Kunal Thakker
Nikita Volodin
Adedamilotun Kanji-Ojelade
Max Bartlik

Intro to Artificial Intelligence
Professor McMahon

Abstract

This report discusses the findings of using multiple heuristics for A* optimal path finding algorithm and the tradeoffs of runtime, memory usage, and efficiency for each of the heuristics.

Implementation

We used pygame to visualize the maps and the selected grids. Also, we added functionality to set different endpoints on the same map, use different heuristics on the same map, use different weights on the same map, and reset the map after running the algorithm once. We also added functionality to display the g-values of the heuristics when the cells are clicked.

Regarding the code, we split up the project into three parts: a runner class, a grid object and a cell object. The grid object contained a 2d list of cells of size 160x120. Each of the cells contained the type of cell, the cost traverse the cell, the g,h, and f values of the cell, and the color of the cell. The search algorithms were implemented in the grid class.

The a* algorithm was implemented similar to the pseudo code. An open list sorted by the f values of the cells was created and a 2d array of booleans was used for the closed list. The loop ran until the goal was reached or the open list was empty. In the loop, the element with the lowest f value was popped from the open list, added to the closed list, and its successors were found. Another loop went through the list of successors, updated their parent and g-value, and added them to the open list if applicable.

The weighed a* algorithm was implemented through the use of parameters in the a* algorithm. The uniform cost algorithm was implemented by using a separate travel-cost function in the a* algorithm.

Optimizations

1. For the closed data structure, we used a 2d boolean list that keeps track of the cells that have been closed. This way, we had O(1) runtime to access the data structure at the cost of space usage

Heuristics

We tested a total of 5 different heuristics for for the uniform cost, A*, Weighted A* algorithms. The following discusses the 2 admissible heuristics and 3 inadmissible heuristics that were used to test the results

Admissible Heuristic

1. **Manhattan_min**: This heuristic computes the Δx and Δy distance between the current point and the end point and adds it together. It then multiplies this value by .25. This is an admissible heuristic because the minimum distance between any two points would be when the path from those 2 points contains a highway. Also, the h-value at the end point would be Δx=0,Δy=0, giving us 0. Therefore, (Δx+Δy)*.25 would be an admissible heuristic.
2. **Euclidean_min**: The euclidean_min heuristic computes the pythagorean theorem value between the current and the end points and multiplies this value by .25. This gies us $.25*((\Delta x)^2+(\Delta y)^2)^{½}$.

Inadmissible Heuristic

1. **Manhattan**: Manhattan is the same as the manhattan_min heuristic, except that it does not multiply the Δx and Δy by 0.25.
2. **Euclidean**: Again, this heuristic is the same as euclidean_min without the .25 multiplier
3. **Diagonal**: This finds the difference between Δx and Δy and uses that to create the diagonal instead of the direct diagonal from the 2 endpoints. It then uses the pythagorean theorem to find the diagonal distance of this triangle and adds it to

Benchmarking

To run the benchmark tests, we created a loop that would go through 5 different maps, each with 10 different start and end points, for each algorithm. We used the python package cProfile to track runtime and guppy package to track memory usage. After the path was created for each set of start and endpoints, we took a screenshot of the path and uploaded the data to a csv file. Below are the results of the benchmark testing.

Note: Tracking memory increased the overall runtime of the program

Experimental Evaluation

The following chart contains the average runtime, memory usage, and % error compared to the optimal value for each of the heuristics. The full chart used to calculate this data is included as a spreadsheet in the zip file.

A*							
Weight	**Heuristic**	**Average Runtime (Sec)**	**Average Memory:Bytes**	**Average G-Value**	**Average Nodes Expanded**	**H-Efficiency**	**Path Len/Optimal**
Uniform	Manhattan_Min	4.240	2391341.72	165.298	12105.8	0.012	1
	Euclidean_Min	4.401	2513130.2	165.298	12769.62	0.011	1
	Diagonal	1.682	1071379.902	164.804	1737.019608	0.115	1.003001466
	Manhattan	0.434	919888.98	168.014	582.24	0.443	0.983834833
	Euclidean	2.131	1318382.04	165.298	3119.7	0.057	1
W=1	Manhattan_Min	7.001	1714572.58	122.549	10634.08	0.027	1
	Euclidean_Min	6.632	1852254.36	122.549	11425.68	0.023	1
	Diagonal	1.267	571967.06	152.641	1606.94	0.201	0.8028602518
	Manhattan	0.632	479032.56	166.432	975.34	0.324	0.7363309226
	Euclidean	1.844	760628.34	138.318	2512.68	0.141	0.8859957614
W=1.5	Manhattan_Min	5.599	1552624.86	122.853	8179.02	0.101	1
	Euclidean_Min	6.261	1804543.72	122.941	9817.22	0.052	0.9992867699
	Diagonal	0.233	537719.94	172.523	193.92	0.780	0.7120971826
	Manhattan	0.227	557737.94	191.930	205.38	0.754	0.6400946522
	Euclidean	0.220	602086.56	171.423	194.58	0.782	0.7166648026
W=2	Manhattan_Min	4.019	1409484.44	125.066	5877.24	0.120	1
	Euclidean_Min	5.300	1735585.02	124.304	8084.62	0.076	1.006129729
	Diagonal	0.172	692024.5	189.350	150.42	0.945	0.6605024101

	Manhattan	0.183	754815.72	180.879	149.14	0.952	0.6914365684
	Euclidean	0.170	713996.16	198.383	157.02	0.919	0.6304274649

Conclusion for A*, weighted A*, and uniform cost search

The g-values for euclidean_min and manhattan_min were the same for w=1. This proves that both of them are admissible heuristics. Overall, we can see that for the two admissible heuristics, there is a longer runtime, as well as memory usage, but the tradeoff is in having a lower g-value in the end. The inadmissible heuristics demonstrate inverse behavior with shorter runtimes and memory usage, while having a higher g-value (on average).

The uniform search algorithm gave the expected behaviour, with all of the different heuristics performing about the in terms of g-value. However, the runtime and memory usage of euclidean_min and manhattan_min was much higher. This means that for a uniform map, where the cost to traverse any 2 cells is the same, it may be more efficient to use an inadmissible heuristic, such as the manhattan or diagonal heuristic.

For the A* algorithm without weight (W=1), we found that the inadmissible heuristic performed worse than the admissible heuristics by about 21% in terms of average g-value. However, the average nodes expanded for the inadmissible heuristic was also only about quarter of that for the admissible heuristic.

For the weighted A* algorithms, the difference between g-value of the admissible heuristic and inadmissible heuristic was about 35% for w=1 and 41% for w=2. The speed of all the heuristics increases compared to the lower weights. However, the g-value compared to the optimal-g value also increases. So overall, a higher weight gives a quicker result and saves memory, but it may not find as optimal of a path.

Sequential Heuristic A*

	Sequential				
Weight (w1,w2)	**Average Runtime**	**Average Mem(bytes)**	**Avg Nodes Expand**	**Avg G_val**	**Path Len/Optimal**
(2,1)	1.262	1316423.44	3664.14	165.782	0.926
(1,1.75)	1.254	1316753.84	3664.14	165.782	0.926

	Sequential Run 2				
Weight (w1,w2)	**Average Runtime**	**Average Mem(bytes)**	**Avg Nodes Expand**	**Avg G_val**	**Path Len/Optimal**
(2,1.75)	2.399	1641922.98	6544.64	135.477	1.133
(1,2)	2.406	1642361.08	6544.64	135.477	1.133

Generally, the sequential algorithm appears to prioritize the use of highways much more efficiently than the other algorithms, expanding fewer cells along the highway if the path of the highway is biased towards the end goal. In some cases, it will achieve minimal expansions (when the start and end-goal are roughly along the same highway). Otherwise, the algorithm exhibits similar behavior to A* using an admissible heuristic, but with a significantly smaller *area* of nodes (not necessarily number of expansions because of overlapping). This algorithm appears to perform a dense* search around the start, then resorting to a less dense* search expanding outwards, biased towards the end-goal, until it finds a highway to prioritize.

*Note: "dense" search is used to indicate how frequently a node is expanded among the different open/closed sets. More dense (darker color) means that the node has been considered closed in multiple lists.

Proof for 8.9:

We want to prove that for some state $\text{Key}(s, 0) \leq \text{Key}(u, 0)\ \forall u \in \text{OPEN0}$, given that $g_0(s) \leq w_1 * c^*(s)$, where $c^*(s)$ is the optimal cost to state s.

We will use $c_{opt}(s)$ and $g_{opt}(s)$ to indicate optimal cost and g-value of some state s, respectively.

Proof.

Proving by contradiction we will assume that

$$Key(s, 0) = g_0(s) + w_1 \cdot h_0(s) > w_1 \cdot g_{opt}(s_{goal}),$$

Meaning that the *f* value for some state s is assumed to be larger than $w_1 \cdot$ *total cost of optimal path.*

We will call the optimal path '*P*,' which consists of the states that form the path.

When we pick some s_i from OPEN_0 that hasn't yet been expanded, and $s_i \in P$, s_{i+1} will always be inserted into OPEN_0.

When the state s_{goal} is the first in the priority queue, execution of the algorithm will terminate and a path can be considered found.

We will proceed by using rules we already know to be true:

$$g_0(s) = 0$$
$$\leq w_1 \cdot g_{opt}(s_i)$$

$$g_0(s_i) \leq g_0(s_{i-1}) + c(s_{i-1},s_i)$$
$$\leq w_1 \cdot g_0(s_{i-1}) + c(s_{i-1},s_i)$$
$$\leq w_1 \cdot g_0(s_i)$$

Where s_{i-1},s_i are in P, so it is true that travelling from s_{i-1} to its successor s_i would make s_i's g-value larger than its predecessor.

By definition

$$Key(s_i, 0) = g_o(s_i) + w_1 \cdot h_0(s_i)$$

So we continue

$$Key(s_i, 0) \leq w_1 \cdot g_{opt}(s_i) + w_1 \cdot h_0(s_i)$$
$$\leq w_1 \cdot g_{opt}(s_i) + w_1 \cdot g_{opt}(s_{goal})$$
$$= w_1 (g_{opt}(s_i) + g_{opt}(s_{goal}))$$
$$= w_1 \cdot g_{opt}(s_{goal})$$

Thus,

$$Key(s_i, 0) \leq w_1 \cdot g_{opt}(s_{goal}).$$

Earlier we assumed that

$$Key(s, 0) > w_1 \cdot g_{opt}(s_{goal}),$$

Thus,

$$Key(s_i, 0) \leq w_1 \cdot g_{opt}(s_{goal}) < Key(s, 0).$$

We can now see that this is a contradiction to the assumption that $Key(s, 0) \leq Key(u, 0)$ $\forall u \in OPEN_0$, as the conclusion we came to states that key(s,0) is <u>always</u> greater than some key(u,0), thus concluding our proof.

■