Phase 1 Report

Matthew Ya & Aleece Randall

February 12, 2017

1 Quick How-to for Interface and Images

Blocked: Black
Regular unblocked: White
Hard to traverse: Orange
Regular unblocked with a highway: Cyan
Hard to traverse with a highway: Blue
Start: Lime Green
Path: Purple
End: Red

2 Implementation of Algorithms

We designed an abstract class called Search and 3 concrete classes UniformCostSearch, AStarSearch, and WeightAStarSearch which implemented the abstract class. Each concrete class shares the findPath() and findSuccessorSet() methods to traverse the map and expand the fringe. The abstract class Search has two abstract methods updateVertex() and setupFringe() that are implemented and defined by the concrete class. The reason for this is that each Search algorithm has its own way of calculating the value of h(n), the value of f(n) and setting up the fringe.

3 Optimization

The data structures used to implement the algorithms were chosen to achieve the best time and space complexities. The fringe or open list is a priority queue that is implemented using a binary heap. This data structure provides worst case complexities of O(log N) for search, inserts, and delete and O(1) for removal of the cell with the lowest value. The successors of each node are stored in a HashSet because of its O(1) insert. A HashSet was also chosen for the closed list after time/space complexity tradeoff. The HashSet has an average case search time of O(1) but worst case time O(n). The HashSet was chosen over a 2D boolean array due to the array's higher space requirement. Although, the Boolean array would yield a better worst case search time of O(1) , the space complexity is always the total number of nodes in the graph. On average, the HashSet is a much smaller value. The HashSet and Priority Queue were initialized to a higher but reasonable initial capacity in an attempt to reduce possible rehashing.

4 Heuritistics

There are several possible ways to calculate the distance between the start node and goal node. Let dx represent the horizontal distance and let dy represent the vertical distance between the start node and goal node. Let D represent the vertical/horizontal cost and D2 represent diagonal cost. The costs used in the following heuristics focus on regular cells to try to avoid creating an inadmissible heuristic.

4.1 Euclidean Distance Formula

$$h = D2 * \sqrt{dx * dx + dy * dy} \tag{1}$$

Pro: considers diagonals
Con: computationally expensive, only considers a straight line

Using the diagonal cost, this function fails to take into account the faster speeds of the highways. Using an average of the highway horizontal/ vertical costs and unblocked cell diagonal s cost, the heuristic tries to consider the fact that an L-shaped movement on a highway is less costly than moving on a diagonal. We consider this to be the worst heuriistic.

4.2 Manhattan Distance Formula

$$h = D * (dx + dy) \tag{2}$$

Pro: computationally inexpensive
Con: does not consider diagonal movement

This formula considers the faster movements of highways. If the optimal path does not make use of the highways, this heuristic is not close to the actual cost. This heuristic is admissible in most cases, see below for exceptions.

4.3 Diagonal Distance Formula

$$h = D * (dx + dy) + (D2 - 2 * D2) * min(dx, dy) \tag{3}$$

Computes the number of steps if a diagonal is not taken and adds the minimum diagonal steps. The cost of the minimum diagonal steps is the net cost of not moving vertical/ horizontal. The equation tries to take into account both directions of travel. Considering all directions, highways for vertical/horizontal and regular cells for diagonal (extremely low probability for diagonal highway movement), might provide a good estimate that is admissible in most cases. However, using all the costs for a highway, this is the best optimal heuristic. It largely underestimates most paths but holds up for rare cases such where an optimal path is diagonal movements through parallel highways.

5 Experimental Results from 50 benchmarks

5.1 Heuristics

$$h1 = \sqrt{2} * min(dx, dy) + max(dx, dy) - min(dx, dy) \tag{4}$$

$$h2 = 0.25 * (dx + dy) \tag{5}$$

$$h3 = ((0.25 + \sqrt{2})/2) * \sqrt{dx * dx + dy * dy} \tag{6}$$

$$h4 = 0.25 * (dx + dy) - (\sqrt{2} - 2 * 1) * min(dx, dy) \tag{7}$$

$$h5 = \sqrt{2} * \sqrt{dx * dx + dy * dy} \tag{8}$$

$$h6 = .25 * (dx + dy) + (\sqrt{2} * .25 - 2 * .25) * min(dx, dy); \tag{9}$$

5.2 Table for Results

UniformCost Search	
Average Run Time	20.7ms
Average Path Length	180 nodes
Average Path Cost	295.54
Average Nodes Expanded	12636 nodes
Average Memory Used	5346.78 KB

A* Search						
Averages	h1	h2	h3	h4	h5	h6
Run Time	4.04ms	19.14ms	6.46ms	13.12ms	1.04ms	21.2ms
Path Length	153 nodes	180 nodes	162 nodes	171 nodes	134 nodes	180 nodes
Path Cost	354.71	295.54	306.97	304.01	400.76	295.54
Nodes Expanded	1202 nodes	8740 nodes	2957	5747 nodes	224 nodes	9600 nodes
Memory Used	431.78KB	3311.81KB	1101.76KB	2114.45KB	106.57KB	3871.26KB

Weighted A* Search with Weight 1.25						
Averages	h1	h2	h3	h4	h5	h6
Run Time	0.64ms	12.82ms	2.74ms	6.56ms	0.16ms	12.86ms
Path Length	132 nodes	178 nodes	154 nodes	164 nodes	126 nodes	178 nodes
Path Cost	402.93	295.98	342.58	321.03	439.24	295.89
Nodes Expanded	360 nodes	7518 nodes	1413 nodes	4388 nodes	137 nodes	8616 nodes
Memory Used	159.50KB	3017.65KB	431.34KB	1695.47KB	26.49KB	3504.37KB

Weighted A* Search with Weight 2.00						
Averages	h1	h2	h3	h4	h5	h6
Run Time	0.04ms	6.56ms	0.02ms	3.24ms	0.02ms	9.06ms
Path Length	125 nodes	168 nodes	126 nodes	141 nodes	123 nodes	174 nodes
Path Cost	463.33	301.86	436.88	400.35	484.69	298.39
Nodes Expanded	129 nodes	4541 nodes	149 nodes	2753 nodes	123 nodes	5904 nodes
Memory Used	53.06KB	1909.11KB	106.68KB	955.82KB	52.99KB	2442.98KB

6 Discussion

Overall, it's obvious that the most expensive search algorithm is the non-heuristic Uniform Cost Search algorithm. This algorithm has the highest average run time, nodes expanded, and memory used. The average performance of the A* search algorithm sits between the average performance of the Uniform Cost search algorithm and the average of the Weighted A* algorithm. For Weighted A* search algorithm, we noticed that the algorithm does significantly better in terms of memory and run time when the weight is set to 2 compared to when the weight is set to 1.25.

Out of the 5 tested heuristics, heuristic 5 had a positive influence in the behavior of the A* search algorithms. The heuristic with the worst result was heuristic 2. It seems that heuristic functions that come closest to predicting the actual path of two points will have a positive influence of the algorithm. Heuristic 5 is the least conservative heuristic function in regards to cost. Heuristic 5 provided the best computational performance but grossly overestimates the optimal cost.

In contrast, Heuristic 6 is an admissible/consistent function and a drastically worst performance. From these results, we can conclude that the more weight/value that originates from the heuristic leads to a better computational performance. The higher the heuristic values, the more influence it has in directing the algorithm. Heuristic 6 provides a fairly low estimate of the true optimal cost. The Weighted A* at 1.25 increased the computational performance without sacrificing the optimal cost nearly all of the trials. The Weighted A* at 2.00 more accuracy is lost. Heuristic 3 provides lower overestimates compared to Heuristic 5. As a result, the optimal path deviated a lot less than Heuristic 5 despite having the same Euclidean Distance Formula basis. The simulations demonstrate a tradeoff between computational performance and optimality.