# Weighted A* Search: The Effect of Weight on Performance

Michael Spiese

December 17, 2022

**Abstract**

When searching for paths through a state-space using A* search, the time it takes to find a solution can be very large. However, applying a weight to the heuristic can result in useful boosts in performance. This paper studies the effect on search performance of the weight placed on the heuristic function in the cost function of Weighted A* search. To test this, custom software is used to search a map of Minneapolis to compare the execution time, path cost, and path length of each solution when searching with Weighted A* and varied weights. It is shown here that, as long as the heuristic is admissible, there is a notable decrease in execution time when compared to standard A* search as weight increases. However, it is also shown that path costs and lengths are not correlated to increased weight, but instead are dependent on the graph.

## Introduction

A* search [2] is a widely-used heuristic search algorithm that is both optimal and complete. Weighted A* search [4] is a variation of A* search in which the heuristic function used for the search is weighted by an integer that is greater than 1. The cost function, therefore, becomes $f(n) = g(n)+\omega*h(n)$ where $\omega$ is the weight being applied. For this to be effective, the heuristic function $h(n)$ must be at least admissible, but it is preferred that $h(n)$ is a perfect estimate of the cost from any node $n$ to the goal. The weight applied to the heuristic implies that the estimated cost to the solution may not be an underestimate, and the solution that the algorithm finds is likely not optimal. However, weighting $h(n)$ more heavily means that more focus is placed on the estimated cost to the goal than the cost to reach node $n$ from the starting node. As a result, Weighted A* remains complete like normal A*, but the path produced is likely to be shorter than the one produced by A* as long as the heuristic function is admissible. That is, the path taken may consist of fewer nodes as paths that are slightly higher cost may now be considered because of the weight. As a result of the shorter path length, it is generally believed that the time it takes to find a solution decreases as a greater weight is applied to $h(n)$ in most cases. [8]

This is an interesting problem to consider due to the performance increase that may

be possible when the weight is increased. A faster search through a problem's state-space is always desirable, and there are many times when it would be better to use a satisficing search algorithm that sacrifices path cost to speed up a search.[8] When searching through a state-space graph using A* search, the time complexity is $O(b^d)$ which can lead to a lot of time spent searching the graph for a solution when the branching factor and depth are large. However, the implications of Weighted A* search are that by simply placing a weight on a heuristic function that provides an admissible and nearly-optimal heuristic guess, the time complexity can ideally be as low as $O(bd)$, leading to a sharp reduction in time for large state-spaces. Of course, if the optimal path is required, then Weighted A* search should not be used because of this compromise in path cost. However, if there is some room for error and a slightly longer path is acceptable, the time reduction that is possible may make Weighted A* search a better option than standard A* search. The goal of this paper is to show the effect of increasing the weight on two different heuristic functions by comparing the time each algorithm takes, as well as the total path cost and number of nodes visited for each applied weight.

# Background

A heuristic function uses information about a problem to make an informed guess about the cost from any node in a graph to a goal node. Because of this, the heuristic function can drastically cut down on the number of nodes searched and therefore the time spent searching when applied to search algorithms. However, this performance increase can be highly dependent on the heuristic function that is used. With specific regards to A* search, the difference in performance obtained from different heuristics can often be seen by observation. Additionally, in certain conditions, a weight applied to the heuristic in A* search can introduce a bound on the search time. First, consider A* search.

### A* Search

A* search [2] is a heuristic search algorithm that is both complete and optimal. To search through a graph, A* uses a cost function $f(n) = g(n) + h(n)$. Here, $g(n)$ is the cost from the start node to node $n$ and $h(n)$ is the heuristic estimate from node $n$ to the solution. To search for a solution, A* expands the node in its frontier with the smallest cost value until a solution node is expanded. Since the first solution node expanded ends the search, then the path found to this node must be an optimal path because the algorithm has not found a solution node with a smaller cost. Additionally, assuming a finite branching factor and path costs that are greater than or equal to some $\varepsilon > 0$, then A* search must be complete. However, for A* to be complete and optimal, it must be the case that the heuristic function $h(n)$ is admissible, or an underestimate of the true cost to the goal state from any node $n$.

There are many ways to find a heuristic function for any one problem as any admissible heuristic function is acceptable. For example, in the trivial case, $h(n) = 0$ for all $n$. Here, the cost function becomes $f(n) = g(n)$ which is exactly the same as the uniform-cost algorithm since the cost function depends only on the distance traveled from the start to any one node $n$. However, a better heuristic leads to better performance since $h(n)$ can provide more information to steer the search in the optimal direction and ignore nodes that are in the wrong direction

or of higher cost. An example of a heuristic function that works well for applications in pathfinding through a map is the geometric distance $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ between two points $(x_1, y_1)$ and $(x_2, y_2)$. This is always an underestimate of actual the distance needed to travel between two points through a graph because it is not constrained in the same way, and therefore can be used as a fairly accurate heuristic. [2] Let us further consider the effect of more informed heuristics on A* search.

### Heuristic Performance

A well chosen heuristic function used for an implementation of A* search can have drastic improvements on the performance of the algorithm. Liu and Gong proved this to be the case in their work on search and rescue in perfect mazes.[3] Here, a perfect maze is one that has no loops or inaccessible areas. In their experiment, ten 24x24 perfect mazes were generated and the average times to find a target from the starting point for depth-first search and A* search with three different cost functions were found. The cost functions used for the A* implementations were $f_1(i) = g_1(i) + h_1(i)$, $f_2(i) = g_2(i) + h_2(i) + h_2(j)$, and $f_3(i) = g_3(i) + \omega_1 \frac{l}{L} + \omega_2 \frac{\theta}{\alpha}$. The heuristic $h_1(n)$ and $h_2(n)$ used are both the distance of the straight line connecting any node $n$ to the target. Additionally, node $i$ is the current node, $j$ is the father node to the current node, $L$ and $l$ are the euclidean distance from the starting node and current node to the target point respectfully, $\alpha$ and $\theta$ are their respective angles, and $\omega_1$ and $\omega_2$ are weights chosen deliberately. The results of this paper show that all cost functions well outperformed depth-first search, regardless of the heuristic function used. Additionally, they found that $f_2(n)$ was more efficient than the other two cost functions. They concluded that this was because $f_2(n)$ provided more information than $f_1(n)$, and the problem contained deviations in distance and angle that made $f_3(n)$ not advantageous. Using cost functions with combined heuristics means that technically the heuristic value used for cost assessment is not always guaranteed to be admissible. However, assuming all combined heuristic values are themselves admissible, then performance can be greatly improved since the search is provided an incentive to move in the right direction, which is seen in the results here. A noteworthy example of this behavior can be seen in the Weighted A* search algorithm.

### Weighted A* Search

Weighted A$*$ search [4] is a natural extension of A* with the addition of a weight applied to the heuristic function. The cost function of Weighted A* is $f(n) = g(n) + \omega * h(n)$ where $\omega$ is a user-specified weight. Again, let $h(n)$ be an admissible heuristic function. Just like standard A* search, Weighted A* search is guaranteed to find a solution assuming a finite branching factor and strictly positive path costs. However, since the weight applied to the heuristic may make the heuristic guess an overestimate, this algorithm is not guaranteed to find an optimal solution. But even though $\omega * h(n)$ itself may not always be admissible, the relationship between heuristic values remains the same. This allows the heuristic to have a larger effect on the search since all heuristic guesses are themselves admissible. Additionally, it is theorized that the higher the weight applied to the heuristic will correlate to a faster search through the state-space.[8] This is because the heuristic is valued much higher than the cost traveled and the algorithm is more willing to consider nodes that align with paths that are not quite

optimal, but have a more direct path to the solution node.

Since Weighted A* assigns more value to the heuristic, this algorithm is known as a greedy search algorithm. Additionally, like normal A*, Weighted A* belongs to the best-first class of search algorithms in which the algorithm considers and expands nodes that are better, or more specifically, lower cost. When comparing different best-first search algorithms with several benchmarks, Wilt, Thayer, and Ruml [9] found that the best choices were $A_\epsilon*$, which will not be discussed here, and Weighted A*. Additionally, when compared to other types of greedy algorithms with the same benchmarks, the best-first search algorithms outperformed hill-climbing approaches and beam search for problems where the solution cannot be reached from all states.

However, there are cases where an admissible heuristic is actually a bad guess. An example of this is when a node is surrounded by other nodes that are not on solution paths that have lower heuristic values than the node on the solution path. For these bad heuristic guesses, it was proven that Weighted A$*$ with higher weights can fail to perform better than normal A*.[8] In general this occurs when the heuristic function $h(n)$ and the true distance to the solution $d^*(n)$ are not strongly correlated, or the heuristic is not very accurate. When these bad guesses are valued higher, the algorithm considers nodes that are irrelevant to the solution more frequently, and it can lead to a drop in performance instead of an increase. Thus, to see positive effects of the weight on performance, it is important for the heuristic function used for Weighted A* search to be as accurate as possible.

**Improvements to Weighted A***

There have been several recent ideas to improve Weighted A*. First, by using a distance estimate to break ties for Weighted A* [7] the performance can be improved. If there are not many ties present in the state space, this does not help performance, but checking distance does not have a significant negative effect on performance either. Next, in an algorithm proposed by Thayer and Ruml called optimistic search [6], if the search begins with a much higher weight on the heuristic than specified by the user and then "cleans up" the result by lowering the weight, a solution can be found that adheres to the user-specified bound. Finally, applying a random weight to Weighted A* [1] can improve performance because the algorithm does not need the ideal weight. Finding the ideal weight to make a search run as fast as possible is difficult as it depends on many factors that are not inherently observable. The Random Weight A* algorithm randomly adjusts the weight used on the heuristic during runtime so that the weight can sometimes be ideal, or at least something close, leading to momentary boosts in performance. While these are indeed novel ideas for increasing performance, since they all introduce additional complexity into the pathfinding problem, they will not be considered here any further.

**Background Summary**

The A* search algorithm and the effect of using different heuristic functions on performance [2] can have drastic effects on the performance of a search. The heuristics that supply A* with more relevant information while remaining admissible perform much better than those that do not.[3] Heuristics that are nearly-optimal when used in combination

with Weighted A* search [4] can provide developers with a way to speed up a search through a graph without significantly increasing the complexity of the algorithm used to search the state-space. In most cases, simply applying a higher heuristic weight will lead to an increase in performance at the expense of a sub-optimal path. However, it is important that the heuristic function correlates with the direct distance of traveling to the solution from any node $n$. If it is not, then performance may actually be reduced due to the heuristic steering the search in directions that are not in solution paths.[8]

# Approach

To solve this problem, software was written in Python to search a map of Minneapolis using Weighted A* with many different weights.[5] The program consists of four different parts: data organization, searching, heuristic functions, and data output and visualization.

### Data Organization

Before the map can be searched, the data must first be imported and organized in a way that is usable for Weighted A* search. The map data of Minneapolis used for the search comes from a CSV file and is organized in 5 columns: if a street is one or two way, starting x position, starting y position, ending x position, and ending y position. As is, this data is unusable since the program does not know which points are connected. In the implementation created here, the program iterates through every line and stores the information into a Python dictionary that maps nodes in the form of $(x, y)$ position tuples to a list of $(x, y)$ position tuples that the node is connected to. It is also important to note that for simplicity in this problem, all streets are treated as a two-way streets, and it is assumed that the speed limit for all of the roads are the same. The only aspect of the roads that are considered in this problem is the length of each road which is treated as the path cost of the edge connecting one node to another.

### Weighted A* Search

After the data is organized, Weighted A* must search the problem state-space for the solution node from a start node that are both specified by the user. In addition to the start and end points, the user also must specify a heuristic function to use for the search, and a weight to be applied to the heuristic. The specific implementation of Weighted A* has not been altered significantly from normal. For clarity, the pseudocode in Figure 1 shows a basic implementation of Weighted A*.

First, the frontier is created in the form of a priority queue ordered by the Weighted A* cost function $f(n) = g(n) + \omega * h(n)$, and the start node is inserted at the front. Also, an empty dictionary is created to keep track of reached nodes and their path cost from the start node. One difference with this implementation, however, is that a dictionary containing every node in the graph mapping to an empty list is created to keep track of the path taken to each node. Next, while the frontier is not empty, the node with the lowest cost function value is popped. If it is a solution, it is returned along with the number of loop iterations and a list of the path taken to reach the node. Otherwise, its children are inserted into the frontier as long as they have not been reached or the path cost from the parent node is smaller than previously calculated.

```
def weighted_astar_search(problem,start,solution,hn,weight=1):
    i = 0
    frontier = PriorityQueue(fn)
    frontier.put(start)
    reached = {}
    reached[start] = 0
    while not frontier.empty():
        i += 1
        node = frontier.get()
        if node == solution: return (node)
        for child in problem.nodes[node]:
            path_cost = reached[node]+dist(node,child)
            if child not in reached or path_cost < reached[child]:
                reached[child] = path_cost
                frontier.put(child)
    return None
```

**Figure 1: Pseudocode for Weighted A***

### Heuristic Functions

Two separate heuristics were used to compare the effect of the weight on the performance of the algorithm, one admissible and one inadmissible. The admissible heuristic function used for the searches is the length of the direct path from any node $n$ to the goal node, ignoring all roads. More specifically, for a child node $(x_1, y_1)$ and an solution node $(x_2, y_2)$, the heuristic value $h_1(n)$ is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. This heuristic function is guaranteed to be admissible because the length of the direct path between two points will always be shorter than the actual cost of the path to the goal when constrained by the roads. The inadmissible heuristic is the sum of the distance along the x axis and the y axis from the start point to the end point. This function is inadmissible because the sum of the sides of a box bounding two points will almost always be longer than the distance traveled between the points inside the box.

### Testing and Data Generation

To run tests and generate data, a start node, solution node, list of weights to test, and heuristic function must be defined in main. The program iterates through every weight in the list and searches the graph using Weighted A*. After each search, the execution time, final path cost and length, and number of iterations to search through the graph for the solution are appended to lists. After all runs have been completed, these lists are written to an output file where they can be compared to the other runs in order to view the effect of the weights on the algorithm. Additionally, all of the generated paths through the graph are visualized on top of the Minneapolis map to better understand the effect of the weight on the generated paths.

## Experiments

All experiments were performed on an Intel i9-10900k CPU running at 3.70 GHz. Four separate tests were performed on the graph. For each heuristic function described above, the graph was searched with two different start and end node pairs using all integer weights between 0 and 150. Pair 1 starts at the intersection of Osseo Rd and Penn Ave N and ends at the intersection of E 54th St

and Hwy 55. Pair 2 starts at the intersection of NE Stinson Blvd and 37th Ave NE, and ends at the intersection of Xerxes Ave S and Hwy 62. Although it was important that the pairs of start and end points were far away from each other for effective data generation, these pairs were otherwise chosen arbitrarily. The map of Minneapolis being searched with visualized start and end pairs and intersection names is shown in Figure 2.

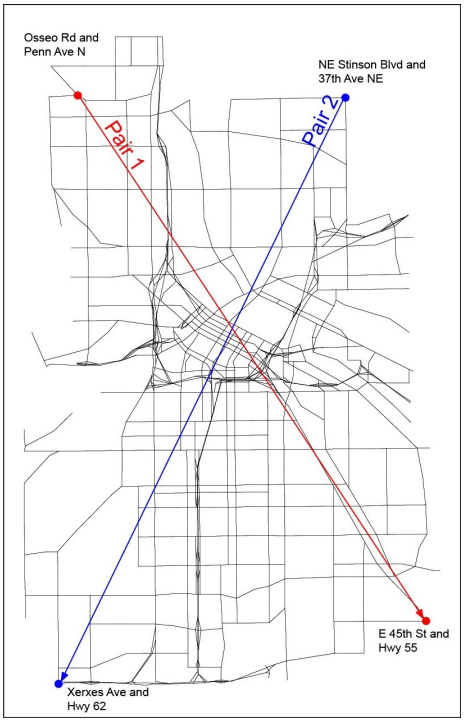


**Figure 2: A graphical representation of the map of Minneapolis being searched.**

## Results

The first two tests performed were searching start/solution pair 1 using the two different heuristic functions. Figure 3 shows the resulting paths computed for varying the weight applied to each heuristic function.

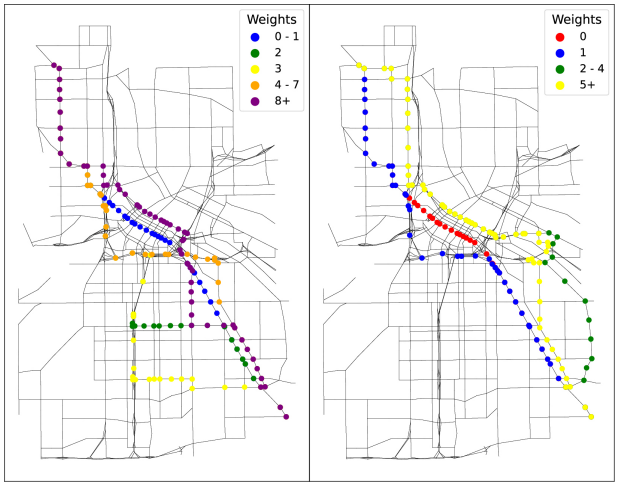


**Figure 3: Pathfinding over start/solution pair 1 with varying weights using the admissible heuristic (left) and the inadmissible heuristic (right).**

Tables 1 and 2 show the first 10 rows of execution data output by searching the map with start/solution pair 1 and both heuristic functions. Weights 11 through 150 are omitted because the path lengths do not change for either heuristic with applied weights over 10.

| Weight | Cost | Length | Iterations | Time($\mu$s) |
|---|---|---|---|---|
| 1 | 5546.716 | 53 | 137 | 1111.5 |
| 2 | 6548.979 | 49 | 51 | 505.7 |
| 3 | 6872.111 | 46 | 47 | 468 |
| 4 | 6255.793 | 47 | 48 | 470.1 |
| 5 | 6255.793 | 47 | 48 | 469.3 |
| 6 | 6255.793 | 47 | 48 | 473.9 |
| 7 | 6255.793 | 47 | 48 | 466.8 |
| 8 | 6199.458 | 58 | 58 | 548 |
| 9 | 6199.458 | 58 | 58 | 566.5 |
| 10 | 6199.458 | 58 | 58 | 542.3 |

**Table 1: The execution data for pair 1 and the admissible heuristic.**

| Weight | Cost | Length | Iterations | Time($\mu$s) |
|---|---|---|---|---|
| 1 | 5899.799 | 45 | 47 | 441 |
| 2 | 6770.97 | 54 | 80 | 598.4 |
| 3 | 6770.97 | 54 | 83 | 607.7 |
| 4 | 6770.97 | 54 | 87 | 632.5 |
| 5 | 6648.418 | 59 | 77 | 589.2 |
| 6 | 6648.418 | 59 | 77 | 586.5 |
| 7 | 6648.418 | 59 | 77 | 585.9 |
| 8 | 6648.418 | 59 | 77 | 610.8 |
| 9 | 6648.418 | 59 | 73 | 598.8 |
| 10 | 6648.418 | 59 | 73 | 578.6 |

**Table 2: The execution data for pair 1 and the inadmissible heuristic.**

Next, the tests for searching start/solution pair 2 using the two different heuristic functions were performed. The resulting computed paths found by varying the weight applied to each heuristic function are once again shown in Figure 4 below.

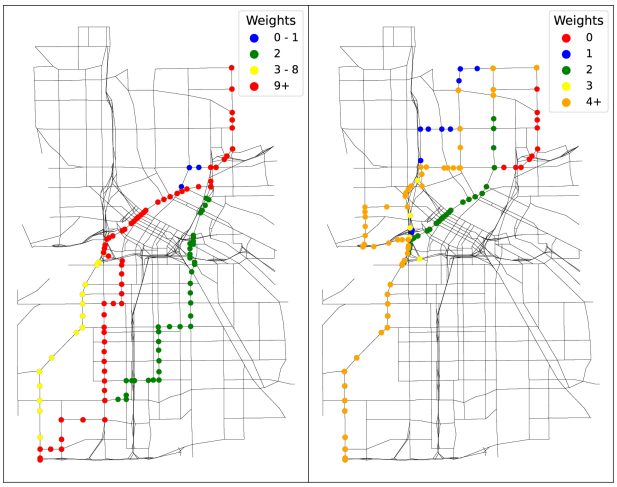


**Figure 4: Pathfinding over start/solution pair 2 with varying weights using the admissible heuristic (left) and the inadmissible heuristic (right).**

Tables 3 and 4 show the first 10 rows of execution data output by searching the map with start/solution pair 2 and both heuristic functions. Once again, weights 11 through 150 are omitted because the path lengths do not change for either heuristic with applied weights over 10.

| Weight | Cost | Length | Iterations | Time($\mu$s) |
|---|---|---|---|---|
| 1 | 6015.343 | 52 | 637 | 3975.4 |
| 2 | 6915.284 | 62 | 63 | 598.6 |
| 3 | 6058.962 | 54 | 54 | 521.1 |
| 4 | 6058.962 | 54 | 55 | 508.2 |
| 5 | 6058.962 | 54 | 55 | 508 |
| 6 | 6058.962 | 54 | 55 | 505.5 |
| 7 | 6058.962 | 54 | 55 | 507.6 |
| 8 | 6058.962 | 54 | 55 | 505.7 |
| 9 | 6891.028 | 63 | 63 | 617.7 |
| 10 | 6891.028 | 63 | 63 | 583.1 |

**Table 3: The execution data for pair 2 and the admissible heuristic.**

| Weight | Cost | Length | Iterations | Time($\mu$s) |
|---|---|---|---|---|
| 1 | 6525.636 | 41 | 125 | 851.2 |
| 2 | 6136.5 | 48 | 48 | 418.2 |
| 3 | 6804.805 | 38 | 38 | 336.8 |
| 4 | 7856.992 | 55 | 76 | 559.4 |
| 5 | 7856.992 | 55 | 74 | 547.3 |
| 6 | 7856.992 | 55 | 74 | 545.1 |
| 7 | 7856.992 | 55 | 74 | 544.7 |
| 8 | 7856.992 | 55 | 74 | 567.7 |
| 9 | 7856.992 | 55 | 74 | 546.4 |
| 10 | 7856.992 | 55 | 74 | 545.7 |

**Table 4: The execution data for pair 2 and the inadmissible heuristic.**

It may be helpful to note that although every weight in the range was tested every time, the generated maps only contain a few of these weights' paths because many of them are the same as other paths generated by another weight.

# Analysis

## Admissible Heuristic

The results portrayed above clearly show that when using an admissible heuristic, an increase in weight leads to a general decrease in execution time when compared to normal A* search. Figure 5 below shows a graph of the execution times versus weight for both tests with the admissible heuristic function.

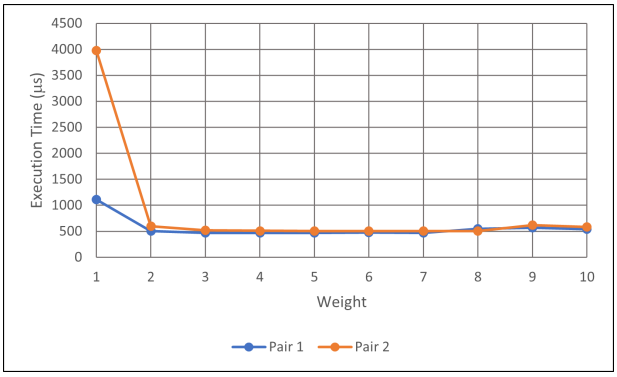


**Figure 5: The execution times versus weight for searching through paths 1 and 2 with the admissible heuristic.**

Here, it is clear that the time spent searching for a path between the pairs is greatly reduced for any weight greater than 1. Now, consider the effect of the weight on the cost of each solution path. Figure 6 shows a graph of path cost versus weight for both start/solution pairs and the admissible heuristic.

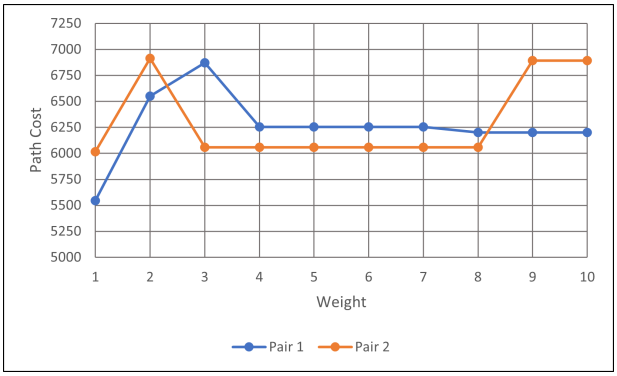


**Figure 6: The path cost versus weight for searching through paths 1 and 2 with the admissible heuristic.**

Theoretically, one would expect that as weight increases, so does the path cost of the computed solution. However, looking at Figure 6, this does not necessarily seem to be the case. For both pairs, the path costs for searching with weights greater than 1 are indeed more than with a weight of 1, but the costs do not correlate with weight. For example, the path cost for pair 1 peaks with a weight of 3 before coming down to about 6250 for weights 4 through 7 and again to about 6200 for any weight above 7. Finally, consider the effect of weight on the path length of nodes visited in the solution. This is shown below in Figure 7.

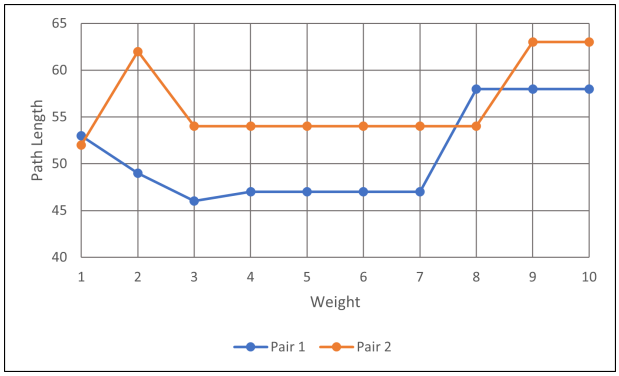


**Figure 7: The path length versus weight for searching through paths 1 and 2 with the admissible heuristic.**

Once again, the data shows weight does not have the expected theoretical effect on path length. By increasing the weight, the length of the paths should go down due to the weight influencing the search to move in the direction of reducing distance to the solution. However, Figure 7 shows that this is not always what happens. The path lengths of pair 1 follow what is expected until any weight greater than 8 when the lengths are increased to 57, a length higher than the optimal solution. Additionally, the path lengths generated with pair 2 and weights higher than 1 fluctuate, but are all longer than the optimal path.

The effect of increasing weight on the path cost and lengths is an interesting result because it implies that not only is the performance increase dependent on the weight applied to the heuristic function, but also the graph being searched as well.

### Inadmissible Heuristic

When using the inadmissible heuristic function, both the cost and length of each computed path follows what is generally expected, with some fluctuation. That is, path cost goes up and path length goes down. However, the execution times for these tests is notable and shown below in Figure 8.

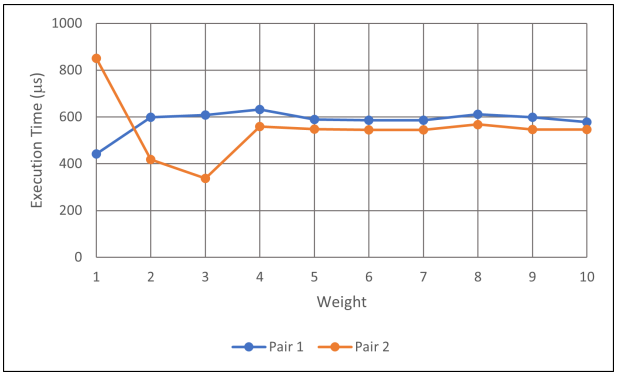


**Figure 8: Execution times of the searches using the inadmissible heuristic for pairs 1 and 2.**

Here, it is clear that increasing the weight on an admissible heuristic does not always have the effect of decreasing the execution time of a search. More accurately, depending on where the search begins and ends, the time spent searching may actually increase as seen from the execution times of path 1 in Figure 8. This is likely due to how the cost function makes decisions because of the information provided by the heuristic.

Using inadmissible heuristics means that the guess of the cost to the solution is not accurate. Figure 9 highlights the effect of inaccurate guesses by searching with an inadmissible heuristic with start/solution pair 2. A weight of 0 is included because it removes the heuristic from the cost function, turning the algorithm into uniform-cost and generating the optimal path.

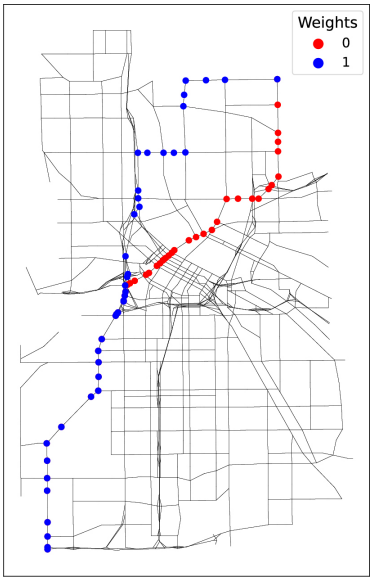


**Figure 9: Pathfinding on start/solution pair 2 with the inadmissible heuristic and weights 0 and 1.**

In Figure 9, the inadmissible heuristic deviates from the optimal path right away. This is because the heuristic values paths that minimize the distance to the solution in one single direction, x or y. Thus, it attempts to take the left path since it is longer. Additionally, it continues along the path it does because it contains more longer roads than the optimal path, which generally moves in a more diagonal path.

## Conclusion

The effect of weight on the performance of searching a state-space while using Weighted A* search has been examined in this study. As the weight applied to the heuristic function increases, the time the algorithm takes to complete a search is noticeably reduced. Additionally, although the final path costs and lengths of solutions with weights higher than 1 are ultimately dependent on the graph and the start/solution pair, some reductions

in path length and inflation of path costs were observed as expected. These results suggest that there is no one weight that will provide performance increases that benefit all start/solution node pairs, but that there are some benefits across all weights when comparing execution data from each search using Weighted A* to that of standard A*. However, these results only apply to heuristic functions that are admissible and do not provide inaccurate cost estimates that are higher than the actual cost.

When using an inadmissible heuristic function, path costs and lengths were for the most part affected by the weight as expected, but search times were often not correlated with increasing the weight on the heuristic function. To be more specific, the execution time was more dependent on the graph and the start/solution pair than increasing the weight. Since Weighted A* search is a satisficing search algorithm in that it intentionally trades path cost for reduced execution time, this solidifies the fact that it is crucial to choose a heuristic function that is admissible to not only avoid decreased performance, but also make the paths found by the algorithm more predictable and closer to the optimal solution.

# References


[1] Abhinav Bhatia, Justin Svegliato, and Shlomo Zilberstein. On the Benefits of Randomly Adjusting Anytime Weighted A∗. In *Proceedings of the Fourteenth International Symposium on Combinatorial Search*, 12(1), pages 116–120, July 2021.

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems, Science, and Cybernetics*, 4(2):100–107, July 1968.

[3] Xiang Liu and Daoxiong Gong. A comparative study of A-star algorithms for search and rescue in perfect maze. In *2011 International Conference on Electric Information and Control Engineering*, pages 24–27, April 2011.

[4] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3-4):193–204, 1970.

[5] Michael Spiese. Csci4511w final project. github.com/michaelspiese/Csci4511w-final-project, December 2022. Github Repository.

[6] Jordan Thayer and Wheeler Ruml. Faster than Weighted A∗: An Optimistic Approach to Bounded Suboptimal Search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 355–362, September 2008.

[7] Jordan Thayer and Wheeler Ruml. Using Distance Estimates in Heuristic Search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 19(1), pages 382–385, September 2009.

[8] Christopher Wilt and Wheeler Ruml. When does Weighted A∗ Fail? In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, 3(1), pages 137–144, July 2012.

[9] Christopher Wilt, Jordan Thayer, and Wheeler Ruml. A Comparison of Greedy Search Algorithms. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 1(1), pages 129–136, August 2010.