Real-Time Heuristic Search

Richard E. Korf
Computer Science Department, University of California, Los Angeles, CA 90024-1596, USA.

ABSTRACT

We apply the two-player game assumptions of limited search horizon and commitment to moves in constant time, to single-agent heuristic search problems. We present a variation of minimax lookahead search, and an analog to alpha-beta pruning that significantly improves the efficiency of the algorithm. Paradoxically, the search horizon reachable with this algorithm increases with increasing branching factor. In addition, we present a new algorithm, called Real-Time-A, for interleaving planning and execution. We prove that the algorithm makes locally optimal decisions and is guaranteed to find a solution. We also present a learning version of this algorithm that improves its performance over successive problem solving trials by learning more accurate heuristic values, and prove that the learned values converge to their exact values along every optimal path. These algorithms effectively solve significantly larger problems than have previously been solvable using heuristic evaluation functions.*

1. Introduction

Heuristic search is a fundamental problem-solving method in artificial intelligence. For most AI problems, the sequence of steps required for solution is not known a priori but must be determined by a systematic trial-and-error exploration of alternatives. All that is required to formulate a search problem is a set of states, a set of operators that map states to states, an initial state, and a set of goal states. The task is to find a sequence of operators that map the initial state to a goal state. The performance of search algorithms is greatly improved by the use of a heuristic evaluation function. A heuristic evaluator is a function that is inexpensive to compute and estimates the relative merit of different states relative to the goal. Heuristic search has traditionally been employed in two types of domains under very different assumptions: single-agent problems and two-player games.

1.1. Single-agent heuristic search

Common examples of single-agent search problems are the Eight Puzzle and its larger relatives the Fifteen and Twenty-four Puzzles (see Fig. 1). The Eight

Artificial Intelligence **42** (1990) 189–211

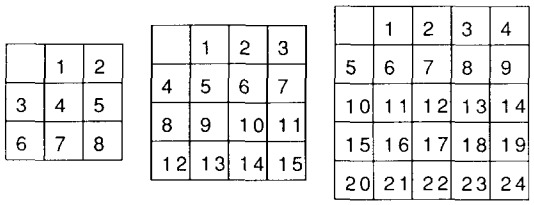

Fig. 1. Eight, Fifteen, and Twenty-four Puzzles.

Puzzle consists of a 3×3 square frame containing eight numbered square tiles and an empty position called the "blank". The legal operators slide any tile horizontally or vertically adjacent to the blank into the blank position. The task is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. A common heuristic function for this problem is called Manhattan Distance. It is computed by counting, for each tile not in its goal position, the number of moves along the grid it is away from its goal position, and summing these values over all tiles, excluding the blank.

A real-world example is the task of autonomous navigation in a network of roads, or arbitrary terrain, from an initial location to a desired goal location. The problem is typically to find a shortest path between the initial and goal states. A typical heuristic evaluation function for this problem is the Euclidean or air-line distance from a given location to the goal location.

The best known single-agent heuristic search algorithm is A* [1]. A* is a best-first search algorithm where the merit of a node, $f(n)$, is the sum of the actual cost in reaching that node from the initial state, $g(n)$, and the estimated cost of reaching the goal state from that node, $h(n)$. A* has the property that it will always find an optimal solution to a problem if the heuristic function never overestimates the actual solution cost. Its major drawback is that it requires exponential space in practice.

Iterative-Deepening-A* (IDA*) [2] is a modification of A* that reduces its space complexity from exponential to linear. IDA* performs a series of depth-first searches, in which a branch is cut off when the cost of its frontier node, $f(n) = g(n) + h(n)$, exceeds a cutoff threshold. The threshold starts at the heuristic estimate of the initial state, and is increased each iteration to the minimum value that exceeded the previous threshold, until a solution is found. IDA* has the same property as A* with respect to solution optimality, and expands the same number of nodes, asymptotically, as A* on an exponential tree, but uses only linear space.

A serious drawback of both A* and IDA*, however, is that they take exponential time to run in practice. This is an unavoidable cost of obtaining optimal solutions, and restricts the applicability of these algorithms to relatively small problems in practice. For example, while A* with the Manhattan

Distance heuristic function can solve the Eight Puzzle, and IDA* can solve the Fifteen Puzzle, any larger puzzle is intractable on current machines. As observed by Simon [3], however, it is relatively rare that optimal solutions are actually required, but rather near-optimal or "satisficing" solutions are usually perfectly acceptable for most real-world problems.

A related drawback of A* and IDA* is that they must search all the way to a solution before making a commitment to even the first move in the solution. The reason is that an optimal first move cannot be guaranteed until the entire solution is found and shown to be at least as good as any other solution. As a result, A* and IDA* are run to completion in a planning or simulation phase before the first move of the resulting solution is executed in the real world.

1.2. Two-player games

In contrast, heuristic search in two-player games adopts an entirely different set of assumptions. The canonical example is a program that plays chess. The classical two-player game algorithm is full-width, fixed-depth minimax search with alpha-beta pruning [4,5]. Since the chess tree is so large, 10^{43} positions by one estimate [6], the idea of searching all the way to checkmate is never entertained. Rather, the problem solver faces a limited search horizon, and makes suboptimal decisions.

A related characteristic is that in a two-player game, actions must be committed before their ultimate consequences are known. For example, in a chess tournament, moves must be made within a certain time limit, and can never be revoked.

While research in single-agent problems has focussed on increasing the size of problems for which optimal solutions can be found, research in two-player games has concentrated on making the best decisions possible given the limited amount of computation available between moves. Whereas the focus on optimal solutions has seriously limited the applicability of heuristic search in single-agent problems, the search paradigm has been spectacularly successful in two-player games. For example, the Hitech machine plays chess better than all but one-half of one percent of rated human tournament players [6].

1.3. Real-time single-agent search

The goal of this research is to apply the assumptions of two-player games, namely limited search horizon and commitment to moves in constant time, to single-agent heuristic searches.

A limited search horizon may be the result of computational or informational limitations. For example, even a slightly larger version of the Fifteen Puzzle, such as the Twenty-four Puzzle, would preclude searching all the way to the goal state from an initial state, due to the computation required. Alternatively, in the case of autonomous vehicle navigation without the benefit of completely

detailed maps, the search horizon is due to the information limit of how far the vehicle sensors can see ahead.

The commitment to moves in constant time is a further constraint. In a real-world setting this corresponds to executing a physical action, such as moving a chess piece. In our experiments, however, while actions are not physically executed, they are simulated as if they were. For example, when backtracking occurs, both the original moves and the backtracking moves are counted in the solution lengths, much as the odometer of a vehicle records every mile driven, whether ultimately productive or not. Furthermore, the only information available to the algorithm is information that would be available had the simulated moves actually been executed. For example, the information available to the sensors of a vehicle depends on its location.

Preliminary results from this research have appeared in [7] and [8].

2. Minimin Lookahead Search

The first step is to specialize the minimax algorithm for two-player games to the case of a single problem-solving agent. We call this algorithm *minimin* search in contrast to minimax. At first we will assume that all edges have the same cost.

The algorithm searches forward from the current state to a fixed depth determined by the computational or information resources available for a single move, and applies the heuristic evaluation function to the nodes at the search frontier. Whereas in a two-player game these values are minimaxed up the tree to account for alternate moves among the players, in the single-agent setting, the backed-up value of each node is the minimum of the values of its children, since the single agent has control over all moves. Once the backed-up values of the children of the current state are determined, a single move is made in the direction of the best child, and the entire process is repeated. The reason for not moving directly to the frontier node with the minimum value is to employ a strategy of least commitment, under the assumption that after committing the first move, additional information from an expanded search frontier may result in a different choice for the second move than was anticipated by the first search.

Note that the search proceeds in two quite different, but interleaved modes. Minimin lookahead search occurs in a planning mode, where the postulated moves are not actually executed, but merely simulated in the machine. After one complete lookahead search, the best move found would actually be executed in the real world by the problem solver. This is followed by another lookahead simulation from the new current state, and another actual move, etc.

In the more general case where the operators have non-uniform cost, we must take into account the cost of the path from the current state to the

frontier, in addition to the heuristic estimate of the remaining cost. To do this we adopt the A* cost function of $f(n) = g(n) + h(n)$. The algorithm looks forward a fixed number of moves, and backs up the minimum $f(n)$ value of each frontier node. An alternative scheme to searching forward a fixed number of moves would be to search forward to a fixed $g(n)$ cost. We adopt the former algorithm under the assumption that in the planning phase the computational cost is a function of the number of moves rather than the actual execution costs of the moves.

If a goal state is encountered before the search horizon, then the path is terminated and a heuristic value of zero is retuned for the goal state. Conversely, if a path ends in a non-goal dead-end before the horizon is reached, then a heuristic value of infinity is returned for the dead-end node, guaranteeing that the path will not be chosen.

3. Alpha Pruning

An obvious question is whether every frontier node must be examined to find the one with minimum cost, or does there exist an analog of alpha-beta pruning that would allow the same decisions to be made while exploring substantially fewer nodes? If our algorithm uses only frontier node evaluations, then a simple adversary argument establishes that no such pruning algorithm can exist, since to determine the minimum cost frontier node requires examining every one.

However, if we allow heuristic evaluations of interior nodes, then substantial pruning is possible if the cost function is *monotonic*. A cost function f is monotonic if it never decreases along a path away from the root. In other words, if n' is a child of n, then $f(n') \geq f(n)$, or $g(n') + h(n') \geq g(n) + h(n)$. Since $g(n') = g(n) + k(n,n')$, where $k(n,n')$ is the cost of the edge from n to n', we have $g(n) + k(n,n') + h(n') \geq g(n) + h(n)$, or $k(n,n') + h(n') \geq h(n)$. This property is referred to as *consistency* of h [9]. Thus, monotonicity of f is equivalent to consistency of h.

In practice, almost all naturally occurring heuristic functions are consistent, including Manhattan Distance and Euclidean Distance. The reason is that consistency is a simple variation of the familiar triangle inequality characteristic of all cost functions. Thus, monotonicity of f is usually not a restriction in practice.

A monotonic f function allows us to apply branch-and-bound to significantly decrease the number of nodes examined without effecting the decisions made. The algorithm, which we call *alpha pruning* by analogy to alpha-beta pruning, is as follows: In the course of generating the tree, maintain in a variable α the lowest f value of any node encountered on the search horizon so far. As each interior node is generated, compute its f value and terminate search of the corresponding branch when its f value equals or exceeds α. The reason is that

since f is monotonic, the f values of the frontier nodes below that node can only be greater than or equal to the cost of that node, and hence cannot be lower than the value of the frontier node responsible for the current value of α. As each frontier node is generated, compute its f value as well and if it is less than α, replace α with this lower value and continue the search.

In practice, α is reinitialized to infinity at the beginning of the search of each immediate child of the current state. While this is strictly not necessary to determine the best move, and results in some loss of efficiency, it is required in order to determine an actual value for each child. Otherwise, all the frontier nodes below a particular child may be cutoff, leaving it with no backed-up value. Values for each child are required for RTA*, to be described below.

3.1. Efficiency of alpha pruning

Figure 2 shows a comparison of the total number of nodes examined as a function of search horizon for several different sliding tile puzzles, including the 3×3 (8 tile), 4×4 (15 tile), 5×5 (24 tile), and 10×10 (99 tile) versions.

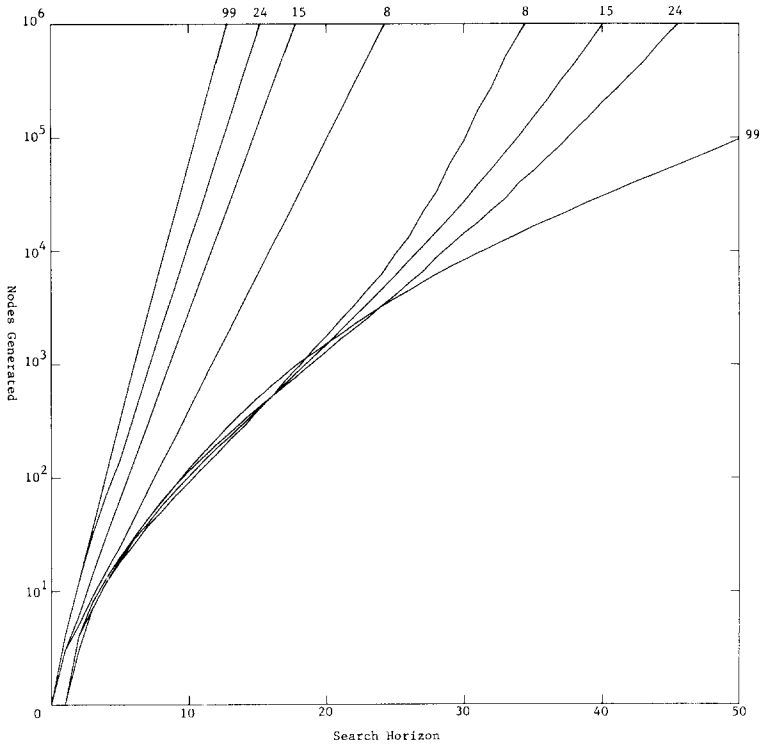

Fig. 2. Search horizon vs. nodes generated for brute-force and alpha pruning search.

The straight lines on the left represent brute-force search with no pruning and indicate branching factors of 1.732, 2.130, 2.368, and 2.790 for the Eight, Fifteen, Twenty-four, and Ninety-nine puzzles, respectively. The curved lines to the right represent the number of nodes generated with alpha pruning using the Manhattan Distance heuristic function. In each case, the values are the averages of 1000 random solvable initial states.

One remarkable aspect of this data is the effectiveness of alpha pruning. For example, if we fix the available computation at one million nodes per move, requiring about a minute of CPU time on a one million instruction per second machine, then alpha pruning extends the reachable Eight Puzzle search horizon almost 50 per cent from 24 to 35 moves, more than doubles the Fifteen Puzzle horizon from 18 to 40 moves, and triples the Twenty-four Puzzle horizon from 15 to 45 moves. Fixing the amount of computation at 100,000 nodes per move, the reachable Ninety-nine puzzle search horizon is multiplied by a factor of five from 10 to 50 moves. In comparison, even under perfect ordering, alpha-beta pruning only doubles the effective search horizon.

Even more surprising, however, is the fact that given a sufficiently large amount of computation, the search horizon achievable with alpha pruning actually *increases* with increasing branching factor! In other words, we can search significantly deeper in the Fifteen Puzzle than in the Eight Puzzle, and even deeper in the Twenty-four Puzzle, in spite of the fact that the brute-force branching factors are larger. Another way of viewing this is that the effective branching factor is smaller for the larger problems.[1]

3.2. An analytic model

Is this paradoxical effect an artifact of the sliding tile puzzles, or a more general phenomenon? The standard analytic model for the analysis of alpha-beta pruning is a tree with uniform branching factor and uniform depth, whose leaf values are chosen independently from a common distribution. For the analysis of alpha pruning, however, we need a model that assigns values to interior nodes as well and ensures the monotonicity property along a path. Such a model can be found in [10]. It consists of a tree of uniform branching factor and depth where the edges are assigned a value of zero or one independently with some probability p. The value of any node of the tree is the sum of the edge costs from the root to that node.

At least at a superficial level, this model characterizes our problem. Moving any tile of the Eight Puzzle either increases its h value by one or decreases it by one. Since every move increases the g value by one, the $f = g + h$ value either increases by two or stays the same. If we divide these increments by 2, and subtract the h value of the root from every node, we get edge costs of zero or

[1]This observation is due to Stuart Russell.

one whose sum represents the f value of a node. Initially, the probability of an increment in f is one half.

What does this analytic model predict about the performance of alpha pruning? Results for this model come from the theory of *branching processes* [11]. In particular, if the expected number of zero cost edges in a typical expansion of a node is less than one, which will occur when the product of the branching factor and the probability of a zero is less than one, then the minimum leaf value will grow linearly with tree depth. Conversely, if the expected number of zero-cost edges is greater than one, however, then with high probability the minimum leaf value will remain bounded even as the tree depth goes to infinity. Thus, increasing the branching factor while holding constant the probability of a zero tends to reduce the cost of the minimum leaf value.

After a single iteration of minimin search with alpha pruning, the value of alpha is equal to the cost of the minimum frontier node. In practice, however, alpha tends to reach its minimum value early in the search. Increasing the branching factor tends to reduce the value of alpha, since increasing the size of the frontier set tends to reduce its minimum value. The total number of nodes generated decreases with decreasing alpha, since the expected length of any particular path is alpha divided by the probability of a one-cost edge. This suggests why increasing the branching factor may decrease the number of nodes generated in searching to a certain depth. Similarly, increasing the branching factor while holding the number of nodes constant may increase the reachable search horizon. Empirically, this effect more than compensates for the increased branching factor.

Unfortunately, this analytic model is only valid in the limit where the problem size grows to infinity. In that case, one could argue that the probability of increasing or decreasing Manhattan Distance with a given move would remain at one-half and be independent of other moves. For practical size problems, however, a string of moves that decrease the Manhattan Distance tends to make an increasing move more likely. The mismatch between the analytic model and the actual data is evidenced by the fact that the model predicts that for the Fifteen and larger size puzzles, the minimum leaf value should remained bounded with increasing depth, whereas in practice it grows linearly. The analytic determination of the effective branching factor of alpha pruning is an open problem.

3.3. Node ordering

As in alpha-beta pruning, the efficiency of alpha pruning ought be improved by *node ordering*. The idea is to order the successors of each interior node in increasing order of their immediate f values, as opposed to their eventual backed-up values, hoping to achieve a low value of alpha early and hence

prune more branches sooner. This algorithm was run on 1000 random Fifteen Puzzle instances up to a search horizon of forty moves with disappointing results. The average reduction in number of nodes generated was only 5 per cent, which was much less than the additional overhead per node required to do the ordering. The result was that the program ran slower with node ordering than without. At least two possible explanations suggest themselves for this result. One is that ordering decisions based on immediate values may not be very effective at deep search horizons. The other is that even without node ordering, small values of alpha may be achieved fairly quickly anyway, thus limiting the potential effectiveness of any node ordering.

4. Other Real-Time Search Algorithms

The key property of minimin search is that it takes constant time to run. By modifying the search horizon, the value of that constant can be set by the constraints of the problem at hand. Note that even though its complexity is an exponential function of the search horizon, the search horizon is bounded by a constant and hence so is the running time of the algorithm. This property, however, is not unique to this algorithm, and is shared by at least two other more obvious algorithms.

The first may be called time-limited-A*. The idea is to run A* until time runs out, and then to make a single move in the direction of the best node on the OPEN list, breaking ties in favor of smaller h values. The main drawback of this algorithm is the same as that of A*: its memory requirement is exponential in the search depth and thus in practice it will exhaust the available memory, unless the amount of time allotted per move is quite small.

To get around the memory limitation without sacrificing constant time complexity, one could run an algorithm called threshold-limited-IDA*. The idea is to run a single iteration of IDA* with a threshold which is always a constant amount greater than the heuristic estimate of the current state. At the end of the iteration, all the frontier nodes will have the same f value, and hence a move would be made in the direction of the minimum h value.

The relation between these two algorithms is as follows. In practice, large numbers of nodes often have the same f values. As a result, A* can be viewed as running through cycles of expanding all nodes of a given value before expanding the first node with the next higher value. If time-limited-A* timed out exactly in between two such cycles, it would make the same decision as an iteration of threshold-limited-IDA* with the corresponding threshold. If it timed out in the middle of such a cycle, it would make a decision based on an incomplete search, having arbitrarily expanded some nodes of a given cost without expanding others. Thus, the essential difference between the two algorithms is one of granularity of stopping points, at the cost of possibly incomplete searches.

Threshold-limited-IDA* and minimin with alpha pruning are also very similar. The only difference is that in alpha pruning the cutoff threshold is dynamically determined by the minimum value of the frontier nodes, as opposed to being static and set in advance by threshold-limited-IDA*. Since the decisions made by minimin with alpha pruning are independent of node order, consider an order where the cost of the first leaf generated equals the final value of alpha. In that case, minimin with alpha pruning will generate the same fraction of the tree that threshold-limited-IDA* would with the threshold equal to alpha, and they would make a move toward the same node, viewed as a maximum g node in one case, and a minimum f node in the other case. The essential difference is again one of granularity of stopping points since there will be more feasible search horizons than alpha thresholds.

Thus, all three of these algorithms are essentially equivalent in terms of the decisions they make, and differ primarily in the granularity of computation per decision. Note that if the search horizon is deep enough to include a goal state, then the minimum cost frontier node will be a goal and the path to it will be an optimal one. Thus, A* and IDA* can roughly be viewed as a special case of minimim lookahead search when the search horizon encompasses a goal.

Furthermore, the effect of any one of these algorithms is to compute a backed-up heuristic value for each root node it is applied to. Thus, any of these algorithms can be encapsulated and viewed as simply a more accurate heuristic evaluation function. This gives rise to an entire family of heuristic functions, one for each search horizon, time-limit, or threshold, that vary in accuracy and computational complexity. The greater the search, the more accurate the evaluation, and the more expensive it is to compute. For the remainder of this paper, we will adopt this point of view and simply refer to an evaluation function, without explicitly mentioning the fact that it may be computed using minimin search with alpha pruning, time-limited A*, or threshold-limited IDA*.

5. Real-Time-A*

So far, we have addressed the issue of how to make individual move decisions, but not how to make a sequence of decisions to arrive at a solution. The obvious approach of simply repeating the minimin algorithm for each decision won't work; the problem solver will eventually move back to a previously visited state and loop forever. Simply excluding previously visited states won't work either since all successors of the current state may have already been visited. Furthermore, since decisions are based on limited information, an initially promising direction may appear less favorable after gathering additional information in the process of exploring it, thus motivating a return to a previous choice point. The challenge is to prevent infinite loops while permitting backtracking when it appears favorable, resulting in a form of single-trial learning through exploration of the problem space.

The basic principle of rationality is quite simple. One should backtrack to a previously visited state when the estimate of solving the problem from that state plus the cost of returning to that state is less than the estimated cost of going forward from the current state. Real-Time-A* (RTA*) is an efficient algorithm for implementing this basic strategy. While the minimin lookahead algorithm is an algorithm for controlling the planning phase of the search, RTA* is an algorithm for controlling the execution phase. As such, it is independent of the planning algorithm chosen.

In RTA*, the merit of a node n is $f(n) = g(n) + h(n)$, as in A*. However, unlike A*, the interpretation of $g(n)$ in RTA* is the actual distance of node n from the current state of the problem solver, rather than from the original initial state. The key difference between RTA* and A* is that in RTA* the merit of every node is measured relative to the current position of the problem solver, and the initial state is irrelevant. RTA* is simply a best-first search given this slightly different cost function. In principle, it could be implemented by storing on an OPEN list the h values of all previously visited states, and every time a move is made, updating the g values of all states on OPEN to accurately reflect their actual distance from the new current state. Then at each move cycle, the problem solver selects next the state with the minimum $g + h$ value, moves to it, and again updates the g values of all nodes on OPEN.

The drawbacks of this naive implementation are: 1) the time to make a move is linear in the size of the OPEN list, since the g value of every node must be updated, 2) it is not clear exactly how to update the g values, and 3) it is not clear how to find the path to the next destination node chosen from OPEN. These problems, however, can be solved in constant time per move using only local information in the graph.

The algorithm maintains in a hash table a list of those nodes that have been visited by an actual move of the problem solver, together with an h value for those nodes. At each cycle of the algorithm, the current state is expanded, generating its neighbors, and the heuristic function, possibly augmented by lookahead search, is applied to each state which is not in the hash table. For those states in the table, the stored value of h is used instead. In addition, the cost of the edge to each neighboring state is added to this value, resulting in an f value for each neighbor of the current state. The node with the minimum f value is chosen for the new current state and a move to that state is executed. At the same time, the previous current state is stored in the hash table, and associated with it is the second best f value. The second best f value is the best of the alternatives that were not chosen, and represents the estimated h cost of solving the problem by returning to this state, from the perspective of the new current state. If there is a tie among the best values, then the second best will equal the best. The algorithm continues until a goal state is reached.

Note that RTA* only requires a single list of previously visited nodes. The size of this list is linear in the number of moves actually made, since the

lookahead search saves only the value of its root node. Furthermore, the running time is also linear in the number of moves made. The reason for this is that even though the lookahead requires time that is exponential in the search depth, the search depth is bounded by a constant.

One can construct examples to show that RTA* could backtrack an arbitrary number of times over the same nodes. For example, consider the graph in Fig. 3, where the initial state is node a, all the edges have unit cost, and the values at each node represent the original heuristic estimates of those nodes. Since lookahead only makes the example more complicated, we will assume that no lookahead is done to compute the h values. Starting at node a, nodes b, c, and d are generated and evaluated at $f(b) = g(b) + h(b) = 1 + 1 = 2$, $f(c) = g(c) + h(c) = 1 + 2 = 3$, and $f(d) = g(d) + h(d) = 1 + 3 = 4$. Therefore, the problem solver moves to node b, and stores with node a the information that $h(a) = 3$, the second best f value. Next, nodes e and i are generated and evaluated at $f(e) = g(e) + h(e) = 1 + 4 = 5$, $f(i) = g(i) + h(i) = 1 + 5 = 6$, and using the stored h value of node a, $f(a) = g(a) + h(a) = 1 + 3 = 4$. Thus, the problem solver moves back to node a, and stores $h(b) = 5$ with node b. At this point, $f(b) = g(b) + h(b) = 1 + 5 = 6$, $f(c) = g(c) + h(c) = 1 + 2 = 3$, and $f(d) = g(d) + h(d) = 1 + 3 = 4$, causing the problem solver to move to node c, storing $h(a) = 4$ with node a. The reader is urged to continue the example to see that the problem solver continues to backtrack over the same terrain, until a goal node is reached. The reason it is not in an infinite loop is that each time it changes direction, it goes one step further than the previous time, and gathers more information about the space. This seemingly irrational behavior is produced by a rational policy in the presence of a limited search horizon, and a somewhat pathological set of heuristic values.

Note that while consistency of the heuristic function is required for alpha pruning in minimin lookahead search, RTA* places no such constraint on the values returned by the heuristic function. Of course, the more accurate the heuristic function the better the performance of RTA*, but even with no heuristic information at all, RTA* will eventually find a solution, as the result below will show.

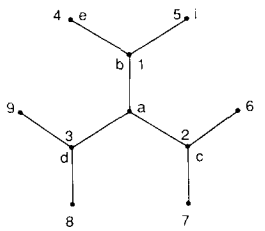

Fig. 3. Real-Time-A* example.

5.1. Completeness of RTA*

Under what conditions is RTA* guaranteed to eventually find a goal state? One caveat is that in an infinite problem space, unlike A*, RTA* may not find a solution. For example, consider a simple straight-line graph where every node has the same h value, such as zero. Once the algorithm arbitrarily chooses one direction, it will continue in that direction forever, regardless of whether or not the goal lies in that direction. A second caveat is that even in a finite problem space, if there are one-way edges with dead-ends, RTA* could end up in a part of the graph from which the goal node is no longer reachable. Furthermore, if there are cycles in the graph with zero or negative cost, RTA* could remain in such a cycle forever without increasing its estimate of reaching the goal. Finally, the heuristic values returned must be finite, since a barrier of infinite heuristic values could surround the goal, making it inaccessible to the algorithm. These are the only restrictions, however, and for all other problems, RTA* is guaranteed to eventually find a solution if one exists. Note that the only constraint placed on the heuristic evaluation function is that it return finite values.

Theorem 1. *In a finite problem space with positive edge costs and finite heuristic values, in which a goal state is reachable from every state, RTA* will find a solution.*

Proof. Assume the converse, that there exists a path to a goal state, but that RTA* will never reach it. In order for that to happen in a finite problem space, there must exist a finite cycle that RTA* travels forever, and that does not include the goal state. Also, since a goal state is reachable from every state, there must be at least one edge leading away from the cycle. We will show that RTA* must eventually leave any such cycle. At any point in the algorithm, every node in the graph has a value associated with it, either explicitly or implicitly. For the nodes already visited, this will be their value in the hash table, and for the unvisited nodes it will be their original heuristic evaluation. Consider an individual move made by RTA*. It reads the value of each of its neighbors, adds the corresponding positive edge costs, stores the second lowest of these values with the current state, and moves to the new state. Thus, the value stored with the old state must be greater than the minimum of the values of its neighbors at the time, since positive edge costs were added to each neighboring value. Now consider a state on the hypothesized infinite cycle with the lowest value. By definition, its value is less than or equal to the value of its neighbors. When the algorithm reaches such a state, it increases its value, since the value written is greater than the minimum of its neighboring values. Furthermore, no value can be written that is less than or equal to the previous lowest value, since every value written is greater than that of its neighbors. Thus, every trip of the algorithm around the cycle increases the lowest value on

the cycle. Therefore, the values of all nodes on the cycle increase without bound. At some point, since the heuristic values are finite, the value of a node on a path that leads away from the cycle will be lower than the competing neighbor on the cycle. At that point, the algorithm will leave the cycle, violating our assumption of an infinite loop. Thus, there can be no infinite loops in a subset of the problem graph. Therefore, in a finite problem space, every node, including a goal node if it exists, must eventually be visited. When the algorithm visits a goal, it will terminate successfully. □

5.2. Correctness of RTA*

Since RTA* is making decisions based on limited information, the best we can say about the quality of decisions made by the algorithm is that RTA* makes optimal moves relative to the part of the search space that it has seen so far. This is true in the special case of a tree, but does not extend to graphs with cycles. We begin with some definitions required to precisely state and prove our result. The key difference between these definitions and the standard ones for A* is that in our case both g and h values are relative to the current state of the problem solver.

As usual, a node is *generated* when the data structure corresponding to that node is created in the machine. A node is *expanded* when all of its children are generated. Define the search *frontier* as the set of all nodes that have been generated but not expanded since the beginning of the search, including all lookahead phases. This is analogous to the OPEN list in A*. Let $g_x(n)$ be the cost of the path in the graph from node x to node n, which is unique in a tree. Note that $g(n)$ in A* is $g_s(n)$ where s is the initial state. If n is a frontier node, let $h(n)$ be the heuristic static evaluation of node n. If n is an interior node, or in other words both generated and expanded, and x is the current state of the problem solver, define $h_x(n)$ as the minimum, over all frontier nodes m below node n in the tree rooted at node x, of $g_n(m) + h(m)$. Finally, define $f_x(n)$ as $g_x(n) + h_x(n)$.

Theorem 2. *Each move made by RTA* on a tree is along a path whose estimated cost of reaching a goal is a minimum, based on the cumulative search frontier at the time.*

Proof. We will show by induction on the number of moves made by the algorithm that when x is the current state of the problem solver, $h_x(n)$ is the value stored by RTA* with each previously visited node n in the tree. Consider the first move of the algorithm, starting from an initial state s, before any values are stored in the table. Minimin search generates a tree rooted at s and terminating in a set of frontier nodes m at the search horizon, possibly including dead-ends and goal states within the horizon. Associated with each frontier node m is a value $f_s(m) = g_s(m) + h(m)$. In the case of a dead-end, this

value will be infinity, and in the case of a goal node it will be $g_s(m)$. The backed-up value associated with each child c_i of the start state s is the minimum value of $f_s(m)$ over all frontier nodes m below c_i in the tree rooted at s. Note that alpha pruning has no effect on the values returned by the lookahead search, but simply computes them more efficiently. The problem solver moves to the child with the minimum value, say c_1 without loss of generality, which is a move along a path whose estimated cost of reaching the goal is a minimum. This move changes the root of the tree from s to c_1, as if the tree was picked up by node c_1. This leaves all parent-child relationships the same, except that c_1 is now the parent of s instead of vice-versa. Therefore, the children of s are now c_i for $i > 1$. At the same time, the second best $f_s(m)$ value is stored with state s. The second best value is the minimum value of $f_s(m)$ for all frontier nodes m below c_i for $i > 1$, which is the minimum $f_s(m)$ value of all frontier nodes m below s in the tree rooted at c_1. Thus, the value stored with s is $h_{c1}(s)$ after the move to c_1. For the induction step, assume that at a given point in the algorithm, the state of the problem solver is x, and the value stored with each node y in the hash table is $h_x(y)$. For each of the children c_i of x, if c_i is not in the table, minimin search will be used to compute $f_x(c_i)$. Otherwise, $h_x(c_i)$ will be read from the table, and $g_x(c_i)$ will be added to it, to produce $f_x(c_i)$. In either case, the correct value of $f_x(c_i)$ will be returned. Then by following exactly the same argument as above, replacing s with x, the value stored with x will be $h_{ci}(x)$ after a move to c_i. Finally, RTA* always moves from its current state x to the the neighbor c_i for which $f_x(c_i)$ is a minimum. This is the same as moving along a path whose estimated cost to the goal is a minimum, given the cumulative search frontier at that point. □

This result is for the special case of trees and does not extend to general graphs with cycles. Both minimin lookahead search and RTA* must be significantly modified to guarantee locally optimal decisions on a graph. This is an object of current research. The guarantee of finding a solution (Theorem 1), however, is not specialized to trees, but applies to graphs as well. Thus, these algorithms can be applied to graphs, with the only caveat being that some decisions may not be locally optimal. For example, the empirical results reported below are for the above algorithms applied to the sliding tile puzzles, whose problem spaces are graphs with cycles.

5.3. Tie-breaking

In practice, this algorithm often faces decisions among equally good alternatives, as evaluated by the heuristic function. While the specification of the algorithm only requires that it choose one of the best alternatives, the manner in which these ties are broken can have a large effect on the quality of decisions made.

A naive implementation would typically resolve ties by the arbitrary order in

which moves are generated. For example, in a road navigation application, the roads leading from a given intersection may be generated in left-to-right order from the perspective of the problem solver entering the intersection. Without an explicit tie-breaking mechanism, ties are likely to be resolved by always choosing the leftmost (or rightmost) road leading from the intersection. The effect of this is to introduce a systematic bias to decisions, and if ties are frequent enough, the problem solver will tend to spiral around the initial state. Alternatively, if roads from an intersection were generated in order of the compass points, there would be a bias in a particular linear direction, which could be equally detrimental to performance in the worst case.

The only way to remove such systematic biases, without further information, is to break ties randomly. A more sophisticated approach to tie-breaking would use a secondary search to discriminate between tied alternatives. This secondary search, however, must also be depth-limited and then a tie-breaking rule might still have to be applied at the secondary search horizon. A completely general and satisfying solution to this problem is elusive, particularly since often several alternatives may in fact be equally good.

5.4. Solution quality

In addition to efficiency and completeness of the algorithm, the quality of solutions generated by RTA* is of central concern. The most important factor affecting solution quality is the accuracy of the heuristic function itself. In addition, for a given heuristic function, solution quality increases with increasing search horizon.

One thousand solvable initial states were randomly generated for each of the Eight, Fifteen, and Twenty-four Puzzles. For each initial state, RTA* with minimin search and alpha pruning was run for various search depths using the Manhattan Distance heuristic function. The search depths ranged from one to twenty-five moves. Ties were broken randomly, and the resulting number of moves made was recorded, including all backtracking moves. Figure 4 shows a graph of the average solution length over all problem instances versus the depth of the search horizon for each of the three puzzles.

The overall shape of the curve confirms the intuition that increasing the search horizon decreases the resulting solution cost. For the Eight Puzzle, searching to a depth of 10 moves produces solution lengths (44) that are only a factor of two greater than optimal (22). In the case of the Fifteen Puzzle, finding solution lengths that are two times optimal (53 moves) requires searching to a depth of 23 moves. While no practical techniques exist for computing optimal solutions for the Twenty-four Puzzle, we can reasonably estimate the length of such solutions at about 100 moves. Searching to a depth of 25 moves produces average solution lengths of 433 moves.

One unusual aspect of the data is the spike at depth three on both the Eight

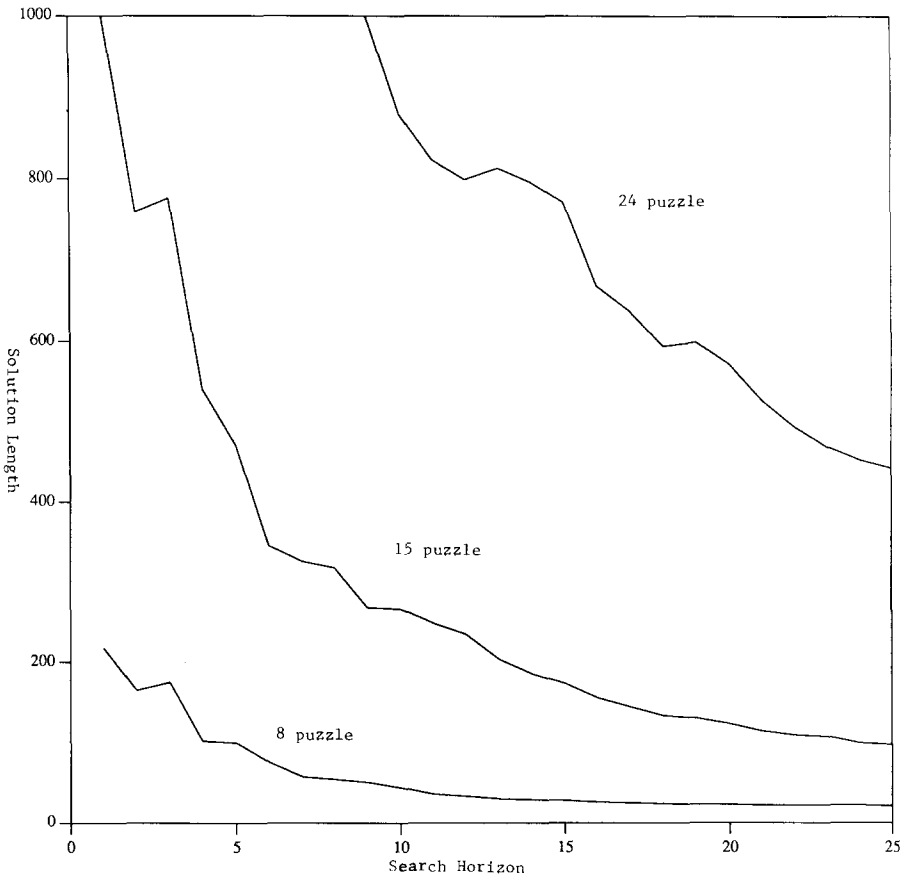

Fig. 4. Search horizon vs. solution length for RTA*.

and Fifteen Puzzle curves. The interpretation is that searching to depth three produces longer solutions on the average than searching to depth two. This appears to be an anomaly of these particular puzzle sizes, since it doesn't appear in the Twenty-four Puzzle data.

A more troubling feature of the results of these experiments is that even though when averaged over a large number of problem instances the solution lengths tend to decrease with increasing search depth, in individual problem instances solution lengths often vary quite erratically with search depth. This produces a great deal of noise in the data, giving rise for example to the spike in the Twenty-four Puzzle line, even when averaged over a thousand initial states. It appears to be due to the fact that large numbers of random decisions must be made among tied alternatives. This interpretation is borne out by the fact that different trials of the same individual problem instances, differing only in the random number seed, tend to produce quite different results.

6. Computation versus Execution Tradeoff

Viewing a heuristic evaluation function with lookahead search as a single, more accurate heuristic function generates a whole family of heuristic functions, one corresponding to each search depth. The members of this family vary in computational complexity and accuracy, with the more expensive functions generally being more accurate. How do we make the best choice from among this family?

Increasing the search horizon increases the amount of computation per move, but decreases the number of moves required to solve the problem. Thus, the choice of how far to look ahead amounts to a tradeoff between the cost of performing the search and the cost of executing the resulting solution. The optimal search horizon depends on the relative costs of computation and execution, and hence is problem dependent. The fact that computation per move grows exponentially with search depth, but that solution length asymptotically approaches the optimal with increasing horizon, guarantees that the optimal search horizon will remain bounded in any application.

For example, in our implementation of RTA*, "execution" of moves is only slightly more expensive than "simulating" moves in lookahead search, mostly due to the additional time required to access the hash table. Thus, in practice the running time of the program is proportional to the number of nodes generated in lookahead search. Empirically, the search horizon that minimizes the total time to find a solution is one for the Eight Puzzle and two for the Fifteen and Twenty-four Puzzles. This can be roughly arrived at by multiplying the corresponding curves in Figs. 2 and 4, since Fig. 2 gives the number of nodes per move and Fig. 4 gives the number of moves per problem instance.

The average clock times to find solutions using optimal search horizons is less than a tenth of a second for the Eight Puzzle, less than half a second for the Fifteen Puzzle, and less than 2.5 seconds for the Twenty-four Puzzle. These values are for a C program running on an HP-9000/350 workstation with a 20 megahertz clock. This is in stark contrast to the hours required to find optimal solutions to the Fifteen Puzzle running IDA* on a comparable machine [2].

7. Learning RTA*

We now turn our attention to the problem of successively solving multiple problem instances in the same problem space with the same set of goals. The key question is to what extent performance improvement, or learning, will occur over multiple problem solving trials. The information saved from one trial to the next is the hash table of values recorded for previously visited states.

Unfortunately, while RTA* as described above is well suited to single problem solving trials, it must be slightly modified to accommodate multi-trial learning. The reason is that the algorithm records the second best estimate with

the previous state, which represents an accurate estimate of the previous state looking back from the perspective of the next state. However, if the best estimate turns out to be correct, then storing the second best value can result in inflated values for some states. These inflated values will direct the agent in the wrong direction on subsequent problem solving trials.

This difficulty can be overcome simply by modifying the algorithm to store the best value in the previous state instead of the second best value. We call this algorithm Learning-RTA* or LRTA* for short. LRTA* retains the completeness property of RTA* described above (Theorem 1), and the same proof is valid for LRTA*. It does not, however, always make locally optimal decisions. For example, if LRTA* starts in one of the central 1 states in Fig. 5, then it will bounce back and forth between the 5 states until it "fills in the hole", or in other words, until the original 1 values equal or exceed 5, at which point it will escape to the rest of the graph. The reader is encouraged to verify this.

7.1. Convergence of LRTA*

An important property that LRTA* does enjoy, however, is that repeated problem solving trials cause the heuristic values to converge to their exact values. We assume a set of goal states, and a set of initial states, from which the problem instances are chosen randomly. This assures that all possible initial states will actually be visited. We also assume that the initial heuristic values are admissible, or do not overestimate the distance to the nearest goal. Otherwise, a state with an overestimating heuristic value may never be visited and hence remain unchanged. Finally, we assume that ties are broken randomly. Otherwise, once an optimal solution from some initial state is found, that path may continue to be traversed in subsequent problem solving trials without discovering additional optimal solution paths. Under these conditions, we can state and prove the following general result:

Theorem 3. *In a finite problem space with positive edge costs, and non-overestimating initial heuristic values, in which a goal state is reachable from every state, over repeated trials of LRTA*, the heuristic values will eventually converge to their exact values along every optimal path.*

Proof. As before, if node n has never been visited, then $h(n)$ is its heuristic static evaluation. Otherwise, $h(n)$ is the heuristic value stored with node n the last time it was visited by LRTA*. A node n is *correct* if $h(n) = h^*(n)$, where $h^*(n)$ is the actual cost of an optimal path from node n to a goal node. When a node n is visited by LRTA*, its new value $h(n)$ is equal to the minimum value

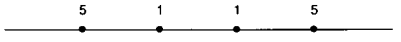

Fig. 5. Learning-Real-Time-A* Example.

of $k(n,n') + h(n')$ over all neighbors n' of n, where $k(n,n')$ is the cost of the edge from n to n'.

The first observation is that when LRTA* visits a node n and updates its value, it preserves the non-overestimating property of h. In other words, if all the neighboring values of a node are non-overestimating, then its updated value with also not overestimate cost to the goal. An optimal path from n must pass through one of its neighbors, say node m. If $h(n)$ is the new value of node n after it is visited by LRTA*, then $h(n)$ by definition is the minimum value of $k(n,n') + h(n')$ over all neighbors n' of n, which is less than or equal to $k(n,m) + h(m)$ since m is one of the neighbors of n. This is less than or equal to $k(n,m) + h^*(m)$ since we assume neighboring h values to be non-overestimating, which is equal to $h^*(n)$ since we assume that m is on an optimal path from n. Thus, $h(n) \leq h^*(n)$, and LRTA* preserves non-overestimating heuristic values with every move.

Next, consider two adjacent nodes i and j, such that j would be the successor of i on a path traversed by LRTA*, and assume that node j is correct. Thus, the new value of node i after updating by LRTA*, $h(i)$, will equal $k(i,j) + h(j)$ since j is the successor of i, which equals $k(i,j) + h^*(j)$ since j is correct. This is less than or equal to the minimum value of $k(i,i') + h(i')$ over all neighbors i' of n_i, since j is the successor of i, which is less than or equal to $k(i,i') + h^*(i')$ since h values are non-overestimating. Therefore, since $h(i)$ is less than or equal to $k(i,i') + h^*(i')$ over all neighbors i' of i, $h(i) = h^*(i)$, and i will be correct after LRTA* updates its value. Thus, if the successor of a given node is correct, the value of the given node will be correct after it is visited by LRTA*.

Since the value stored with a state by LRTA* is the minimum of the values of its neighbors plus the associated edge costs, the value stored with a state is always strictly greater than the minimum value of its neighbors. Applying the same argument from the proof of Theorem 1, this implies that LRTA* will eventually reach a goal state during every problem solving trial.

Now consider a path $(n_s,n_2,\ldots,n_g)$ about to be traversed by LRTA* from an initial state n_s to a goal state n_g. Since we assume non-overestimating heuristic values, $h(n_g) = 0 = h^*(n_g)$. Working backward from the goal, there will be a non-empty continuous sequence of correct nodes until an incorrect node is reached. Let node j be the last correct node in this continuous sequence, and i be the first incorrect node, where j is the successor of i. Once LRTA* reaches node j it will not change its value, or any subsequent value on the path, since they are all correct and their successors are correct. Furthermore, when LRTA* reaches node i, it will correct its value in passing, since its successor, node j, is correct. Thus, every pass of LRTA* over a path to a goal changes another node on the path from incorrect to correct, without disturbing the values of the following nodes. This continues until all nodes along the path are correct.

Assume that there is a node n on an optimal path p from some initial state s to a goal state, and that after an infinite number of trials, its value is not correct. Therefore, node n must must only have been visited a finite number of times. Of all nodes on p that have only been visited finitely many times, there must be a node j which is closest to s. It can't be node s, since initial states are chosen randomly and hence every initial state is chosen infinitely often. Therefore, node j must have a predecessor, i along path p, and by definition, i has been visited an infinite number of times. Thus, at least one of the other neighbors of i, say l, must have been visited an infinite number of times and hence $h(l) = h^*(l)$. Since $h(j)$ does not overestimate, $k(i,j) + h(j) \leq k(i,j) + h^*(j)$, which is less than or equal to $k(i,l) + h(l)$, since j is on an optimal path from i to a goal, which equals $k(i,l) + h^*(l)$ since l is correct. Therefore, j would be chosen as the successor of i at least as often as l since ties are broken randomly, and hence must also have been visited infinitely often. But this violates our assumption that it has only been visited a finite number of times. Thus, there can be no node on an optimal path whose value is not correct after an infinite number of trials. □

7.2. Efficiency of LRTA*

How long does this convergence to exact values take? The answer depends on the structure of the problem space, the distribution of initial and goal states, and the initial heuristic values. By way of example, we present below the analysis of one special case.

The problem space is a square grid N units on a side, with all edges having unit cost. A single initial state is in one corner and a single goal state in the opposite corner. The initial heuristic values are all zero, demonstrating that perfect values can be learned with no initial information. When the goal state is reached, the problem solver starts over again at the the initial state. Every state in the grid is along some optimal solution path. When learning is complete, the final space contains a regular pattern of diagonal stripes of equal heuristic values.

On the average, each move of each trial increases the value of one state by one unit. This persists even as most of the states reach their correct values, since the algorithm preferentially traverses the incorrect, underestimating paths over the correctly estimating paths. The average final value of a state is N, with the range being between zero and $2N - 2$. Since there are N^2, states, the average learning time is $O(N^3)$ moves. This result is borne out by experimental results.

Over the course of the learning time, the average time to reach the goal decreases to the optimal time of $2N - 2$. Empirically, solution times very close to this optimal value are actually achieved very early in the learning episode. This result is based on observations of the graphics interface to the system

described above. Even though perfect values appear as regular diagonal stripes, long before the learning is complete, an imperfect but very recognizable banding occurs in the remainder of the space. The interpretation of this is that even though only a small percentage of the states have achieved their exact values, the values of a large percentage of the states are consistent with those of their neighbors. This amounts to a relative accuracy that is very effective in driving the problem solver efficiently towards the goal.

8. Conclusions

Existing single-agent heuristic search algorithms cannot be used in large-scale real-time applications, due to their computational cost and the fact that they cannot commit to an action before its ultimate outcome is known. Minimin lookahead search is an effective algorithm for such problems. Alpha pruning drastically improves the efficiency of the algorithm without effecting the decisions made, and furthermore the achievable search horizon with this algorithm increases with increasing branching factor. Real-Time-A* efficiently solves the problem of when to abandoned the current path in favor of a more promising one, and is guaranteed to eventually find a solution. In addition, RTA* makes locally optimal decisions on a tree. Extensive simulations on three different sizes of sliding tile puzzles show that increasing search depth increases solution quality, and that solution lengths comparable to optimal ones are achievable in practice. Lookahead search can be characterized as generating a family of heuristic functions that vary in accuracy and computational complexity. The optimal level of lookahead depends on the relative costs of simulating vs. executing moves, and hence is a function of the particular application. Finally, Learning RTA* is a slight modification of RTA* which preserves its completeness properties while learning exact heuristic values over repeated problem solving trials. These algorithms effectively solve larger single-agent problems than have previously been solvable using heuristic evaluation functions.

ACKNOWLEDGEMENT

Subsequent to this work, Hermann Kaindl pointed out that minimin lookahead search was implemented by Rosenberg and Kestner [12] in the context of A*. Our discovery of the idea arose out of discussions with Bruce Abramson. A similar algorithm was independently implemented by Andrew Mayer and Othar Hansson, with whom I have had many discussions concerning this research. Judea Pearl provided valuable comments on a draft of this manuscript. Valerie Aylett prepared the figures. This research was supported by an NSF Presidential Young Investigator Award, NSF Grant IRI-8801939, an IBM Faculty Development Award, and an equipment grant from the Hewlett-Packard Corporation.

REFERENCES

1. P.E. Hart, N.J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Syst. Sci. Cybern.* **4** (1968) 100–107.

2. R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence* **27** (1985) 97–109.
3. H.A. Simon, *The Sciences of the Artificial* (MIT Press, Cambridge, MA, 2nd ed., 1981).
4. C.E. Shannon, Programming a computer for playing chess, *Philos. Mag.* **41** (1950) 256–275.
5. T.P. Hart and D.J. Edwards, The alpha-beta heuristic, MIT Artificial Intelligence Project Memo, MA (1963).
6. H. Berliner and C. Ebeling, Pattern knowledge and search: The SUPREM architecture, *Artificial Intelligence* **38** (1989) 161–198.
7. R.E. Korf, Real-time heuristic search: First results, in: *Proceedings AAAI-87*, Seattle, WA (1987) 133–138.
8. R.E. Korf, Real-time heuristic search: New results, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 139–144.
9. J. Pearl, *Heuristics* (Addison-Wesley, Reading, MA, 1984).
10. R.M. Karp and J. Pearl, Searching for an optimal path in a tree with random costs, *Artificial Intelligence* **21** (1983) 99–117.
11. T. Harris, *The Theory of Branching Processes* (Springer, Berlin, 1963).
12. R.S. Rosenberg and J. Kestner, Look-ahead and one-person games, *J. Cybern.* **2** (4) (1972) 27–42.

Received October 1987; revised version received September 1988