Heuristic Search Using Information From Many Heuristics

Quintanilla, Arthur (`netid: artquint`)
Eisen, Stephen (`netid: see58`)

February 20, 2017

A* Algorithm Assignment - Phase 1

In this assignment we were tasked with implementing the A* path-finding algorithm. We will discuss the different heuristics that were used to implement this algorithm, optimizations that were used to decrease its running time, and provide an in depth analysis of each heuristic based on 50-benchmark grid worlds that are supplied in this project's zip folder.

Heuristics

This section will give a brief overview of the different heuristics we chose to use for the demonstration of our A* Algorithm Assignment. There will be no comparison using hard data from our 50 benchmark comparisons here. Such content can be found in the Experimental Evaluation section.

1. A* Given Heuristic

   $$h(n) = \sqrt{2} * min(| s^x - s^x_{goal} |, | s^y - s^y_{goal} |) + max(| s^x - s^x_{goal} |, | s^y - s^y_{goal} |) - min(| s^x - s^x_{goal} |, | s^y - s^y_{goal} |)$$

   This heuristic is neither admissible nor is it consistent. It is inadmissible because it over-estimates the true lowest cost from s to s_{goal}, and it is not consistent because $\nexists\ s' \in Successors(s)$ such that the cost from s to $s' + h(s') < h(s)$.

   In general the given A* heuristic from this assignment was by far the most balanced. When using this heuristic it will produce a path length that is only slightly sub-optimal, but few nodes will be expanded and the running time will much shorter than other heuristics we discuss in this report.

2. Manhattan Distance

   $$h(n) = 0.25(|s^x - s^x_{goal}| + |s^y - s^y_{goal}|)$$

   This was the first heuristic that we found to be admissible. It adds the vertical and horizontal distances between s and s' and adds them together, effectively giving us the Manhattan distance to the goal node like city blocks in Manhattan.

   However this sum by itself would not be admissible because $\exists$ s to s' transitions such that the cost is less than the euclidean difference between cells (i.e. transitions on highways can result in 0.25, 0.325 or 0.5 cost where the minimum euclidean distance would be 1).

   In order to simplify this heuristic to address a slightly easier problem, but still maintain consistency, we modified it to assume that all node transitions cost 0.25 (i.e. all nodes would be highways in this simpler sub-problem).

   In this way the heuristic will only ever under-estimate and not over-estimate the true cost by adding a 0.25 multiplier. This maintains its admissibility and consistency at the cost of under-estimating the true cost whenever the s to s' transition is greater than 0.25.

3. Euclidean Distance

$$h(n) = 0.25\sqrt{(s^x - s^x_{goal})^2 + (s^y - s^y_{goal})^2}$$

This heuristic uses the distance formula to calculate the Euclidean distance between two cells and uses that as an estimate of the distance from any node s to s_{goal}.

However, because the Euclidean distance between two nodes s and s' will overestimate the true cost of the transition, unless we add a multiplier to account for this the heuristic will be inadmissible.

To fix this we added 0.25 multiplier to the heuristic function such that any Euclidean distance from s to s_{goal} will be scaled down as though all transitions are between two directly adjacent, non-diagonal highway nodes (the lowest possible cost that can exist for any s to s_{goal} path). Since, with this multiplier, the true cost cannot possibly be over-estimated it is both admissible and consistent.

4. Diagonal Distance

$$\begin{gathered} d_{max} = max(\mid s^x - s^x_{goal} \mid, \mid s^y - s^y_{goal} \mid) \\ d_{min} = min(\mid s^x - s^x_{goal} \mid, \mid s^y - s^y_{goal} \mid) \\ h(n) = c_d * d_{min} + c_n * (d_{max} + d_{min}) \end{gathered}$$

This heuristic considers movement in an 8-directional grid world such as ours where the minimum cost to move horizontally or vertically from s to s' is different than the cost to move diagonally from s to s'.

Here c_n is defined as the lowest cost to transition between two adjacent non-diagonal nodes and c_d is defined as the lowest cost to transition between two adjacent diagonal nodes. In our grid world $c_n = 0.25$ and $c_d = \sqrt{2}$.

This heuristic however is inadmissible and inconsistent as it will in certain cases overestimate the true cost from s to s_{goal}.

5. Chebyshev

$$h(n) = max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$$

The Chebyshev heuristic takes the maximum value of either the horizontal or vertical distance between any s and s_{goal}.

This heuristic is not admissible, because it does not take into account that the minimum transition cost between any two s and s' may be less than one. If we were to add a constant multiplier of 0.25 to this heuristic it would make it admissible, as the minimum transition cost between any two s and s' is 0.25.

However, we chose to leave this heuristic as inadmissible in order to gather data regarding inadmissible heuristics.

Optimizations

- One opportunity for optimization was changing the method with which we searched through the closed list for nodes that have already been fully expanded.

  Originally we used an ArrayList to implement our closed list. The time complexity for searching through the ArrayList to determine whether or not a node existed in the closed list is O(n). Instead of using this ArrayList to keep track of the closed list, we instead used a 2-dimensional array with dimensions 120x160, called closedArray.

  Each element in the array represents one node on the grid. When a node on the grid has been entered into the closed list, the array element with indices matching the position of this node is set to true. Here, true represents the corresponding node on the grid has been entered into the closed list and is fully expanded, and false represents the opposite. By using this 2-dimensional array, instead of an ArrayList, the search time complexity for determining a nodes existence in the closed list is reduced from O(n) to O(1).

- We found another similar opportunity for optimization when determining existence in the fringe (which we implemented as a minimum binary heap).

  When we first implemented the .Contains method for our heap (which returned true if an item was contained within the heap) we searched through the array that we used to implement our heap and checked each element for equality and returned true if an equal item was found and false otherwise. This resulted in an O(n) time complexity for determining if an item existed in the fringe.

  Instead of using this .Contains method however we changed our strategy for determining existence to be similar to our previous optimization. We again used a 2-dimensional array to flag nodes that have been pushed into the heap and de-flag nodes that have been removed from the heap. The time complexity for accessing this array and determining a nodes existence in the fringe was then reduced from O(n) to O(1).

- Another opportunity for optimization we discovered was with the constant $\sqrt{2}$, $\sqrt{8}$ and $\frac{\sqrt{2}+\sqrt{8}}{2}$ values used for determining cost between two diagonal cells.

  Originally we were using Math.Sqrt(2), etc... calls each time we wanted to calculate diagonal cost. We learned however that the Math.Sqrt call is very expensive during runtime, and were able to reduce our average elapsed time by storing these values in compile-time constants and using these constant values rather than making Math.Sqrt calls.

- We also realized that in our implementation of the A* algorithm we were re-calculating several values such as Cost and H-values for each node s several times. To reduce running-time we calculated these values only once and stored the results in local variables.

Experimental Evaluation

Below are the experimental result averages across all 50 benchmarks.

A*

Heuristic	Elapsed Time (ms)	Nodes Expanded	Path Length	Mem Used (kB)
Default	11.66	1109.46	113.70	2388.64
Manhattan	67.00	8501.62	97.51	2316.30
Euclidean	71.34	9403.66	97.50	2442.02
Diagonal	43.88	4427.42	104.61	2434.66
Chebyshev	26.72	2983.88	111.79	2387.24

Weighted A*

Heuristic/Weight	Elapsed Time (ms)	Nodes Expanded	Path Length	Mem Used (kB)
Default/w = 1.25	2.24	248.92	129.54	2390.64
Default/w = 2.25	1.18	117.38	145.96	2391.14
Manhattan/w = 1.25	57.44	7346.56	97.58	2334.48
Manhattan/w = 2.25	33.76	3678.18	101.09	2336.56
Euclidean/w = 1.25	67.74	8570.78	97.54	2440.00
Euclidean/w = 2.25	46.62	5286.38	99.12	2442.16
Diagonal/w = 1.25	29.22	3252.78	112.89	2435.08
Diagonal/w = 2.25	11.76	1392.02	128.33	2435.34
Chebyshev/w = 1.25	13.08	1217.90	131.58	2413.96
Chebyshev/w = 2.25	1.20	131.74	148.51	2412.10

Uniform Cost

Algorithm	Elapsed Time (ms)	Nodes Expanded	Path Length	Mem Used (kB)
Uniform Cost	78.32	12045.84	97.51	2406.06

1. Evaluation of given A* heuristic

   Overall the given A* algorithm was the most balanced of all A*-family algorithm heuristics that we implemented for this assignment.

   The average running time of the A* algorithm using this heuristic was 9.72ms. This was a faster time than all other heuristics except for Chebyshev with a weight of 2.25 and a running time of 1.00ms. However, the average path length retrieved by the given A* heuristic was 113.18 cost, while for Chebyshev with a weight of 2.25 it was 147.48 cost. This implies that even though Chebyshev with weight 2.25 was much faster than A*, its retrieved path was far less optimal.

So, while the given A* heuristic is certainly not the fastest, it does produce a lower cost path than other alternatives and this can be considered a good trade off for the extra run time.

The Manhattan and Euclidean heuristics with weight 1.25 produced path length averages of 97.74 cost and 97.68 cost respectively. These paths have a much more optimal cost than the given A* heuristic but their running times were much larger with an average run time of 53.80ms and 60.26ms respectively.

Here, the slightly less optimal cost path can be considered a good trade off since the given A* heuristic has a running time of roughly 1/6th of the Manhattan and Euclidean heuristics with weight 1.25 while only increasing the path cost by about a factor of 1.15.

2. Evaluation of Manhattan heuristic

   The benefits of using the Manhattan heuristic were its very close to optimal path costs. The average optimal path cost was 97.65 and Manhattan with a 1.25 weight and 2.25 weight returned paths of average cost 97.74 and 100.64 respectively. This is only a .0009% error and .03% error respectively.

   However, the trade-off here is the large increase in average run time. We don't see that large of a decrease in path cost from the given A* heuristic but we have a 553% and 312% increase in runtime when compared to the given A* heuristic considering Manhattan with weight 1.25 and 2.25 respectively. This is a very large increase in run time for a very small increase in path length. In order for the trade-off to be worth it, run time must not be a large factor when considering which heuristic to choose

   If we follow this trend it is easy to see that the more weight we give the Manhattan heuristic, the faster the run time will be. However, the trade off is less-optimal path costs.

3. Evaluation of Euclidean heuristic

   The benefits of using the Euclidean heuristic (similar to the Manhattan) become obvious when run-time is not a heavy factor.

   Increasing the weight of the Euclidean heuristic will yield less optimal paths, that are computed using less node expansions, and as a result shorter running times. However, like Manhattan, these small gains in path-cost optimization might not be that significant when considering that the runtime of the given A* heuristic has an average of 9.72ms, and the average runtime for the Euclidean heuristic with weights 1.25 and 2.25 are 60.26ms and 41.72ms respectively.

   The benefit of the Euclidean heuristic when run-time is not as significant is that it can produce very close to optimal path-costs the closer the weight is to 1. However as the weight is reduced, the number of node expansions and running time will increase.

4. Evaluation of Diagonal Distance heuristic

The benefits of using the Diagonal heuristic in A*-family algorithms lies solely in the reduced running time. This heuristic has very poor performance compared to the previous two because of its inadmissibility and inconsistency.

With average running times of 27.66ms and 10.58ms when using weights 1.25 and 2.25 respectively neither has a better running time than the given A* heuristic which has an average running time of 9.72ms.

In the case of Diagonal with weight 1.25, the running time is about 2.85 times longer and only reduces the average path length from 113.18 cost to 112.46 cost and there are about 2.98 times as many node expansions when compared to the given A* heuristic.

Increasing the weight to 2.25 will yield a running time roughly similar to that of the given A* heuristic, but also increases the average path-cost by about 1.12 times.

5. Evaluation of Chebyshev heuristic

   The Chebyshev heuristic is another inadmissible and inconsistent heuristic. Just like the Diagonal heuristic which was also inadmissible, the Chebyshev heuristic yielded much faster run-times than the heuristics that were admissible but also had a larger number of node expansions and a much greater average path-cost.

   The most notable result of the Chebyshev heuristic evaluation was that when the weight was increased to 2.25ms the average running time was reduced to about 1.00ms. This was by far the fastest running time of any heuristic that we tested.

   Chebyshev with 2.25 weight also had the least amount of node expansions with only 131.46 node expansions. However, it also had the worst path-cost with a large 147.48 cost. This is 1.51 times larger than the optimal path-cost of 97.65 cost.

   The Chebyshev heuristic can be a good choice if finding the optimal path-cost is very insignificant and an answer just needs to be determined as fast as possible. Reducing the weight of Chebyshev to 1.25 will make the results more reasonable, but considering that even with this weight it will have a greater run-time, more node expansions and a greater path-cost than the given A* heuristic, there is no reason to use Chebyshev with such a low weight.

6. Evaluation of Uniform Cost

   The uniform cost implementation of the A* algorithm does not use any heuristic. Rather it is guided only by finding the $s' \in Successors(s)$ with the lowest cost.

   In this way it is guaranteed to find the optimal path in terms of cost but will also have a large number of node expansions (since it has no heuristic to guide it) and a very large run time (since so many nodes need to be expanded).

   This implementation is recommended when running-time is not at all a factor and it is only important to retrieve the path with the optimal cost.

On average the uniform cost implementation had 12,158.40 node expansions and took 70.76ms. This was the greatest running time of all other algorithm implementations and took 7.23 times longer and expanded 10.99 times more nodes than the most balanced heuristic, the given A*.

If finding the optimal path-cost is necessary than uniform cost is the best implementation to use. However, if path-cost can be sub-optimal, it is much better to use Manhattan or Euclidean with low weights. Then, if running time is important, it is still better to use the given A* heuristic.

Conclusions

We found that the given A* heuristic for this assignment was by far the most balanced heuristic to use for A*-family algorithms. However, it is important to note that this heuristic is not admissible. If the admissibility is important then the Manhattan heuristic should be used.

Manhattan is the best admissible heuristic that we found in our evaluation. The only other admissible heuristic we used was the Euclidean. With similar weights, Manhattan performed with a faster run time, less node expansions, and only a slightly less optimal path.

For a weight of 1.25 Manhattan had a path that only cost 1.0006 times more and for a weight of 2.25 it had a path that only cost 1.0154 times more. The better path-cost yielded by the Euclidean heuristic does not offset the much faster run-time of the Manhattan heuristic. For a weight of 1.25 the Manhattan had a run-time that ran in 89% of the time yielded by Euclidean with a weight of 1.25 and for a weight of 2.25 it had a run-time that ran in 73% of the time yielded by Euclidean with a weight of 2.25.

Based on this cost-benefit analysis, the Manhattan heuristic is generally the better choice over the Euclidean heuristic in our 8-direction grid world.

A* Algorithm Assignment - Phase 2

In the second phase of this assignment we were tasked with implementing A* family algorithms that used multiple heuristics as metrics for prioritization of node expansion. The two implementations were the Sequential and Integrated methods.

Efficiency

A* Sequential

Heuristics	Time (ms)	Nodes Expanded	Path Length	Mem Used (kB)
Anchor=Manhattan, Weights=1.25, 2.00	82.30	7830.44	121.60	4632.76
Anchor=Manhattan, Weights=1.50, 2.25	42.08	4309.12	128.33	4558.58
Anchor=Euclidean, Weights=1.25, 2.00	100.55	9669.20	119.52	4664.02
Anchor=Euclidean, Weights=1.50, 2.25	60.36	6148.14	127.59	4590.12

A* Integrated

Heuristics	Time (ms)	Nodes Expanded	Path Length	Mem Used (kB)
Anchor=Manhattan, Weights=1.25, 2.00	271.04	7840.72	97.80	2980.52
Anchor=Manhattan, Weights=1.50, 2.25	230.73	6537.40	97.66	2947.62
Anchor=Euclidean, Weights=1.25, 2.00	298.04	9218.22	97.92	3017.16
Anchor=Euclidean, Weights=1.50, 2.25	269.80	8115.00	97.71	2990.16

In general we found that the sequential and integrated heuristic methods would take longer than other A* algorithm implementations discussed in this report, especially with low weights. First we will compare the sequential and integrated algorithms to each other, and then we will compare them to the algorithms from phase 1.

The faster of these two implementations was clearly the sequential heuristic implementation based on the observations of our 50 benchmarks. To test this implementation we used four cases. For two cases we used the Manhattan heuristic as our anchor and for the other two cases we used the Euclidean heuristic as our anchor. For both of these groups we tested with two different sets of weights, one set was weight 1 = 1.25 and weight 2 = 2, and the other was weight 1 = 1.5 and weight 2 = 2.25. We then used these same specifications when testing the integrated heuristic method.

In both implementations, using the Manhattan heuristic as an anchor was much faster than using the Euclidean heuristic, regardless of what weight values were assigned. For our sequential tests with low weights we found the average runtime to be 82.30ms using the Manhattan heuristic as an anchor compared to the equivalent test using the Euclidean heuristic as an anchor which had an average of 100.55ms across all benchmarks. For high weights this difference was 42.08ms vs 60.36ms respectively across all benchmarks.

We found in general that as the weights were increased, runtime decreased, the average nodes expanded decreased, and the path length increased. Making the sequential implementation efficient then relies heavily on selecting a proper anchor heuristic and choosing good weights.

Our low weight/Manhattan heuristic test expanded on average 7830.44 nodes which was fewer than the comparable test with a Euclidean anchor which expanded on average 9669.20 nodes across all 50 benchmarks. This pattern held true with high weights as well. The difference in path lengths for this comparison was 121.60 vs 119.52 respectively. So while Manhattan had a faster runtime with less nodes expanded, Euclidean did manage to find a better path for both high and low weights.

The low weight/Manhattan test used 4632.76kB of memory which was only slightly more than the high weight test which used 4558.58kB of memory. This trend followed a similar pattern for our Euclidean test. Raising the weight will decrease the amount of memory needed but not by much.

When testing the integrated algorithm there was a similar trend (i.e., as weight increased the runtime decreased, path length shortened, and nodes expanded decreased). However, the results were still very different.

The runtimes for integrated were overall much longer, with the shortest being the high weight/Manhattan anchor heuristic test with an average run time of 230.73ms across all benchmarks. With low weights the nodes expanded were roughly similar to the results for sequential.

The most important difference in the integrated tests however are the differences in path length. Even with low weights of w1 = 1.25 and w2 = 2.00, for both the Manhattan and Euclidean anchor heuristics the path lengths were very close to the true shortest path cost.

The average true path cost derived from the uniform cost search across all 50 benchmarks was 97.51. As seen in the above A* integrated table all tests averaged to anywhere between 97.66 to 97.92 path length. This is very close to the true shortest cost, but the runtime to reach this is very expensive, and may not be worth it.

In order to make integrated more appealing, higher weights should be used, this way the path length will but slightly longer than the true shortest cost but the runtime will be less than the time for a uniform cost search. The implementer of this algorithm should ideally increase the weight until a desired average runtime is reached for a suitable sub-optimal path cost.

Sequential A* Efficiency Compared to Phase 1

1. Compared with A* - Manhattan Heuristic

   The A* algorithm from phase 1 utilized a single heuristic with no weight factor. Depending on the heuristic used, the efficiency of the algorithm varied. This is similar to the sequential implementation and choosing a anchor heuristic. The anchor heuristic of the sequential algorithm will be the base factor in determining the efficiency of the algorithm.

   In the single heuristic A* approach, using the Manhattan heuristic yielded an average time of 67.00ms across all benchmarks. For low weight (w1 = 1.25 and w2 = 2) A* sequential using Manhattan as an anchor, the average time was 82.30ms and for high weight (w1 = 1.5 and w2 = 2.25) the average time was 42.08ms.

   Evidently, A* sequential with low weights using the Manhattan heuristic as an anchor will have a longer run time than using the Manhattan heuristic as an anchor for regular A*. However, increasing the weights of the sequential implementation will eventually bring the run time below that of regular A*.

   Regular A* Manhattan had an average of 8501.62 nodes expanded across all benchmarks. Low weight sequential using Manhattan as an anchor would expand an average of 7830.44 nodes and high weight sequential with the Manhattan heuristic as an anchor would expand an average of 4309.12 nodes. Thus, even though regular A* using the Manhattan heuristic has a lower run time than sequential, it still expands on average more nodes than sequential using Manhattan as an anchor even when sequential uses lower weights.

   The average path length for the regular A* using the Manhattan heuristic was 97.51 cost, which was exactly the average true path cost across all 50 benchmarks. The average path length for the sequential implementation using the Manhattan heuristic as an anchor with low weights was 121.60 cost. This path is far longer (and thus more sub-optimal) than the regular A* algorithm using the Manhattan heuristic. Considering that sequential with low weights and Manhattan as an anchor runs slower and finds a worse path, it is not efficient to use the sequential A* algorithm using the Manhattan heuristic as an anchor unless higher weights are used.

   For example, when the A* sequential algorithm using the Manhattan heuristic is used with higher weights, the average path length was not much worse at 128.33 cost. Compared to the 51.13% drop in run time down to 42.08ms this is a trade off that might make sequential A* using the Manhattan heuristic as an anchor more suitable than the regular A* using the Manhattan heuristic in certain situations.

   The nature of sequential A* is that it will use much more memory than the regular A* algorithm since it has n fringes rather than a single fringe. The average memory used by the regular A* algorithm using the Manhattan heuristic was

2316.30kB compared with the sequential A* algorithm using the Manhattan heuristic as an anchor which had 4632.76kB and 4558.58kB for low and high weights respectively.

Overall Sequential A* might be preferred over regular A* with similar heuristics if a slightly longer path length is acceptable. To achieve this sequential must be used with high enough weights such that the run time will be less than its regular A* algorithm equivalent yet small enough that its path length will not be that much worse when compared to its A* algorithm equivalent.

2. Compared with A* - Euclidean Heuristic

   The comparisons here are very similar to the Manhattan heuristic except for when considering the cost-benefit. The Manhattan heuristic shines when fast run-times, few nodes expanded, at the cost of a slightly worse path length is desired. Euclidean however is a preferable heuristic when the run times can be slightly longer and the nodes expanded can be slightly greater with the benefit of having a better path length.

   This can be seen when comparing the benchmark results of the sequential A* algorithm using the Euclidean heuristic as an anchor for both high weights (w1 = 1.5 and w2 = 2.25) and low weights (w1 = 1.25 and w2 = 2) with the benchmark results of the regular A* algorithm using the Euclidean heuristic.

   The average runtime for the A* algorithm using the Euclidean heuristic was 71.34ms. Compared to the average runtimes for the sequential A* algorithms using the Euclidean heuristic as an anchor for both low and high weights, which were 100.55ms and 60.36ms respectively, it is demonstrated how adjusting the weights of the sequential A* algorithm will greatly affect the efficiency of the algorithm.

   Using A* sequential with the Euclidean as an anchor with low weights will yield a slower runtime, more nodes expanded, a longer path length, and more memory used than its A* algorithm counterpart. In general, there is no scenario in which using A* sequential with the Euclidean heuristic as an anchor with low weights is beneficial. The benefits only appear when higher weights are used.

   For instance, using A* sequential with the Euclidean heuristic as the anchor with high weights will yield an average runtime of 60.36ms, an average nodes expanded of 6148.14 nodes, an average path length of 127.59 and an average of 4590.12kB used. Compared to the regular A* algorithm using the Euclidean heuristic, which had an average run-time of 71.34ms, an average nodes expanded of 9403.66 nodes, an average path length of 97.50 cost and an average of 2442.02kB used, the benefits become more apparent.

   With these higher weights we see a 15.4% reduction in average runtime, a 34.6% reduction in average nodes expanded, at the expense of a 46.8% increase in memory used, and a average path length that is 23.6% more sub-optimal. As the weights increase these reductions will decrease more but the path length and memory used will also increased, thus the weights should not be set too high.

Overall, with medium weights this implementation could be comparable to the regular A* algorithm using the Euclidean heuristic, however the regular A* is still most likely more balanced for general purpose.

3. Compared with Weighted A* - Manhattan Heuristic

   The weighted A* algorithm using the Manhattan heuristic is a very balanced implementation of the A* algorithm. In most respects it is one of the best implementations we tested with our 50 benchmark evaluation. However, the benefits of adding weights to this single heuristic implementation did not translate well to the sequential implementation using Manhattan as an anchor.

   The average runtimes for A* sequential using low and high weights were 82.30ms vs 42.08ms respectively. The average runtime for weighted A* using the Manhattan heuristic for low and high weights were 57.44ms and 33.76ms respectively. This was a 43% and 24.6% increase in runtime respectively when using A* sequential vs the weighted A*.

   Despite this increase in runtime, we did not see a reduction in average path length. In fact, nodes expanded, path length, and memory used all increased in all cases when using sequential A* with a Manhattan heuristic compared to weighted A* with the Manhattan heuristic. For comparable weights there was a 6.6% and 17.1% increase in average nodes expanded, a 24.6% and a 26.9% increase in average path length and a 98.4% and a 95% increase in memory used when comparing weighted A* with sequential A*.

   Thus for all metrics, sequential A* using the Manhattan heuristic as an anchor performed worse when compared with weighted A* using the Manhattan heuristic.

4. Compared with Weighted A* - Euclidean Heuristic

   Similar to the above findings, we found that using a sequential A* algorithm with the Euclidean heuristic as its anchor would perform worse in all metrics when compared to its single heuristic weighted A* counterpart.

   For low and high weights respectively there was a 48.4% increase and a 29.4% increase in run-time, a 31.6% increase and a 16.3% increase in average nodes expanded, a 22.5% increase and a 28.7% increase in path length, and a 91.1% increase and a 88% increase in memory used.

   Thus for each metric, using the sequential A* algorithm with the Euclidean heuristic as an anchor performed worse compared to its weighted A* counterpart.

5. Compared with Uniform Cost

   With higher weights the sequential A* algorithm will run faster than the uniform cost algorithm. It will expand fewer nodes granted both weights are above 1, but it will use more memory and will find a far more sub-optimal path than uniform cost.

For example, our 4 A* sequential evaluations across the 50 benchmarks yielded path lengths of 121.60, 128.33, 119.52, and 127.59. None of these come close to the path cost found by uniform cost which was 97.51. This is do largely to the fact that the heuristics were weighted and some non-anchor heuristics were inadmissible.

Integrated A* Efficiency Compared to Phase 1

In general we found that for all Integrated evaluations across the 50 benchmarks the average runtimes were much longer than any other algorithm discussed in this report. However, this algorithm used very little memory and found among the most-optimal paths (alongside uniform cost and weighted A* Manhattan).

1. Compared with A* - Manhattan Heuristic

   The integrated A* algorithm using Manhattan heuristic took 4.05 times and 3.43 times longer than the regular A8 algorithm using the Manhattan heuristic for low and high weights respectively.

   However, with this large increase in runtime there was virtually no increase in path length optimality. All path lengths calculated were within 0.01% of the path length calculated by the regular A* algorithm.

   Due to the huge increase in runtime and virtually no path length gain, we found that the regular A* using the Manhattan heuristic performed better in all metrics except for nodes expanded.

2. Compared with A* - Euclidean Heuristic

   The integrated A* algorithm using the Euclidean Heuristic had the longest runtimes of any algorithm tested in our evaluation. For low and high weights, compared to the regular A* implementation using the Euclidean heuristic, there was an increase in runtime by a factor of 4.17 and 3.78 respectively.

   However, for low and high weight integrated A* using the Euclidean heuristic as an anchor there were 9218.22 and 8115.00 nodes expanded on average, both of which were less than the average nodes expanded for the regular A* algorithm using the Euclidean heuristic which was 9403.66 nodes expanded.

   Still, there was again virtually no gain in path length optimality. The average path length across all benchmarks for our integrated A* algorithm using the Euclidean heuristic as an anchor for both low and high weights were within 0.01% percentage difference from the path length generated by the regular A* algorithm using the Euclidean heuristic.

   There was also no improvement in memory usage, since here the integrated tests all used more memory then our regular A* test using the Euclidean heuristic.

In general we found that because of the dramatically increased run times and the virtual no gain in path length optimization, this algorithm would not be a good choice when compared with the regular A* algorithm using the Euclidean heuristic.

3. Compared with Weighted A* - Manhattan Heuristic

   When compared with weighted A*, again this implementation had runtimes which were far too high and net-gains in path length optimality that were far too low to be considered a good trade off.

   The runtimes for low and high weights increased by a factor of 4.72 and 6.83 respectively. For low weight the path length was slightly higher (by a factor of < 1.01) and for high weight there was only a 3.39% decrease in path length.

   The weighted A* algorithm using the Manhattan heuristic also expanded fewer nodes and used less memory.

4. Compared with Weighted A* - Euclidean Heuristic

   When compared with the weighted A* implementation using the Euclidean heuristic, this algorithm still took for too long and saw too little benefit to be considered a viable alternative.

   For low and high weights the average run-time for the A* integrated algorithm using the Euclidean heuristic as an anchor was 4.40 times and 5.78 times slower than the weighted A* algorithm using the Euclidean heuristic for comparable weights.

   Compared to the $< 2\%$ gain in path length, this was not a good alternative considering there are far better choices discussed in this evaluation.

   On average more nodes were expanded and more memory was used using the integrated A* algorithm with the Euclidean heuristic as an anchor compared to the weighted A* algorithm using the Euclidean heuristic for comparable weights.

5. Compared with Uniform Cost

   The only metric that integrated A* using the Euclidean heuristic as an anchor beat the Uniform Cost algorithm in was nodes expanded. There were on average 23.4% and 32.6% fewer nodes expanded for low and high weights respectively.

   However, the comparable runtimes saw an increase by a factor of 3.81 and 3.44 and the comparable path lengths only differed by $< .01$. There was no benefit besides nodes expanded in this evaluation for which the integrated A* using the Euclidean heuristic outperformed the Uniform Cost search.

Question I

Let s_i be a state in the $OPEN_0$ queue. s_i must have been pushed onto $OPEN_0$ from some previous state s. When s was expanded it pushed all nodes $s' \in succ(s)$ where s' was not previously in $CLOSED_i$ and $g(s') > C(s, s')$.

Consider s_{start} when it is first expanded. All nodes s' are pushed onto $OPEN_0$. One of these nodes must be on the shortest path to s_{goal} since there are no other paths to take from s except through s' (unless there is no solution for the grid).

Thus there always exists $s_i \in OPEN_0$ such that s_i is on the shortest path from s_{start} to s_{goal}. We can say that the key value of the state with the minimum key in $OPEN_0$ is $Key(s, 0) \leq Key(u, 0) \forall u \in OPEN_0$.

Since $Key(s_i, 0) = g(s_i) + w_1 * H_0(s_i)$

and $\exists$ s such that $s \in$ the shortest path to s_{goal},

we can say that $g(s) + w_1 * H_0(s) \leq w_1 * g^*(s_g)$.

$g(s) \leq w_1 * g^*(s_g) - w_1 * H_0(s)$

$g(s) \leq w_1(g^*(s_g) - H_0(s))$

When $s = s_{goal}$ (a requirement for the terminating condition), $H_0(s) = 0$.

$g(s) \leq w_1 * g^*(s_g)$

Therefore, for the terminating condition of anchor search, we know that $g_i(s_{goal}) \leq w_1 * g^*(s_{goal})$.

Now we must consider the inadmissible search (This is our other of the two terminating conditions).

In order for the algorithm to terminate in the *ith* iteration where $i \neq 0$ we must have $OPEN_i.MinValue \leq w_2 * OPEN_0.MinValue$.

Let s be the state with the minimum key value in $OPEN_i$.

Then, $Key(s, i) = g(s) + w_1 * H_i(s) = OPEN_i.MinValue$

In order to reach the terminating condition $\exists$ $s \in OPEN_i$ such that $Key(s, i) \leq w_2 * OPEN_0.MinValue$.

We know from the previous condition that

$OPEN_0.MinValue = Key(s, 0) = g(s) + w_1 * H_0(s)$

Then, $g(s) + w_1 * H_i(s) \leq w_2 * (g(s) + w_1 * H_0(s))$

$g(s) + w_1 * H_i(s) \leq w_2 * g(s) + w_1 * w_2 * H_0(s)$

$g(s) \leq w_2 * g(s) + w_1 * w_2 * H_0(s) - w_1 * H_i(s)$

$g(s) \leq w_2 * g(s) + w_1 * w_2 * H_0(s)$

We know that when $s = s_{goal}$ (as it will in the terminating condition), $H_0(s) = 0$.

Then $g(s) \leq w_2 * g(s)$.

Therefore, for the terminating condition of the inadmissible search, we know that $g_i(s_{goal}) \leq w_2 * g^*(s_{goal})$.

Since for the terminating condition for the anchor search we know that $g_i(s_{goal}) \leq w_1 * g^*(s_{goal})$ and for the terminating condition of the inadmissible search we know that $g_i(s_{goal}) \leq w_2 * g^*(s_{goal})$, we can bound $g_i(s_{goal}) \leq w_1 * w_2 * c^*(s_{goal})$.

Therefore, the solution cost obtained by the algorithm is bounded by a $w_1 * w_2$ sub-optimality factor.

Question J

i. We know that there are only two closed lists for the A* integrated algorithm. One represents nodes that have been expanded with key values using the anchor heuristic, and another represents nodes that have been expanded with key values using the other heuristics. When the algorithm begins, both of these lists are empty.

   In the algorithm there are two paths a node can take. It can first be expanded in the anchor search. When this happens, it will be expanded for both the anchor and the non-anchor heuristics and will be placed into the anchor's closed list. Once a node is in the anchor's closed list it will not be expanded again.

   The other path a node can take is if it is first expanded in a non-anchor ith iteration. In this case it will be expanded onto both the anchor fringe and the non-anchor fringes, but will only be placed into the non-anchor closed list. Thus in a following iteration it can still be expanded a second time.

   Since these are all the paths a node can take through the algorithm, we can see that it will never be expanded more than twice.

ii. When a node is expanded in the anchor search, it will be pushed into the $fringe_{anchor}$, and all other $fringe_i$ for $i \in 1...n$ where $n =$ the total number of heuristics used.

   After this expansion it is placed into the anchor's closed list. Even if the node is popped off of any $fringe_i$, whether in an anchor or non-anchor search, it will not be re-expanded, since in ExpandState(s) we check to see if it is in the anchor closed list. As seen above, in this case, we know that will always be true.

Therefore, if a node that has been expanded in an anchor search first, is attempted to be re-expanded in a non-anchor search, it will see that it has been inserted into the anchor closed list, will not expand it, and will place it into the non-anchor closed list.

Thus such a node is only expanded once.

iii. In order for a state s to be expanded in an inadmissible search, the min-value on $fringe_i$ must be less than the min value on $fringe_{anchor}$ multiplied by a factor of weight $w2$.

After s is expanded it is removed from the fringe and inserted into the non-anchor closed list so that it cannot be expanded in a non-anchor search again. The only case where s can be expanded again is if is on the anchor fringe with a lower value than the min-value on $fringe_i$. In order for this to occur, it's G value must have been lowered in some previous search, otherwise $fringe_i$ min-value would be lower.

Thus, s will be expanded in an anchor search after it was expanded in a non-anchor search if and only if its G-value was lowered by the expansion of some other node s'.