is spent, use the same heuristic function to make the move decision. This single-heuristic approach has also been suggested by Russell and Wefald (Russell & Wefald 1991) for use in real-time search algorithms (*e.g.*, *DTA**).

We now discuss two pathological behaviors that can result from using a single heuristic function for both exploration and decision making. First, consider *node-cost BFS* which uses the heuristic function $f_{exp}(x) = f_{dec}(x) = node\_cost(x)$ for both exploration and decision making. When searching the tree in figure 3, *node-cost BFS* will first explore the paths below node a until the path cost equals 0.3. At this point, the best-cost path swaps to a path below node b. If the computation time runs out before the nodes labeled x and y are generated, then *node-cost BFS* will move to node b instead of node a, even though the expected cost of a path through node a is the lowest (for a uniform $[0, 1]$ edge cost distribution and binary tree). We call this behavior the best-first "swap pathology" because the best decision based on a monotonic, non-decreasing cost function will eventually swap away from the best expected-cost decision. This pathological behavior is a direct result of comparing node costs of frontier nodes at different depths in the search tree.

As an alternative to *node-cost BFS*, we considered *estimated-cost BFS* which uses an estimate of the expected total solution cost ($f_{exp}(x) = f_{dec}(x) = \hat{E}(total\_cost(x))$) for both exploration and decision making, in order to better compare the value of frontier nodes at different depths. The estimated total cost of a path through a frontier node x can be expressed as the sum of the node cost of x plus a constant c times the remaining path length, $f(x) = node\_cost(x) + c \cdot (tree\_depth - depth(x))$, where c is the expected cost per decision of the remaining path. This heuristic is only admissible when $c = 0$. In general, we don't know or can't calculate an exact value for c, so it must somehow be estimated. This *estimated-cost* heuristic function, which is used to estimate the value of frontier nodes, should not be confused with the $E(MCP)$ decision method, which combines the distributions of path costs to find the expected cost of a path through a child of the current decision node. In some sense, though, minimin decisions based on the *estimated-cost* heuristic function can be viewed as an approximation of the $E(MCP)$ decision method.

The problem with *estimated-cost BFS* is that when the exploration heuristic is non-monotonic, the exploration will stay focussed on any path it discovers with a non-increasing estimated-cost value. The result is often a very unbalanced search tree with some paths explored very deeply and others not explored at all.

Alternative Best-First Algorithms

Our approach to the real-time decision-making problem is to adapt the best-first method so that it avoids the pathological behaviors described above. The main idea behind our approach is to use a different heuristic function for the exploration and decision-making tasks.

Hybrid best-first search (*hybrid BFS*) avoids the pathological behaviors of a single-heuristic best-first search by combining the exploration heuristic of *node-cost BFS* with the decision heuristic of *estimated-cost BFS*. The intuition behind *hybrid BFS* is that the node-cost exploration will be more balanced than estimated-cost exploration, while the estimated-cost decision heuristic will avoid the swap pathology by effectively comparing the estimated total costs of frontier nodes at different depths.

Another best-first search variant is *Best-deepest BFS* which explores using the node-cost heuristic and moves toward the lowest-cost frontier node in the set of deepest frontier nodes. The idea behind *best-deepest BFS* is to mimic the way DFBnB makes decisions. In fact, if the move decision is toward the lowest-cost node in the set of deepest nodes that have already been expanded,

Algorithm	Exploration Rule: f_{exp}	Decision Rule: f_{dec}
DFBnB	$depth(x)$	$node\_cost(x)$
node-cost BFS	$node\_cost(x)$	$node\_cost(x)$
estimated-cost BFS	$\hat{E}(total\_cost(x))$	$\hat{E}(total\_cost(x))$
hybrid BFS	$node\_cost(x)$	$\hat{E}(total\_cost(x))$

Table 2: Table of algorithms considered.

then *best-deepest BFS* will make the same decision as DFBnB with the same depth bound. In our experiments, the policy of moving toward the lowest cost node in the set of deepest frontier nodes resulted in slightly lower cost solution paths. Since neither version of *best-deepest BFS* resulted in any significant performance improvement over *hybrid BFS*, we will not discuss them further.

Experimental Results

In order to evaluate the performance of *hybrid BFS*, we conducted a set of experiments on the random tree model described above. The expected cost of a single decision was estimated as the cost of a greedy decision ($c = E(greedy) = 1/(b+1)$ for a tree with branching factor b), and edge costs were chosen uniformly from the set $\{0, 1/2^{10}, ..., (2^{10}-1)/2^{10}\}$. We tested the four algorithms listed in figure 2 over a range of time constraints (*i.e.*, available generations per decision). The results in figure 4a show the average over 100 trials of the error per decision as a percentage of optimal solution path cost ($(solution\_cost - optimal\_cost)/optimal\_cost$), versus the number of node generations allowed per decision for a tree of depth 20 with branching factor 2. Figure 4b shows the same results for a branching factor of 4. Note that the results are presented with a log-scale on the horizontal axis. All algorithms had sufficient space to save the relevant explored subtree from one decision to the next. The leftmost data points correspond to a greedy decision rule based on a 1-level lookahead (*i.e.*, 2 or 4 generations per decisions). Similar results have been obtained for a variety of constant branching factors, tree depths, and also for random, uniformly distributed branching factors and deadlines.

The results indicate that *hybrid BFS* performs better than *node-cost BFS*, *estimated-cost BFS*, and slightly better than DFBnB. *Node-cost BFS* produces average solution costs that are initially higher than a greedy decision maker. This is due to the best-first "swap pathology", because the initial computation is spent exploring the subtree under the greedy root child, eventually making it look worse than the other root child. *Estimated-cost BFS* does perform better than greedy, but its performance quickly levels off well above the average performance of *hybrid BFS* or DFBnB. This is due to fact that the estimated-cost exploration heuristic is not admissible and does not generate a balanced tree to support the current decision. Thus *estimated-cost BFS* often finds a sub-optimal leaf node before consuming the available computations and then commits to decisions along the path to that node without further exploration. *Hybrid BFS* probably outperforms DFBnB because iterative-deepening DFBnB can only update its decision after it has finished exploring to the next depth bound, whereas best-first strategies can update the decision at any point in the exploration.

The results for best-first search are not surprising since *node-cost* and *exploration-cost BFS* were not expected to perform well. What is interesting is that a previous decision-theoretic analysis of the exploration problem (Russell & Wefald 1991) suggested that, for a given decision heuristic and the *single-step* assumption (*i.e.*, that the value of a node expansion can be determined by assuming that it is the last computation before a move decision), the best node to explore should be determined by the same heuristic function. Our experimental results and pathological examples contradict this suggestion.

Related Work

Our initial work was motivated by Mutchler's analysis of how to spend scarce search resources to find a complete solution path (Mutchler 1986). He has suggested a similar sequential decision problem and advocated the use of separate functions for exploration and decision making, thus our algorithms can be seen as an extension of his analysis. Our work is also related to Russell and Wefald's work on DTA* (Russell & Wefald 1991), and can be viewed as an alternative interpretation within their general framework for decision-theoretic problem solving. The real-time DFBnB algorithm is an extension of the minimin lookahead method used by RTA* (Korf 1990). Other related work includes Horvitz's work on reasoning under computational resource constraints (Horvitz 1987), and Dean and Boddy's work on anytime algorithms (Dean & Boddy 1988).

Conclusions

Incremental real-time problem solving requires us to reevaluate the traditional approach to exploring a problem space. We have proposed a real-time decision-making problem based on searching a random tree with limited computation time. We have also identified two pathological behaviors that can result when the same heuristic function is used for both exploration and deci-

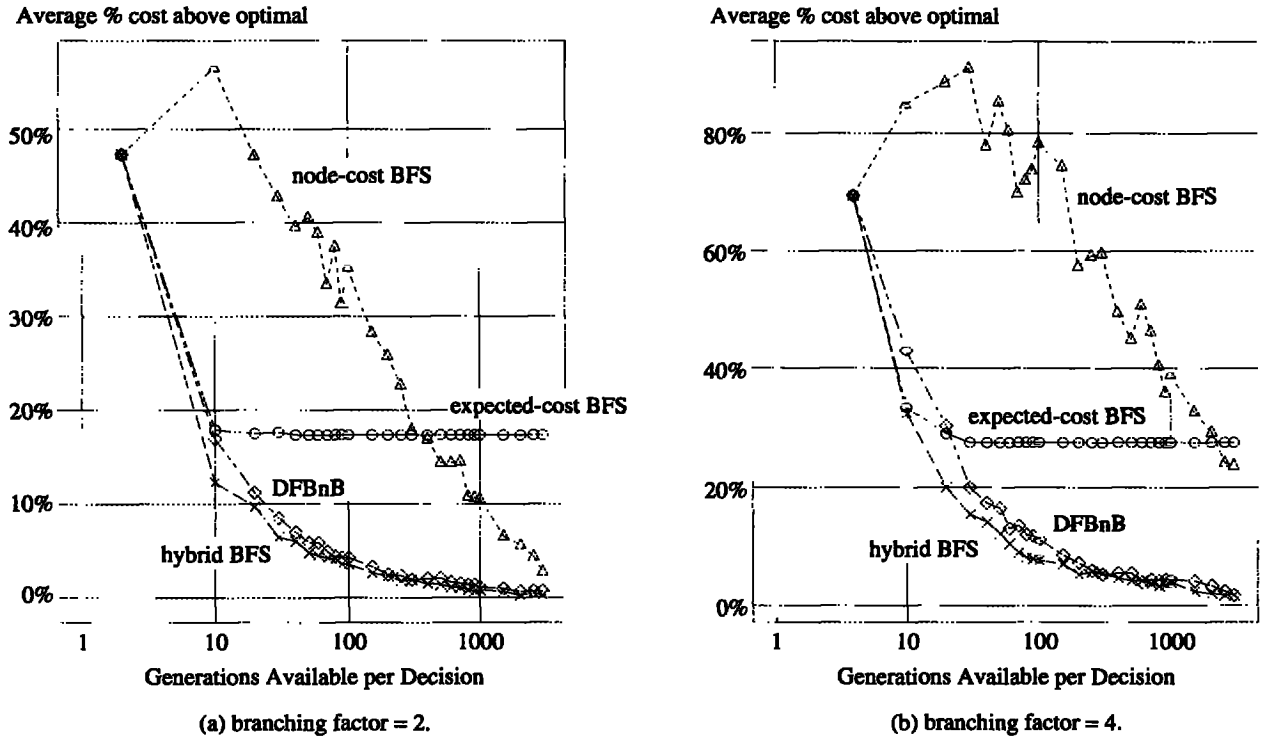

Figure 4: Percent cost above optimal versus generations available for a depth 20 random tree.

sion making. An alternative to minimin decision making, based on propagating minimum-cost path distributions, was presented. Preliminary results showed that minimin decision making performs nearly as well as the distribution-based method. An alternative best-first search algorithm was suggested that uses a different heuristic function for the exploration and decision-making tasks. Experimental results show that this is a reasonable approach, and that depth-first branch-and-bound with iterative deepening and node ordering also performs well. Although DFBnB did not perform as well as the new best-first algorithm, its computation overhead per node generation is typically smaller than for best-first methods because it doesn't have to maintain a heap of unexpanded nodes. The choice between DFBnB and a best-first approach will depend on the relative cost of maintaining a heap in best-first search to the overhead of iterative deepening and of expanding nodes with costs greater than the optimal node cost at a given cutoff depth.

Acknowledgements

We would like to thank Moises Goldszmidt, Eric Horvitz, David Mutchler, Mark Peot, David Smith, and Weixiong Zhang for helpful discussions, William Cheng for *tgif*, David Harrison for *xgraph*, and the Free Software Foundation for *gnuemacs* and *gcc*. Finally, we would like to thank the reviewers for their comments.

References

Dean, T., and Boddy, M. 1988. An analysis of time-dependent planning. In *Proceedings, 7th National Conference on Artificial Intelligence (AAAI-88), St. Paul, MN*, 49–54.

Horvitz, E. J. 1987. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings, 3rd Workshop on Uncertainty in AI, Seattle, WA*, 301–324.

Karp, R., and Pearl, J. 1983. Searching for an optimal path in a tree with random costs. *Artificial Intelligence* 21:99–117.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2–3):189–211.

Mutchler, D. 1986. Optimal allocation of very limited search resources. In *Proceedings, 5th National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA*, 467–471.

Pemberton, J. C., and Korf, R. E. 1993. An incremental search approach to real-time planning and scheduling. In *Foundations of Automatic Planning: The Classical Approach and Beyond*, 107–111. AAAI Press Technical Report #SS-93-03.

Russell, S., and Wefald, E. 1991. *Do the Right Thing.* Cambridge, MA: MIT Press.

Shannon, C. E. 1950. Programming a computer for playing chess. *Philosophical Magazine* 41(7):256–275.