Assignment 1:

Heuristic Search Using Information from Many Heuristics

Team:

Eric Cajuste and Thurgood Kilper

Project GitHub link: https://github.com/UncappingEric/AI-Project1

Phase 1 Report

A). Map Interface

We used Java as the platform for visualizing the generated test maps. In addition, the map displays nodes that are in the open and closed lists while the algorithm operates.
Blocked squares = black
Weighted squares = gray
Unblocked squares = white
Highway squares = blue
Start/goal/path = red
closed node = light purple
fringe node = dark purple

B). Abstract Heuristic Algorithm Implementation

We define HeuristicAlgorithm.java as an abstract class which contains the base implementation of the A* algorithm so that we may instantiate three different versions of it. UniformSearch.java and AStar.java each extend the HeuristicAlgorithm abstract class. The pathfinding algorithm is inherited from HeuristicAlgorithm.java, which means that the inheriting classes only need to handle the calculation of f and h values for checked nodes. In UniformSearch, h is ignored, and f=g. For A*, f=g+h, and h is determined by a heuristic function. WeightedAStar.java extends Astar.java, and simply multiplies h by a weight value.

C). Algorithm Optimization

In practice, the main operations that affect performance are accessing the open(fringe) and closed lists. This is due to the fact that the algorithm is constantly checking for certain nodes in each list. The efficiency of accessing these lists has a noticeable impact on the performance. For the fringe, we use the Java library PriorityQueue class, which uses a heap data structure. As a result, inserting a node takes O(logn) time. This is improved over an Arraylist implementation, which would need to keep the nodes in sorted order, taking O(nlogn) time. For accessing the closed list, we utilize hashing with Java's HashSet class. For hashcodes we use the node coordinates, which ensures that each node has a unique code with no duplicates. It also makes checking for set membership take O(1) time, which is also vastly improved on an Arraylist implementation that would take O(n).

D). Heuristic Considerations

-Euclidean Distance

The optimal heuristic for an 8-way grid problem is the euclidean distance, which calculates the straight line distance from a node to the goal. We use h = sqrt[(x1-x2)^2 + (y1-y2)^2], and multiply by 0.25, the

smallest cost of moving to an adjacent node, in order for the heuristic to be admissible.

-Manhattan Distance
The manhattan distance counts the minimum number of nodes between a node and the goal moving only left, right, up or down. Compared to euclidean distance, this heuristic takes longer to expand due to ignoring diagonals, as well as resulting in many equal cost nodes.

-Beeline Distance
The beeline heuristic calculates the distance of a path traveling across the smallest number of cells from the starting cell to the goal, without taking into account cell types. It does this by favoring nodes in the direction of a wider gap.

-Unscaled Distance
Here we use a euclidean heuristic that does not scale to the lowest movement cost. As a result, the heuristic performs a search that performs closer to a flat graph search that does not account for the g cost weights. As a result it is inadmissible but finds a path very quickly.

-WideScale Distance
This heuristic is similar to manhattan distance in that it calculates the number of cells between a cell and the goal. It differs by accounting for diagonal movement. It also gives a weight to nodes in a horizontal direction, due to the graph being wider horizontally than vertically. In this case the weight is the ratio between the dimensions. The resulting heuristic is inadmissible only by a tiny margin.

E). Benchmark Test Results
-Path Cost
In general, we observe that the total costs of uniform search and A* are equal when using admissible heuristics. This is because A* is an optimal algorithm when using admissible heuristics and will return the same optimal path. Weighted A* will often return a higher total cost as its weight breaks admissibility, making the path not necessarily optimal.

-Runtime
In general, we observe that uniform search is the slowest, while weighted A* is the fastest. This is expected, as the heuristics provide a way for the search to prefer checking nodes closer towards the goal. On average, the searches complete within a fraction of a second. At this speed, in individual trials, we sometimes notice that one of the A* algorithms may perform much slower than expected- this is due to the occasional activation of Java's garbage collector, which may skew the runtime performance.

```
Grid #0 Trial #3
      Uniform:          Runtime: 0.363sec
      AStar:(1):        Runtime: 0.388sec ←Java GC creates unexpected result
      W-AStar (1.5):    Runtime: 0.361sec
      W-AStar (2.0):    Runtime: 0.04sec
```

-Cells Traveled
In terms of cells traveled, uniform and A* will provide the same optimal amount. Weighted A* will often provide a path that is shorter in cell length, but not in cost. In favoring the heuristic, Weighted A* will sometimes miss paths that are longer but cost less overall.

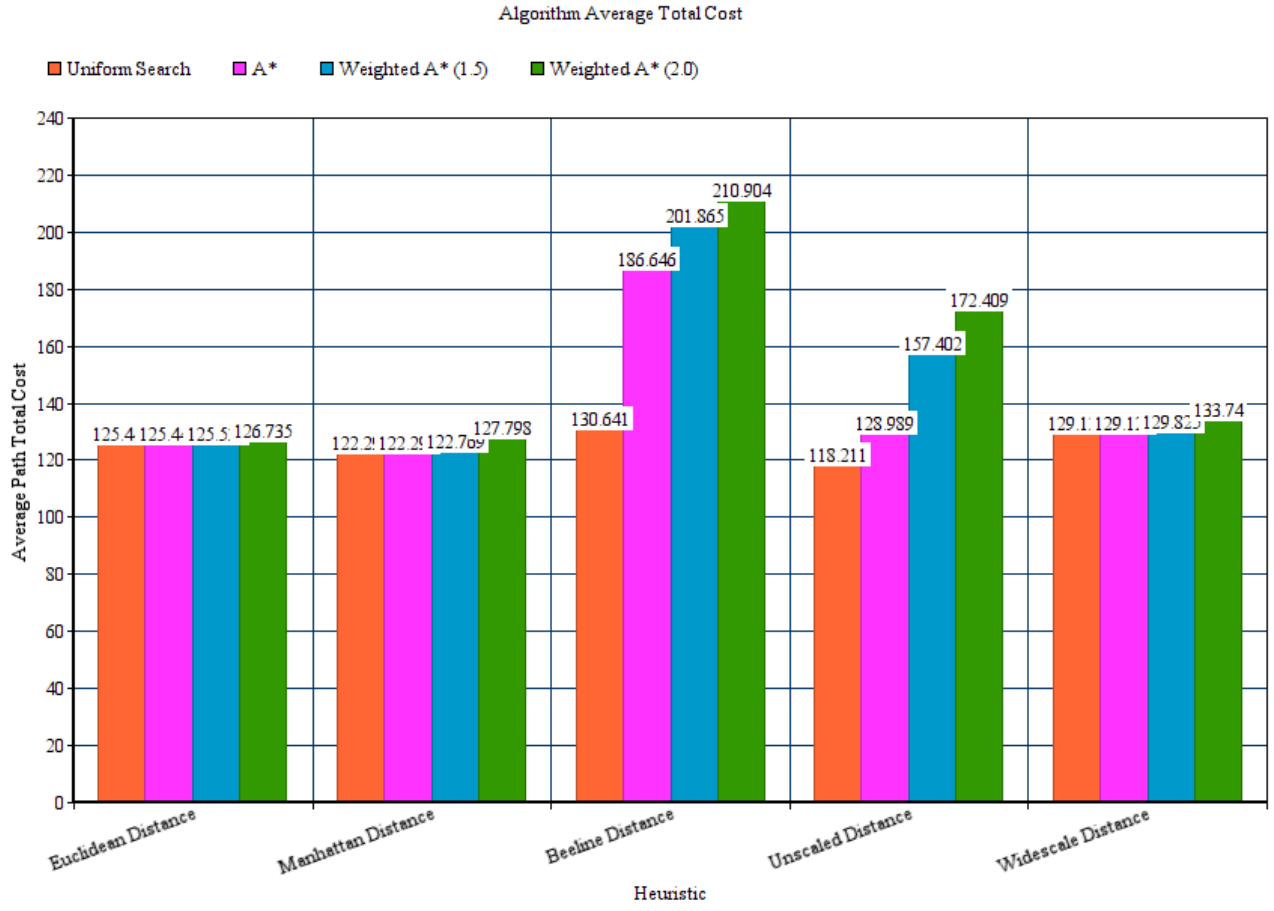
-Nodes Expanded
For expanded nodes, uniform expands the most, while weighted A* expands the least. The differences

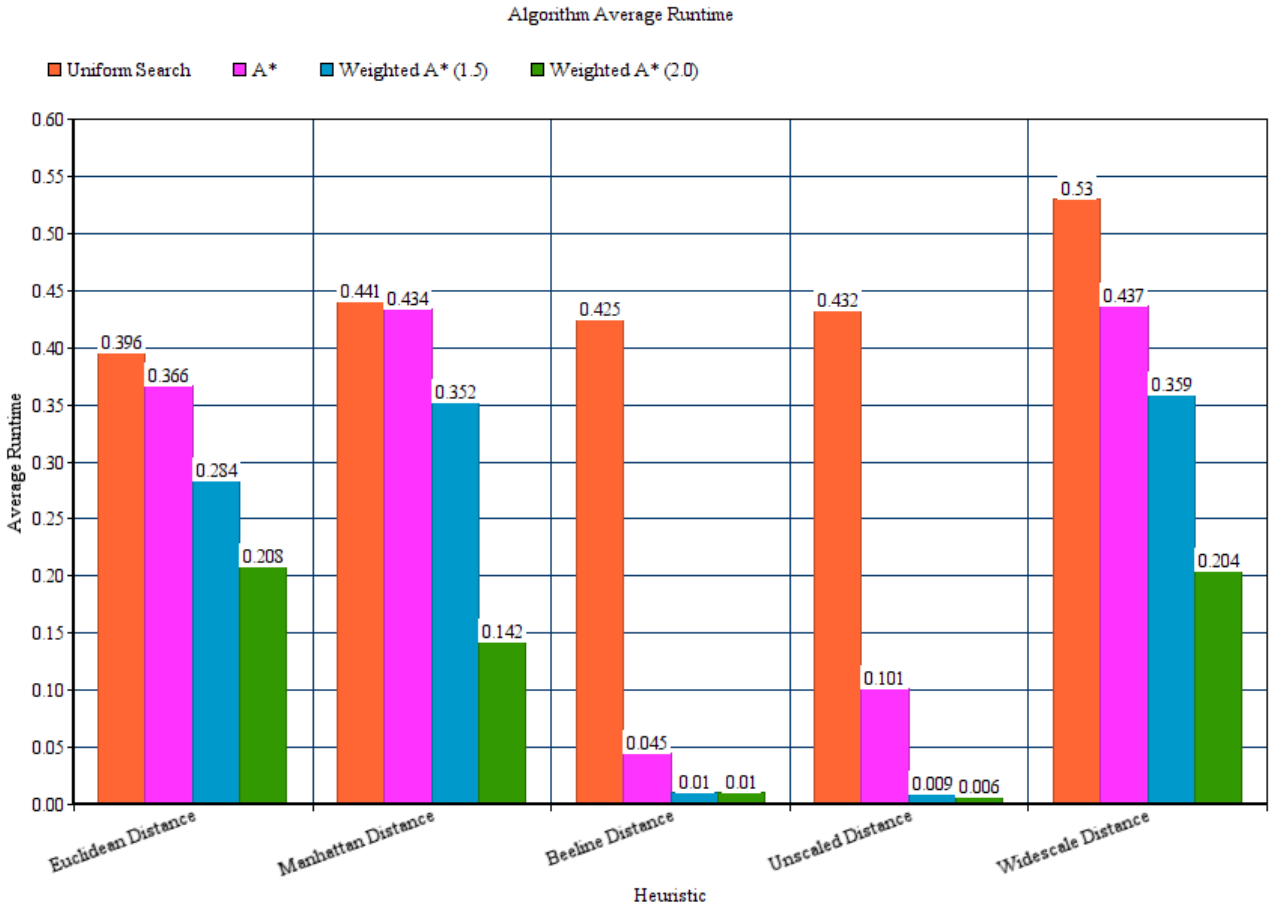
between weighted A* and A* increase greatly at higher weight values. This is expected, as the heuristics allow the A* algorithms to favor expanding towards the goal, whereas uniform search takes no preference.

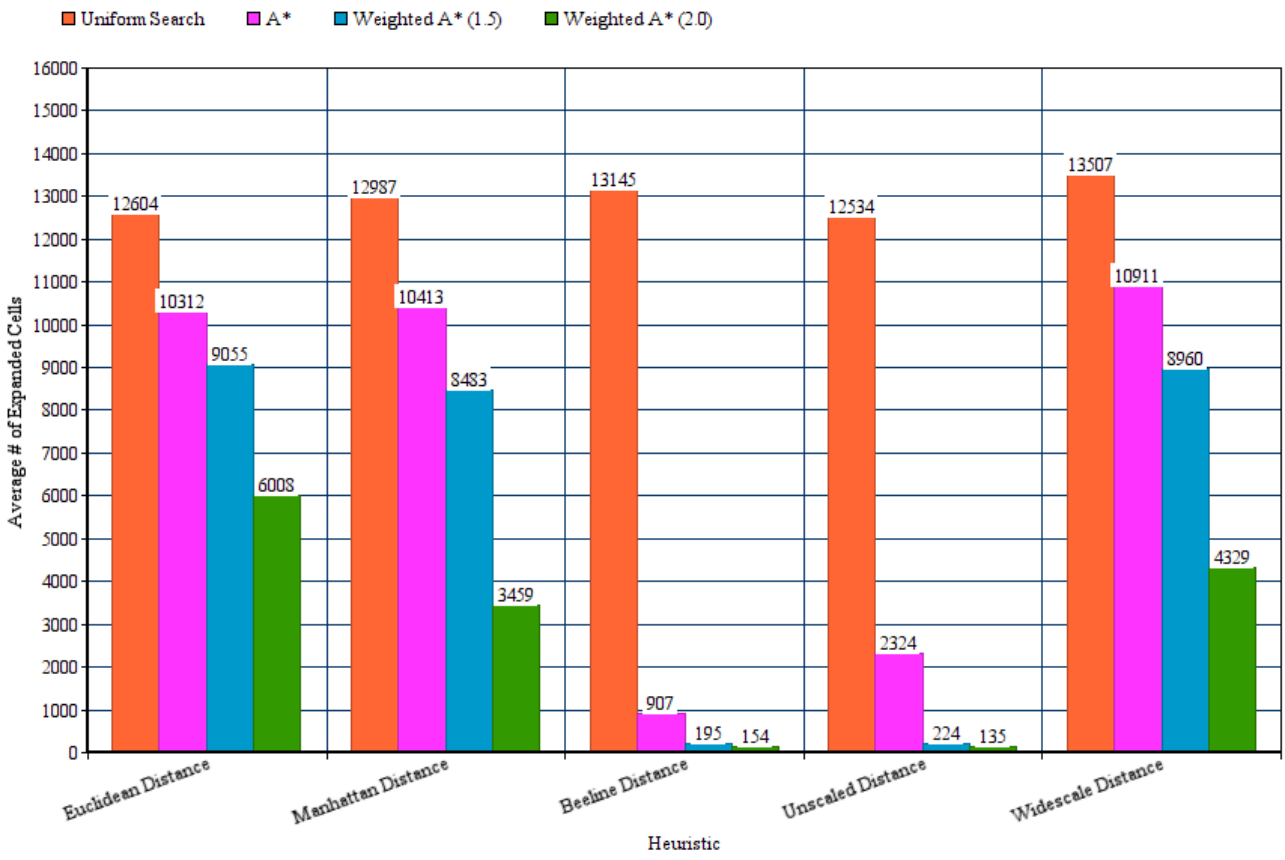
-Memory Usage

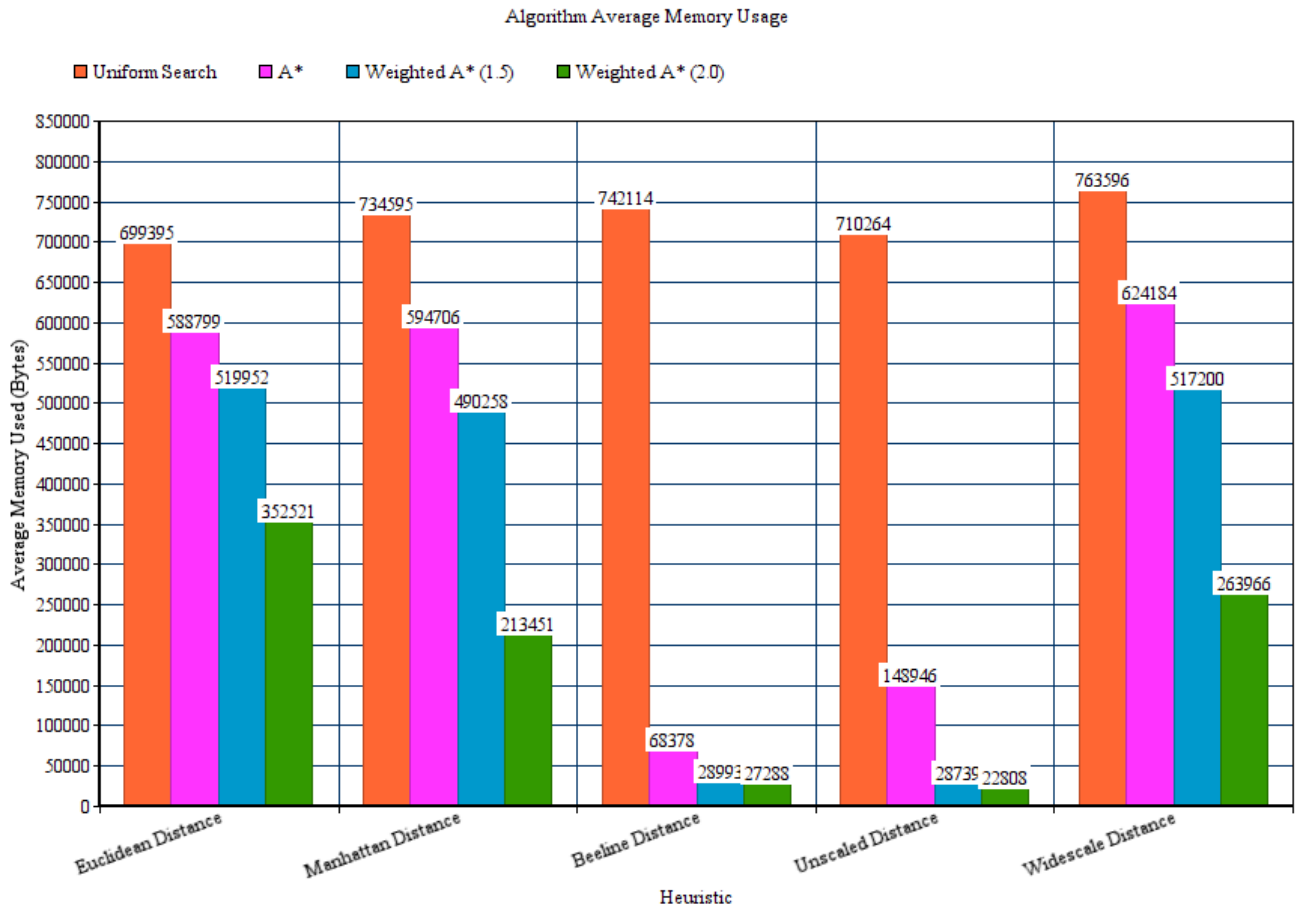
All of the algorithms use the same base implementation, differing only in calculation of f and h values for nodes. As a result, the difference in memory usage is mainly determined by the size of the open and closed lists. To calculate memory usage, we take the size of the open and closed lists used by the algorithms and multiply by the number of bytes used by a Node object. In our implementation, a node uses 56 bytes. As expected, the algorithms that run faster use less memory, due to expanding less nodes and making the lists larger.

-Heuristic Benchmark Results:




Algorithm Average Runtime
Uniform Search
A*
Weighted A* (1.5)
Weighted A* (2.0)
Average Runtime
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.396
0.366
0.284
0.208
0.441
0.434
0.352
0.142
0.425
0.045
0.01
0.01
0.432
0.101
0.009
0.006
0.53
0.437
0.359
0.204
Euclidean Distance
Manhattan Distance
Beeline Distance
Unscaled Distance
Widescale Distance
Heuristic


Algorithm Average Cells Expanded
Uniform Search
A*
Weighted A* (1.5)
Weighted A* (2.0)
Average # of Expanded Cells
16000
15000
14000
13000
12000
11000
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
12604
10312
9055
6008
12987
10413
8483
3459
13145
907
195
154
12534
2324
224
135
13507
10911
8960
4329
Euclidean Distance
Manhattan Distance
Beeline Distance
Unscaled Distance
Widescale Distance
Heuristic



F). Benchmark Result Discussion

Uniform search does not use a heuristic. As a result, it expands without preference towards the goal. Across the board, its results are not affected by the heuristic being used. Compared to A* which uses heuristics, it will expand many more nodes before reaching the goal, resulting in longer run times. The most noticeable differences are apparent between admissible and non admissible heuristics. In general, we observe that inadmissible heuristics provide the most efficient results in terms of speed and memory while sacrificing path optimality. Admissible heuristics are faster than not using heuristics while still finding an optimal path. Relatively, weighted A* provides a great increase in performance while in some cases sacrificing very little in terms of optimality. For inadmissible heuristics, we observe that unscaled distance is more cost effective, while maintaining comparable efficiency to beeline distance. For admissible heuristics, the results vary marginally, though Euclidean distance provides the best unweighted results.