Assignment 3 Heuristic Search

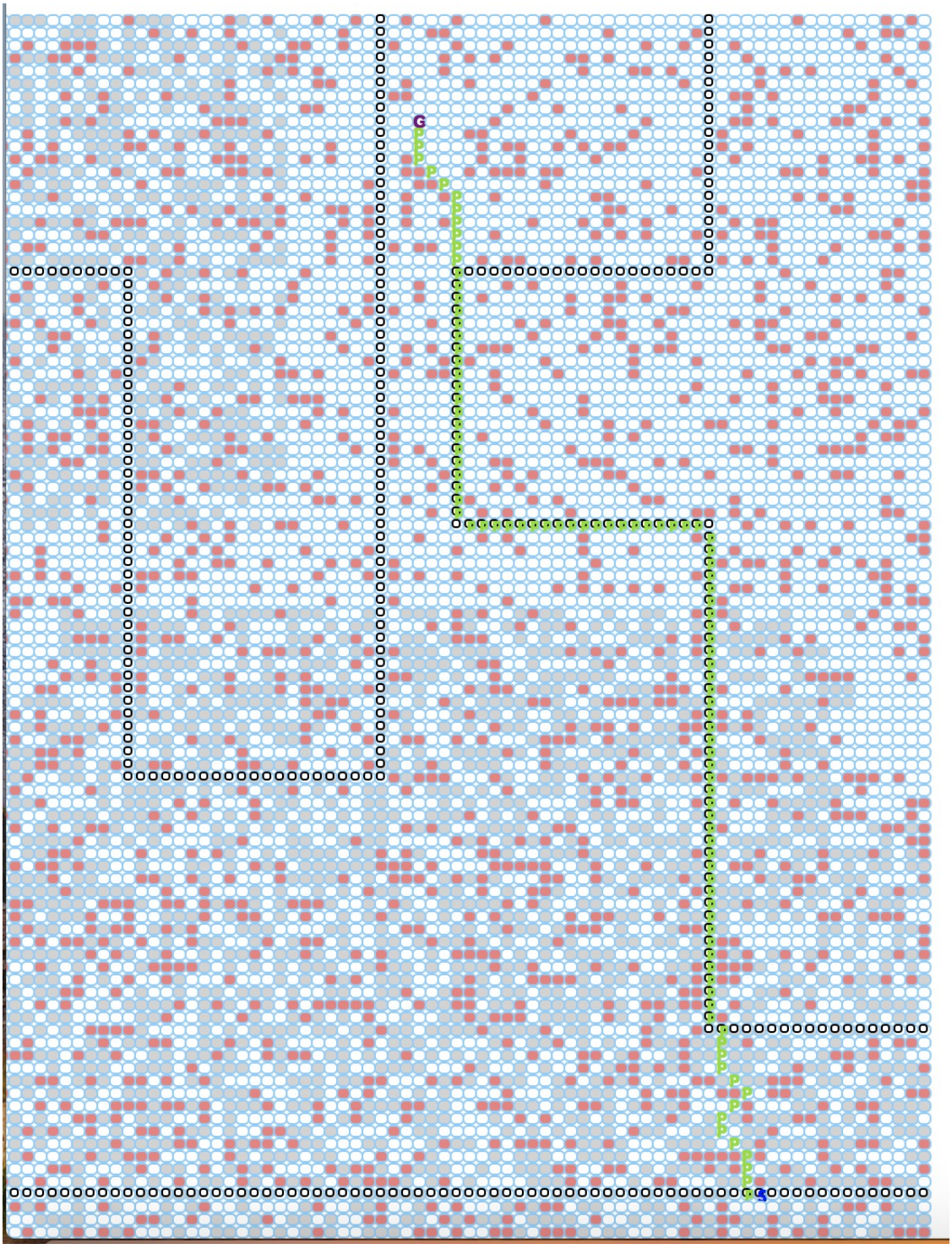
By Brandon Young and Ruicheng Wu

A. Interface

Here are three examples of the GUI. The first image shows the overall look of the GUI. The buttons located on the top-left of the interface are used to interact with the GUI. Clicking on a cell on the grid shows its h, g and f values below the buttons and above the grid.

Below is the control panel of the GUI. All of the I/O interactivity is located here. The red "Show Path" button is used to upload a path file to show a green path on the grid. The yellow "File Output" is used to output a file representing the current grid. The dark blue "Upload" button is used to upload and show a grid from an existing file. The light blue "Cell Info" button is for uploading a file with the f,g and h values for cells in the grid.

This third image shows a closer look at the actual grid. Red squares represent blocked cells, gray for hard-to-traverse cells and white for unblocked cells. A black circle within a square shows that the cell has a highway on it. Green P's show the path. The start (in blue) and goal (in purple) are either represented by S and G respectively or pie-shaped symbols.

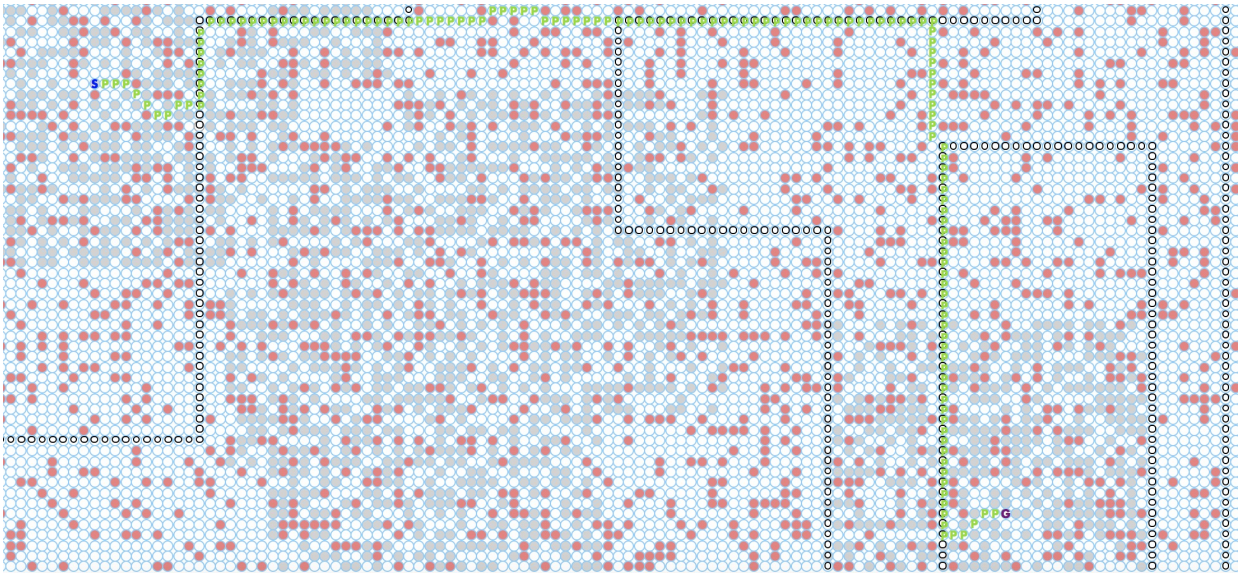

G
S

B. UCS, A* and Weighted A* Implementations

There are four examples shown below show the path from running uniform-cost search (UCS), A* and weighted A* respectively on the same map.

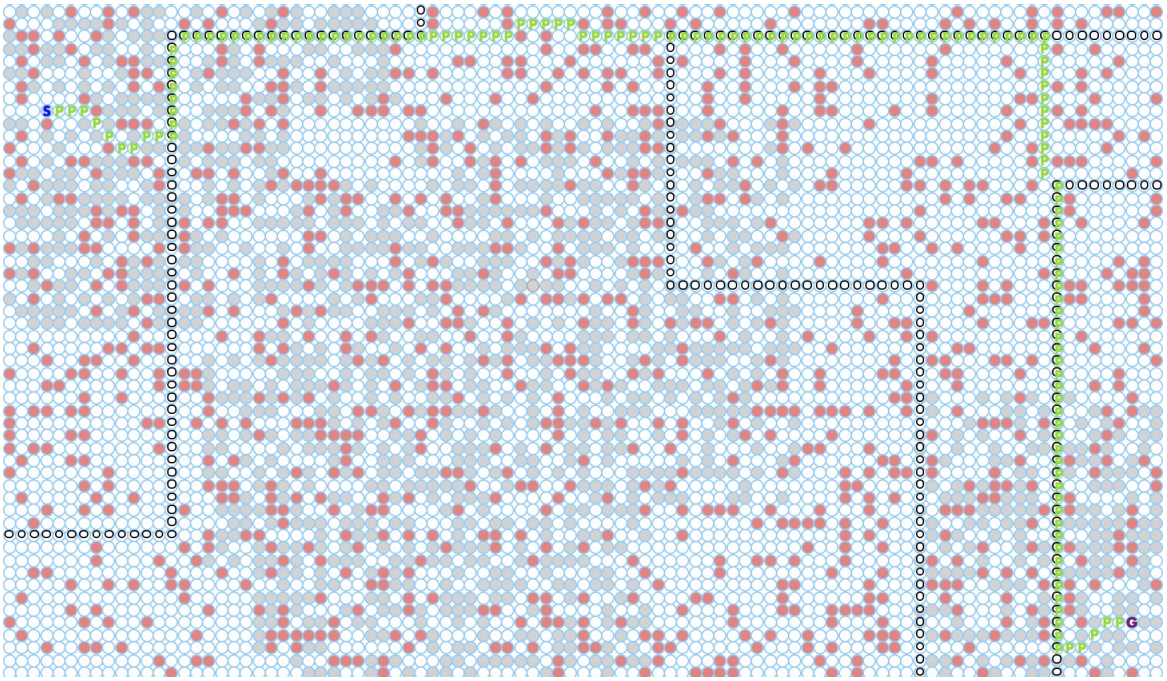
1. Uniform-cost search:

Path length: 81.08, Nodes Expanded: 11595, Run Time: 0.384272



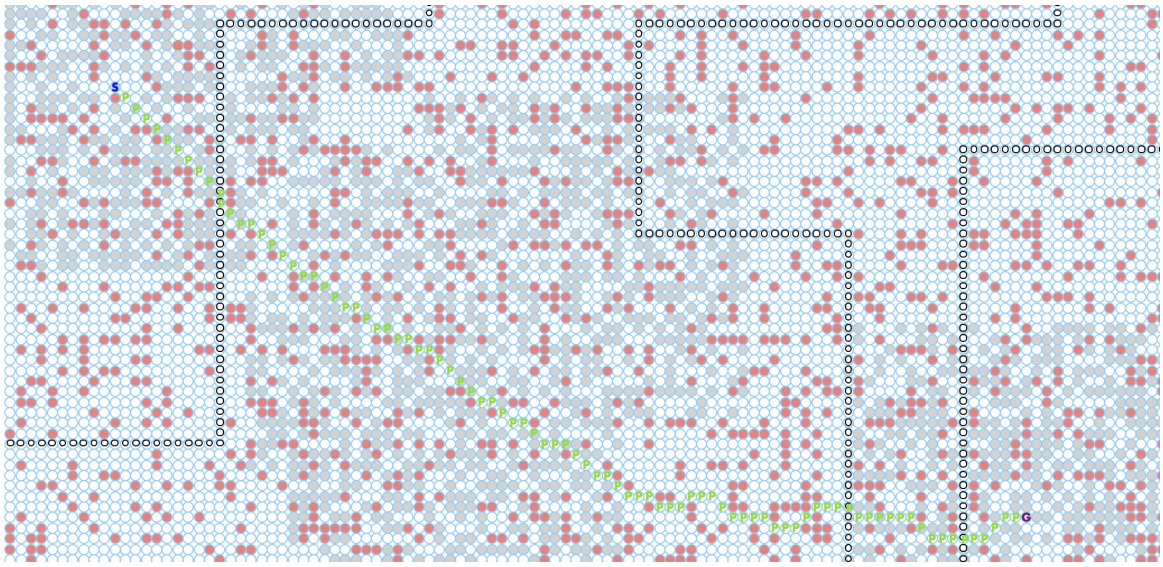
2. A * search ($h = 1$, Euclidean distance):

Path length: 81.08, Nodes Expanded: 8155, Run Time: 0.287288s



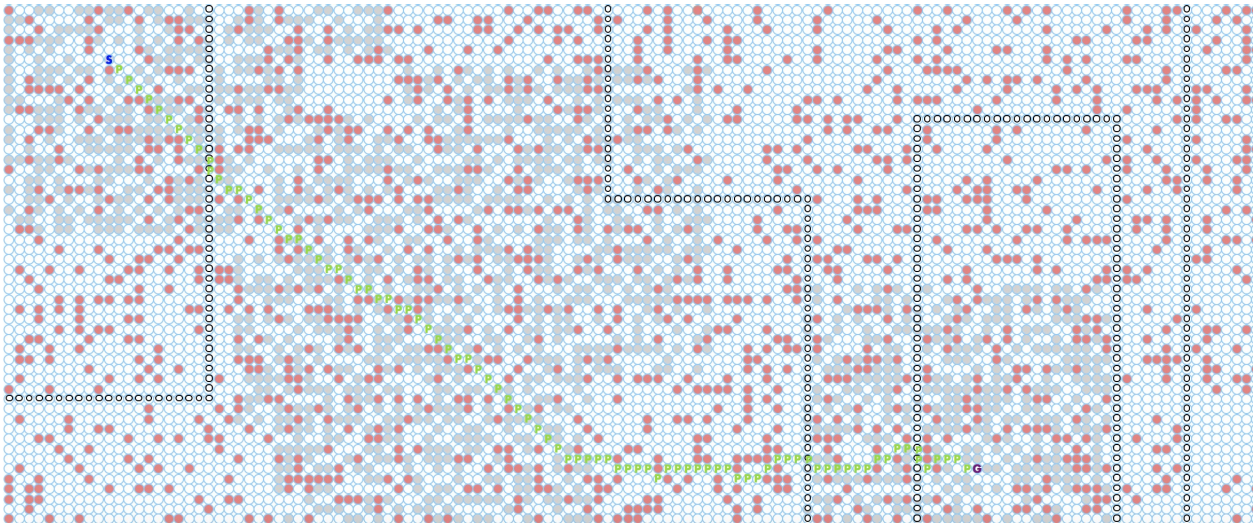
3. A* search ($h = 4$, Euclidean squared):

Path length: 137.33, Nodes Expanded: 234, Run Time: 0.056771s



4. Weighted A * search ($h = 4$, $w = 2.0$):

Path length: 144.03, Nodes Expanded: 107, Run Time: 0.05289s



C. Optimizations

To optimize the search algorithms, we used a Python dictionary as a heap, rather than a Python list (array), to implement the closed list more efficiently. Since we just needed to insert or check if a pair of coordinates existed inside the close list, a heap would speed up the check and may help the insertion. Overall, checking if a element exists takes O(n) for an array and O(1) for a heap.

For the heap, we initially used the sum of coordinate values to use as the key. However, this meant for every (x, y) there is a matching pair (y, x) with the same key. In addition, as the sums increased, there were a greater number of pairs that summed to the same value. As a result, we decided to distinguish (x, y) by multiplying x with a hash code value. With some testing, higher hash codes lowered the average size of a list for each key so we ended up using 91 as the hash code to speed up search checks.

We also optimized the way the heuristic was applied to the grid. At first we applied the heuristic over the entire graph before running the search algorithm. However, this wasted resources when only a portion of the overall grid was searched. Instead, we compute the heuristic during the search, to ensure only relevant cells in the grid have the heuristic applied to them.

D. Heuristics

1. Euclidean Distance

Euclidean distance is also a admissible/consistent heuristic as it works on a subproblem of the grid problem where any degree of movement is allowed. We also cut the distance by 4 to mimic movement along a highway, since the best possible case is if a highway leads from start to finish.

The formula for cell (x_1, y_1) and goal (x_2, y_2) is:

$$h((x_1, y_1)) = \frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{4}$$

2. Manhattan Distance

This heuristic assumes every cell in the grid is an unblocked cell with a highway on it. Then it computes the shortest path using only horizontal and vertical steps. We assumed every cell is a highway because using only unblocked cells results in higher than usual h-values. Often a large portion of the optimal path is along a highway, so the heuristic should reflect this.

The formula for cell (x_1, y_1) and goal (x_2, y_2) is:

$$h((x_1, y_1)) = 0.25 * (dx + dy)$$

where $dx = |x_2 - x_1|$ and $dy = |y_2 - y_1|$. The 0.25 comes from the cost it takes to move horizontally or vertically in one step.

The heuristic is inadmissible for the grid problem because diagonal movements are normally allowed. As a result, Manhattan distance slightly overestimates the actual cost to the goal. Nevertheless, this metric can come in handy for sequential A* search because of that pessimism.

3. Diagonal Distance Heuristic

The best admissible/consistent heuristic we used was the diagonal distance heuristic. The diagonal distance heuristic is similar to the Manhattan distance heuristic, except diagonal steps are allowed.

The formula for cell $(x1, y1)$ and goal $(x2, y2)$ is:

$$h((x1, y1)) = 0.25 * (dx + dy) + (\frac{\sqrt{2}}{4} - 2(0.25)) * min(dx, dy)$$

where $dx = |x2 - x1|$ and $dy = |y2 - y1|$. The 0.25 comes from the cost to move horizontally and vertically between highway cells and $\frac{\sqrt{2}}{4}$ is the cost from moving diagonally between highway cells.

Diagonal distance is better than Manhattan distance because it is admissible and Manhattan is not. Diagonal distance was preferred over Euclidean distance because the movements it allows is closer to the movements allowed on the actual grid. Therefore, the heuristic value will come closer to the actual optimal distance than the value from Euclidean distance.

4. Euclidean Squared

The formula for cell (x_1, y_1) and goal (x_2, y_2) is:

$$h((x_1, y_1)) = \frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{16}$$

This heuristic is similar to Euclidean distance, except it squares the output from Euclidean distance. As a result, it output very high values and it is not admissible. However, because the square root operation is not used, some computation time is saved. In general, this heuristic is pretty bad because its h-values will tend to be much higher than the g-values, but that may come in handy with sequential A* search.

5. Sample Heuristic

The heuristic given in the assignment instructions. Not admissible because it does not consider highway movement.

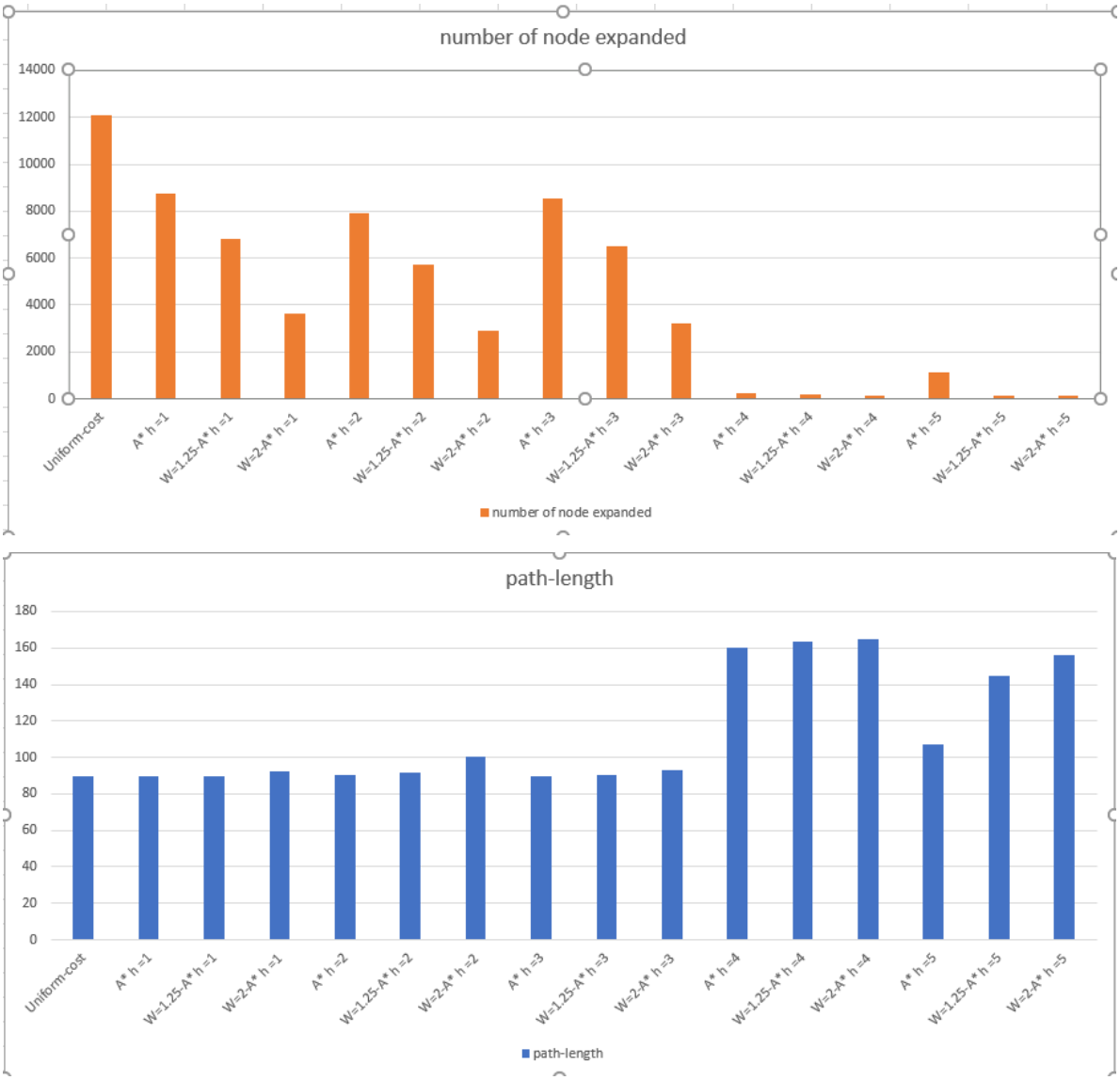
The formula for cell (x_1, y_1) and goal (x_2, y_2) is:

$$h((x_1, y_1)) = \sqrt{2} * min(dx, dy) + max(dx, dy) - min(dx, dy)$$

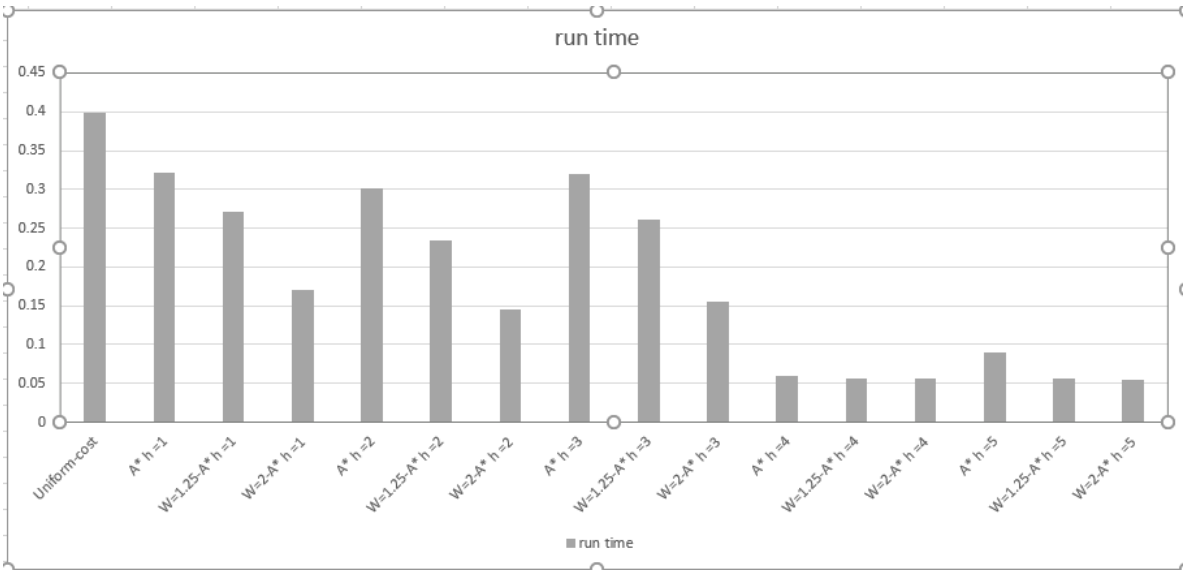
where $dx = |x_2 - x_1|$ and $dy = |y_2 - y_1|$.

E. UCS, A*, Weighted A* Statistics

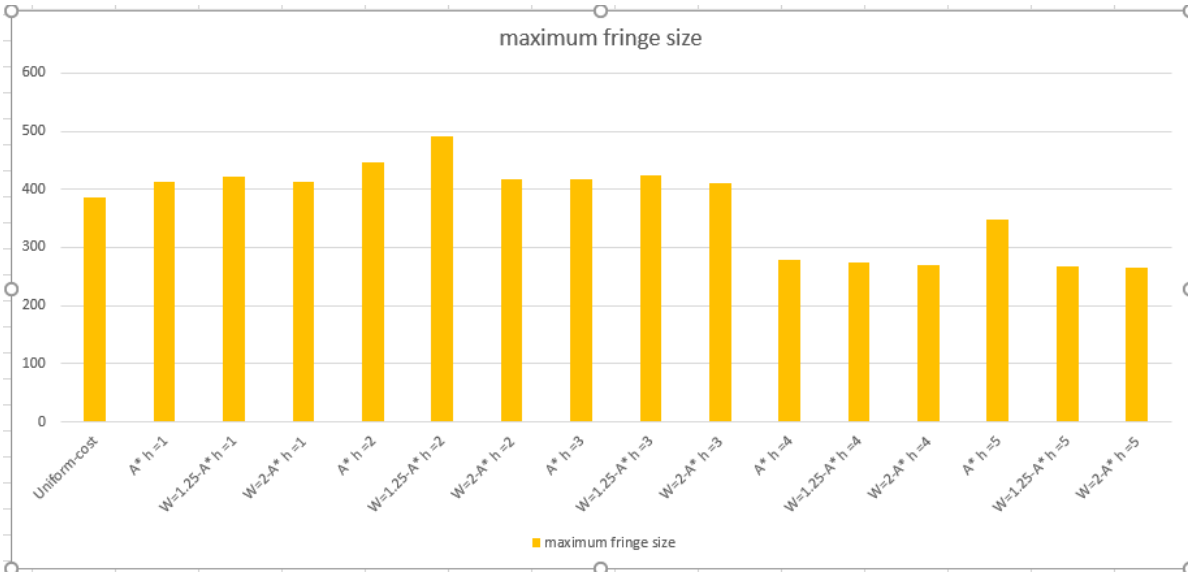
For the graphs below, h1 = Euclidean distance, h2 = Manhattan distance, h3 = diagonal distance, h4 = Euclidean squared, h5 = sample heuristic. Also, UCS = uniform-cost search, a = A* search, wa = weighted A* search and w = weight used for weighted A*



The time graph below was recorded in seconds.



The memory graph below was recording the maximum size of the fringe running the algorithm



F. Analysis of UCS, A* & Weighted A*

At first glance, uniform-cost search (UCS) finds the shortest path, but it has the highest number of nodes expanded and time taken by far, compared to the other methods. This provides evidence how UCS sacrifices speed and memory for the guarantee of finding an optimal path.

In comparison, A* (heuristic search using the diagonal distance or euclidean distance heuristics), tended to run much quicker than UCS, while still finding the shortest path. A* also expanded roughly a third less nodes than UCS. When using inadmissible heuristics, the heuristic search ran very quickly and expanded very few nodes (like with heuristic 4, Euclidean squared), but at the expense of longer paths.

Weighted A* search is similar to heuristic search with the Euclidean squared heuristic in that its runtime was quicker than A*, as expected, but the path length suffers slightly. With the additional weight factor bias the search, the heuristic can search more greedily with a good heuristic, but may return a sub-optimal path. Just a slightly higher weight like 1.5 (weight = 1 is the same as A*) reduces the time and number of nodes expanded significantly, while the average path length increases slightly. With a larger weight, like 2.5, the effects of weighted A* search are even more noticeable. A higher weight leads to even fewer nodes expanded and an even shorter run time with only a slight increase from the optimal path. In general, for the "good" heuristics (heuristics 1, 2 and 3), the sacrifice is not as severe as for other heuristics (heuristics 4 and 5).

Between the heuristics themselves, heuristics 1 (Euclidean distance), 2 (Manhattan distance) and 3 (diagonal distance) performed the best in terms of path length. Heuristic 4 (Euclidean squared) and heuristic 5 (sample heuristic) performed the best in terms of time and nodes expanded.

The heuristic results were mostly expected, with diagonal distance performing slightly better than Euclidean distance. In heuristic search with no weights, diagonal distance and Euclidean distance both found the optimal path. Surprisingly, Manhattan distance's average path length comes really close to the optimal path length and, at the same time, it performs better in terms of nodes expanded and time in comparison to diagonal and Euclidean distance. With higher weights, Manhattan distance resulted in a noticeably worse path than the admissible heuristics with the same weights.

Euclidean squared and the sample heuristic perform the worst in terms of path length by almost a factor of 2. It should be noted that the sample

heuristic did not consider highways, which resulted in its inadmissibility. Moreover, with normal heuristic search, the sample heuristic's path comes fairly close to the optimal one. On the other hand, the Euclidean squared heuristic's path is consistently much longer than the optimal path, with or without weights. While these heuristic find longer paths, they also have huge saves in time and expand practically zero nodes relative to other searches.

Out of all of the heuristics, the sample heuristic is the only one whose average path length is significantly affected with weights 1.5 and 2.5. Other heuristics only have a slight change in average path length with the same weights.

In conclusion, Euclidean distance and diagonal distance are the best two heuristics to use among all of the five proposed ones. Diagonal distance outperforms Euclidean distance in nearly every aspect, except with weights involved, Euclidean distance's average path length fairs a bit better. To save on time, either weight the heuristic slightly (w = 2.5) or use the Euclidean squared or sample heuristics.

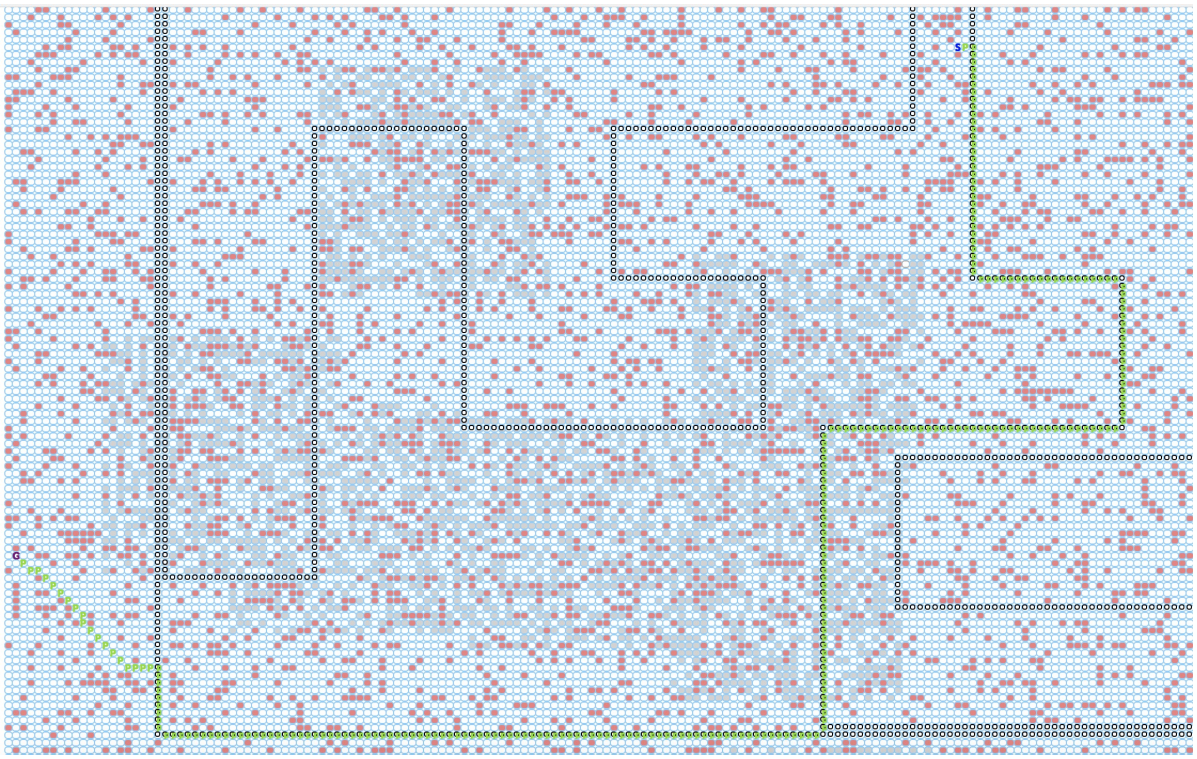
G. Sequential A* Implementation & Comparison

The path below was found using sequential A* with $w1 = 1.5$ and $w2 = 1.5$.

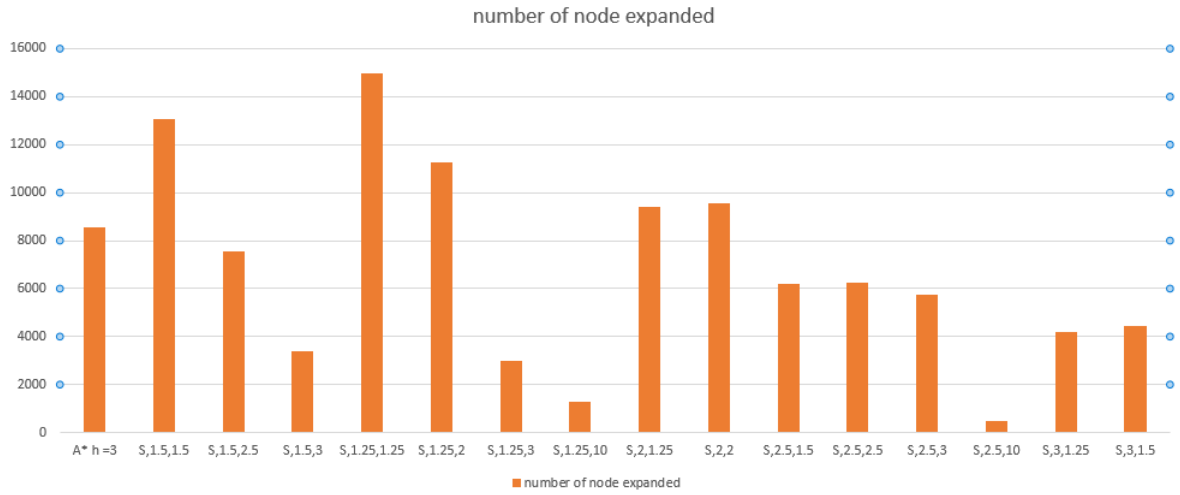
Path length: 95.024

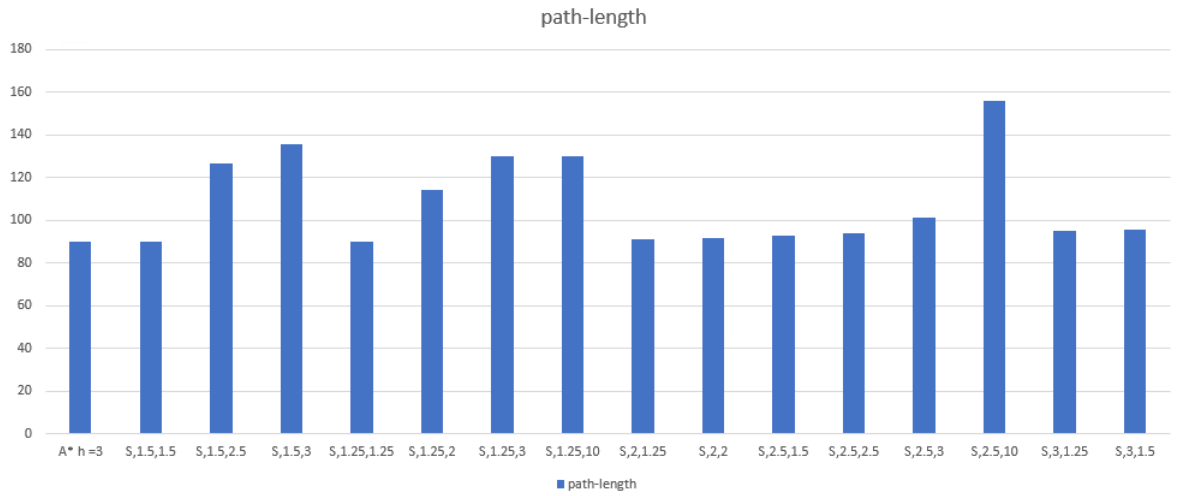
Time (number of nodes expanded): 11095

Time (In seconds):0.653

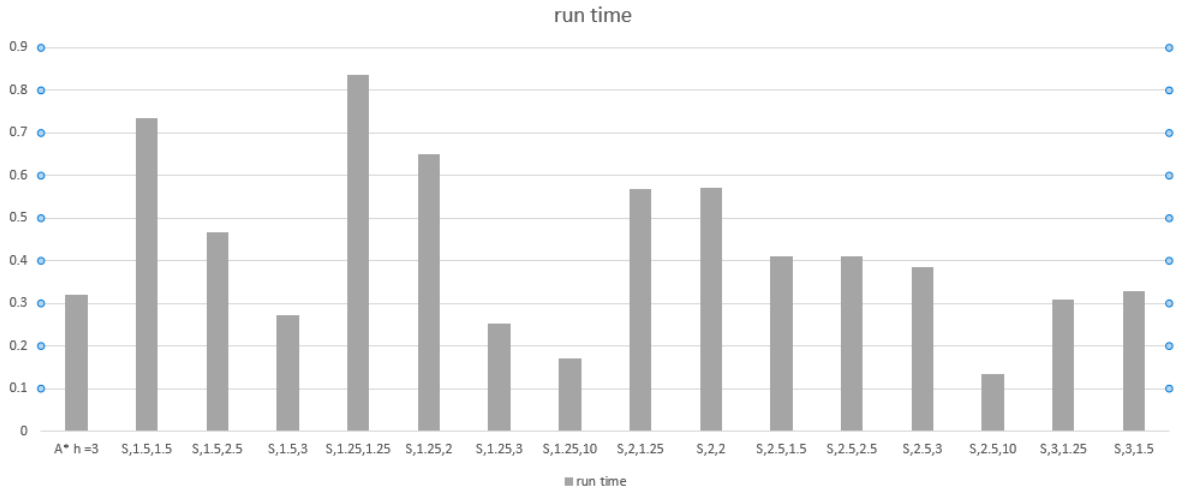


The experimental results are provided below, with S representing searches using sequential A*, the first value after the S representing w1 and the 2nd value after the S representing w2. For example, S,1.25,3 indicates the data for sequential search using $w1 = 1.25$ and $w2 = 3$. We compared sequential search with A* using the diagonal distance heuristic.

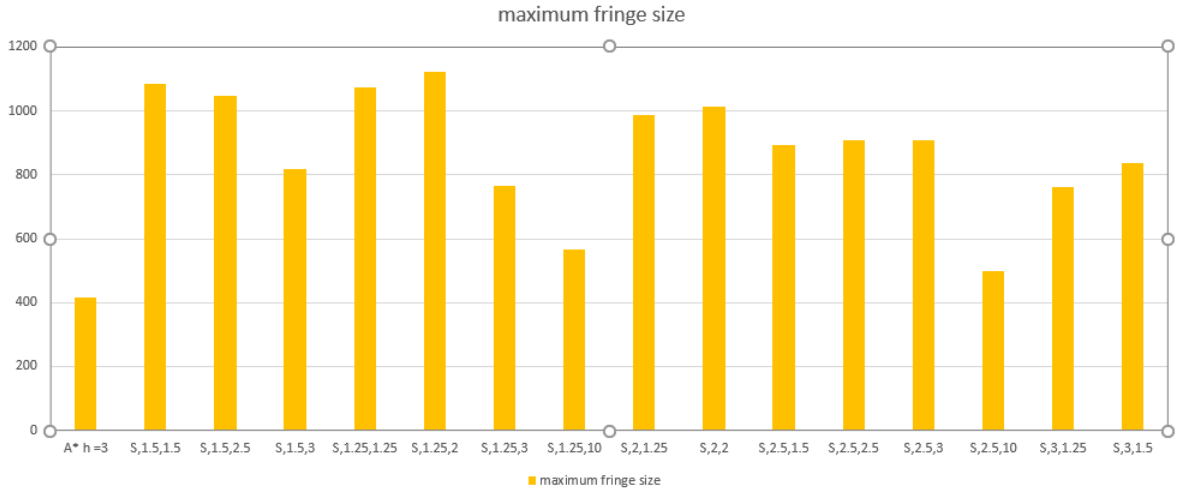




The time graph below was recorded in seconds.



The memory graph below was recording the maximum count of cells stored over all fringes at any point in time.



We ran 16 different w_1, w_2 pairs to collect the data. Originally, we predicted that sequential A* would generally be slower than regular A* search since it runs multiple heuristics, instead of just one. However, after we increased w_1 to 2.5 and kept w_2 as low as possible, the run time becomes shorter than A* search, while maintaining a relative short path.

Surprisingly, a higher w1 value does not increase the path length much. This is likely because of multiple heuristics balancing out and finding a better path than one that would be found by weighted A* search with weight $w = w_1$.

Sequential A* search's path length suffers the most from increases in w_2. There is a jump in path length when using the same w_1, but going from $w_2 = 1.5$ to $w_2 = 2.5$. With some w_1 values, like $w_1 = 2.5$, changes in w_2 did not impact the path length as much.

Depending on the values of w_1 and w_2, sequential search can run slower (and expand more nodes) or faster (and expand fewer nodes) than A*. In general, increasing w_1 appears to decrease the nodes expanded as well as the runtime. Increasing w_2 also causes a drop in nodes expanded and runtime, but to a greater effect than with w_1.

We also observed that $w_1 = 2.5$ about the best average performance in nearly every aspect, when compared to other runs of sequential search. Both w_1 and w_2 can affect the performance, but by adjusting the value of w_2, we can improve on runtime performance while impacting path length very little. This does not mean w_2 should be too high, as shown by examples where $w_2 = 10$, which result in significant increases in path length. So this means there is a trade-off between increasing w_2 and maintaining a decent path length.

We expected A* to use less memory than sequential search, since sequential search has to keep track of multiple fringes at a time. The evidence we found supports this, showing that A* uses the least memory while sequential A* search used more, no matter what the w_1, w_2 values were.

More specifically, increasing w_1 decreases the memory required somewhat, whereas increasing w_2 decreases the memory required significantly.

In conclusion, we found that the best weight paring to be $w_1 = 2.5$ and $w_2 = 2.5$.

H. Sequential A* Optimization

Similar to UCS, A* and weighted A*, a dictionary was used for each closed list and a min heap was used as a priority queue for each fringe. Dictionaries allowed for quick insertions and checks for whether an element is contained in the dictionary. Min heaps improved performance of insertions, removals and searches for the cell with the lowest priority.

A list of closed lists and a separate list of fringes were maintained to provide quick access to either the ith closed list or the ith fringe. Lists for each cell were created to store g's and h's that provide quick access and modifications to the contents.

For a more general comparison between different w_1 and w_2 values used during sequential search, see part G above. For the statements below, we assume that $w_1 = 2.5$ and $w_2 = 2.5$ were used for sequential search. Compared to A* and UCS, sequential A* search is not optimal, but it does provide a very short path nevertheless. While sequential search does take up more memory and slightly more time than the two methods, the path length it finds is more resilient than weighted A*. As weights in weighted A* increase somewhat, the path length increases dramatically. In comparison, as w_1 or w_2 increase, unless they have a huge jump in value (like with $w_2 = 10$), the path length increases, but not as much as in weighted A* and usually staying below 140. Weighted A* with higher weights and inadmissible heuristics had path lengths that hovered around 160.

w_1 is similar to the weight used in weighted A*, so increasing that should increase path length and decrease runtime. However, we observed that w_1 does not have the same impact as the weight in weighted A*. This is likely because with lower w_2 values, sequential search relies more on its anchor and, because weights did not impact diagonal distance heuristic as much, the path length does not change.

With higher w_2 values, sequential search relies less on its anchor and acts more like weighted A* with an inadmissible heuristic. As a result, sequential search runs in less time, uses less memory, but finds poorer paths.

I. Sequential A* Proof
Part A:
At first we assume $key(s,0) = w_1 * h_0(s) + g_0(s) > w_1 * g^*(s_{goal})$

Then we assume a least path from s_{start} to s_{goal} as $\mathrm{P} = \pi(s_0 = s_{start}, ..., s_k = s_{goal})$. Then we pick a state s_i that is not unexpanded by the anchor search.It should also be in the open set at index 0,which means $s_i \subset OPENSET_0$. It is always possible to find such state since s_{start} is inserted at the initialization.Once any consecutive state s_j is expanded doing the anchor search, s_{j+1} is guaranteed to be inserted in the $OPEN_0$ set.

In the meantime, $s_k = s_{goal}$ won't be expanded in the anchor search.The pseudocode states that whenever s_{goal} have the MinKey in the $OPEN_0$, it will terminate and return the path.

Concerning of $g_0(s_i)$. When i = 0,

$$g_0(s_i) = g_0(s_{start}) = 0 \leq w_1 * g*(s_i)$$

because $g*(s_{start}) = g_0(s_{start}) = 0$.

If $i \neq 0$, we can certainly know s_{i-1} has already been expanded in the anchor search. When chose the s_{i-1} to expand, we would have $g_0(s_{i-1}) \leq w_1 * g^*(s_{i-1})$ from the given property.

We can conclude(s‘ as successor of s)

$$\begin{aligned} g_0(s_i) &\leq g_0(s_{i-1}) + c(s_{i-1}, s_i) \\ &\leq w_1 * (g^*(s_{i-1}) + c(s_{i-1}, s_i)) \\ &= w_1 * g^*(s_i) \end{aligned}$$

because $s_{i-1}, s_i \subset optimalpath$

So we now have $g_0(s_i) \leq w_1 * g^*(s_i)$. We apply this equation,

$$\begin{aligned} key(s_i, 0) = w_1 * h_0(s_i) + g_0(s_i) &\leq w_1 * g^*(s_i) + w_1 * h_0(s_i) \\ &\leq w_1 * g^*(s_i) + w_1 * c*(s_i, s_{goal}) \end{aligned}$$

h_0 is our anchor search heuristic, so it is consistent and admissible,thus,
$= w_1 * g^*(s_{goal})$

Since $s_i \subset OPEN_0$ and $key(s_i, 0) \leq w_1 * g^*(s_{goal}) < key(s, 0)$, we have a contradiction to our initial assumption. Hence it proves $key(s, 0) \leq w_1 * g^*(s_{goal})$.

Part B:
We first clarify that there are 3 cases the sequential A* search will terminate, find path with anchor search, or by other inadmissible search or no solution.

If it is terminated with anchor search finding the path, with the given property, we will obtain,

$$\begin{aligned} g_0(s_{goal}) &\leq w_1 * g^*(s_{goal}) \\ &\leq w_1 * w_2 * g^*(s_{goal}) \end{aligned}$$

since both w_1 and w_2 are greater than 1.

if it is terminated with other search, then

$$\begin{aligned} g_i(s_{goal}) &\leq w_2 * OPEN_0.Minkey() \\ &\leq w_2 * w_1 * g^*(s_{goal}) \end{aligned}$$

So we have the final path cost to be within $w_1 * w_2$ factor of the optimal cost.

If we encountered the 3rd case, from part A we just proved, we know $OPEN_0.Minkey() \leq w_1 * g^*(s_{goal})$. $OPEN_0 = \emptyset$ if and only if $OPEN_0.Minkey() = \infty ==> g*(s_{goal}) \geq \infty$,and we could still say it is bound within $w_1 * w_2$ although there is no finite solution.