AIRS: Anytime Iterative Refinement of a Solution

Sam J. Estrem and Kurt D. Krebsbach
Department of Mathematics and Computer Science
Lawrence University
Appleton, Wisconsin 54911
{samuel.j.estrem,kurt.krebsbach}@lawrence.edu

Abstract

Many exponentially-hard problems can be solved by searching through a space of states to determine a sequence of steps constituting a solution. Algorithms that produce optimal solutions (e.g., shortest path) generally require greater computational resources (e.g., time) than their sub-optimal counterparts. Consequently, many optimal algorithms cannot produce any usable solution when the amount of time available is limited or hard to predict in advance. *Anytime algorithms* address this problem by initially finding a suboptimal solution very quickly and then generating incrementally better solutions with additional time, effectively providing the best solution generated so far anytime it is required. In this research, we generate initial solutions cheaply using a fast search algorithm. We then improve this low-quality solution by identifying subsequences of steps that appear, based on heuristic estimates, to be considerably longer than necessary. Finally, we perform a more expensive search between the endpoints of each subsequence to find a shorter connecting path. We will show that this improves the overall solution incrementally over time while always having a valid solution to return whenever time runs out. We present results that demonstrate in several problem domains that AIRS (Anytime Iterative Refinement of a Solution) rivals other widely-used and recognized anytime algorithms and also produces results comparable to other popular (but not anytime) heuristic algorithms such as Bidirectional A* search.

Motivation: Greedy Plateaus

Inexpensive searches can be used to generate low-quality solutions quickly. In particular, we begin by using best-first greedy search ("Greedy")—in which the search is guided solely by the heuristic estimated distance to the goal (denoted h)—to generate an initial low-quality solution. In domains in which "low-quality" implies "longer" (e.g., more actions), these long solutions often contain one or more greedy plateaus. A greedy plateau is comprised of a sequence of states that all remain at approximately the same *estimated* distance (h value) from the goal. This apparent "orbit" of the goal can often make up the majority of the solution.

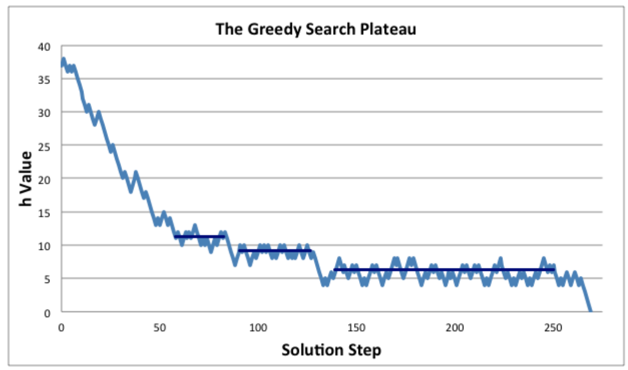

Figure 1: This graph shows a low-quality (270-step) solution with three greedy plateaus. For each step on the x-axis, the estimated distance from that state to the goal (h value) is plotted.

Figure 1 provides a Greedy Solution in which we observe three such plateaus: one from steps 50-80, another from 90-130, and the third from 150-250. Greedy plateaus often result from greedy searches where h, while admissible, badly underestimates the actual remaining distance to the goal. After most of the states on the "orbit" are visited, h values eventually improve only to settle on other plateaus later, as shown above.

The motivation for the algorithm we present, called AIRS, is based on the observation that most plateaus should be fairly easy to identify and to shorten with a better, more memory-intensive search (e.g., Bidirectional A*). We also observe that the larger the number of states on a plateau (with the same h value), the greater is the probability that pairs of states near the extremes of the plateau will have a much shorter path between them than is reflected in the greedy solution. The unnecessarily longer path can then be replaced with the short-cut, eliminating the wasteful segment. This is the crux of iterative refinement as embodied in the AIRS algorithm.

AIRS

Initial vs. Refinement Search

AIRS is a modular algorithm which allows the use of any two searching algorithms as the *initial* and the *refinement* algorithms. The initial algorithm is used to generate an inexpensive but low-quality solution. The refinement search algorithm is generally more expensive (both in time and memory) and attempts to "patch" the current solution by searching between chosen points to find a shorter path between them. We note that memory-intensive search algorithms in particular—which might require far too much memory to be used to generate an entire solution—can often be exploited in the refinement stage because the depth of the sub-solution search is only a small fraction of the overall solution depth, drastically reducing the exponent of the chosen algorithm's exponential memory requirement.

In this paper, we use two versions of the A* algorithm: Weighted A* (WA*) for the initial solution and Bidirectional A* (BidA*) for the refinement algorithm. Recall that A* (Hart, Nilsson, and Raphael 1968; 1972) defines the state evaluation function $f = g + h$, with includes the accumulated distance along a path (denoted g) together with the estimate h. WA* is a version of A* that weights the g and h values. Standard WA*uses $f = \epsilon * g + h$ where $\epsilon \leq 1$. When $\epsilon = 1$, the algorithm becomes A* and produces optimal results. Anytime Weighted A*(AWA*) uses a generated ϵ value which increases after each search. We use a starting value of $\epsilon = 0.3$ with a 0.2 increase in each iteration.

BidA* (Pohl 1971) is a version of A* where instead of one search, two A* searches are initiated in opposition to one another. On a sequential processor, the two searches alternate, each expanding a node and then checking whether or not there exists an intersection between their respective fringes. The version we use for this paper checks to see if the children of each expanded node intersect the opposing fringe to determine if there exists a solution path. While this often results in a near-optimal solution, it does not guarantee optimality, as the full-blown BidA* would. Because we only seek to refine a particularly wasteful segment of the solution, and because we cannot expect the incremental (refined) solution to be optimal anyway, we cannot justify using the optimal version because the time and memory required to guarantee optimality (i.e., to continue searching after the first fringe intersection is found) has been shown to require exponential space just like unidirectional A* (Kaindl and Kainz 1997) in the worst case, while the near-optimal version is a superior trade-off as a refinement algorithm.

Iterative Refinement

The number of states explored with an expensive search is exponential in the length of the solution. AIRS suggests an alternative by first generating a quick solution using a cheap search like Greedy or WA* with a small weight on g. AIRS then analyzes that solution and computes what it believes to be a "balloon" defined by two points at the extremes of the plateau the appear to be in close proximity based on estimated distance between them (which we call h_2). Then, using a more expensive search such as A*, BidA*, or WA*, AIRS attempts to replace the balloon with a "bridge" connecting these apparently close states, and shortening the overall solution.

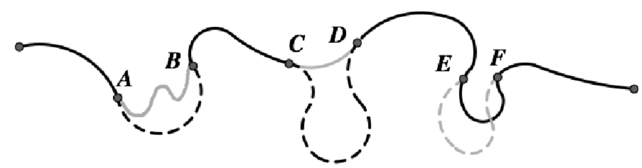

Figure 2: An intuitive example of an AIRS solution path after refinement process is complete. Solid lines depict the final path, with dashed lines showing either discarded balloons or unacceptable bridges.

Figure 2 depicts a solution path at the end of the AIRS refinement process. The continuous line represents the final solution. Three pairs of points representing balloons—(A, B), (C, D), and (E, F)—had been chosen at some point by AIRS as candidates for refinement. In part, these balloons were chosen because the endpoints appear to be close to each other in the state space. The dashed lines between pairs (A, B) and (C, D) indicate the paths found between these points in the initial (low-quality) solution that were then later replaced with a shorter path, as shown with a grey, solid line. In contrast, the grey dashed line shown between states E and F is a refinement that was actually worse than the original sequence, and so the original is retained. In cases where a refinement is ignored, the endpoints (E, F) are cached and later used by AIRS to avoid repeating already-failed (but good-looking) searches.

Pairs of points on the current solution continue to be chosen in prioritized order and (possibly) refined in this manner. When refinements are made, new sequences of states are introduced, opening up new possibilities for further refinement.

The AIRS Algorithm

We now provide a description of the AIRS algorithm. In Line 1, we must store the minimum cost for the domain. The minimum cost is the smallest action cost within the domain. For example, the 15-Puzzle's minimum cost would be 1 because there is no action which costs less than 1. This is used later in computing the *Ratio* (Line 11) to weight it toward larger refinements in case two pairs of states return the same ratio. In Line 2, AIRS initially computes a low-quality solution generated by a fast and suboptimal algorithm such as Greedy ($f = h$) or an appropriately-weighted WA*. AIRS also stores all previous failed searches in a list

Function AIRS(Problem)

```
1  p ← Minimum Cost;
2  Sol ← GreedySearch(Problem_Start,Problem_Goal);
3  FS ← ∅ ;
4  while TimeRemaining > 0 do
5      (Bx, By) ← (−1, −1) ;
6      Br ← Intetger.MAXVALUE ;
7      x ← 0;
8      while x < Length(Sol ) − 2 do
9          y ← x + 2;
10         while y < Length(Sol ) do
11             Ratio ← h2(Sol_x, Sol_y) / (g(Sol_y) − g(Sol_x) − p) +
                   h2(Sol_x, Sol_y) / ((g(Sol_y) − g(Sol_x) − p) − MaxOverlap(x + 1,y − 1,Sol,FS ));
12             if Ratio < Br then
13                 Br ← Ratio;
14                 (Bx,By)← (x, y);
15             y ← y + α;
16         x ← x + β;
17     FixedSect ← BidirectionalA*(Sol_Bx, Sol_By);
18     if g(FixedSect_last) < g(Sol_By) − g(Sol_Bx) then
19         Sol ← Sol_0,Bx−1 + FixedSect + Sol_By+1,last;
20         clear(FS);
21     else
22         FS ← FS ⋃(Bx, By);
23 return Sol;
```

Function MaxOverlap(start, end, Sol, fails)

```
24 Greatest ← 0;
25 x ← 0;
26 while x < Length(fails ) do
27     q ← fails_x.second;
28     s ← fails_x.first;
29     overlap ← Min(g(Sol_end),g(Sol_q)) −
       Max(g(Sol_start),g(Sol_s));
30     if overlap > Greatest then
31         Greatest ← overlap;
32     x ← x + 1;
33 return Greatest;
```

(FS) which is initially empty (Line 3). Given the initial solution, AIRS then attempts to identify which pair of states along the solution appear to be in close proximity in the state space, but for which the current path appears disproportionately long. By computing a shortcut between these states with a more expensive algorithm (e.g., BIDA*), AIRS attempts to shorten the overall solution with a minimum of additional search.

The function $h_2(s_i, s_j)$ is the function within a problem domain which estimates the distance between two states. In contrast, the function h is a function of one argument that estimates the distance between a state and the nearest goal. To determine the best candidate set of points, it performs an $O(n^2)$ computation by iterating through a subset of all pairs of states (s_i, s_j) on the solution path $s_1, s_2, \ldots, s_n$ where $i < j$ and n is the current solution length (Lines 8-16), and where α and β (Lines 15-16) parameterize the resolution of the subset selected. For each pair, it computes a specialized ratio to determine how much a search between the two states would benefit the solution (Line 11). This ratio compares the estimated distance between the states—according to the function h_2— by the current distance along the current solution and then adds on a weighting factor designed to keep the algorithm from repeating previous failed searches by using the function $MaxOverlap$.

The function $MaxOverlap$ first iterates through all the failed searches since the most recent successful refinement (Line 26). Given each pair, it computes the degree of overlap between that segment and the search defined by the two inputs *start* and *end* (Line 29). If the overlap is larger than the greatest overlap so far, it updates the greatest overlap with that value (Lines 30-31). Once it iterates through all the failed searches, it return the greatest overlap found (Line 33).

Once the two states have been selected, AIRS uses a second, more expensive search algorithm to find a path between them (e.g., BiDA* search, as in Line 17). If the solution returned is shorter than the current path between the chosen states, we update the solution to use the new path and clear the list of previous failed searches (Lines 18-20). If the returned solution is longer than the current path, we add the pair of chosen states to the list of failed searches in Lines 21-22. AIRS repeats this process in anytime fashion until time runs out, at which time it returns the current solution.

AIRS as an Anytime Algorithm

We take this opportunity to observe that AIRS is ideally-suited for use as an *anytime algorithm.* Anytime algorithms are flexible algorithms designed to return the best solution possible in the time available, but without knowing how much time is available in advance. We will show that AIRS has the two crucial properties required of an effective anytime algorithm. First, AIRS finds an initial solution very quickly: a crucial property of a algorithm that might be asked to return a solution in very little time. Second, the iterative refinements performed by AIRS are also relatively fast as compared to common competing algorithms (e.g., WA*). In particular, we will show that performing expensive search over short segments results is shorter iterations, which is beneficial so that when the clock runs out, the probability of wasting time on an "almost completed refinement" is minimal. (This can be a problem for WA*, in which consecutive searches take longer and longer as ϵ grows).

Still, the AIRS algorithm—like any algorithm—has important trade-offs to make. For an initial (poor) solution length of n steps, AIRS could choose to perform an entire $O(n^2)$ computation to check all orderings of pairs states to consider as the best "balloon" to refine. As will be seen shortly, this computation is generally small in comparison to the resulting expensive search that follows it.

We also observe that many of the expensive searches fail to find a significantly better solution on a subsequence of the current solution. As we show in the empirical results, however, *it is still worthwhile* in terms of time and space to attempt these refinements. In other words, even counting the time wasted in generating and attempting to short-circuit balloons in vain, the successful refinements are still cheaper and more effective than just using a better but more expensive algorithm in the first place.

Empirical Results

We now discuss some empirical results obtained in two domains: Fifteen Puzzle and Terrain Navigation.

Fifteen Puzzle Domain

Figure 3 shows the comparison of solution lengths for AIRS against Anytime Weighted A* given the amount of time it took Bidirectional A* to complete each of the 100 Korf Fifteen Puzzles (Korf 1985).

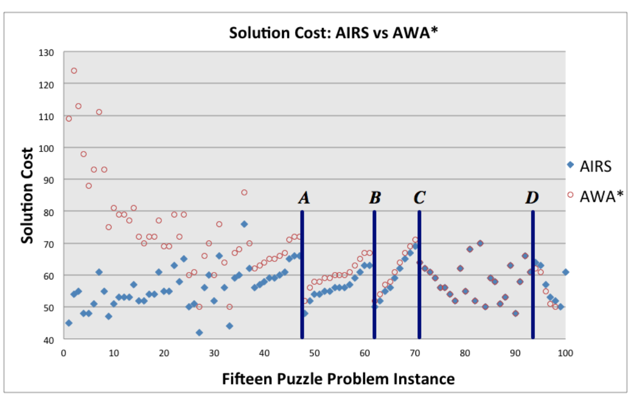

Figure 3: AIRS vs WeightedA* for 100 Korf 15-Puzzles.

As shown in Figure 3, AIRS compares favorably to another anytime algorithm, WA*, on the Korf Fifteen Puzzle benchmark. Because anytime algorithms are designed to return a solution at "any time," we ran a non-anytime algorithm, BidA*on each of the 100 Korf problems first to establish a baseline. Then, for each problem, we gave both AWA* and AIRS this amount of time to produce a solution in anytime fashion. Figure 3 shows the solution quality for each algorithm for each problem.

The x-axis represents a specific Fifteen Puzzle problem instance and the corresponding y is the cost of the final solution produced by each algorithm. The 100 problem instances are sorted from left to right based on the difference in cost between AIRS and AWA*; i.e., problems in which AIRS outperformed AWA* are further to the left. We see from this that AIRS significantly outperforms AWA* in 47% of the problems (i.e., as indicated by vertical line A). In approximately 23% of the problems, AIRS outperforms AWA* by a small margin. In about 22%, both algorithms achieve solutions of the same length and in the last 8%, AWA*'s solutions are better.

In general, AWA* suffers from infrequent iterations and thus takes much longer to complete each iteration to produce a shorter solution. AIRS largely outperforms Anytime Weighted A* due to its ability to make continuous small improvements, which is a useful property in "anytime" situations. Counter-intuitively, we have also observed that AIRS can actually, at times, converge to a higher-quality solution more quickly starting with a terrible solution than it can with a better initial solution, even given the same amount of time in each case. This can happen in domains that are structured in a way that large "near cycles" can appear in the initial solution, but can easily be refined away in a single refinement iteration.

Terrain Navigation Domain

We now turn our attention to a second domain, namely Terrain Navigation (TerrainNav). TerrainNav is inspired by so-called "grid world" domains, but has been elaborated for our purposes here, as we note that in a standard grid world without obstacles or non-uniform costs, an iterative refinement approach is not appropriate. At one extreme (e.g., a minimum spanning tree), there may be a unique path which, by definition, admits no possibility of iterative refinement. At the other extreme, in domains where many equally good paths exist, even Greedy often finds one in the initial search, again leaving little or no room for refinement.

In TerrainNav, each coordinate on a grid is given a weight to represent its height. A larger difference in heights between subsequent steps means a larger cost. "Mountains" of various height and extent are placed throughout the map, sharply raising the weight on a specific coordinate and probabilistically raising the weights around the peak proportional to the peak's height and the distance from the peak. For TerrainNav, we use Greedy for the initial search with Euclidean distance as the heuristic estimate (h). Because h ignores terrain costs, Greedy walks "through" each mountain

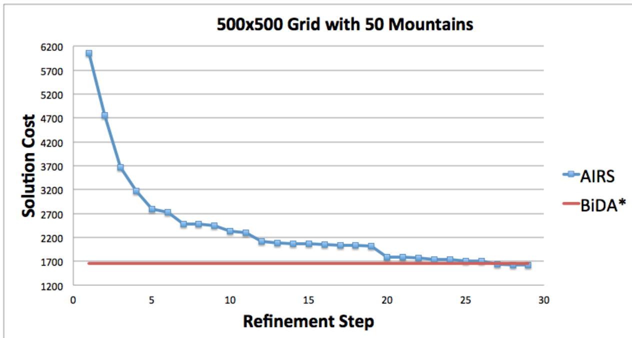

Figure 4: Solution cost after each AIRS refinement step when AIRS is used as an anytime algorithm and given the same amount of time as BidA*.

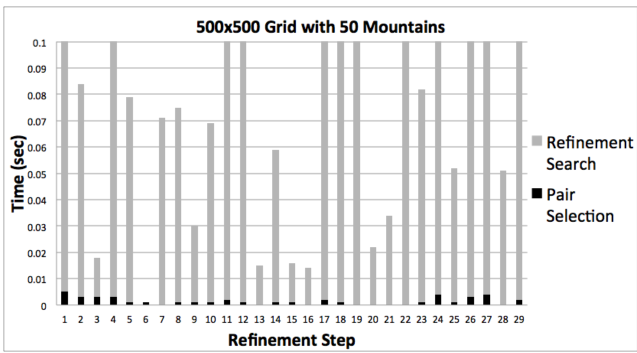

Figure 5: For each refinement, the black section shows time spent on pair selection, with the grey showing refinement (BidA*) search time.

on its way to a fast and suboptimal solution.[1]

In Figures 4 and 6, we compare the AIRS solution cost after each refinement step to that of the overall BidA* solution cost. (Note that horizontal lines for BidA* are $y \approx 1700$ and $y \approx 400$, and are shown for reference even though BidA* is only run once, and takes the same amount of time to run as we allow AIRS to run.) The y-axis represents the solution costs for each method while the x-axis represents the i^{th} iteration of successful AIRS refinement. We do not plot failed attempts at refinement, but rather consider them (and their time) as part of the process of a successful refinement. We note that early refinements produce a drastic reduction in the solution cost, with later ones continuing the refinement at a reduced pace.

Figures 5 and 7 compare the time spent on each phase of refinement. For each refinement, the black section shows time spent on pair selection, with the grey showing the localized refinement (BidA*) search time. The y-axis represents time in seconds, and the x-axis represents the i^{th} iteration of refinement. Again, the i^{th} iteration consists of the total time spent doing all pair selection and BidA* (refinement) searches performed between *successful* improvements of the solution.

[1]We note that this method of solving a *relaxed* version of the problem quickly is reminiscent of current planning approaches that solve a simplified planning problem quickly, and then use distances in the "relaxed solution" as heuristic estimates during the real planning search. The difference here is that we choose portions of the relaxed solution to refine directly using shorter searches.

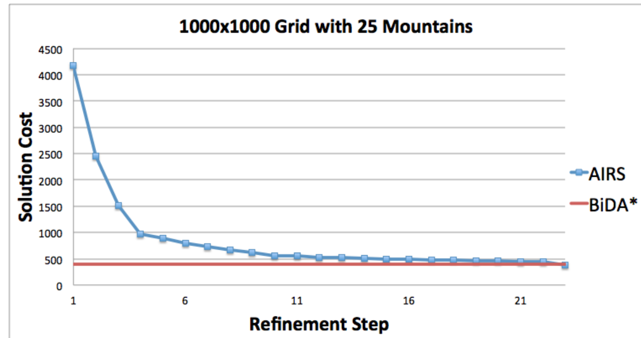

Figure 6: Same as Figure 4 with different terrain parameters.

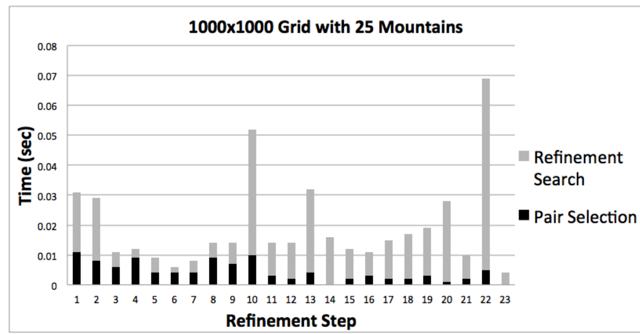

Figure 7: Same as Figure 5 with different terrain parameters.

We observe from the time-based graphs (Figures 5 and 7) that approximately the same amount of time was used in pair selection, regardless of the solution length from one iteration to the next. This is due to two factors. First, as the current solution length decreases, the $O(n^2)$ search space decreases quadratically. Second, as mentioned earlier, the specific pair that AIRS selects is not from the set of all states, but from a uniform sampling which is restricted based on several parameters, including resolution parameters α and β (Lines 15-16), and the current solution length. By restricting the space in a systematic way, we can drastically reduce the time to identify the next pair, while slightly increasing the probability of having to perform multiple BidA* searches to find a successful one. We strike a balance in our algorithm, but any AIRS search should include this as a tunable parameter for best results.

We see an example of this in Figure 7 from steps 3-9. In this range, we can see that both pair selection and BidA* took about the same amount of time because

AIRS finds a successful refinement with only a single BidA* search. Comparing this against Figure 6, we see that those same refinements drastically reduced the length of the solution. This is the balance we want. In contrast, in Figure 5, we seen that BidA* takes much more time than pair selection. This specific problem is, in fact, a difficult one. Early on, BidA* does not take an excessive amount of time even though there are multiple searches happening per refinement, but later, the trade-off does not work ideally because the BidA* searches are hard. We are currently working on a more flexible mechanism to more intelligently trade off time between these two AIRS phases.

Related Work

Anytime algorithms were first proposed as a technique for planning when the time available to produce a plan is unpredictable and the quality of the resulting plan is a function of computation time (Dean and Boddy 1988). While many algorithms have been subsequently cast as anytime algorithms, of particular interest to us are applications of this idea to heuristic search.

For example, Weighted A*, first proposed by Pohl (1970), has become a popular anytime candidate, in part because it has been shown that the cost of the first solution will not exceed the optimal cost by a factor of greater than $1 + \epsilon$, where ϵ depends on the weight (Pearl 1984). Hansen and Zhou provide a thorough analysis of Anytime Weighted A* (WA*), and make the simple but useful observation that there is no reason to stop a non-admissible search after the first solution is found (Hansen and Zhou 2007). They describe how to convert A* into an anytime algorithm that eventually converges to an optimal solution, and demonstrate the generality of their approach by transforming the memory-efficient Recursive Best-First Search (RBFS) into an anytime algorithm. An interesting variant of this idea, called Anytime Repairing A* (ARA*) was proposed initially for real-time robot path planning, and makes two modifications to AWA* which include reducing the weight between searches and limiting node re-expansions (Likhachev, Gordon, and Thrun 2004).

Conclusion

We introduce a new algorithm, AIRS (Anytime Iterative Refinement of a Solution), that divides a search problem into two phases: Initial Search and Refinement Search. The user is free to choose specific search algorithms to be used in each phase, with the idea being that we generate an initial solution cheaply using a fast but sub-optimal search algorithm, and refine relatively short portions of the evolving solution with a slower, more memory-intensive search. An important contribution of our method is the efficient identification of subsequences of solution steps that appear, based on heuristic estimates, to be considerably longer than necessary. Once identified, if the refinement search computes a shorter connecting path, the shorter path is substituted and the current solution path is incrementally improved. We emphasize that this method is ideal as an anytime algorithm, as it always has a valid solution to return when one is required, and, given the way subsequences are chosen, refinement iterations are kept quite short – reducing the probability of wasting expensive search time by failing to complete a refinement just before time happens to run out. Finally, we present results that demonstrate in several problem domains that AIRS rivals other popular search choices for anytime algorithms.

References

Dean, T., and Boddy, M. 1988. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, 49–54. St. Paul, Minnesota: Morgan Kaufmann.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28:267–297.

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

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. Correction to A formal basis for the heuristic determination of minimum cost paths. *SIGART Newsletter* 37:28–29.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* cs.AI/9712102.

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

Likhachev, M.; Gordon, G.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proc. Neural Information Processing Systems,(NIPS-03*. MIT Press.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading, Massachusetts: Addison-Wesley.

Pohl, I. 1970. First results on the effect of error in heuristic search. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 5*. Amsterdam, London, New York: Elsevier/North-Holland. 219–236.

Pohl, I. 1971. Bi-directional search. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 6*. Edinburgh, Scotland: Edinburgh University Press. 127–140.