PBA*: Using Proactive Search to Make A* Robust to Unplanned Deviations

Paul Breimyer
pwbreimy@ncsu.edu
North Carolina State University
Raleigh, NC 27695

Peter R. Wurman
wurman@ncsu.edu
North Carolina State University
Raleigh, NC 27695

Abstract

Many path planning algorithms leverage A* to determine optimal paths. However, when an actor deviates from the optimal path, a typical application of A* executes a new search from the deviation point to the goal. This approach redundantly calculates paths that may have been examined during the initial search, rather than leveraging previous information. We introduce Plan-B A* (PBA*), which uses A* for the initial search, and substantially reduces the number of searched states during all subsequent searches, while incurring minimal space overhead. PBA* not only remembers certain states it has examined, it proactively creates solution paths for the most likely deviations. In our experiments, PBA* searches only 10% of the A* search space when recovering from execution errors by storing a limited amount of search history.

Introduction

A* (Nilsson 1971) is a powerful, informed search algorithm that uses heuristics to guide the search along paths that are most likely to lead to the goal. Applications of A* are found in countless areas, including the computer gaming industry, robot path planning, GPS navigation devices, and a wide variety of other optimization applications. In some of those environments, the execution of the plan is either left to a human user, who may make an error (e.g., missing a turn while following GPS directions), or can be impacted by the actions of other actors in the environment (e.g., player actions can affect non-player characters in computer games). These environments may also have the property that there is adequate planning time at the beginning, or as a background process during execution, but if a deviation occurs while executing the plan the system needs to recover quickly.

If we were to apply A* in a naive way in these environments, we would simply search again from the point of the deviation, thereby recalculating many paths that we looked at the first time. Depending on the complexity of the environment, this may not satisfy the need to recover quickly. When humans make plans, we will often proactively make a backup plan—a plan "B"—to follow if an undesirable but not unlikely event occurs that foils the execution of our preferred plan. These B-plans are the inspiration for Plan-B A* (PBA*). During the initial planning phase, PBA* intentionally searches the most likely deviations from the optimal path and caches the best paths. If a deviation occurs, we are more likely to have the new answer already computed. If the deviation was not along one of the plan-B paths, then we search from the point of the deviation, but leverage what we learned in previous searches to more quickly identify the optimal recovery path.

Alternative Approaches

There are several other approaches that leverage A* for various types of problem spaces. For example, after a deviation local-repair strategies search for a route that returns to the original path, however, they do not support proactive paths, nor do they preserve path optimality. Dynamic A* (D*) (Stentz 1995; Likhachev et al. 2005) and LifeLong Planning A* (LPA*) (Koenig, Likhachev, and Furcy 2001; Likhachev and Koenig 2005) are designed for environments in which the world changes. If the number of edge cost updates is small, the time performance of these algorithms is generally quite good, but many changes to the world state will dramatically increase both the storage and time penalties.

In contrast, PBA* is designed for environments in which the world is static, but the execution of the path may fail. For example, D* recalculates f and g values for states on OPEN and re-sorts them as the 'robot' traverses the expected path. However, if the robot deviates, OPEN is invalidated and recalculating state values is insufficient to resolve discrepancies. Instead, D* would have to re-start, and since D* introduces storage and time overhead during the initial search, this overhead is compounded by new searches after deviations. Therefore, PBA* may be preferable in storage and time restricted static domains.

Markov decision processes (MDPs) represent another class of problems in which the optimal path is defined as the one that minimizes the expected overall cost. However, the minimal expected-cost path often differs from the best non-stochastic path. In a military setting, missions often have a primary target and a secondary target. If circumstances prevent the soldiers from achieving the primary objective, they quickly move on to the next-best target. When a football quarterback runs a passing play, he generally tries to throw to a primary receiver. If that man is covered by a de-

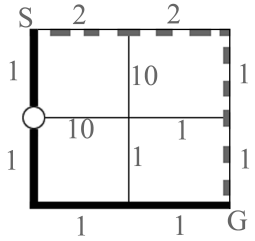

Figure 1: An example grid traversal problem. The circle represents a 50% chance of deviating at the intersection. The bold line path is optimal with cost 4 and expected utility 8.5. The actual and expected utility cost for the dashed line is 6.

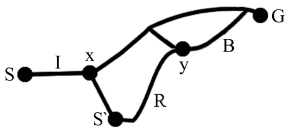

Figure 2: The three classes of paths in PBA*. The top line from *S* to *G* is the initial goal path, *I*. During the proactive phase, PBA* finds a plan-B path, *B*. When executing the path, the actor deviates at cell *x* and finds himself off the known path and in state *S'*. *S'* becomes the new start node for a search that discovers path *R*, which connects back to a cell *y* on path *B*. At that point, the optimal reactive path is known.

fender, he quickly targets the secondary receiver, and maybe a tertiary receiver if necessary. Figure 1 illustrates the difference between the two approaches in a simple grid environment. The objective is to travel from S to G. The circle on the middle-left of the grid represents a stochastic event; all other transitions are deterministic. When attempting to travel through the circle, there is a 50% chance of being directed down the wrong path. Algorithms that minimize expected utility will return the dashed line path with expected cost 6. PBA* will return the bold line with ideal cost 4 but expected cost of 8.5. PBA* would also store a plan-B path through the middle of the grid should the stochastic event not go in favor of the traveler.

PBA* Algorithm

There are three phases to the algorithm: the planning phase that finds the initial goal path from the start node to the goal; the proactive phase, which finds plan-B backup paths from deviation-prone cells to the goal; and the reactive phase, which determines a reactive goal path following a user deviation. These phases produce three classes of paths: initial goal paths *I*, backup paths that are proactively discovered, *B*, and reactive goal paths found following a deviation, *R*. Figure 2 illustrates the three classes of paths.

For the purposes of this explanation, and the experimental results that follow, we present the algorithm in the context of searching large square grids with varying edge weights and error probabilities on the nodes.

Planning Phase

In the planning phase, the initial search uses A* to find an optimal path from the start node to the goal node; this path is designated *I*. Once a goal path is determined, the algorithm traverses the path, updates each node's *h* value with the actual distance to the goal, and stores each node and its goal path in a hash table called KNOWN.

In the context of a square grid, the length of the goal path is sublinearly proportional to the number of cells in the grid, and therefore KNOWN requires a relatively small storage overhead when compared to the size of OPEN.

Proactive Phase

In the proactive phase, the algorithm investigates the goal path for potential deviations. It begins at the start node and checks the probability of transitioning to each of its children given the intention to follow the goal path. If the probability of making that particular mistake is greater than a probability threshold, α, the child is added to a priority queue of potential deviation cells called BACKUP. BACKUP is ordered by the probability of reaching each deviation cell, which ensures that the highest risk cells are processed first.

If the cell in question is separated from *I* by two deviations, the probability of reaching it is the product of the two errors that were required to reach the branch: the first error to deviate from path *I*, followed by a second error to deviate off a proactive path *B*. For example, if a cell is a grandchild of the start node, and the two nodes traveled to reach the deviant grandchild each have a mistake probability of 0.3, then the probability of reaching the grandchild is 0.3*0.3=0.09. If the algorithm encounters a cell more than once, the highest probability occurrence persists and is used to calculate the deviation probability for all descendants. In the example grid world, this occurs frequently; every turn in a path has at least two cells on the goal path that share a neighbor. Clearly, with error probabilities compounding in this way, the likelihood of looking at paths that require more than a few deviations drops off dramatically.

After evaluating a path and storing the potential deviations, the proactive phase finds plan-B goal paths for each deviation cell in BACKUP. These paths are added to the set *B*, and the cells are added to KNOWN. Once the process is complete, there is a known goal path for all the cells on each *B* path, which PBA* can leverage if the user deviates into any of these cells.

Also note that this approach allows PBA* to run in the background, which lets the user start their route as soon as the initial *I* path is found. The proactive phase of PBA* can run with a fixed time limit or fixed memory limit. Furthermore, it can run continuously, looking ahead at the next steps in the action plan and computing deviation paths for the next actions. In our simulations, we chose to let the proactive phase complete without an imposed time limit.

Reactive Phase

When a user deviates from the goal path, the algorithm reacts by finding an optimal detour path; these paths are the set *R*. This search behaves exactly the same as A*, and the

PBA* initial search, except it checks every cell to determine if it is on KNOWN before it is added to OPEN. If it is, the cell is added to OPEN using its actual *h* value, instead of the heuristic estimate of *h*. This is one of the central features in the algorithm: when a cell *y* is examined that is on both OPEN and KNOWN, the search immediately halts and determines the optimal path from the start cell to *y*, and concatenates the known path from *y* to the goal. This path is guaranteed to be optimal because *y*'s *h* value is actual and not a heuristic value. Therefore, all other nodes that could have resulted in shorter goal paths have already been explored.

Pseudocode

The pseudocode for PBA* is shown below.

START(Start_Node, Goal_Node)
1: Initialize graph γ, KNOWN
2: path = SEARCH (*Start_Node*, *Goal_Node*)
3: **return** path

SEARCH(Start_Node, Goal_Node)
1: path = GETPATH (*Start_Node*, *Goal_Node*)
2: TRAVERSEPATH(path)
3: PROACTIVE(*Start_Node*, *Goal_Node*)
4: **return** path

TRAVERSEPATH(Path)
1: **for** every node *n* in *Path* **do**
2: **if** KNOWN does not contain *n* **then**
3: Store *n* and its goal path in KNOWN
4: **end if**
5: **end for**

PROACTIVE(Start_Node, Goal_Node)
1: Initialize Backup
2: Backup.Push(*Start_Node*)
3: **while** Backup is not empty **do**
4: Current_Node = Backup.Pop
5: **if** Current_Node.ProbReached $\geq \alpha$ **then**
6: **for** every child *m* of Current_Node **do**
7: m.ProbReached = Current_Node.ProbReached * m.ProbDeviation
8: Backup.Push(m)
9: **end for**
10: SEARCH (Current_Node, *Goal_Node*)
11: **end if**
12: **end while**

GETPATH(Start_Node, Goal_Node)
1: Initialize Open, Closed
2: Open.Push(*Start_Node*)
3: **if** Open is empty **then**
4: **return** failure
5: **end if**
6: *n* = Open.Pop()
7: Closed.Push(n)
8: **if** *n* is *Goal_Node* **then**
9: **return** the goal path obtained by following parent pointers from *n* to *Start_Node* in γ
10: **end if**
11: **if** *n* is on KNOWN **then**
12: **return** the goal path obtained by following parent pointers from *n* to *Start_Node* in γ concatenated with the known path from *n* to *Goal_Node*
13: **end if**
14: Expand node *n* and generate the set M of its successors
15: **for** every node *m* of M **do**
16: **if** *m* on KNOWN **then**
17: overwrite *m*'s heuristic h value with its actual G distance
18: **end if**
19: **end for**
20: **for** every node *m* of M not on Open or Closed **do**
21: Open.Push(*m*)
22: **end for**
23: **for** every node *m* of M on Open and not in KNOWN **do**
24: **if** the new *m*'s h value is cheaper than *m*'s h value on Open **then**
25: update *m*'s h value on Open
26: **end if**
27: **if** the new estimated path is cheaper than the previous path **then**
28: update the path value on Closed and go to step 20 with each of *m*'s successors
29: **end if**
30: **end for**
31: Reorder Open in order of increasing f values
32: Go to Step 3

There may be a significant time overhead during the proactive phase depending on the specific problem space, however this was usually less than 20% of the corresponding A* search space in our simulations using the cost and deviation probability allocation techniques discussed.

Experimental Setup

Our environment is static, and therefore it is unnecessary to employ a storage intensive algorithm or to store the entire search space to support edge cost updates. PBA* only stores the grid cells along known goal paths, thus limiting the amount of space required.

We tested PBA* in a 4-connected square grid world that supports movements up, down, right, and left, and does not allow diagonal transitions, although the algorithm has no dependencies on a square grid, nor these directional specifications. The start and goal nodes are defined within the grid world. Additionally, the environment maintains edge costs and the probability of deviating when traveling between any two cells. The edge deviation probability signifies the chance that a user deviates when traveling over the edge, and is used to determine which edge is traversed after a deviation. For example, if an edge has deviation probability 0.25, than there's a 25% chance that the user would go the wrong way, and a 75% chance of the user traversing the edge successfully. If the user deviates, PBA* evaluates the potential next cells using the weighted inverse of the error probability

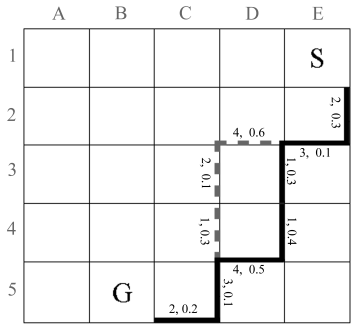

Figure 3: PBA* in a Grid World. The solid line is the optimal goal path from start node *S* to goal node *G*, and the dotted line at *D2* represents the new path after deviating to node *C2*. The dotted line rejoins the original path at *C5* because PBA* found an optimal path through *C5*, and the search halted. The tuples represent *Edge Cost, Edge Deviation Probability*. In reality, all edges have a tuple, but only the tuples along the discussed paths are shown.

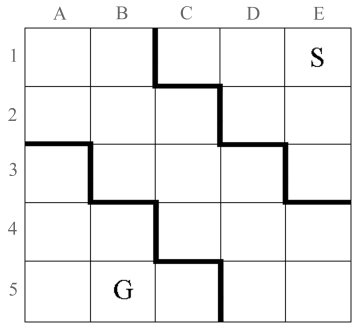

Figure 4: The *unavoidable deviation* test case: the bold lines represent edges with mistake probability one; all other edges have probability zero. Therefore, there will be at least two deviations when traveling from the start, *S*, to the goal node, *G*, given that the distance from *S* to *G* is at least three.

of each deviant cell. For example, if there are three possible deviant children with deviation probabilities $\{0.2, 0.1, 0.3\}$ then their weighted probabilities are $\{1/3, 1/6, 1/2\}$.

Figure 3 depicts a simple 5x5 environment. Every edge has a bidirectional tuple *Transition Cost, Probability of Deviating* that is available at runtime. For simplicity, the figure only shows edge tuples for the two paths discussed in the diagram. We did not incorporate obstacles, although the environment can easily support known obstacles by setting edge costs to infinity.

To test the algorithm in our environment, we ran numerous simulations while varying α, the grid size, and the probability distribution used to assign edge deviation probabilities. We used the Manhattan distance heuristic for *h*. A single run creates a new, random grid world, with random start and goal nodes. Edge costs are real numbers uniformly distributed in [1-200] inclusive and are known at runtime.

For modeling purposes, we used several different distributions to set the deviation probabilities in the experiments. Results are shown for uniform, normal and exponential distributions. The distributions are meant to represent realistic variations, and a real domain would have its own, natural distribution. The mean for both the uniform and normal distributions is 0.5, and the standard deviation for the normal distribution is 0.15. If any of the randomly generated values are below 0 or above 1, they are set to 0 or 1, respectively. Additionally, a controlled test case was used that creates two unavoidable sets of edges between the start and goal nodes that allows us to analyze results with at least two deviations. All edges in the unavoidable sets have deviation probability one, and all other edges have probabilities equal to zero. Therefore, the search will only deviate at the edges that have probability one, unless there is not enough space to create the unavoidable edges because the distance between start and goal nodes is less than three moves. We will call this test case *unavoidable deviation* and Figure 4 demonstrates the unavoidable sets of edges in a sample grid world.

For the exponential distribution, the lambda parameter was determined using a range of correctness probabilities between 0.80 and 0.95, in increments of 0.05, such that the probability of correctly traversing any given edge is equal to the current correctness value. For example, if the correctness value is 0.80, then the simulation sets lambda such that 80% of all moves are expected to be correct, and 20% will result in a deviation.

The distinct traits exhibited by these distributions determine how different factors affect PBA* and in which environments it should be used; we will use the GPS motivating example to demonstrate different distribution uses. The PBA* reactive phase yields very few *R* class paths using the exponential distribution because the deviation probabilities are generally low, but still allows for some high probability transitions. This distribution may be used to simulate particularly bad intersections, or perhaps country roads that are often easy to navigate, until the user reaches a town. Similarly, using the normal distribution yields many more *R* class paths and can represent city environments where deviations, both intentional and accidental, are more likely. The uniform distribution may represent an unknown environment and provides a control to which we compare the other distributions. Finally, the *unavoidable deviation* test case guarantees deviations in most runs, which could simulate closed roads or obstacles that are not known until the user encounters them.

The simulations generate results through numerous repetitions to calculate reliable and comparable performance metrics. The algorithm finds an optimal path, traverses it from the start node, and then identifies all possible deviations along every search branch until the deviation probability is less than α. When the traversal is complete, the simulation walks the goal path looking for deviations. It uses a random number generator and the edge probabilities to determine when the user deviates, at which point it searches for an *R* path. If the user deviates, it randomly selects the next cell based on the weighted inverse of each child's deviation probability, as previously explained. This process continues until the traversal reaches the goal node.

The above process occurs for each grid dimension be-

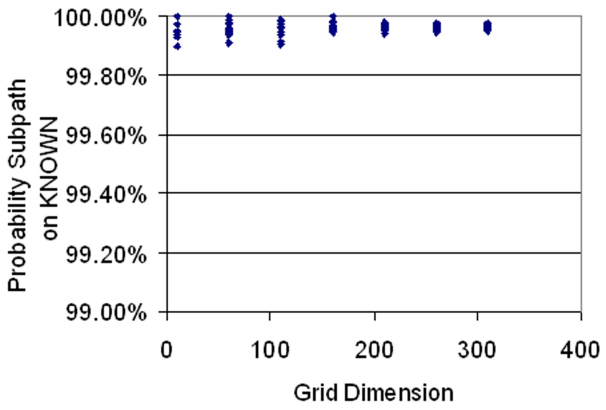

Figure 5: The probability PBA* finds a path on KNOWN.

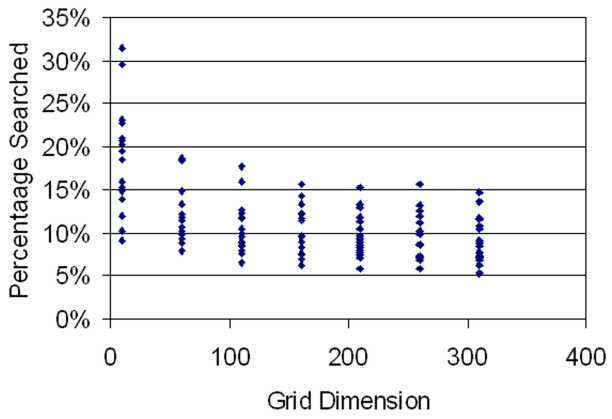

Figure 6: Percentage PBA* searched of the A* search space as a function of grid size.

tween 10 and 310 units, in increments of 50, and the results are averaged and returned to the running simulation. Grid dimension refers to the length of one side of the square grid world: a 10x10 grid has grid size 100 and grid dimension 10. We tested the algorithm using α values between 0.01 and 0.40. We did not test α values greater than 0.40 because generally cells with deviation probabilities greater than 40% are not common, few *B* paths would be found because most deviations would not meet the α threshold, and planning has limited benefits in such deviation-prone environments.

Results

When a path on KNOWN is not found, PBA* empirically behaves the same as A*. As grid sizes increase, so does the probability that PBA* will encounter a cell on KNOWN while calculating a new reactive path. The more frequently the algorithm encounters paths on KNOWN, the greater the performance gain. To evaluate the algorithm, it is important to determine how far along the discovery usually occurs. The algorithm is more useful if it reconnects with paths on KNOWN closer to the start node than if it reconnects near the goal node. Figure 5 shows the probability of reconnecting to a path on KNOWN using an exponential distribution. The average length of the KNOWN path compared to the final goal path was 97%, excluding grid dimension 10. Therefore, PBA* only searched an average of 3% of the actual goal path before reconnecting. Figure 6 shows the corresponding percentages searched of the A* search space. We compare against A* because it is leveraged by PBA*, it is the standard search algorithm in the field, and to show that PBA* improves performance over conventional A*.

PBA* requires a limited amount of memory overhead to store KNOWN. Average performance results for all distributions are shown in Table 1, including the storage overhead per distribution. These percentages represent averages over all α and grid dimensions, except dimension 10 because it skewed the values and PBA* is not suggested for such small environments. Overheads are calculated by comparing the PBA* values to A*. If A* requires 1000 nodes of storage and PBA* has 0.50% storage overhead, then PBA* requires 1005 nodes.

Distribution	Storage Overhead	Proactive Phase Time Overhead	Reactive Phase % of A*
Uniform	0.44%	8.16%	9.51%
Normal	0.52%	13.21%	10.37%
Exponential	0.52%	10.34%	10.28%
Unavoidable	0.51%	1.55%	10.22%
Average	0.50%	8.32%	10.10%

Table 1: Average results using different deviation probability assignments.

We varied α between 0.1 and 0.4, in increments of 0.1, to gauge the impact of searching different size areas around potential deviations; we also set α=0.01 in a separate experiment, which is discussed below. The results do not show a significant performance distinction within this range; for the exponential distribution, there was about a 2.6% difference in the A* search space explored during the reactive phase between α equaling 0.1 and 0.4. This is likely caused by the fact that multiplying successive edge deviation probabilities quickly yields values below α, regardless of the actual α value. Therefore, when α is set to 0.1, the algorithm does not search significantly more cells than when α is set to 0.4. On average, α=0.1 has a 14% proactive phase time overhead and searches 10% of the A* search space during the reactive phase. α=0.4 has a 10% proactive phase time overhead and searches 12.6% of the search space during the reactive phase. Therefore, PBA* can be used in environments with low α values if the additional overhead is acceptable for the domain.

The exponential distribution favors probabilities closer to zero, which cause fewer overall deviations than the other distributions. Therefore, the algorithm contributed fewer *R* and *P* paths to KNOWN. This resulted in a mere 0.52% space overhead, while searching 10.28% of the A* search space during the reactive phase.

The results discussed so far involve α values between 0.1 and 0.4. To simulate more extreme scenarios, we set α=0.01,

which searches most possible deviations. We used a normal distribution to assign edge deviation probabilities. On average, PBA* searched 9.20% of the A* search space, excluding grid dimension 10, with 1% additional space overhead and 20% proactive phase time overhead. Therefore, less of the A* search space was searched during the reactive phase, at the cost of more time overhead during the proactive phase.

Storing the *I* and *R* paths significantly improves performance, often because the optimal path from a deviated cell often leads immediately (U-turns), or quickly, back to the original *I* or *R* path. In order to determine what effect the *B* paths have on results, we ran several simulations that excluded the proactive phase of the algorithm. For each grid dimension, we looked at the number of states explored during the replan following a deviation, both with and without the proactive phase. Table 2 shows the results using the normal distribution to assign deviation probabilities.

Grid Dim.	PBA* *With* Proactive: # of States Searched	PBA* *Without* Proactive: # of States Searched	Percentage Searched
10	156	201	77.61%
60	11853	17199	68.92%
110	57112	85842	66.53%
160	148969	277650	53.65%
210	352339	591707	59.55%
260	745853	1046693	71.26%
310	976002	1834180	53.21%

Table 2: The table shows the cumulative number of states explored during the reactive phase of 100 simulations, both with and without the proactive phase, using the normal distribution to assign deviation probabilities. The last column compares the search space explored by PBA* *with* the proactive phase to *without* the proactive phase.

By predicting when and where users will deviate, PBA* only searches about 10% of the A* search space following an actual deviation in a normally distributed probability space. The average space overhead associated with this improved performance is less than 1%. It is evident from Table 2 that *B* paths greatly contribute to the overall performance improvements. On average, over 40% of all subpaths found during PBA* are *B* paths; 35% are *R* paths; and the remaining 25% are *I* paths. Therefore, very small space overhead can lead to large performance gains, with no associated time penalty. Techniques exist that improve A* performance, and any A* improvements are likely to improve the performance of PBA*.

PBA* begins looking for potential deviations in the proactive phase, and can run as a background process; even if a deviation occurs during this overhead period, it is likely that the algorithm has already found a goal path.

Conclusion

PBA* yields large performance gains over A* during subsequent searches of static domains by proactively calculating plan-B paths and saving information about known goal paths for use in the reactive phase. When environment deviation probabilities are available, proactively searching the environment for likely deviations and storing optimal plan-B goal paths can result in searching only about 10% of the A* search space following a deviation, while preserving optimality. Our experiments suggest that it may be advantageous to set α to a low value and let the PBA* proactive phase run as a background process.

Other techniques exist for replanning, but PBA* provides strong performance improvements and requires minimal additional storage, usually less than 1%. PBA* is not designed for partial information environments, and does not support dynamically adding and removing obstacles. It is also not designed for especially small domains; our results for grid dimension 10 were worse than all larger dimensions.

To further evaluate and validate our results, we intend to gather actual deviation probability data from live systems and provide more accurate simulations to determine the effectiveness of PBA*.

Acknowledgments

The work of P. Breimyer was funded by the Scientific Data Management Center under the Department of Energy's Scientific Discovery through Advanced Computing program.

References

Bonet, B., and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129(1-2):5–33.

Ferguson, D.; Likhachev, M.; and Stentz, A. 2005. A Guide to Heuristic-Based Path Planning. In *Proceedings of the Workshop on Planning under Uncertainty for Autonomous Systems at The International Conference on Automated Planning and Scheduling (ICAPS)*.

Koenig, S.; Likhachev, M.; and Furcy, D. 2001. Lifelong Planning A*. *Artificial Intelligence Journal* 155(1-2):93–146.

Likhachev, M., and Koenig, S. 2005. A Generalized Framework for Lifelong Planning A* Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 262–271.

Likhachev, M. 2005. *Search-based Planning for Large Dynamic Environments*. Ph.D. Dissertation, Carnegie Mellon University.

Nilsson, N. 1971. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.

Stentz, A. 1995. The Focused D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1652–1659.