

### **Abstract**

This paper describes and analyzes a Web-based JavaScript implementation of an assortment of pathfinding techniques applied to a simple terrain map. The interactive application performs breadth-first search, depth-first search, uniform cost search, and A\* search on a variety of preset and randomly generated terrain maps represented by grid squares with various corresponding movement costs. Details on implementations of each algorithm and their pathfinding effectiveness are discussed herein.

### **Introduction**

The general problem of pathfinding involves planning routes between locations specified on a representation of an environment. In most cases, we seek a route between a start location and a goal location, and we want to minimize some cost associated with moving from the start to the goal. Good solutions to this problem are important in many areas, such as gaming, puzzle-solving, autonomous vehicle control, network topology analysis, mapping, transportation logistics, and robotics. This paper examines the performance of several search algorithms that are often associated with pathfinding applications. These search algorithms are applied to a simplified terrain map represented as a grid. Each grid square corresponds to one of several possible terrain features. Each terrain feature is associated with a movement cost, which is meant to model the relative difficulty of moving into a square having that feature. In this application, the terrain features are: Road (movement cost 1), field (movement cost 2), forest (movement cost 4), hills (movement cost 5), river (movement cost 7), mountains (movement cost 10), and water (no movement possible). These features are entered into the application via a textual map file format. The map file data can be supplied by the user, or generated randomly using a fractal terrain generation technique. Once the data is supplied, the map is rendered and the user interactively invokes an assortment of common search techniques on the map. The found paths, as well as information about the algorithm's search activity, are displayed when each search is completed.

### **Algorithms and Heuristics**

The search algorithms included in the application are breadth-first search, depth-first search, uniform cost search, and A\* search. Each search operates on a search tree that represents possible states of movement between the start location and goal location on the map. The start state is the root of the tree. The goal state is somewhere below the root. A series of movements from the root node through connected nodes of the tree corresponds to a path that visits grid squares associated with each node. The grid squares themselves are represented in a two-dimensional array used to index the state information of each grid square. State information includes cost, open/closed/visited status, and search tree depth for current path.




Each search strategy begins at the root node. Child nodes, representing subsequent grid movement possibilities, are expanded and visited in an order determined by which search strategy is employed. Breadth-first search places expanded nodes onto a FIFO (first in, first out) queue. Since the nodes are retrieved in FIFO order, all nodes of a given depth are explored before nodes of the next-deepest depth are explored. Depth-first search places expanded nodes onto a LIFO (last in, first out) stack. Since the nodes are retrieved in LIFO order, deepest nodes are explored first before sibling nodes are explored. Neither breadth-first nor depth-first search concern themselves with the movement costs. Each will return the first path found that reaches the goal location. Uniform cost search, on the other hand, uses a binary min heap to sort expanded nodes by cost. It is a greedy search strategy that explores the nodes with the cheapest costs first. A\* search uses a similar binary min heap and greedy search, but also computes an estimate (heuristic) of the path cost from the current location to the goal location, and adds this estimate to the path cost from the start to the current location. Three heuristics are used. The first heuristic computes the minimum Manhattan distance between the current location and the goal. This is equivalent to the total number of steps required to reach the goal location via a path of minimum length. The second heuristic computes the Euclidian distance between the current location and the goal. This is the shortest straight-line

distance between the current location and the goal. The third heuristic computes the sum of costs of all squares surrounding each expanded node. This is an attempt to gauge the relative difficulty of the nearby terrain when selecting nodes to explore.

When each search strategy is performed, cycles in the graph search are actively avoided by preventing the search from visiting grid squares that were already visited on the current path. Such a loop back to an already visited square will cause the algorithm to search repeated states unnecessarily, and result in a search that fails to terminate. The search is further improved by performing pruning of the search tree. For breadth-first and depth-first search, once a grid square on a search path has been expanded, the grid square is marked as closed. If the square is encountered again on a different path, it is ignored and not added to the queue of open nodes. Searching in this manner, breadth-first search returns a path of minimum length without needing to search the entire tree. Depth-first search, on the other hand, returns a path of very suboptimal length, because it is locating the first of the possible goal states at the bottom of the search tree. Goal states at these bottom levels represent paths of large length. Minimal length and cost could be obtained by modifying these algorithms to search the entire tree, keeping the best results in the process. But this is an extremely inefficient proposition. Pruning during uniform-cost searches is done in a fashion similar to BFS/DFS – once a path through a given grid square is examined, the grid square is placed on a closed list. Other pathways that pass through the closed grid squares are not examined, because the first path encountered is the path with the best cost, due to the nature of uniform-cost's greedy search. A\* search cannot use this method of tree pruning, because the first path encountered is not guaranteed to be the one of minimal cost. So, for each visited grid square, a best cost is tracked. When the search examines a later path through the same grid square, the computed heuristic cost is compared to the best cost for that grid square. If the best cost is equaled or exceeded by the current path's heuristic cost, the current node is ignored (its children are not expanded). If the best cost is less than the current path's heuristic cost, then the current node is put on the open list (its children are expanded), and the best cost for that grid square is updated with the current path's heuristic cost.

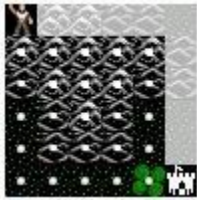
## **Results**

I implemented an iterative (non-recursive) breadth-first search by using a FIFO queue of expanded nodes. This section describes results of running a raw BFS implementation that does not perform any search tree pruning. As a result, many nodes are searched, and the algorithm is very slow with anything other than very small map sizes. Each visited node is marked on a two-dimensional array to track the number of nodes visited. I began with a 4x4 section of the sample map data file given for this project. BFS does not consider the cost of the terrain features (except for the impassible water), so the terrain detail is not important in these tests. In each case, I set the start location to the upper left grid square, and the goal location to the lower right grid square. I increased the grid size to determine the effect of grid size on BFS execution time. Results are tabulated in Table 1 and plotted in Figure 1.

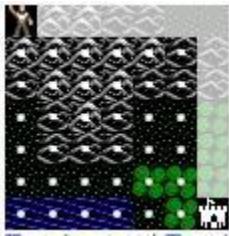
path found	grid size	# of visited nodes	# of grid squares covered by path	max queue length observed	max tree depth reached	search time (seconds)
	2	4	3	2	3	0.001
	3	17	5	15	5	0.002
	4	91	7	85	7	0.006



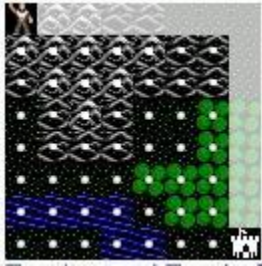
5            517            9            561            9            0.056



6            3187            11            3659            11            1.348

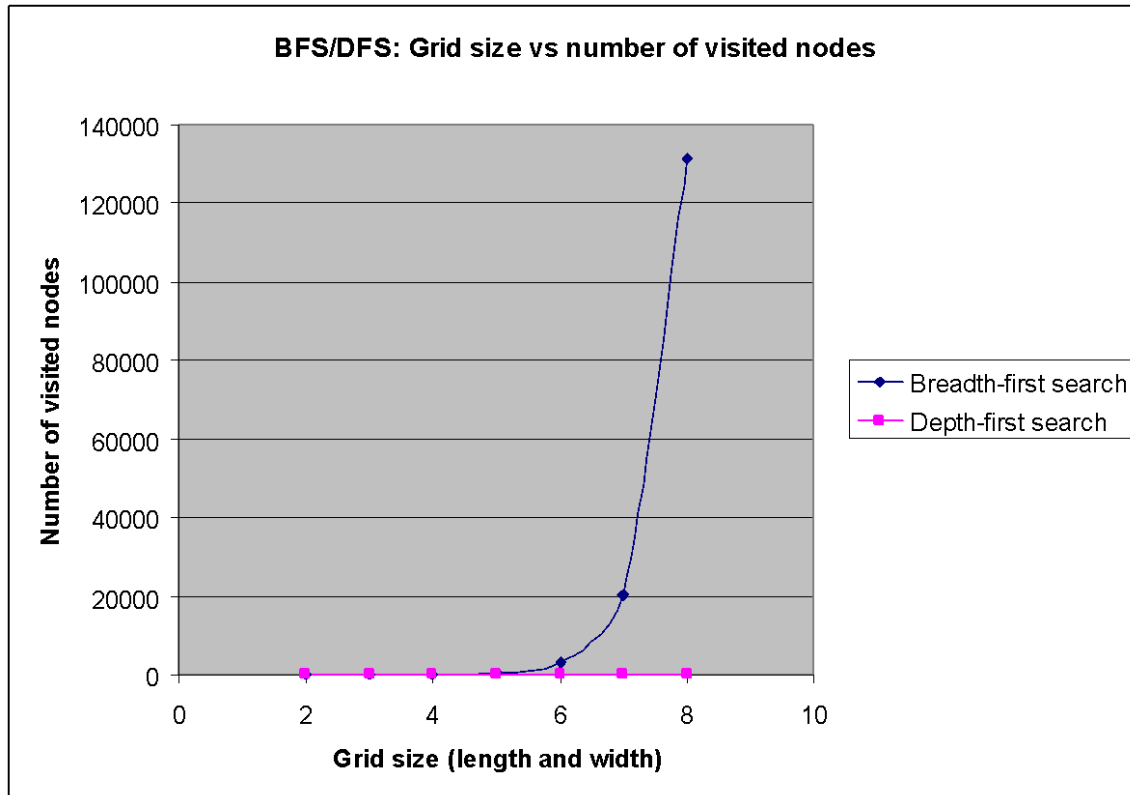


7            20147            13            25103            13            51.509



8            131309            15            170027            15            2805.499




*Table 1 – BFS results*


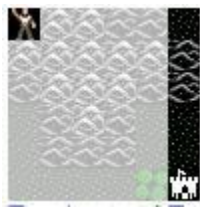
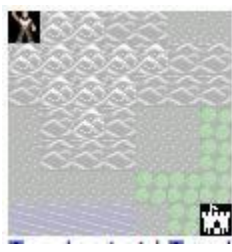
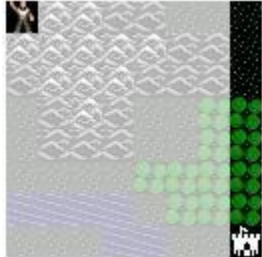


**Figure 1 – BFS and DFS results**

BFS performs poorly at large map sizes because the goal node, located at a significant depth in the search tree, is not reached until many nodes have been expanded and placed on the queue. BFS execution is much faster if the goal is placed closed to the start node on the map, because BFS will reach this node at a shallower level on the search tree before expanding a large number of new nodes for the FIFO queue. The found path is always of minimum length. However, in general, the found path does not have minimum cost.

To perform DFS, BFS was modified to use a LIFO stack. Results on the same test cases are tabulated in Table 2 and plotted in Figure 1.

path found	grid size	# of visited nodes	# of grid squares covered by path	max queue length observed	max tree depth reached	search time (seconds)
	2	3	3	2	3	0.001
	3	9	9	5	9	0.001
	4	13	13	10	13	0.002

	5	25	25	17	25	0.001
	6	31	31	26	31	0.002
	7	49	49	37	49	0.004
	8	57	57	50	57	0.005

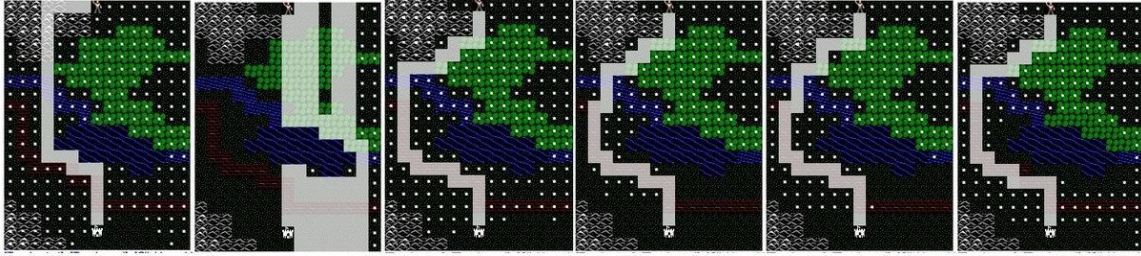
**Table 2 – DFS results**

DFS performance is optimal for the same test cases. This is because the goal is coincidentally placed on the bottom (or near the bottom) of a path on the search tree corresponding to the first search path DFS applies to the search tree. However, the path is not useful, because it consistently has neither least cost nor minimum length.

DFS can be shown to be a poor performer by modifying the test cases to purposefully place the goal at a location that is pushed early onto the LIFO stack and popped much later in the search.

As a curiosity, I modified BFS and DFS searches to track path costs as paths to goals are discovered. The path with best cost is kept, and the entire search tree is processed. The result is an extremely inefficient BFS / DFS search which locates a minimum cost path via brute force. At grid size 2, 7 nodes are explored. At grid size 3, 79 nodes are explored. At grid size 4, 2111 nodes are explored. This exponential effect becomes unwieldy at grid sizes 5 and higher.

I then applied tree pruning modifications to the search algorithms, which speeds up their execution times considerably. For BFS, DFS, and UCS, this is done by closing explored nodes as soon as their children are expanded. If the corresponding grid squares are encountered on later paths, they are ignored, because each of the search strategies will have already determined a path of equal or better cost (except in the case of DFS, which simply returns the first path found to the goal). A\* search uses the best-cost pruning strategy described earlier in this paper. **Figure 2** shows the paths located on Map 1 (the sample map given with the project) by each of the search strategies.

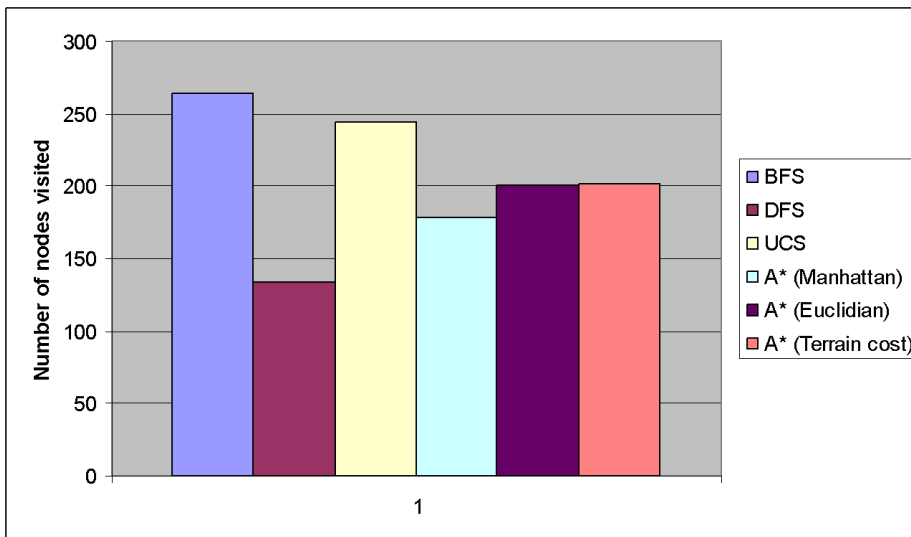


**Figure 2** - Paths found on Map 1 by search strategies. From left to right: BFS, DFS, UCS, A\* (Manhattan distance heuristic), A\* (Euclidian distance heuristic), and A\* (surrounding terrain cost heuristic). White dots indicate grid squares visited during the searches.

**Table 3** and **Figure 3** summarize the total number of tree nodes visited in each algorithm.

Algorithm	# of nodes visited
BFS	264
DFS	134
UCS	244
A* (Manhattan)	178
A* (Euclidian)	200
A* (Terrain cost)	202

**Table 3** – Map 1 results – Comparison of number of nodes visited for each algorithm.

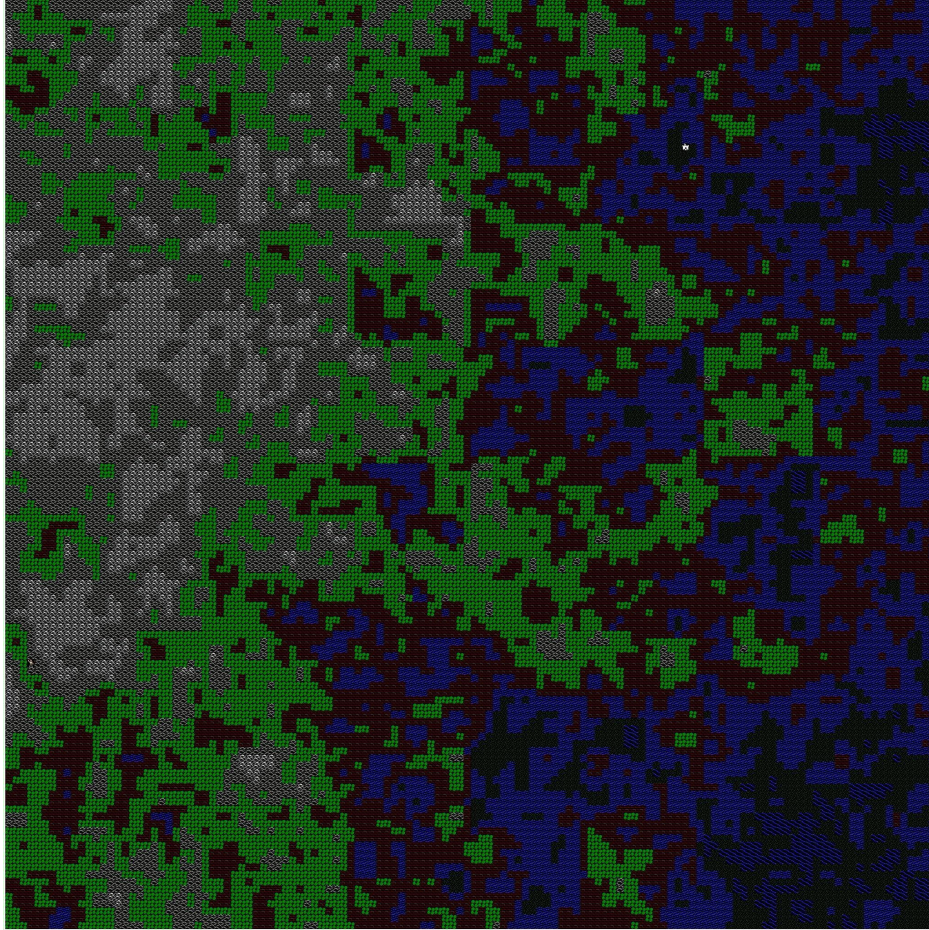


**Figure 3** – Comparison of nodes visited during searches on Map 1

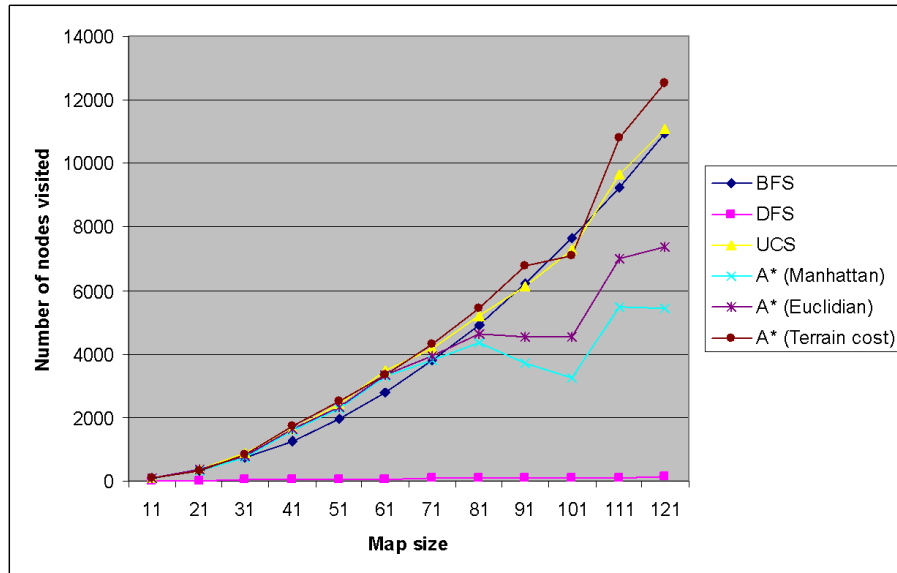
For Map 1, DFS visited the least number of nodes, resulting in the fastest execution time. But it produced the worst path - it covers 101 grid squares and has cost 268 (the best path covers 31 grid squares and has cost 56). This leaves the A\* searches as the best performers, since they produce the best cost path while visiting the least number of nodes. Of the three heuristics used, the Manhattan distance heuristic results in the least number of visited nodes. The terrain cost heuristic results in a path of cost 57, which is not the best cost path. The terrain cost heuristic likely overestimates or misrepresents the true cost of reaching the goal, and thus finds suboptimal paths.



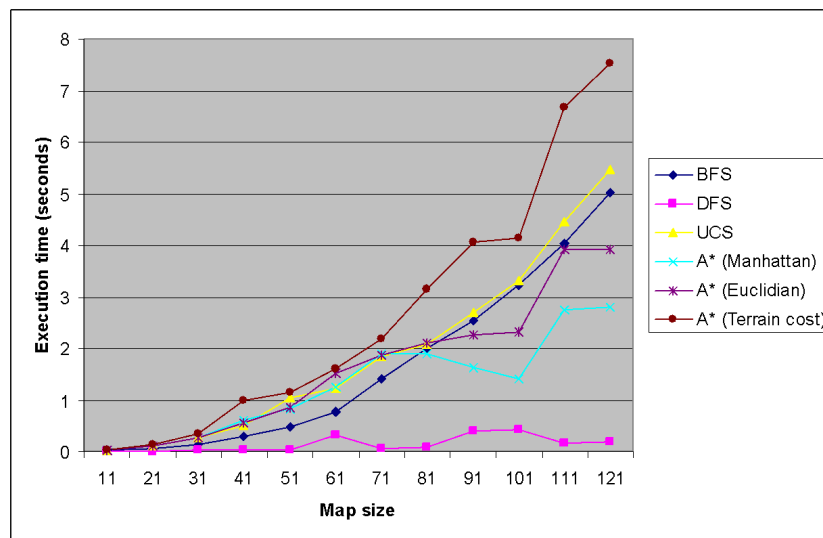
I compared the performance of each search strategy by running each algorithm against map sizes ranging from 11x11 to 121x121. I extracted each of these maps from the upper left section of the large map shown in **Figure 4**. For each sub-map, the start location was placed in the middle of the topmost row of the grid, and the goal location was placed in the middle of the bottommost row of the grid. Performance results in terms of visited nodes and execution time are shown in **Figure 5** and **Figure 6**.



**Figure 4** – Test map used for algorithm comparisons. Subsets of the map, ranging from size 11x11 to 121x121, were extracted from the upper left section of this map and used as inputs for the algorithm tests.



**Figure 5** – Performance results based on number of visited nodes for each search algorithm. The map size is plotted against the number of nodes visited by each algorithm.



**Figure 6** – Performance results based on execution time for each search algorithm. The map size is plotted against the execution time of each algorithm.

Both sets of performance results suggest that the best overall method of finding the best-cost path is A\* search using a Manhattan distance heuristic. On a few smaller maps, the Euclidian distance heuristic gave marginally improved performance over the Manhattan distance heuristic. The terrain cost heuristic performs very poorly overall; the uniform-cost search has consistently better performance. The terrain cost heuristic is not well-informed about the overall search problem – it knows only about nearby terrain features. This is likely to be the cause of its poor performance.

### Conclusions

Overall, the best search in each experiment in terms of space and time complexity was A\* search using the Manhattan distance heuristic. Use of the Euclidian distance heuristic resulted in slightly poorer performance. It is not as good of a heuristic as Manhattan distance. Possibly, this is due to the lack of a



direct translation of Euclidian distance into the constraints of the map grid environment. Manhattan distance more exactly represents true movement costs. The terrain cost heuristic is the poorest performer of the three heuristics tested. This heuristic does not accurately estimate the cost of the searched path to the goal. Its near-sighted view of neighboring terrain features misrepresents the overall path cost, which results in poor performance and suboptimal path costs. An improved heuristic may be to draw an imaginary straight line from the current location to the goal, and compute the terrain cost of all grid squares that the line passes through.

Uniform-cost search performed slightly worse than each of the three types of A\* searches. I originally believed UCS had a slight advantage, since it closes nodes sooner than A\* search and it avoids the time/space cost of pushing nodes onto a sorted list. But A\* searches do a better job of pruning the search tree, so the extra time and space needed for the min heap add/delete operations in A\* are compensated by the fact that A\* (with a proper heuristic) visits fewer nodes than UCS.

Allowing additional diagonal moves to the searches is likely to speed up the search for a goal location, at the expense of adding up to four new nodes to the open list (for a total of eight possible). Uniformed searches are likely to be quickly overcome by the increased branching factor of the search tree. Even UCS can be overcome this way, since it can be easily misled by low-cost grid squares in the vicinity that lead the search away from the goal. But informed searches will continue to take cost estimates into account. Since there are more possible ways to find lower cost nodes for the open list, the informed searches will more quickly converge on the optimal path.