Real-Time Heuristic Search: First Results

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024

Abstract

Existing heuristic search algorithms are not applicable to real-time applications because they cannot commit to a move before an entire solution is found. We present a special case of minimax lookahead search to handle this problem, and an analog of alpha-beta pruning that significantly improves the efficiency of the algorithm. In addition, we present a new algorithm, called Real-Time-A*, for searching when actions must actually be executed, as opposed to merely simulated. Finally, we examine the nature of the tradeoff between computation and execution cost.

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 typically is to find a lowest cost sequence of operators that map the initial state to a goal state. The complexity of search algorithms is greatly reduced by the use of a heuristic evaluation function, often without sacrificing solution optimality. A heuristic is a function that is relatively cheap to compute and estimates the cost of the cheapest path from a given state to a goal state.

Common examples in the AI literature of search problems are the Eight Puzzle and its larger relative the Fifteen Puzzle. The Eight Puzzle consists of a 3x3 square frame containing 8 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 shorted path between the initial and goal states. A typical heuristic evaluation function for this problem is the air-line distance from a given location to the goal location.

2 Existing Algorithms

The best known 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, $g(n)$, and the estimated cost of reaching the solution from that node, $h(n)$. A* has the property that it will always find an optimal solution to a problem if the heuristic function is admissible, i.e. never overestimates the actual cost of solution.

Iterative-Deepening-A*(IDA*)[2] is a modification of A* that reduces its space complexity in practice from exponential to linear. IDA* performs a series of depth-first searches, in which a branch is cutoff 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 prcperty 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.

The drawback of both A* and IDA* is that they take exponential time to run in practice. This is an unavoidable cost of obtaining optimal solutions. As observed by Simon[4], 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 both 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. This is a serious limitation of these algorithms with respect to real-time applications.

3 Real-Time Problems

In this section we examine several important characteristics of real-time problems that must be taken into consideration by any real-time heuristic search algorithm.

The first characteristic is that in real problems the problem solver must face a limited search horizon. This is due primarily to computational and/or informational limitations. For example, due to the combinatorial explosion of the Fifteen Puzzle, finding optimal solutions using IDA* with Manhattan Distance on a DEC20 required an average of over five hours per problem instance[2]. Any larger puzzle would be intractable. In the case of the navigation problem without the benefit of completely detailed maps, the search horizon (literally, in this case) is due to the informational limit of how far the vision system can see ahead. Even with the aid of accurate maps the level of detail is a limitation. This gives rise to a "fuzzy horizon" where the level of detail of the terrain knowledge is a decreasing function of the distance from the problem solver.

A related characteristic is that in a real-time setting, actions must be committed before their ultimate consequences are known. For example, a chess tournament requires that moves be made within a certain time limit. In the case of navigation, the vehicle must be moved in order to extend the search horizon in the direction chosen.

A final important characteristic is that often the cost of action and the cost of planning can be expressed in common terms, giving rise to a tradeoff between the two. For example, if the goal of the Fifteen Puzzle were to solve it in the shortest possible time, as opposed the smallest number of moves, and we quantified the time to actually make a physical move relative to the time required to simulate a move in the machine, then in principle we could find algorithms that minimized total solution time by balancing "thinking" time and "action" time.

4 Minimin Lookahead Search

In this section we present a simple algorithm for real-time heuristic search in single-agent problems that takes the above characteristics into account. It amounts to a special case of the minimax algorithm for two-player games[3]. This should not be surprising since two-player games share the real-time characteristics of limited search horizon and commitment to moves before their ultimate outcome can be known. At first we will assume that all operators have the same cost.

The algorithm is to search forward from the current state to a fixed depth determined by the computational or informational resources available for a single move, and apply the heuristic evaluation function to the nodes at the search frontier. Whereas in a two-player game these values would then be 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 follow 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 indicated by the first search. We call this algorithm *minimin* search in contrast to minimax search[1].

Note that the search proceeds in two quite different, but interleaved modes. The minimin lookahead search occurs in a simulation mode, where the postulated moves are not actually executed, but merely simulated in the machine. After one complete lookahead search, the best move found is actually 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 a path so far 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 then looks forward a fixed number of moves and backs up the minimum f 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.

To ensure termination, care must be taken to prevent infinite loops in the path actually traversed by the problem solver. This is accomplished by maintaining a CLOSED list of those states that have actually been visited by an actual move of the problem solver, and an OPEN stack of those nodes on the current path from the start state. Moves to CLOSED states are ruled out, and if all possible moves from a given state lead to CLOSED states, then the OPEN stack is used to backtrack until a move is available to a new state. This conservative strategy prohibits the algorithm from undoing a previous move, except when it encounters a dead end. This restriction will be removed later in the paper.

5 Alpha Pruning

A natural question to ask at this point is whether every frontier node must be examined to find the one with min-

[1]This name is due to Bruce Abramson

imum 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 $f(n)$ is monotonic if it never decreases along a path away from the initial state. Monotonicity of $f(n) = g(n) + h(n)$ is equivalent to consistency of $h(n)$, or obeying the triangle inequality, a property satisfied by most naturally occurring heuristic functions, including Manhattan Distance and air-line distance. Furthermore, if a heuristic function is admissible but not monotonic, then an admissible, monotonic function $f(n)$ can trivially be constructed by taking its maximum value along the path.

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 called α the lowest f value of any node encountered so far on the search horizon. As each interior node is generated, compute its f value and cut off the corresponding branch when its f value equals α. The reason this can be done is that since the function is monotonic, the f values of the frontier nodes descending from that node can only be greater than or equal to the cost of that node, and hence cannot effect the move made since we only move toward the frontier node with the minimum value. 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 experiments with the Fifteen Puzzle using the Manhattan Distance evaluation function, alpha-pruning reduces the effective branching factor by more than than the square root of the brute-force branching factor (from 2.13 to 1.41). This has the effect of more than doubling the search horizon reachable with the same amount of computation. For example, if the computational resources allow a million nodes to be examined in the course of a move, the brute force algorithm can search to a depth of 18 moves while alpha pruning allows the search to proceed more than twice as deep (40 moves).

As in alpha-beta pruning, the efficiency of alpha pruning can be improved by *node ordering*. The idea is to order the successors of each interior node in increasing order of their f values, hoping to find low cost frontier nodes early and hence prune more branches sooner.

Although the two algorithms were developed separately, minimin with alpha pruning is very similar to a single iteration of iterative-deepening-A*. The only difference is that in alpha pruning the cutoff threshold is dynamically determined and adjusted by the minimum value of the frontier nodes, as opposed to being static and set in advance by the previous iteration in IDA*.

6 Real-Time-A*

So far, we have assumed that once an action is committed, it is not reversed unless a dead end is encountered, with the primary motivation being the prevention of infinite loops by the problem solver. We now address the question of how to incorporate backtracking when it appears favorable, as opposed to dead-end backtracking, while still preventing infinite loops. The basic idea 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 backtracking 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 simulation phase of the search, RTA* is an algorithm for controlling the execution phase of the search. As such, it is independent of the simulation algorithm chosen. For simplicity of exposition, we will assume that the minimin lookahead algorithm is encapsulated within the computation of $h(n)$, and hence becomes simply a more accurate and computationally more expensive way of computing $h(n)$.

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. 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, 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. Interestingly, these problems can be solved in constant time per move using only local information in the graph. The idea is as follows: from a given current state, the neighboring states are generated, the heuristic function, augmented by lookahead search, is applied to each, and then 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 next best f value is stored at the previous current state.

This represents the estimated h cost of solving the problem by returning to this state. Next, the new neighbors of the new current state are generated, their h values are computed, and the edge costs of all the neighbors of the new current state, including the previous current state, are added to their h values, resulting in a set of f values for all the neighboring states. Again, the node with the smallest value is chosen to move to, and the second best value is stored as the h value of the old current state.

Note that RTA* does not require separate OPEN and CLOSED lists, but a single list of previously evaluated nodes suffices. 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.

Interestingly, one can construct examples to show that RTA* could backtrack an arbitrary number of times over the same terrain. For example, consider the simple straight-line graph in Figure 1, where the initial state is node a, all the edges have unit cost, and the values below each node represent the 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 $a, f(b) = g(b) + h(b) = 1 + 1 = 2$, while $f(c) = g(c) + h(c) = 1 + 2 = 3$. Therefore, the problem solver moves to node b, and leaves behind at node a the information that $h(a) = 3$. Next, node d is evaluated with the result that $f(d) = g(d) + h(d) = 1 + 4 = 5$, and node a receives the value $f(a) = g(a)+h(a) = 1+3 = 4$. Thus, the problem solver moves back to node a, and leaves $h(b) = 5$ behind at node b. At this point, $f(b) = g(b) + h(b) = 1 + 5 = 6$, and $f(c) = g(c) + h(c) = 1 + 2 = 3$, causing the problem solver to move to node c, and leave $h(a) = 6$ behind at node a. The reader is urged to continue the example to see that the problem solver continues to go back and forth, until a goal is reached. The reason it is not 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 rational behavior in the presence of a limited search horizon, and a pathological space.

Unfortunately, the capability of RTA* to backtrack is not exercised by the Fifteen Puzzle with Manhattan Distance as the evaluation function. The reason is that since Manhattan Distance only changes by one in a single move, it can be shown that RTA* will only backtrack at dead ends.

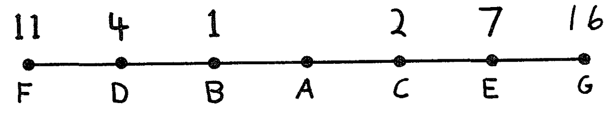

Figure 1: RTA* Example

7 Solution Quality

In addition to efficiency of the algorithm, the length of solutions generated by minimin lookahead search is of central concern. The most natural expectation is that solution length will decrease with increasing search depth. In experiments with the Fifteen Puzzle using Manhattan Distance, this turned out to be generally true, but not uniformly.

One thousand solvable initial states of the Fifteen Puzzle were randomly generated. For each initial state, the minimin algorithm with alpha pruning was run with search depths ranging from 1 to 30 moves. Moves were made until a solution was found, or a thousand moves had been made, in order to limit overly long solutions, and the resulting number of moves made was recorded. Figure 2 shows a graph of the average solution length over all thousand problem instances versus the depth of the search horizon. The line at the bottom represents 53 moves which is the average optimal solution length for a different set of 100 initial states. The optimal solution lengths were computed using IDA*, which required several weeks of CPU time to solve the hundred initial states[2].

The overall shape of the curve confirms the intuition that increasing the search horizon decreases the resulting solution cost. At depth 25, the average solution length is only a factor of two greater than the average optimal solution length. This is achieved by searching only about 6000 nodes per move, or a total of 600,000 nodes for the entire solution. This is accomplished in about one minute of CPU time on a Hewlett-Packard HP-9000 workstation.

However, at depths 3, 10, and 11, increasing the search horizon resulted in a slight increase in the average solution length. This phenomenon was first identified in the case of two-player games and was termed *pathology* by Dana Nau[5]. He found that for certain artificial games, increasing the search depth resulted in consistently poorer play in some cases. Until now, pathology has never been observed in a "real" game.

While the pathological effect is relatively small when averaged over a large number of problem instances, in individual problem instances the phenomenon is much more prominent. In many cases, increasing the search depth by one move resulted in solutions that were hundreds of moves longer.

In an attempt to understand this phenomenon, we performed some additional experiments on *decision quality* as opposed to solution quality. The difference is that a solution is composed of a large number of individual move decisions. While solution quality is measured by the total length of the solution, decision quality is measured by the percentage of time that an optimal move is chosen. Since the optimal moves from a state must be known to determine decision quality, the smaller and more tractable Eight Puzzle was chosen for these experiments, with the same Manhattan Distance evaluation function.

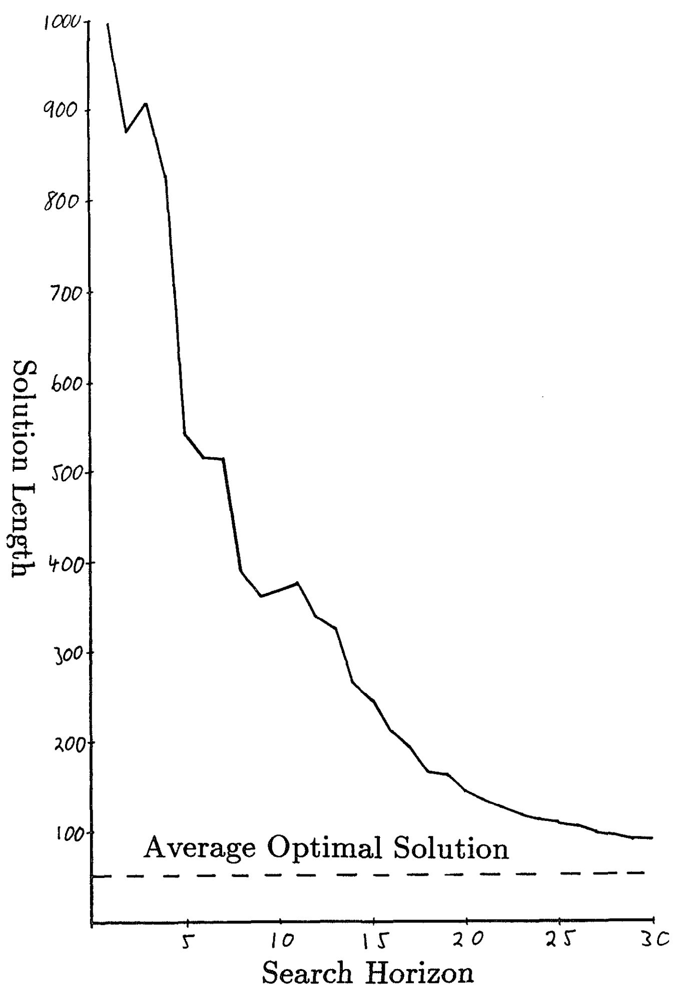

Figure 2: Solution Length vs. Search Horizon

In this case, ten thousand solvable initial states were randomly generated. Instead of examining the entire solution that would be generated by the minimin algorithm, only the first move from these states was considered. In each case, the percentage of time that an optimal first move was chosen was recorded over all initial states for each different search horizon. The search horizons ranged from one move to a horizon one less than the optimal solution length for a given problem. Figure 3 shows a graph of error percentage versus search horizon. As the search depth increases, the percentage of optimal moves also increases monotonically. Thus, pathology does not show up in terms of decision quality.

If decision quality smoothly improves with increasing search depth, why is solution quality so erratic? One explanation is that while the probability of mistakes decreases, the cost of any individual mistake can be quite high in terms of overall solution cost. This is particularly true in these experiments, where backtracking only occurred when a dead end was encountered.

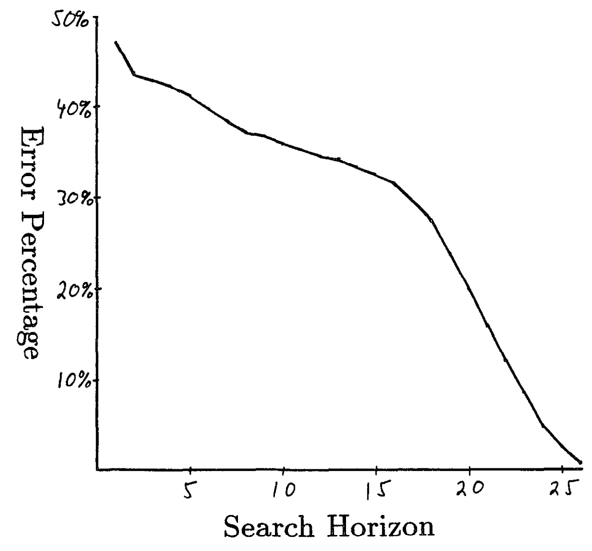

Figure 3: Decision Quality vs. Search Horizon

Another source of error is ties among alternatives. In a situation where moves must be committed based on uncertain information, ties should not be broken arbitrarily. More generally, when dealing with inexact heuristic estimates, two values that are closer together than the accuracy of the function should be considered virtual ties, and dealt with as if they were indistinguishable.

In order to deal with this problem, ties and virtual ties must first be recognized. This means that the alpha pruning algorithm must be changed to prune a branch only when its value exceeds the previous best by the error factor. This will increase the number of nodes that must be generated.

Once a tie is recognized, it must be broken. The most reasonable way to accomplish this is to perform a deeper secondary search on the candidates until the tie is broken. However, this secondary search must also have a depth limit. If the secondary search reaches its depth limit without breaking the tie, a virtual tie may as well be resolved in favor of the lower cost move.

8 Computation vs. Execution

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.

The choice of which evaluation function to use amounts to a tradeoff between the cost of performing the search and the cost of executing the resulting solution. The minimum total time depends on the relative costs of computation and execution, but a reasonable model is that they are linearly related. In other words, we assume that the cost of applying an operator in the real world is a fixed multiple of the cost of applying an operator in the simulation.

Figure 4 shows the same data as Figure 2, but with a horizontal axis that is linear in the number of nodes generated per move as opposed to linear in the search depth. The curve shows that the computation-execution tradeoff is initially quite favorable in the sense that small increases in computation buy large reductions in solution cost. However, a point of diminishing returns is rapidly reached, where further significant reductions in solution cost require exponentially more computation. The effect is even greater than it appears, since overly long solutions were arbitrarily terminated at 1000 moves. Different relative costs of computation and execution will change the relative scales of the two axes without altering the basic L-shape of the curve.

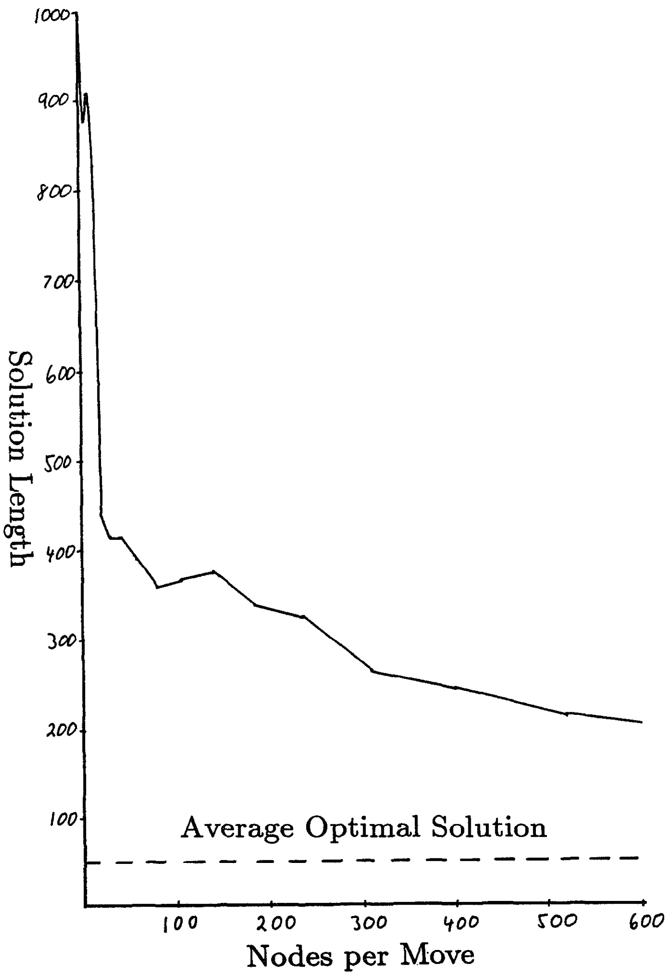

Figure 4: Execution Time vs. Computation Time

9 Conclusions

Existing single-agent heuristic search algorithms cannot be used in 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. Furthermore, alpha pruning drastically improves the efficiency of the algorithm without effecting the decisions made. In addition, Real-Time-A* efficiently solves the problem of when to abandoned the current path in favor of a more promising one. Extensive simulations show that while increasing search depth usually increases solution quality, occasionally the opposite is true. To avoid the detrimental effect of virtual ties on decision quality, additional search is required. Finally, lookahead search can be characterized as generating a family of heuristic functions that vary in accuracy and computational complexity. The tradeoff between solution quality and computational cost is initially quite favorable but rapidly reaches a point of diminishing returns.

10 Acknowledgements

The idea of minimin lookahead search arose out of discussions with Bruce Abramson. A similar algorithm was independently implemented by Andy Mayer and Othar Hansson. I'd like to thank David Jefferson for his comments on an earlier draft of this paper. This research was supported by NSF Grant IST-85-15302, an IBM Faculty Development Award, an NSF Presidential Young Investigator Award, and a grant from Delco Systems Operations.

References

[1] Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, SSC-4, No. 2, 1968, pp. 100-107.

[2] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[3] Shannon, C.E., Programming a computer for playing chess, *Philosophical Magazine*, Vol. 41, 1950, pp. 256-275.

[4] Simon, H.A., *The Sciences of the Artificial*, 2nd edition, M.I.T. Press, Cambridge, Ma., 1981.

[5] Nau, D.S., An investigation of the causes of pathology in games, *Artificial Intelligence*, Vol. 19, 1982, pp. 257-278.