The Journal of Real-Time Systems, 2, 7–24 (1990)

Depth-Limited Search for Real-Time Problem Solving

RICHARD E. KORF*
Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90024

Abstract. We propose depth-limited heuristic search as a general paradigm for real-time problem solving in a dynamic environment. When combined with iterative-deepening, it provides the ability to commit to an action almost instantaneously, but allows the quality of that decision to improve as long as time is available. Once a deadline is reached, the best decision arrived at is executed. We illustrate the paradigm in three different settings, corresponding to single-agent search, two-player games, and multi-agent problem solving. First we review two-player minimax search with alpha-beta pruning. Minimax can be extended to the *maxn* algorithm for more than two players, which admits a much weaker form of alpha-beta pruning. Finally, we explore real-time search algorithms for single-agent problems. Minimax is specialized to *minimin*, which allows a very powerful *alpha pruning* algorithm. In addition, *real-time-A** allows backtracking while still guaranteeing a solution and making locally optimal decisions.

1. Introduction

1.1. Search in AI

Search has a long and distinguished history in Artificial Intelligence. The earliest AI programs, such as the Logic Theorist of Newell, Simon and Shaw (1963), and Samuel's checkers player (1963), were heuristic search programs. The reason behind this is that higher level problem solving, such as theorem proving and game playing, was the first aspect of intelligence to receive the attention of AI researchers.

A classical example of a problem-solving task is the Travelling Salesperson Problem (TSP). The problem is to plan a trip among a set of cities so that every city is visited exactly once, and ending at the starting city. An optimal solution to this problem accomplishes the task in the minimum total distance possible. All known solutions to this problem amount to some sort of systematic trial-and-error exploration of different city sequences, searching for one of minimum distance.

In general, AI deals with problems that are sufficiently complex that the correct next step to be taken in their solution cannot be determined a priori. As a result, some type of search algorithm is employed to find a solution.

While search is a very general problem-solving technique, its efficiency limits its practical application. For example, the simplest algorithm for TSP would be to enumerate all

*This research was supported by an NSF Presidential Young Investigator Award, NSF Grant IRI-8801939, and an equipment grant from Hewlett-Packard. Thanks to Valerie Aylett for drawing the figures.

possible tours, calculate their distances, and return a shortest one. Unfortunately, the running time of this algorithm is proportional to the factorial of the number of cities. Even with the most efficient algorithms known for this problem, the largest supercomputers are incapable of finding optimal solutions to problems with as few as a hundred cities, within practical time limits.

If an optimal solution is not strictly required, however, there are algorithms that can find very good tours very quickly. In general, there is a tradeoff between the quality of solution found, measured in miles, versus the amount of computation necessary to find it, measured in time. The nature of this tradeoff is such that small sacrifices in solution quality often yield huge savings in computation time. Real-time problem solving often places severe constraints on the computation available for each decision. The challenge is to make the best decisions possible within those constraints.

1.2. Real-time search

The most successful artificial intelligence systems, measured by performance relative to humans in a complex domain, are state-of-the-art, two-player game programs. For example, the chess machine Deep Thought was recently rated at 2551, placing it among the top 30 players in the United States (Berliner 1989).

Game programs meet most of the usual criteria for real-time systems. Actions (moves) must be irrevocably committed to in constant time. Games are dynamic in that the strategic and tactical situation changes over time. Finally, there are at least two different sources of uncertainty in such games. One is uncertainty in predicting the opponent's moves, and the other is uncertainty in the value of a position due to the size of the search space.

The basic algorithm employed by these programs is fixed-depth lookahead search. From the current game situation, a tree of possible moves and possible responses is generated. The tree is generated only to a fixed depth, depending on the computational resources and time available per move. Then, based on a heuristic evaluation function applied to the frontier nodes of the tree, the best move from the current situation is decided upon and executed.

The key property of fixed-depth search that makes it amenable to real-time problem solving is that both computational complexity and decision quality is a function of search depth. With a very shallow depth, a decision can be made very quickly. As the search depth increases, decision quality improves, but at the cost of increased time. Thus, given a time deadline for a move, a search horizon is chosen so that the best possible move can be made before the deadline.

Since it is generally difficult to predict exactly how long a search to a given depth will take, iterative deepening is used to set the search horizon. The idea of iterative deepening is that a series of searches is performed, each to a greater depth than the previous. Then, when the deadline is reached, typically in the middle of a search, the move recommended by the last completed iteration is made. Thus, even with completely unpredictable deadlines, a plausible move is always available and can be executed. For a detailed discussion of iterative deepening, see (Korf 1985).

Fixed-depth search combined with iterative deepening forms the basic structure of real-time search programs. The remainder of this paper will describe the application of these ideas in three different settings: two-player games, multi-player games, and single-agent

problem solving. While the material on two-player games will be familiar to many, the latter two areas constitute recent results. Since they have been reported on in detail elsewhere (Korf 1988; Korf 1990; Korf 1990), we will only briefly survey their highlights.

2. Two-player games

One of the original challenges of AI, which in fact predates AI by a few years, was to build a program that could play chess at the level of the best human players. As such, chess is the canonical two-player game in AI.

2.1. Minimax Search

The standard algorithm for two-player games is called minimax search with static evaluation (Shannon 1950). The algorithm searches forward to some fixed depth in the game tree, limited by the amount of computation available per move. Figure 1 shows a two-level game tree for tic-tac-toe, starting from the initial state of the game and taking advantage of board symmetries. At this *search horizon*, a heuristic static evaluation function is applied to the frontier nodes. In the case of a two-player game, a heuristic evaluator is a function that takes a board position and returns a number that indicates how favorable that position is to one player or the other. For example, a very simple heuristic evaluator for chess would count the total number of pieces on the board for one player, appropriately weighted by their relative strength, and subtract the weighted sum of the opponent's pieces. Thus, large positive values would correspond to strong positions for one player whereas large negative values would represent advantageous situations for the opponent.

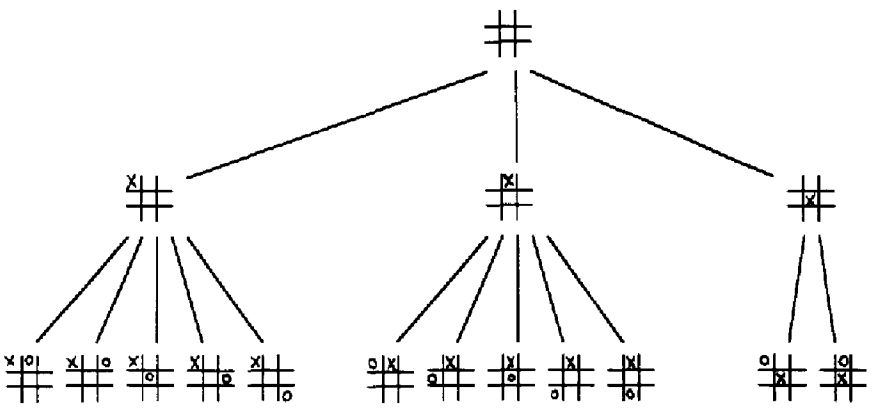

Figure 1. Two-Level Tic-Tac-Toe Game Tree.

Given the static evaluations of the frontier nodes, values for the interior nodes in the tree are computed according to the minimax rule. The player for whom large positive values are advantageous is called MAX, and conversely the opponent is referred to as MIN. The value of a node where it is MAX's turn to move is the maximum of the values of its children, while the value of a node where MIN is to move is the minimum of the values of its children. Thus, at alternate levels of the tree, the minimum and the maximum values of the children are backed up. This continues until the values of the immediate children of the current position are computed, at which point one move to the child with the maximum or minimum value is made, depending on the player to move. Figure 2 shows a minimax tree where square nodes represent positions where MAX is to move and circular nodes correspond to MIN's moves. The values above the bottom row are computed from the frontier values using the minimax rule.

2.2. *Alpha-beta pruning*

One of the most elegant ideas in all of heuristic search is the alpha-beta pruning algorithm. John McCarthy came up with the original idea (Pearl 1984), and it first appeared in print in a memo by Hart and Edwards (1963). The notion is that an exact minimax search can be performed without examining all the nodes at the search frontier.

Figure 3 shows an example of alpha-beta pruning applied to the tree of Figure 2. The search proceeds depth-first to minimize the memory requirement, and only evaluates a node when necessary. After statically evaluating nodes d and e to 6 and 5, respectively, we back up their maximum value, 6, as the value of node c. After statically evaluating node g as 8, we know that the backed up value of node f must be greater than or equal to 8, since it is the maximum of 8 and the unknown value of node w. The value of node b must be 6 then, because it is the minimum of 6 and a value which must be greater than or equal to 8. Since we have exactly determined the value of node b, we do not need to evaluate or even generate node w. This is called an alpha cutoff. Similarly, after statically evaluating nodes j and k to 2 and 1, the backed up value of node i is their maximum or 2. This tells us that the backed up value of node h must be less than or equal to 2, since

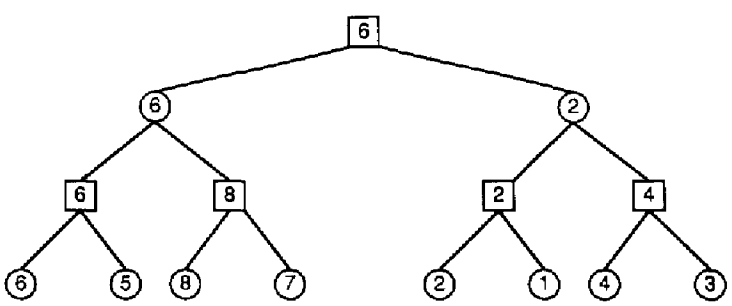

Figure 2. Minimax game tree.

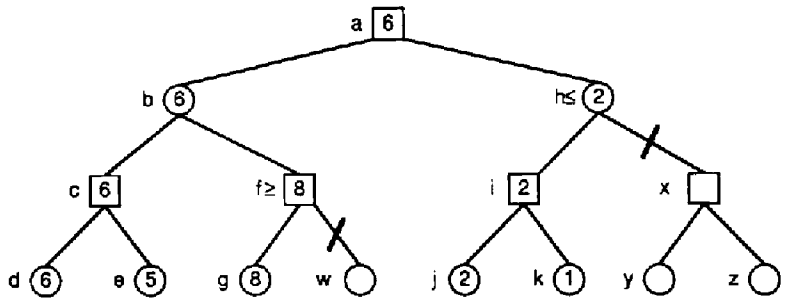

Figure 3. Alpha-beta pruning.

it is the minimum of 2 and the unknown value of node x. Since the value of node a is the maximum of 6 and a value that must be less than or equal to 2, it must be 6, and hence we have evaluated the root of the tree without generating or evaluating nodes x, y, or z. This is called a beta cutoff.

Since alpha-beta pruning allows us to perform a minimax search while evaluating fewer nodes, its effect is to allow us to search deeper with the same amount of computation. This raises the question of how much deeper, or how much does alpha-beta improve performance? This problem has been carefully studied by a number of researchers and finally solved by Pearl (Knath and Moore 1975; Pearl 1982). The best way to characterize the efficiency of a pruning algorithm is in terms of its *effective branching factor.* The effective branching factor is the d^{th} root of the number of frontier nodes that must be evaluated in a search to depth d.

The efficiency of alpha-beta pruning depends upon the order of the node values at the search frontier. For any set of frontier node values, there exists some ordering of the values such that alpha-beta will not perform any cutoffs at all. In that case, all frontier nodes must be evaluated and the effective branching factor is b, the brute-force branching factor.

On the other hand, there is an optimal or perfect ordering in which every possible cutoff is realized. In that case, the effective branching factor is reduced from b to $b^{1/2}$, which is the square root of the brute-force branching factor. Another way of reviewing the perfect ordering case is that for the same amount of computation, one can search twice as deeply with alpha-beta pruning as without. Since the search tree grows exponentially with depth, doubling the search horizon is quite a dramatic improvement.

In between worst-possible ordering and perfect ordering is random ordering, which is the average case. Under random ordering of the frontier nodes, alpha-beta pruning reduces the effective branching factor to approximately $b^{3/4}$. This means that one can search 4/3 as deep with alpha-beta, yielding a 33% improvement in search depth.

2.3. *Node ordering, quiescence, and iterative-deepening*

In practice, however, the effective branching factor of alpha-beta is closer to $b^{1/2}$ due to *node ordering*. The idea of node ordering is that instead of generating the nodes of the

tree strictly left-to-right, the order in which paths are explored can be based on static evaluations of the interior nodes in the tree. In other words, the children of MAX nodes can be expanded in decreasing order of their static values while the children of MIN nodes would be expanded in increasing order of their static values.

Another important notion is quiescence. The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the middle of a piece trade. In those positions, a small secondary search is conducted until the static evaluation becomes more stable.

Iterative-deepening is used to solve the problem of how to set the search horizon, as previously mentioned. In a tournament game, there is a limit on the amount of time allowed for moves. Unfortunately, it is very difficult to accurately predict how long it will take to perform a complete search to a given depth. If one picks too shallow a depth, then time which could be used to improve the move choice is wasted. Alternatively, if the search depth is too deep, time will run out in the middle of a search, and a move based on an incomplete search is likely to be very unreliable. The solution is to perform a series of complete searches to successively increasing depths. When time runs out, the move recommended by the last completed search is made.

Iterative-deepening and node ordering can be combined as follows. Instead of ordering interior nodes based on their static values, the frontier values from the previous iteration of the search can be used to order the nodes in the next iteration. This produces much better ordering than the static values alone.

Virtually all performance chess programs in existence today use full-width, fixed-depth, alpha-beta minimax search with node ordering, quiescence, and iterative-deepening. They make very high quality move decisions under real-time constraints.

3. Multi-player game trees

We now consider games with multiple players. For example, Chinese Checkers can involve up to six different players moving alternately. As another example, Othello can easily be extended to an arbitrary number of players by having different colored pieces for each player, and modifying the rules such that whenever a mixed row of opposing pieces is flanked on both sides by two pieces of the same player, then all the pieces are captured by the flanking player.

3.1. Maxn Algorithm

Luckhardt and Irani (1986) extended minimax to multi-player games, calling the resulting algorithm *maxn*. We assume that the players alternate moves, that each player tries to maximize his return, and is indifferent to the returns of the remaining players. At the leaf nodes, an evaluation function is applied that returns an N-tuple of values, with each component corresponding to the estimated merit of the position with respect to one of the players. Then, the value of each interior node where player i is to move is the entire N-tuple of the child for which the i^{th} component is a maximum. Figure 4 shows a maxn tree for three players, with the corresponding maxn values.

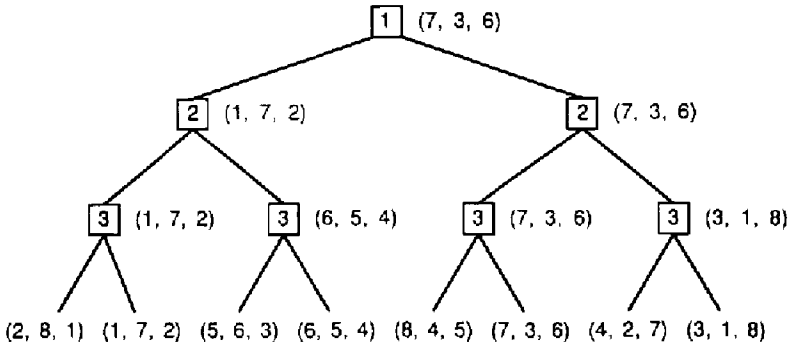

Figure 4. Example of maxn for three-player game.

For example, in Chinese Checkers, the value of each component of the evaluation function might be the negative of the minimum number of individual moves required to move all of the corresponding player's pieces to their goal positions. Similarly, an evaluation function for multi-player Othello might return the number of pieces for each player on the board at any given point.

Two-player minimax can be viewed as a special case of maxn in which the evaluation function returns an ordered pair of x and $-x$, and each player maximizes his component at his moves.

3.2. Alpha-beta pruning in multi-player game trees

Luckhardt and Irani (1986) observed that at nodes where player i is to move, only the i^{th} component of the children need be evaluated. At best, this can produce a constant factor improvement, but it may be no less expensive to compute all components than to compute only one. They correctly concluded that without further assumptions on the values of the components, pruning of entire branches is not possible with more than two players.

If, however, there is an upper bound on the sum of all components of a tuple, and there is a lower bound on the values of each component, then a form of alpha-beta pruning is possible. The first condition is a weaker form of the standard constant-sum assumption, which is in fact, required for two-player alpha-beta pruning. The second is equivalent to assuming a lower bound of zero on each component, since any other lower bound can be shifted to zero by subtracting it from every component. Most practical evaluation functions will satisfy both these conditions, since violating either one implies that the value of an individual component can be unbounded in at least one direction. For example, in the piece-count evaluation function described above for multi-player Othello, no player can have less than zero pieces on the board, and the total number of pieces on the board is the same for all nodes at the same level in the game tree, since exactly one piece is added at each move.

3.2.1. Immediate pruning. The simplest kind of pruning possible under these assumptions occurs when player i is to move, and the i^{th} component of one of his children equals the upper bound on the sum of all components. In that case, all remaining children can be pruned, since no child's i^{th} component can exceed the upper bound on the sum. We will refer to this as *immediate pruning*.

3.2.2. Shallow pruning. A more complex situation is called *shallow pruning* in the alpha-beta literature. Figure 5 shows an example of shallow pruning in a three-player game, where the sum of each component is 9. Evaluating node a results in a lower bound of 3 on the first component of the root, since player one is to move. This implies an upper bound on each of the remaining components of $9 - 3 = 6$. Evaluating node f produces a lower bound of 7 on the second component of node e, since player two is to move. Similarly, this implies an upper bound on the remaining components of $9 - 7 = 2$. Since the upper bound (2) on the first component of node e is less than or equal to the lower bound on the first component of the root (3), player one won't choose node e and its remaining children can be pruned. Similarly, evaluating node h causes its remaining brothers to be pruned. This is similar to the pruning in Figure 3.

3.2.3. Failure of deep pruning. In a two-player game, alpha-beta pruning allows an additional type of pruning known as *deep pruning*. In deep pruning, nodes are pruned based on bounds inherited from their great-great-grandparents. Surprisingly, deep pruning does not generalize to more than two players.

Figure 6 illustrates the problem. Again, the sum of each component is 9. Evaluating node b produces a lower bound of 5 on the first component of node a and hence an upper bound of $9 - 5 = 4$ on the remaining components. Evaluating node e results in a lower bound of 5 on the third component of node d and hence an upper bound of $9 - 5 = 4$ on the remaining components. Since the upper bound of 4 on the first component of node d is less than the lower bound of 5 on the first component of node a, the value of node f cannot become the value of node a. In a two-player game, this would allow us to prune node f.

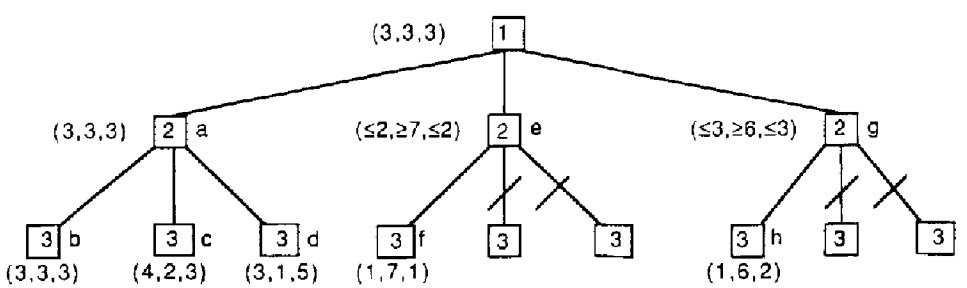

Figure 5. Shallow pruning in three-player game tree.

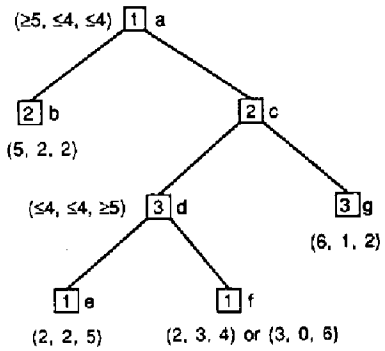

Figure 6. Failure of deep pruning for three players.

With three players, however, the value of node f could effect the value of the root, depending on the value of node g. If the value of node f were (2,3,4) for example, the value of e would be propagated to d, the value of d would be propagated to c, and the value of b would be propagated to a, giving a value of (5,2,2). On the other hand, if the value of node f were (3,0,6) for example, then the value of f would be propagated to d, the value of g would be propagated to c, and the value of c would be propagated to a, producing a value of (6,1,2). Even though the value of node f cannot *be* the maxn value of the root, it can *effect* it. Hence, it cannot be pruned.

In general, shallow pruning, which is understood to include immediate pruning as a special case, is the best we can do, as expressed by the following theorem:

THEOREM 1. Every directional algorithm that computes the maxn value of a game tree with more than two players must evaluate every terminal node evaluated by shallow pruning.

By a directional algorithm we mean one in which the order of node evaluation is independent of the value of the nodes, and once a node is pruned it can never be revisited. For example, a strictly left-to-right order would be directional. The main idea of the proof amounts to a generalization of the above example to variable values, arbitrary depths, and any number of players greater than two.

3.2.4. Performance of shallow pruning. How effective is shallow pruning? The best-case analysis of shallow pruning is independent of the number of players and was done by Knuth and Moore (1975) for two players. The best-case effective branching factor is $(1 + \sqrt{4b - 3})/2$. For large values of b, this approaches $\sqrt{b}$ which is the best-case performance of full two-player alpha-beta pruning.

Knuth and Moore (1975) also determined that in the average case, the asymptotic branching factor of two-player shallow pruning is approximately $b/\log b$. In the case of multiple-players, however, the average-case asymptotic branching factor of shallow pruning is simply b, the brute-force branching factor. Thus, pruning does not produce an asymptotic improvement in the average case with more than two players. Similarly, in the worst case, every terminal node would have to be evaluated and no pruning would take place.

As in the case of two-player minimax, iterative deepening can be applied to multi-player game trees as well. Multiple iterations to successively deeper search depths are performed and the move recommended by the last completed iteration at the deadline is made.

4. Single-agent problem solving

We now turn our attention to real-time problem solving by a single agent. Common examples of single-agent search problems are the Eight Puzzle and its larger relatives, the Fifteen and Twenty-four Puzzles (see Figure 7). The Eight 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.

In a single-agent problem, a heuristic function estimates the cost of a path from a given node to a goal node. A common heuristic function for sliding tile puzzles 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 vehicle 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 airline distance from a given location to the goal location.

Most of the work in single-agent search has not addressed real-time constraints. For example, the best known algorithm is A* (Hart, Nilsson and Raphael 1968). 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.

	1	2
3	4	5
6	7	8

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 7. Eight, fifteen, and twenty-four puzzles.

A serious drawback of A*, however, is that it takes exponential time to run in practice. This is an unavoidable cost of obtaining optimal solutions, and restricts the applicability of the algorithm to relatively small problems in practice. For example, while A* with the Manhattan Distance heuristic function can solve the Eight Puzzle, and a linear-space variation called IDA* (Korf 1985) can solve the Fifteen Puzzle, any larger puzzle is intractable on current machines. A related drawback of A* and IDA* is that they must search all the way to a solution in a planning or simulation phase before executing even the first move in the solution.

In this section we apply the real-time assumptions of 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, larger versions of the Fifteen Puzzle impose a computational limit on the problem solver. 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. This means that after a constant amount of time we commit to a physical action, such as sliding a tile or moving the vehicle. If we later decide to slide the same tile back to where it was, or drive the vehicle back, both moves are counted in the solution cost.

4.1. 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. The resulting algorithm, which we call *minimin search*, was originally developed by Rosenberg and Kestner (1972) in the context of A*. 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 actually executed 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.

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.

4.2. *Alpha pruning*

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. Since monotonicity of f is equivalent to h obeying the triangle inequality characteristic of all metrics, it is satisfied by all naturally occurring heuristic functions including Manhattan Distance and Euclidean Distance. Thus, monotonicity of f is 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.

4.2.1. Efficiency of Alpha pruning. Figure 8 shows a comparison of the total number of nodes examined as a function of search horizon for several different sliding tile puzzles, including the Eight, Fifteen, Twenty-Four and 10 × 10 Ninety-Nine Puzzle. 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 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 percent 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. By 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

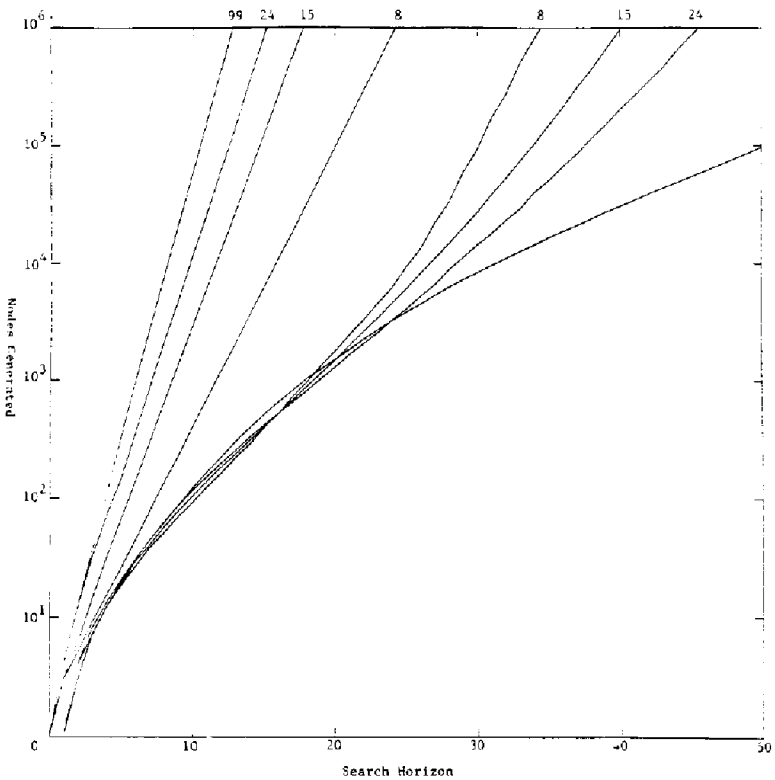

Figure 8. Search horizon vs. nodes generated for brute-force and alpha pruning search.

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.

*4.3. Real-Time-A**

Since minimin search with alpha pruning returns a backed-up value for a node, it can be viewed as simply a more accurate and computationally expensive heuristic function. As such, it doesn't actually execute moves but merely simulates them. We now turn our attention to the problem of controlling the sequence of moves executed. The two-player game approach of simply repeating the lookahead algorithm for each decision won't work in the single-agent case; 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 a best-first search given this different cost function.

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.

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.

For example, consider the graph in Figure 9, 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 node a in the hash table with 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, $f(b) = g(b) + h(b) = 1 + 5 = 6$, $f(c) = g(c) +$

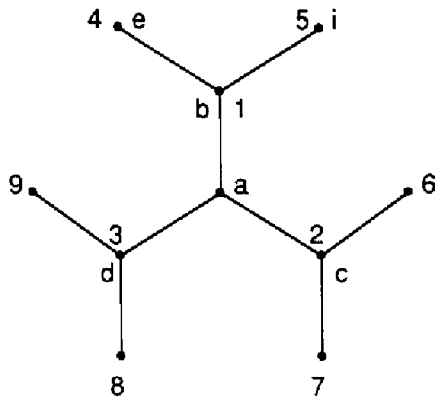

Figure 9. Real-time A* example.

$h(c) = 1 + 2 = 3$, 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.

4.3.1. Completeness and Correctness of RTA*. Note that while monotonicity of the cost 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 indicated by the following result:

THEOREM 2. In a finite problem space with finite positive edge costs and finite heuristic values, in which a goal state is reachable from every state, RTA* will find a solution.

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 locally optimal moves relative to the part of the search space that it has seen so far.

THEOREM 3. 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.

4.3.2. 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 10 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.

The average clock times to find solutions using short search horizons is less than a tenth of a second for the Eight Puzzle, less than half a second for the 15 Puzzle, and less than 2.5 seconds for the 24 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 (Korf 1985). Furthermore, RTA* has successfully solved sliding tile puzzles up to the 10 × 10 ninety-nine puzzle.

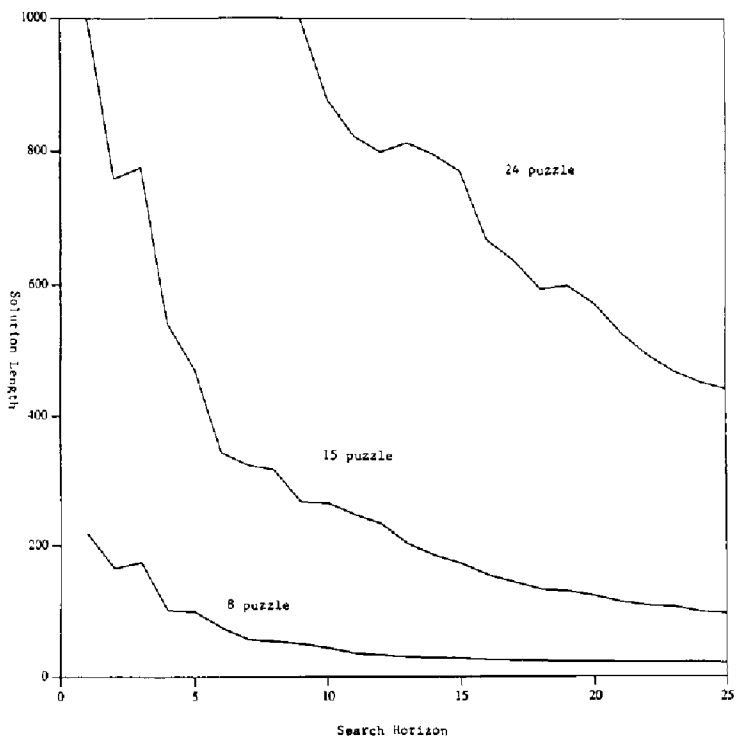

Figure 10. Search horizon vs. solution length for RTA*.

Iterative-deepening can easily be applied to RTA* as well. Instead of a fixed search horizon for minimin and alpha pruning, successive iterations to deeper thresholds are executed until the move deadline is reached. At that point the move recommended by the last completed iteration is executed.

5. Conclusions

We have examined fixed-depth lookahead search as a paradigm for real-time problem solving in three different settings: two-player games, multi-player games, and single-agent problems.

Fixed-depth minimax search is the basic two-player game algorithm. Alpha-beta pruning can double the achievable search horizon without effecting the quality of decisions made. When combined with node ordering, quiescence, and iterative-deepening, these techniques have been remarkably successful in producing very high performance programs under real-time constraints.

Allowing more than two players leads to a generalization of the minimax algorithm called maxn. If we further assume that there is a lower bound on each component of the evaluation function, and an upper bound on the sum of all components, then shallow alpha-beta pruning is possible, but not deep pruning. In the best case, this results in significant savings in computation, but in the average case it does not reduce the asymptotic branching factor.

Most 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. Alpha pruning dramatically improves the efficiency of the algorithm without affecting the decisions made, and the achievable search horizon with this algorithm increases with increasing branching factor. Real-Time-A* efficiently solves the problem of when to abandon 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. These algorithms effectively solve larger single-agent problems than have previously been solvable using heuristic evaluation functions.

In each case, fixed-depth search combined with iterative-deepening allows the problem solver to execute a decision at any point in time, while continuing to refine its decisions as long as a commitment is not required. Furthermore, it focuses the attention of the problem solver on the most immediate consequences of its actions, and only considers later ramifications as time permits.

In general, we have modelled the computational constraints of a real-time problem as simply a strict deadline on each decision. In more complex situations, there may not be strict deadlines, but rather increasing cost with time of delay, and/or a problem of allocating fixed computational resources over multiple decisions. These situations call for more sophisticated resource management, and is being pursued by several groups (Horvitz, Cooper and Heckerman 1989; Boddy and Dean 1989; Russell and Wefald 1989) under the general model of decision-theoretic control of computation.

References

Newell, A., H.A. Simon, and J.C. Shaw. 1963. Empirical explorations with the logic theory machine: A case study in heuristics. In *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.). New York: McGraw-Hill.

Samuel, A.L. 1963. Some studies in machine learning using the game of checkers. In *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.), New York: McGraw-Hill.

Berliner, H. 1989. Deep-Thought wins Fredkin Intermediate Prize. *AI Magazine*, 10, 2, (Summer).

Korf, R.E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27, 1:97–109.

Korf, R.E. 1988. Search in AI: A survey of recent results. In *Exploring Artificial Intelligence*. Los Altos, CA: Morgan-Kaufmann.

Korf, R.E. 1990. Mult-player alpha-beta pruning. *Artificial Intelligence* (to appear).

Korf, R.E. 1990. Real-time heuristic search. *Artificial Intelligence* (to appear).

Shannon, C.E. 1980. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41:256–275.

Pearl, J. 1984. *Heuristics*. Reading, MA: Addison-Wesley.

Hart, T.P., and D.J. Edwards. 1963. The alpha-beta heuristic. M.I.T. Artificial Intelligence Project Memo, Massachusetts Institute of Technology, Cambridge, MA, October.

Knuth, D.E., and R.E. Moore. 1975. An analysis of Alpha-beta pruning. *Artificial Intelligence*, 6,4:293–326.

Pearl, J. 1982. The solution for the branching factor of the Alpha-Beta pruning algorithm and its optimality. *Communications of the Association of Computing Machinery*, 25, 8:559–564.

Luckhardt, C.A., and K.B. Irani. 1986. An algorithmic solution of N-person games. *Proceedings of the National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, (Aug):158–162.

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

Rosenberg, R.S., and J. Kestner. 1972. Look-ahead and one-person games. *Journal of Cybernetics*, 2, 4:27–42.

Horvitz, E.J., G.F. Cooper, and D.E. Heckerman. 1989. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, (Aug):1121–1127.

Boddy, M., and T. Dean. 1989. Solving time-dependent planning problems. In *Proceedings of the International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, (Aug): 979–984.

Russell, S., and E. Wefald. 1989. On optimal game-tree search using rational meta-reasoning. In *Proceedings of the International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, (Aug):334–340.