Finding the Shortest Path Through Minneapolis

Axel Stenson
stens103@umn.edu

December 22, 2017

Abstract

This project was to implement an algorithm to find the shortest path through real world street data. We provided the search algorithm with a list of paths, and the user is able to enter a start and end node. The search then finds the shortest path through the streets provided. This is a difficult problem to solve - an extended version of this problem, the traveling salesman problem, is a common example of an NP-complete problem. Despite this, there are many algorithms that make this type of search much faster than a simple brute force search. To solve this, this project implemented A* and BFS to find shortest paths through the Minneapolis path data provided by the instructor. I then compared the runtimes of the two results and the difference as the path length increased. This led to the expected result that A* vastly outperforms BFS in runtimes, especially for longer paths. The only paths that BFS equalled A* in runtimes were trivial searches with paths of 0 or 1. The increases in runtime for A* were fairly trivial in comparison, with the longest paths having just two to three times the runtime of the trivial paths where the start and goal were the same.

1 Introduction

It is a common but important problem in AI to use search algorithms to find the shortest path between two points. This is critical in mapping software, used by companies like Waze and Alphabet, in their product Google Maps, to find actual paths through the streets and highways that cover our planet. It is also used in less apparent ways, by Facebook to find friend suggestions and in searching, in parsing context free grammars [**?**], and in games.

This is worth studying because the increase in pathfinding performance has been astounding. "Algorithms for route planning in transportation networks have recently undergone a rapid development, leading to methods that are up to three million times faster" [8]. This improvement in these algorithms have made complicated applications much more feasible and accessible to anyone.

Real world pathfinding is a great way to study these search algorithms and the speed increases that the state of art algorithms provide since this is one of the most obvious ways that many of us utilize these searches in everyday life. Google Maps, for example, has over one billion monthly users [10].

To accomplish the search, I built a Path data structure to store the requisite values for A* search. I used this path data structure as the "nodes" in each search algorithm to keep as much uniformity as possible between overhead work. This was mainly done so I could present the differences in search time as purely as possible, but the fact that BFS has lower overhead are attributes for those search types, so a streamlined implementation of BFS may have better performance than

what is presented here. I am not interested in this as a optimization coding exercise though, but purely as a test of algorithm speed so in this experiment I feel that this approach provided the best comparisons.

2 Related Work and Other Algorithm Options

2.1 Why A*?

A* is very widely used, and over performs in comparison to all other algorithms with a similar level of ease of implementation.

A comparative study of A-star algorithms for search and rescue in perfect maze [5] argues for the use of A* for pathfinding in real world situations. So, at least in 2011 when the paper was written, A* was being used a state-of-art pathfinding algorithm in real-world situations.

In *Investigation of the*(star) search algorithms: Characteristics, methods and approaches* [12], the authors lay out many different versions of A* that use different data structures to modify the search for more optimality in specific instances. One of the most widely used of these modifications is that of D*, which allows the heuristic value of each node to change dynamically during search. This is especially helpful in situations where the map is changing and will not help in our case, unless some data like traffic was introduced to the map that could change dynamically during the search.

Path Planning for Virtual Human Motion Using Improved A Star Algorithm* [13] presents a modified version A* for similar purposes, but uses what they call a "weighted processing of evaluation function," which implies the use of minimax algorithms to help find sections of the map that are no longer worth searching. The nature of pathfinding on roads makes this approach not feasible since there may be sections of the map where many paths are dead ends or have too long of a path cost, but there is a "short cut" that leads to the shortest path.

Optimal Reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning [3] gives a large increase in speed for certain types of graphs. This algorithm severs the arcs of a "target" node and attempt to reinsert it into a Directed Acyclic Graph with a new computed shortest path. This, of course, won't work for our problem because the map is not a DAG.

A* then, is a great choice for a easy to implement algorithm that has a vastly superior performance when compared to simple searches like BFS.

2.2 How We Measure Success

In our analysis we will focus on time complexity. This is interesting for these two algorithms in particular, because in theory both have the same time complexity:

$$O(b^d)$$

Where b is the branching factor and d is the depth of goal state. In practice though, because we use an efficient search heuristic to inform our search and our search is:

$$O(nlogn)$$

Where n is the number of nodes. Shown together, it obvious that this means that A* will massively outperform BFS as the path size grows. The growth of these two complexities is shown in Figure 1.

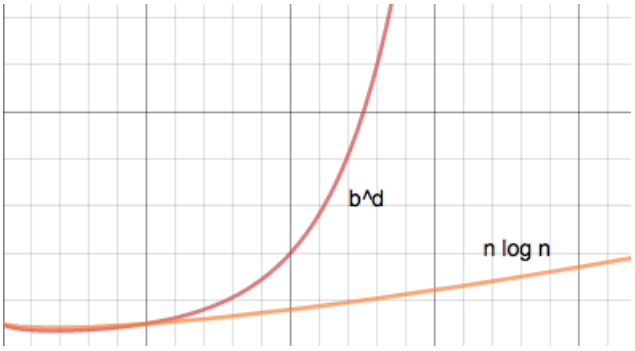


Figure 1: Log and exponential runtimes

This demonstrates the importance of a good search heuristic when searching. I will talk about the heuristic I used in Section 3.

I've chosen not to focus on Space Complexity for three reasons. First, this is much harder to test because space complexity is much harder to define. Second, algorithm quality is most commonly measured by speed. Space complexity has become less important except in the case of errors [7]. Third, A* and BFS both have the same space complexity:

$$O(n)$$

Which makes it much less interesting to compare them.

3 Implementation

3.1 BFS

Breadth First Search, or BFS, is a search algorithm which uses a first-in-first-out queue (FIFO). The process of the algorithm is fairly simple. It starts at the start node and adds all successors to the queue. Then until the queue is empty or the goal is found, it removes the first and adds all their successors to the queue. Since the algorithm uses a FIFO, the successors of the first node will all be searched before any successors to successor nodes are searched. In this way, the search spans the "breadth" of the graph.

The following code was used in our project to run BFS on the map data:

```
def bfs_search(graph, start, goal):
        q = Q.Queue()
        q.put(start)

        while not q.empty():
                temp = q.get()
                successors = find_succ(graph, temp)
                for s in successors:
```

```
                    for t in temp.path:
                        if (s.start_node == t.start_node):
                            successors.remove(s)
                            break

            for next in successors:
                    if (next.end_node == goal.start_node):
                        next.path.append(next)
                        return next.path
                    else:
                        next.path = temp.path
                        next.path.append(next)
                        q.put(next)
    return []
```

3.2 A*

A* really is just a modified version of BFS, this is why their big-O complexities are the same. The difference, and how A* gets increased performance, is that it uses a priority queue instead of a FIFO queue. The priority level is set by the function:

$$f(n) = g(n) + h(n)$$

Where g(n) will be the path distance so far, and h(n) will be calculated based on the search heuristic used. For my search, I used the direct cartesian distance between two nodes. I used this because it will always underestimate the distance, since it is impossible to traverse the graph in a path faster than a straight line. The algorithm will then assign the current node to the node with the lowest f(n) value and recalculate f and g based on that path taken.

The following code is what was used in our project to run A* search:

```
def a_star_search(pathList, start_path, goal_path):

    pq = Q.PriorityQueue()
    close = set()

    start_path.g_value = 0
    start_path.f_value = start_path.h_value
    pq.put((start_path.f_value, random(), start_path))

    while not pq.empty():
        temp = pq.get()
        top = temp[2]
        if is_goal(goal_path, top):
            top.path.append(top)
            print ("DONE?")
```

```
                return top.path
        close.add(top)
        for next in find_succ(pathList, top):
                skip = 0
                for i in close:
                        if (next.start_node == i.start_node):
                                skip = 1
                if skip == 0:
                        next.g_value = top.g_value + top.distance
                        next.f_value = next.g_value + next.h_value
                        for i in top.path:
                                next.path.append(i)
                        next.path.append(top)
                        pq.put((next.f_value, random(), next))

return []
```

4 Analysis

To analyze the search speed of these two algorithms I ran them 100 times with 10 randomized start and goal states. I implemented an algorithm that allowed for a user to enter four numbers and would find the closest nodes to those numbers as x,y values. In Table **??** you can see the runtimes, in seconds, of the two search algorithms on the same start and goal states.

Table 1: Runtime(s) of A* and BFS with varying path lengths.

	0	1	8	10	14	21	24	25	26	27
A*	0.112	0.113	0.119	0.117	0.121	0.188	0.161	0.166	0.166	0.228
BFS	0.113	0.116	0.217	0.172	0.553	5.956	4.573	6.156	4.522	39.439

As you can see, the A* algorithm outperforms BFS in all but the trivial search instances. In Figure 2 I plotted these runtimes against each other, leaving off the highest runtimes of BFS for better detail visibility.

To show how the performance of BFS degrades compared to A* I found difference between the two run times, expressed as a the ratio between A* search time and BFS search time and graphed this in Figure 3.

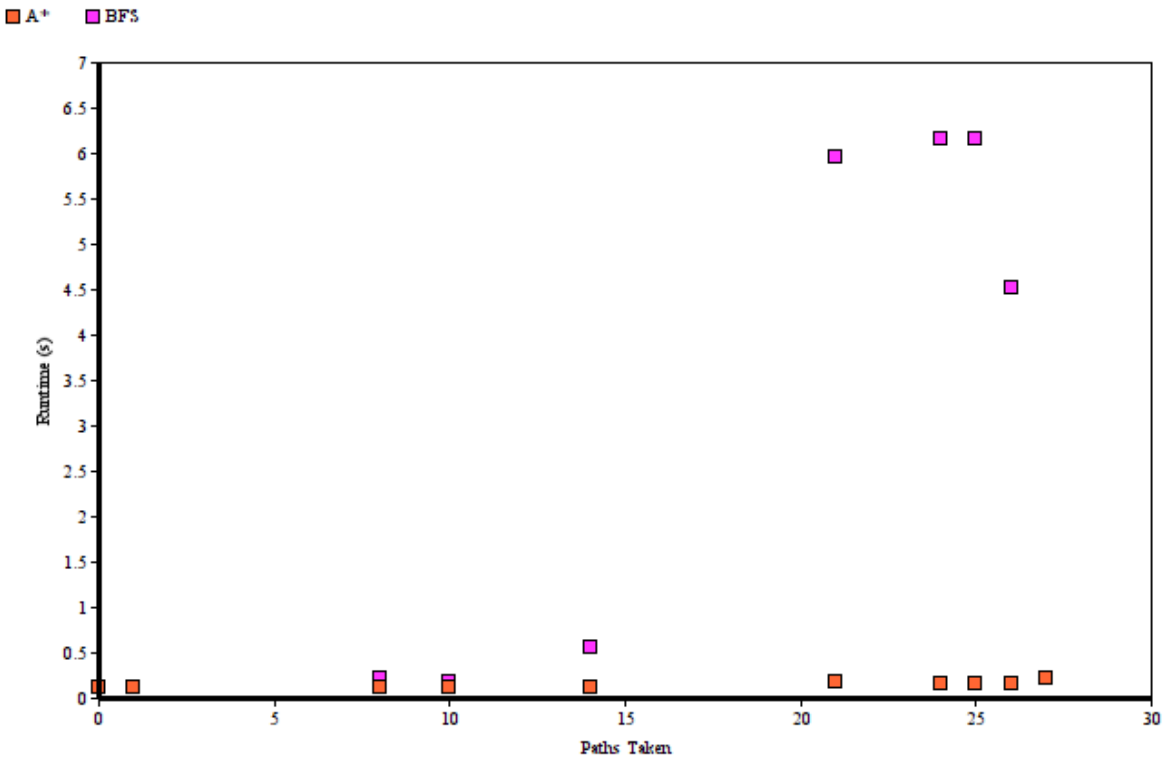


Figure 2: Runtimes (s) of A* and BFS

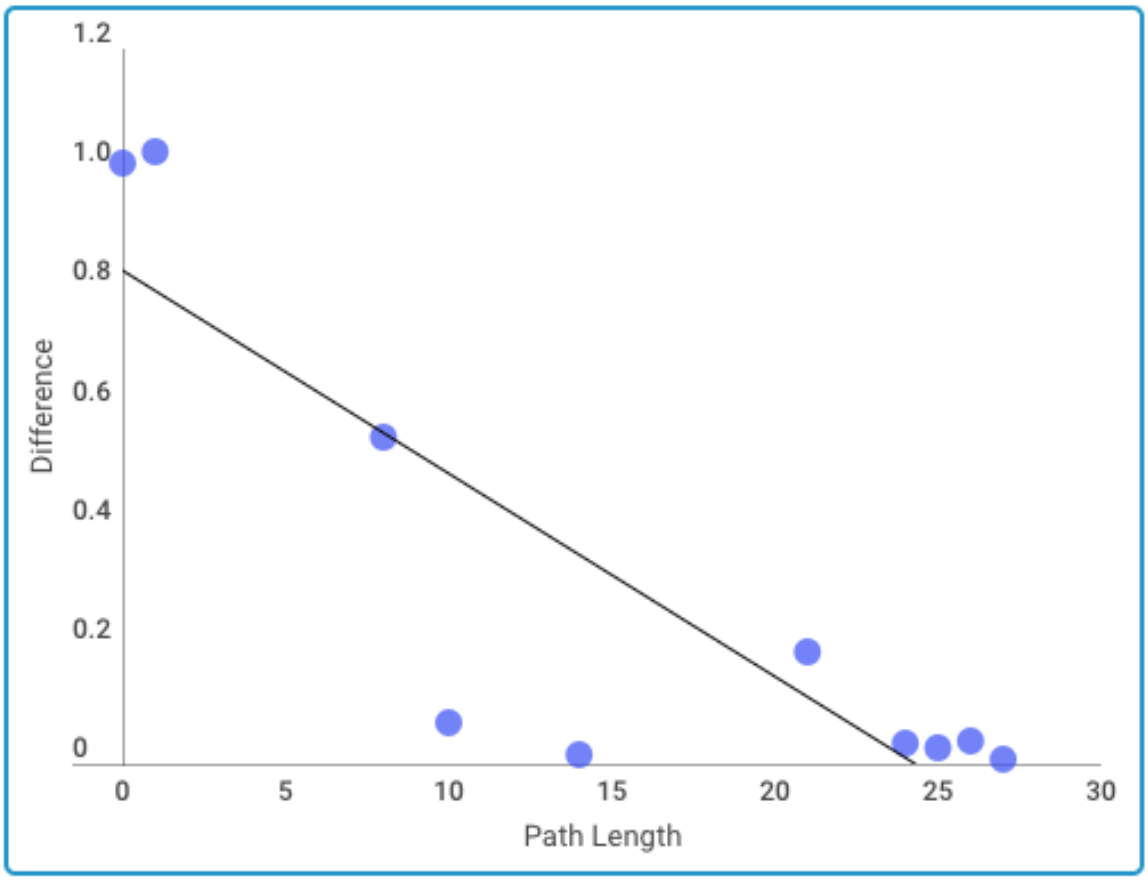


Figure 3: Difference in performance of BFS as compared A*

We can see that there is a steady degradation of the quality of BFS as compared to A* but to prove that there is a direct negative relationship between the two, we can find the correlation coefficient using the equation:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}}$$

$$r = -0.8594$$

This shows that there is a strong negative correlation between in the difference between the runtimes of A* and BFS and is strong mathematical evidence that A* is a much more useful search algorithm for pathfinding.

Mapping the runtimes of BFS and A* onto the data, we can also verify the strength out our heuristic. Figure 4 shows that the data very closely matches the expected result for an A* search with a strong heuristic, of:

$$O(nlogn)$$

While BFS closely resembles it's O time complexity of:

$$O(b^d)$$

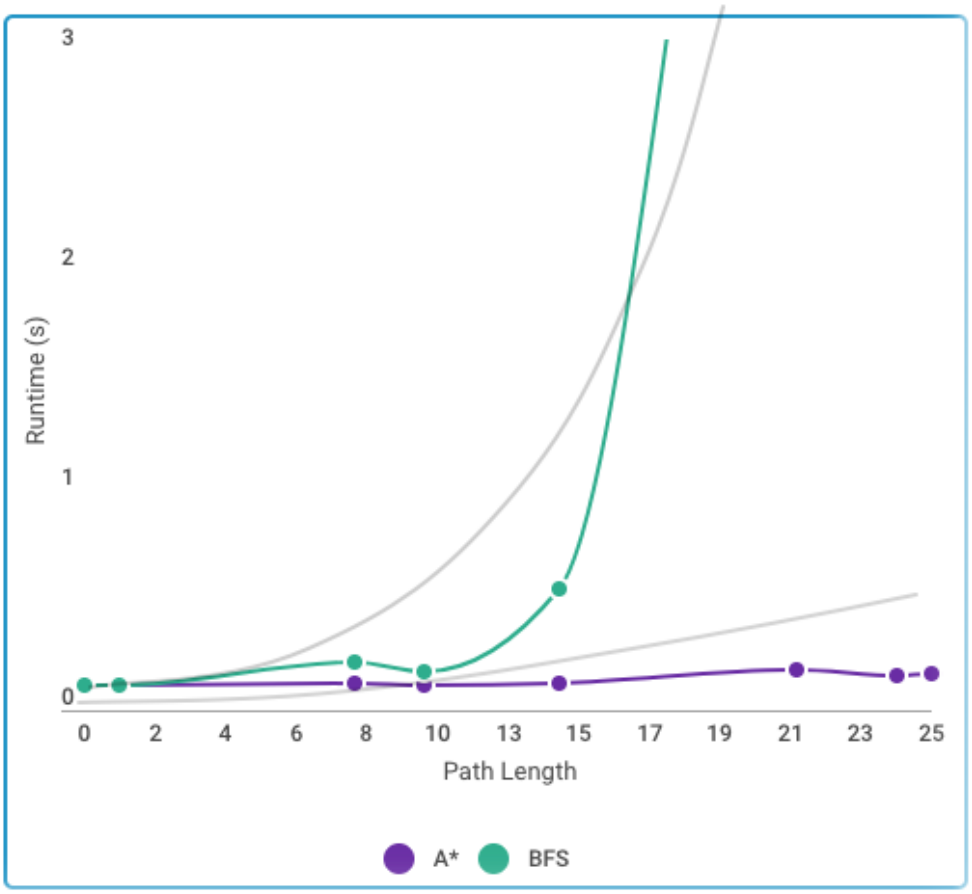
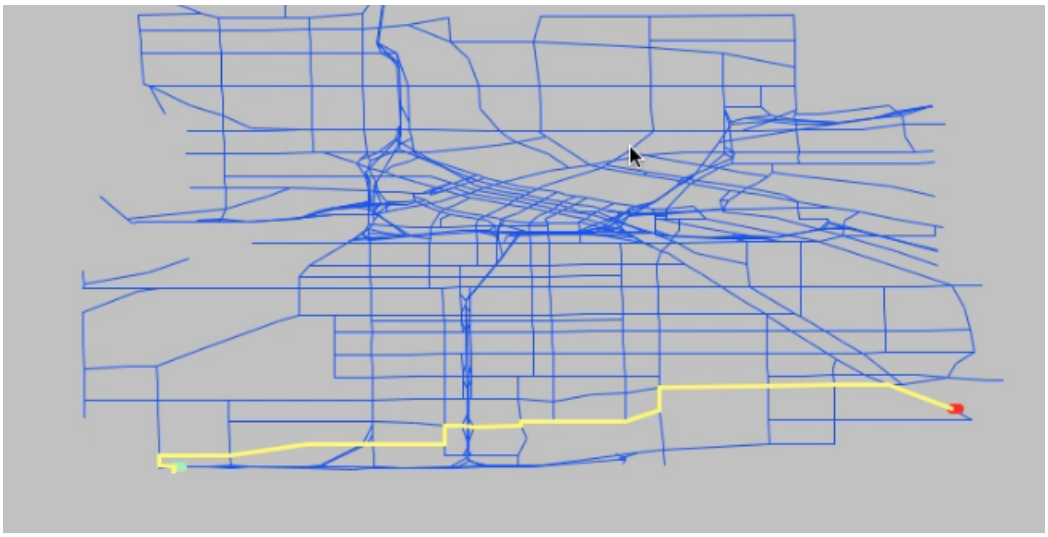


Figure 4: Expected O-time complexity of algorithms against results.

5 Conclusion

The A* search algorithm not only performed better than BFS, which I expected, it performed better than I could have expected. Beating the O-time of A* substantially. BFS was no longer useable after a path of 25 nodes and as such it is not suitable for mapping in a real world scenario. The heuristic we chose performed very well in most instances although I did see evidence of backtracking in some paths, which leads me to believe it needs to be optimized a bit for a real world search scenario.

During the coding process I created a graphical representation of the paths that were found through Minneapolis to help with Debugging. They provided me a concrete demonstration of the usefulness of these search algorithms, so I think it makes sense to include them here. This is approximately the path from Edina to Minnehaha Falls.



References

[1] Bryan Stout. *Smart Moves: Intelligent Pathfinding.* Game Developer, July 1997.

[2] Ajit P. Singh, and Andrew W. Moore. *Finding optimal Bayesian networks by dynamic programming.* (2005).

[3] Andrew Moore and Weng-Keen Wong. *Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning.* ICML. Vol. 3. 2003.

[4] Richard E. Korf, et al. *Frontier search.* Journal of the ACM (JACM) 52.5 (2005): 715-748.

[5] Xiang Liu, and Daoxiong Gong. *A comparative study of A-star algorithms for search and rescue in perfect maze.* Electric Information and Control Engineering (ICEICE), 2011 International Conference on. IEEE, 2011.

[6] Frantisek Duchon, Andrej Babinec, Martin Kajan, Peter Beno, Martin Florek, Tomas Fico, and Ladislav Jurisica. *Path Planning with Modified a Star Algorithm for a Mobile Robot.* Procedia Engineering, page 59-69, Vol. 96, 2014.

[7] Ernst Leiss *Programmer's Companion to Algorithm Analysis* Chapman and Hall CRC, 2007

[8] Sanders P., Schultes D., and Wagner D. *Engineering Route Planning Algorithms.* In: Lerner J., Wagner D., Zweig K.A. Algorithmics of Large and Complex Networks. Lecture Notes in Computer Science, vol 5515.

[9] Ioannis Tsamardinos, Laura E. Brown, and Constantin F. Aliferis. *The max-min hill-climbing Bayesian network structure learning algorithm.* Machine learning 65.1 (2006): 3

[10] Ludovic Privat. *Google Maps: 1 Billion Monthly Users.* `https://www.gpsbusinessnews.com/Google-Maps-1-Billion-Monthly-Users_a4964.html`

[11] Dan Klein and Christopher D. Manning. A parsing: fast exact Viterbi parse selection. Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1. Association for Computational Linguistics, 2003.

[12] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. *Investigation of the*(star) search algorithms: Characteristics, methods and approaches.* World Applied Programming 2.4 (2012): 251-256.

[13] Junfeng Yao, et al. *Path planning for virtual human motion using improved A* star algorithm.* Information Technology: New Generations (ITNG), 2010 Seventh International Conference on. IEEE, 2010.

[14] Yichao Zhou, and Jianyang Zeng. *Massively Parallel A* Search on a GPU.* AAAI. 2015.