From: AAAI-88 Proceedings. Copyright ©1988, AAAI (www.aaai.org). All rights reserved.

Real-Time Heuristic Search: New Results*

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

Abstract

We present new results from applying the assumptions of two-player game searches, namely limited search horizon and commitment to moves in constant time, to single-agent problem-solving searches. We show that the search depth achievable with alpha pruning for a given amount of computation actually *increases* with increasing branching factor. We prove that real-time-A* (RTA*) makes locally optimal decisions given the heuristic information available to it on a tree. We then modify RTA* to perform optimally on a graph as well. We also prove that RTA* is guaranteed to find a solution to any solvable problem regardless of the initial heuristic values. In addition, we develop a learning version of RTA* that improves its performance over multiple problem-solving trials, and prove convergence of the learned heuristic values to the exact values. Finally, we demonstrate that these algorithms effectively solve larger problems than have previously been solvable with heuristic search techniques.

1 Introduction

Heuristic search has been applied both to two-player games and single-agent problems. Research on two-player games assumes insufficient computation to search all the way to terminal positions, and that moves must be irrevocably committed under strict time constraints [1]. Conversely, research on single-agent problems assumes that search can proceed to goal positions, that an entire solution may be computed before even the first move need be executed. As a result, existing single-agent heuristic search algorithms, such as A* [2] and IDA* [3], do not scale up to large problems due to their exponential complexity, a necessary consequence of finding optimal solutions. The goal of this research is to extend the techniques of heuristic search to handle single-agent problems under real-time constraints. By this we mean that computation or information is limited, and that each individual action must actually be executed in constant time. This requires sacrificing solution optimality, and imposing a limited search horizon. A previous paper [4] reported the first results of this research. A more comprehensive treatment can be found in [5].

*This research was supported by an NSF Presidential Young Investigator Award.

2 Minimin with Alpha Pruning

The first step in applying bounded lookahead search to single-agent problems is to specialize minimax search to the case where a single-agent makes all the moves. The resulting algorithm, called minimin search, searches forward from the current state to a fixed depth horizon determined by the computational resources available, and then applies the A* cost function of $f(n) = g(n) + h(n)$ to the frontier nodes. Since a single agent makes all the decisions, the minimum value is then backed up, instead of the minimax value, and a single move is made in the direction of the minimum value. Making only a single move at a time follows a strategy of least commitment, since the backed-up values are only heuristic, and further search may recommend a different second move than anticipated by the first search.

There exists an analog to alpha-beta pruning [6] that makes the same decisions as full minimin search, but by searching many fewer nodes. It is based on the assumption that h is a metric. Since by definition all reasonable cost functions are metrics, this condition is not a restriction in practice. If h is a metric, then $f = g + h$ is monotonically non-decreasing along any path. Therefore, given static evaluations of all interior nodes, branch-and-bound can applied as follows. The value of the best frontier node encountered so far is stored in a variable called α, and whenever the cost of a node equals or exceeds α, the corresponding branch is pruned off. In addition, whenever a frontier node is encountered with a value less than α, α is reset to this lower value.

2.1 Performance of Alpha Pruning

How much does alpha pruning improve the efficiency of minimin search? Figure 1 shows a comparison of the total number of nodes generated as a function of search horizon for several different sliding tile puzzles, including the 3×3 (8 Puzzle), 4×4 (15 Puzzle), 5×5 (24 Puzzle), and 10×10 (99 Puzzle) versions. Each puzzle consists of a square frame containing a number of movable tiles, and one empty position. Any tile that is horizontally or vertically adjacent to the empty position can be slid into that position. The task is to rearrange the tiles from a given initial configuration to a particular goal configuration. The straight lines on the left represent brute-force search with no pruning, while the curved lines to the right represent the number of nodes examined with alpha pruning using the Manhattan Distance heuristic function. It is computed by determining, for each individual tile, the number of grid units it is displaced from its goal position, and summing these values over all tiles. In each case, the values are the

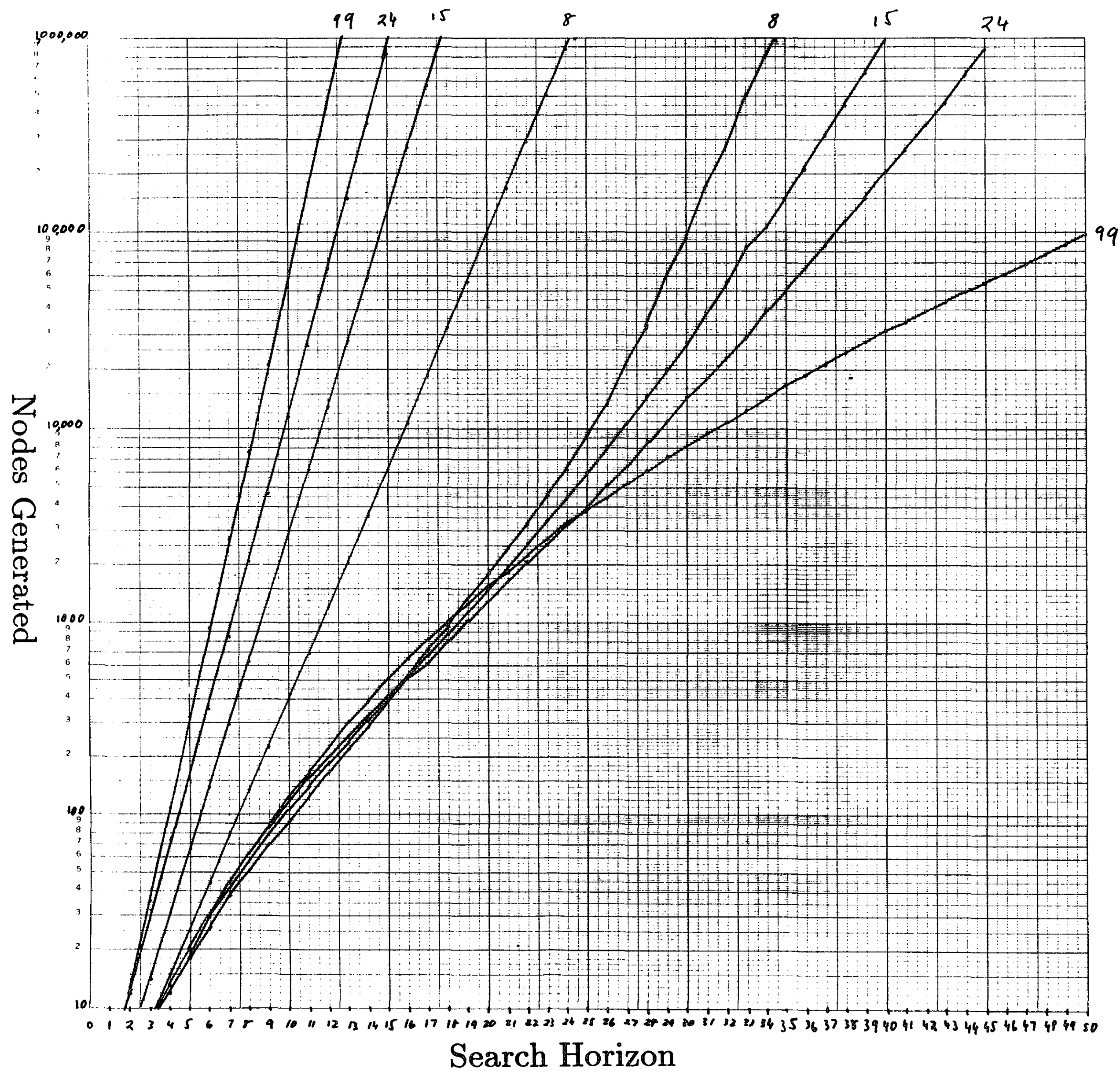

Figure 1: Search horizon vs. nodes generated for brute-force and alpha pruning search

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 amount of computation at 100,000 nodes per move, the reachable 99 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 beyond a certain point 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 24 puzzle, in spite of the fact that the brute-force branching factors are larger.

How general is this paradoxical affect? An analytic model that captures the monotonicity property necessary for branch-and-bound is a tree with uniform branching factor and uniform depth, where the edges are assigned a value of zero or one independently with some probability p. The value of a node is the sum of the edge costs from the root to that node. This model, which is identical to that studied by Karp and Pearl [7] in a different context, predicts the phenomenon observed above, namely that increasing branching factor allows increasing search depth with the same amount of computation. Thus, this counter-intuitive result appears to be a fairly general property of branch-and-bound algorithms, rather than an artifact of sliding tile puzzles.

A simple intuitive explanation for why this occurs is that increasing the branching factor increases the number of frontier nodes. Since alpha is the minimum value of the frontier nodes, increasing the number of frontier nodes tends to decrease alpha. A lower value of alpha in turn causes more branches to be pruned sooner, resulting in greater savings. Empirically, this effect more than compensates for the increased branching factor.

3 Real-Time-A*

Minimin lookahead search with alpha pruning is a strategy for evaluating the immediate children of the current node. It runs in a planning mode where the moves are merely simulated, rather than actually being executed in the real world. As such, it can be viewed as providing a range of more accurate but computationally more expensive heuristic functions, one corresponding to each search horizon.

Real-Time-A* (RTA*) is an algorithm for controlling the sequence of moves actually executed. The neighbors of the current state are generated and a heuristic function, including lookahead search with alpha pruning, is applied to each new state. The neighbor with the minimum $g + h$ value is chosen as the new current state, and the old current state is stored in a table along with the second best $g + h$ value, which is the best value among the remaining children. This represents the best estimate of the cost of finding the solution via the old current state from the perspective of the new current state. This assumes that the edge costs in either direction are equal. The extension to unequal edge costs is straightforward. The algorithm simply repeats this cycle, using the stored h values for previously visited states, and computing it for new states, until a solution is found.

3.1 Correctness of RTA*

Since RTA* is making decisions based on limited information, the best we can say about the quality of decisions made by this algorithm is that RTA* makes optimal moves relative to the part of the search space that it has seen so far. Initially we will assume that the graph is a tree, in other words contains no cycles, and prove such an optimality theorem. Then we will modify RTA* to perform optimally on a graph. The same result holds for the more complex algorithm.

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 nodes that have been generated but not expanded since the beginning of the search, including all lookahead phases of the search so far. This is analogous to the OPEN list in A*. If n is a frontier node, let $h(n)$ be the heuristic static evaluation of node n. Let $g_x(n)$ be the actual distance in the graph from node x to node n. Note that $g(n)$ in A* is $g_s(n)$ where s is the initial state. Similarly, let $f_x(n) = g_x(n) + h(n)$.

Theorem 1 *Given a cumulative search frontier, at every cycle RTA* moves from the current state x toward a frontier node n for which the value of $f_x(n) = g_x(n) + h(n)$ is a minimum.*

Proof: If x is an interior node of the tree, define $h(x)$ to be the minimum value of $f_x(n) = g_x(n) + h(n)$ over all frontier nodes n below x in the tree. This definition of $h(x)$ for interior nodes is relative to a tree rooted at the current state of the problem solver. We will show by induction on the number of moves made by the algorithm that $h(x)$ is the value stored by RTA* with each previously visited node x of the tree. Consider the first move of the algorithm, starting from an initial state s. Minimin search generates a tree rooted at s and terminating in a set of frontier nodes at the search horizon. The backed-up value associated with each child c_i of the start state s is the minimum value of $f_{c_i}(n) = g_{c_i}(n) + h(n)$ for each frontier node below c_i, which is $h(c_i)$. Note that alpha pruning has no effect on the values returned by the lookahead search, but simply computes them more efficiently. The problem solver then adds $g_s(c_i)$ to $h(c_i)$ to compute $f_s(c_i)$, moves to the child with the minimum value, say c_1 without loss of generality, and the stores the second best value with state s. This move changes the root of the tree from s to c_1, as if the tree was now picked up by c_1. This changes c_1 to the parent of s instead of vice-versa, but leaves all other parent-child relationships the same. The children of s are now c_i for $i > 1$. Since the second best value is the minimum value of $f_s(c_i) = g_s(c_i) + h(c_i)$ for $i > 1$, the value stored with s is exactly $h(s)$ after the move. For the induction step, assume that at a given point in the algorithm, the value stored with each node y in the hash table is $h(y)$, and that the current state of the problem solver is x. For each of the neighbors c_i of x, if c_i is not in the table, minimin search will be used to compute $h(c_i)$. Otherwise, $h(c_i)$ will be read from the table. In either case, the correct value of $h(c_i)$ will be returned. Then by

following exactly the same argument as above, the value stored with x will be $h(x)$ after the move. Finally, RTA* always moves from its current state x to the the neighbor c_i for which $f_x(c_i) = g_x(c_i) + h(c_i)$ is a minimum. This is the same as moving toward a frontier node n for which $f_x(n) = g_x(n) + h(n)$ is a minimum.

3.2 RTA* on a Graph

The above theorem holds for RTA* on a tree with no cycles. On a graph, however, this simple version of RTA* may occasionally make a suboptimal move, and must be modified to achieve optimal performance.

The problem is that the h values are relative to the current state of the problem solver, and hence when the algorithm returns to a previously visited state by a different path from which it left, some of the h values in the graph may be incorrect. In particular, those values that are based on a path that passes back through the new current state will be incorrect from the perspective of that current state. When we return to a previously visited state via a different path from which we left, the incorrect values must be modified.

In general, $h(y)$ will be correct from the perspective of x if it is based on a path through some neighbor z other than x. In order for this to be the case, y must have a neighbor z, other than x, such that $h(y) = f_y(z) = g_y(z) + h(z)$. If there is such a neighbor z of y, then $h(y)$ is correct relative to node x. This check must be performed on each neighbor y of x. If all values are found to be correct, then the algorithm can proceed as usual. If any neighbor of the current state fails this test, then the same test must be performed on each of its neighbors, since their values could be incorrect for the same reason. This amounts to recursively exploring the subgraph of incorrect values until each branch terminates in correct values. The correct terminal values are then recursively backed up according to the usual rule until all the h values are correct relative to the new current state. At that point the algorithm continues as usual.

This fixup phase is analogous to the pointer redirection phase of A* when a cycle in the graph is detected. When it is added to RTA*, Theorem 1 becomes true for general graphs as well as trees. Unfortunately, space limitations preclude us from presenting the proof.

3.3 Completeness of RTA*

Under what conditions is RTA* guaranteed to find a goal state? Here we will assume a graph with cycles, but use only the simple version of RTA* that does not correct values in cycles. One caveat is that in an infinite problem space, RTA* may not find a solution, since h values could easily be constructed to send the problem solver down an infinite wrong path. 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. Finally, we must rule out cycles with zero or negative cost, for obvious reasons. These are the only restrictions, however, and for all other problems, RTA* is guaranteed to eventually find a solution if one exists. The only constraint placed on the heuristic evaluation function is that it return finite values.

Theorem 2 *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, if a goal state is not part of the cycle, 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 or computes the value of each of its neighbors, adds the corresponding positive edge costs, and moves to the neighbor with the minimum resulting value (the new state). At the same time, it writes the second best value in the state it left (the old state). Since the second best value is greater than or equal to the best, and the value of the new state must be strictly less than its value after the cost of the edge from the old state is added to it, the value written into the old state must be strictly greater than the value of the new state. Thus, the algorithm always writes a larger value in the state it leaves than the value of the state it moves too. Now consider a state with the lowest value on the hypothesized infinite cycle. Its value must be less than or equal to the value of the next state on the cycle. When the algorithm reaches a node with the lowest value, it must increase its value in passing, in order to move to the next state, which has an equal or greater value. Thus, every trip of the algorithm around the cycle must increase the value of the node on the cycle with the lowest value. Therefore, the values of all nodes on the cycle must increase without bound. At some point, 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.

4 Learning-RTA*

We now turn our attention to the problem of multiple agents 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 will be the table of values recorded for previously visited states.

Unfortunately, while RTA* as described above is ideally suited to single problem solving trials, it must be modified to accommodate multi-trial learning. The reason is that the algorithm records the second best estimate in the previous state, which represents an accurate estimate of that 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 next agents 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*s shown in Theorem 2, and the same proof is valid for LRTA*. It does not, however, always make locally optimal decisions in the sense of Theorem 1.

4.1 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 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 *Given non-overestimating initial heuristic values, over repeated trials of LRTA*, the heuristic values will eventually converge to their exact values along every optimal path from an initial state to a goal state.*

Proof: The first observation is that visiting a state and updating its value preserves the non-overestimating property of h. Assuming that the h values of the neighbors of a given state do not overestimate distance to the goal, then after adding the corresponding edge values to each of the neighboring states, the minimum of the resulting values cannot overestimate the distance from the given state. Define the value $h(n)$ of a state n to be *consistent* with those of its neighbors $h(n')$, if $h(n)$ is equal to the minimum of $h(n') + k(n, n')$ for all neighbors n' of n, where $k(n, n')$ is the cost of the edge from n to n'. Now, assume the converse of the theorem, that after an infinite number of trials, there exists a state along an optimal path from an initial state to a goal whose value is incorrect. Assuming that h of all goal states is equal to zero, if the values of any node along any path to a goal state is incorrect, then some node along the same path must be inconsistent. This follows formally by induction on the distance from the goal. If there exists a state whose value is inconsistent, then there must exist a least such state in an arbitrary ordering of the states. Call such a state x. By assumption, x lies along an optimal path from some initial state, s, to a goal state. In addition, since all the h values are non-overestimating and this property is preserved by LRTA*, the $f = g + h$ values of all nodes along the optimal path from s to x, from the perspective of s, are less than or equal to their exact values. Since starting states are chosen randomly, and ties are broken randomly, this ensures that node x will eventually be visited by RTA*. When it is, its value will become consistent with those of its neighbors, thus violating the assumption that it is the least inconsistent state in some ordering. Therefore, the value of every node along an optimal path from an initial state to a goal state must eventually reach its correct value.

5 Empirical Results

RTA* with minimin lookahead search and alpha pruning has been implemented for the 8, 15, and 24 puzzles. Figure 4 shows a graph of the average solution length over a thousand problem instances versus the depth of the search horizon for each of the three puzzles.

As expected, the solution lengths generally decrease with increasing search horizon. For the Eight Puzzle, searching to a depth of 10 moves, which requires generating an average of 92 nodes per move, produces solution lengths (42) that are only a factor of two greater than average optimal solutions (21). In the case of the Fifteen Puzzle, finding solution lengths (106) that are two times average optimal (53) requires searching to a depth of 22 moves and evaluating 2622 nodes per move on the average. 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, which requires generating an average of 4057 nodes per move, produces solution lengths of about 400 moves. The time required to find solutions is on the order of tenths of seconds for the 8 puzzle, seconds for the 15 puzzle, and tens of seconds for the 24 puzzle. These are the first reported solutions to the 24 puzzle using heuristic search techniques.

6 Conclusions

We present a number of new results in the area of real-time heuristic search. The first is that the search horizon reachable with alpha pruning increases with increasing branching factor of the problem space. Next, we prove that RTA* makes locally optimal decisions on a tree, relative to the complete search horizon visible to the algorithm since it began. We then modify the algorithm to make locally optimal decisions on a general graph. Even without this modification, however, we prove that RTA* is guaranteed to eventually find a solution. We also modify RTA* to learn more accurate heuristic values over multiple problem solving trials, and prove that the learned values will converge to the exact values over every optimal path from an initial state to a goal state. Finally, we present new empirical results that demonstrate that these algorithms are effective at solving larger problems than have previously been solvable with heuristic search algorithms.

References

[1] Shannon, C.E., "Programming a Computer for Playing Chess", *Philosophical Magazine*, Vol. 41, 1950, pp. 256-275.

[2] 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.

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

[4] Korf, R.E. Real-time heuristic search: First results, *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Wash., July, 1987, pp. 133-138.

[5] Korf, R.E., Real-time heuristic search, to appear, *Artificial Intelligence.*

[6] Slagle, J.R., and Dixon, J.K., Experiments with some programs that search game trees, J.A.C.M., Vol. 16, No. 2, 1969, pp. 189-207.

[7] Karp, R.M., and J. Pearl, Searching for an optimal path in a tree with random costs, *Artificial Intelligence*, Vol. 21, No. 1-2, 1983, pp. 99-117.

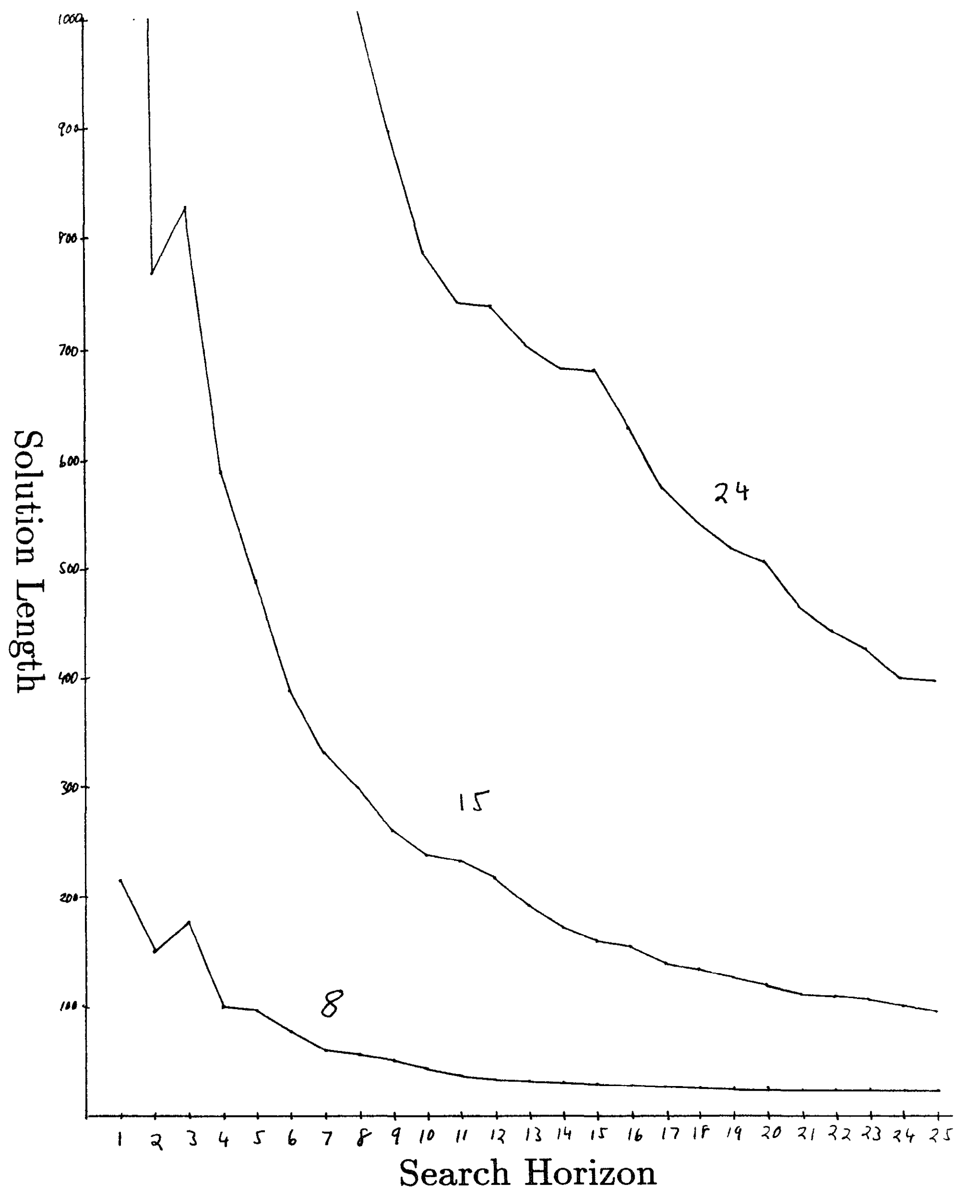

Figure 2: Search horizon vs. solution length for RTA*