David Mead

October 2018

**Search Type Length Time**

FIEBDA0CONKGHLMJ: (35)

A-STAR 35 2.37084

A-STAR RANDOM 35 10.69509

A-STAR RANDOM 35 9.65249

A-STAR RANDOM 35 8.90101

GCBFAEHKDI0JLNMO: (36)

A-STAR 36 15.48136

A-STAR RANDOM 36 43.50388

A-STAR RANDOM 36 49.44474

A-STAR RANDOM 36 47.68785

FIBEALDKJCNGHM0O: (37)

A-STAR 37 4.39186

A-STAR RANDOM 37 6.25867

A-STAR RANDOM 37 7.53885

A-STAR RANDOM 37 6.96895

IBKFGACNDOMJLHE0: (38)

A-STAR 38 0.41671

A-STAR RANDOM 38 2.39986

A-STAR RANDOM 38 2.33582

A-STAR RANDOM 38 2.34974

BKNCAIEJD0FMLHGO: (39)

A-STAR 39 16.41959

A-STAR RANDOM 39 23.58052

A-STAR RANDOM 39 34.12554

A-STAR RANDOM 39 35.38502

MFKCAEOGBH0IDLNJ: (40)

A-STAR 40 10.57953

A-STAR RANDOM 40 48.38795

A-STAR RANDOM 40 41.00462

A-STAR RANDOM 40 33.34451

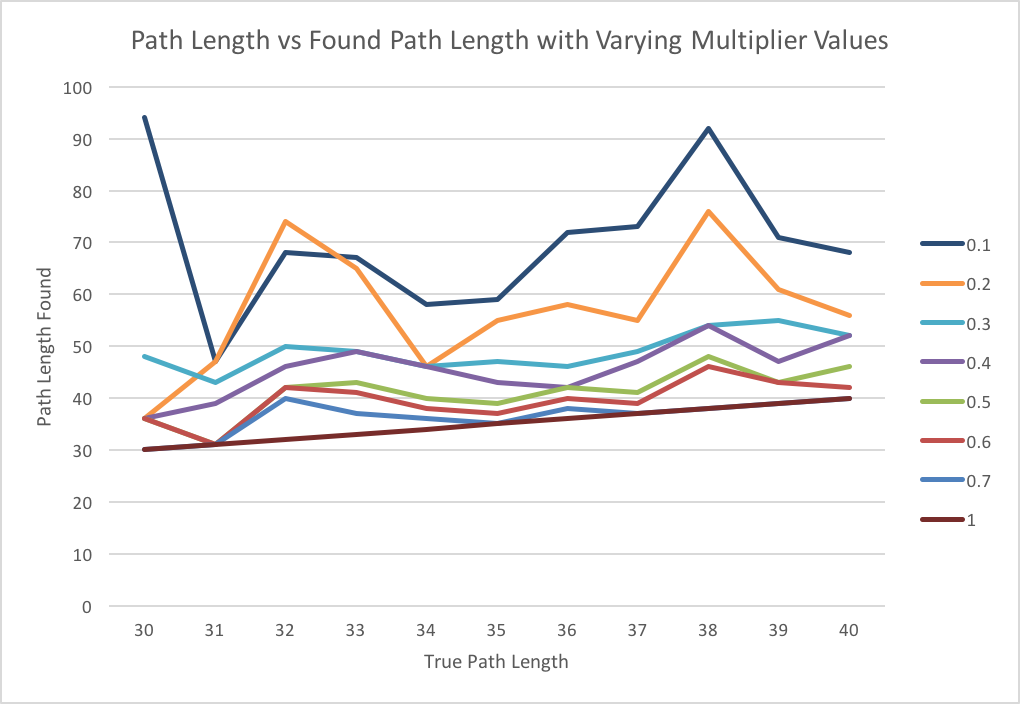
Period 2

Exploration 1: Tie Breaker

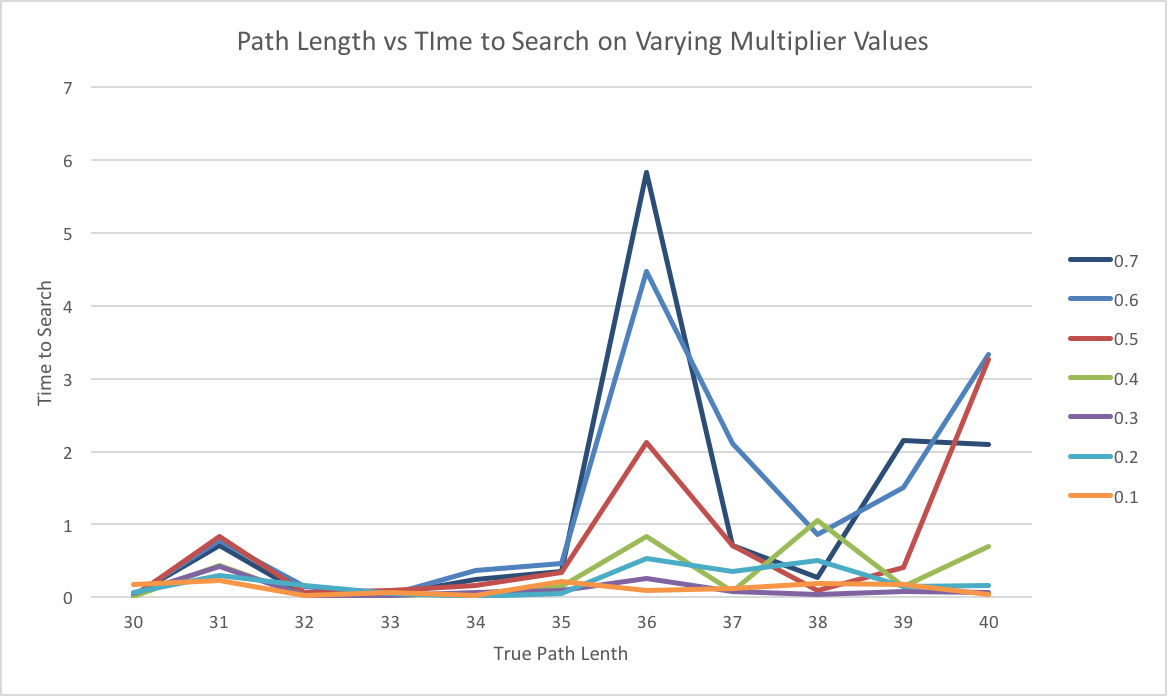
To the left you’ll see the output for my A\* with tie breaker method searching on states with path lengths 35-40. The first output (“A-STAR”) was done by a regular A\* search, and the next 3 (“A-STAR Random”) were done using A\* with the random tie breaker functionality. I used a simple random.randint(0, 1000) to get the random number.

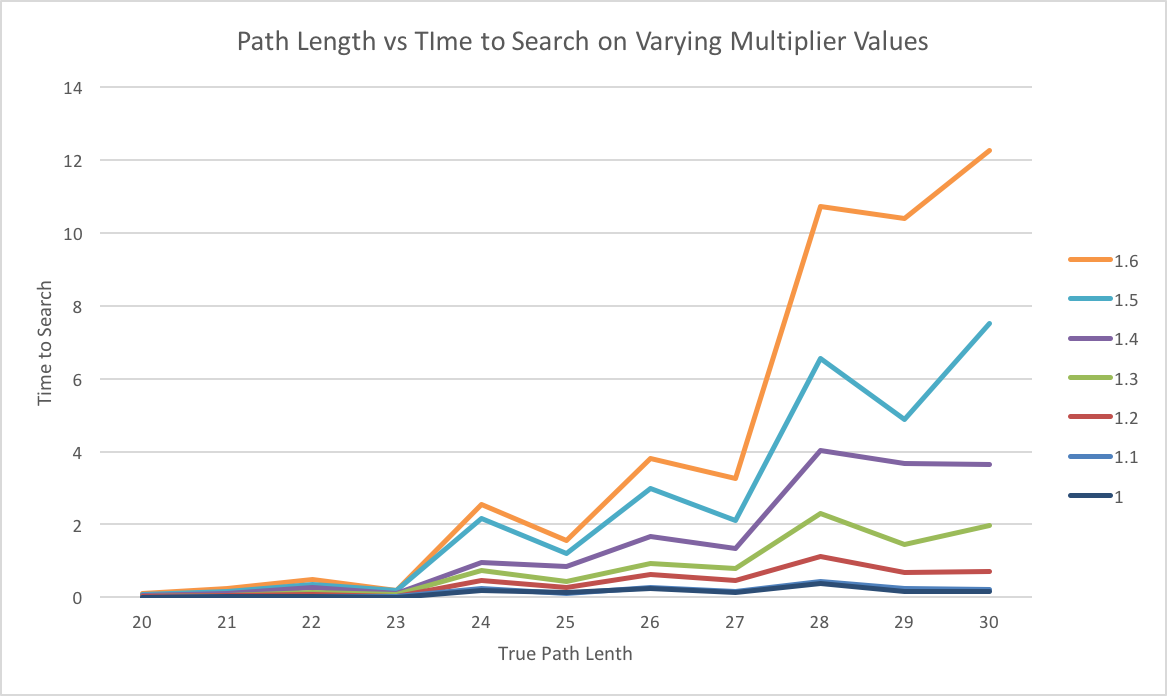
For these relatively low length solutions, the tie breaker A\* was able to maintain the same accuracy as the regular A\*, but was significantly slower (anywhere from 2-5 times slower). While a lower runtime is unfortunate, the tie breaker functionality useful because in the event that a tie is found in the heuristic value, it randomly decides the path to go down rather than choosing based on whichever thing would have been in the second place of the tuple, such as the state string, which could throw off results.

Exploration 2: Multiplier

 This graph shows the change in the path length found by the search compared to the real path length. The different colored lines are each an A\* with a different multiplier value from .7 to .1. The bottom red line is just to show what the real path would be. These states were just the states length 30-40 from 15puzzles.txt.

The graph clearly shows that as we get further from a multiplier of 1, the path length found gets further from the true path length. Unfortunately, the loss of accuracy isn’t consistent enough for us to simply calculate the true path length from the estimated path length.

 This next graph plots the time it took to find the estimated path based on the real path length. In this graph we see how the time erratically increases as the true path length increases, and that the super low multiplier values are constantly speedy, whereas the larger/closer to 1 multiplier values take more time.

 Both these graphs combine to show us the relationship between time and accuracy when estimating the path length. We could extrapolate and say that if we used a multiplier value of .0001, we could get *a* path length almost instantaneously, but it would be way off from the real length.

Once we get to and go past a multiplier value of 1, all the path lengths are accurate, so I omitted a graph for true path length/estimated path length for multiplier values above 1.

In the third graph, we see that as we get further from 1 in the positive direction, the time increases for each search. This makes sense because we are diminishing the advantage of the of taxi-cab distance and focusing much more on only the depth of the nodes, which is not an indicator of whether a state is any closer to solved or not, so we end up going deep into far more incorrect paths than needed, which takes more time.

I ran the A\* with a multiplier of .7 (and the random tie breaker) on a state with a path length of 50 10 times. The output is on the left. As you can see, the estimated path length was either 52 or 50, and it found this in about 5 seconds each time. While the multiplier A\* will not always guarantee the accuracy of the path length found, when we run it multiple times using a tie breaker, we get a range of results, the lowest of which will be close to the true path.

0KBJFECLINMGHAOD: (0)

        A-STAR 1        52      7.55215

        A-STAR 2        50      5.654

        A-STAR 3        50      5.14814

        A-STAR 4        50      5.07027

        A-STAR 5        50      4.97133

        A-STAR 6        52      5.36356

        A-STAR 7        52      4.92116

        A-STAR 8        50      5.7281

        A-STAR 9        50      5.51651

        A-STAR 10     52      5.02532

WITHOUT MULTIPLIER:

A-STAR 1 50 133.64114

If we did not have the tie breaker, we may have got 52 as our estimated path length 10 times, but by sacrificing a few seconds per search, we are able to get a much more accurate path length.

Additionally, I added the output for when I run A\* without the multiplier, which took 133 seconds. In the time it took to find the guaranteed shortest path, we could have gotten around 26 estimates, one of which has a good chance of being correct or close to correct.

|  |  |  |
| --- | --- | --- |
| true length | minimum estimate | difference |
| 39 | 44 | 5 |
| 40 | 43 | 3 |
| 41 | 44 | 3 |
| 42 | 45 | 3 |
| 43 | 46 | 3 |
| 44 | 49 | 5 |
| 45 | 58 | 13 |
| 46 | 51 | 5 |
| 47 | 54 | 7 |
| 48 | 55 | 7 |
| 49 | 56 | 7 |

Exploration 3: korf100

To estimate the path lengths of the first 10 states in the korf100, I first ran A\* with a multiplier of .5 on the states with lengths 39-49 from 15puzzles.txt 5 times each and took the minimum. Since I knew the real path length, I could calculate the difference in my estimate and the real length. I found that as we get up to real lengths of around 50, my estimate is about 7 off, and as the real length rises, the difference continues to rise as well.

After this, I ran the same A\* with a multiplier of .5 on the 10 states from korf100 5 times and collected the results. The minimums found are bolded in the 2nd table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| # | 1 | 2 | 3 | 4 | 5 | **min** |
| 0 | 63 | 63 | 63 | 63 | 63 | **63** |
| 1 | 67 | 67 | 63 | 67 | 65 | **63** |
| 2 | 69 | 71 | 73 | 71 | 71 | **69** |
| 3 | 72 | 70 | 72 | 72 | 72 | **70** |
| 4 | 68 | 60 | 68 | 70 | 60 | **60** |
| 5 | 62 | 60 | 58 | 58 | 62 | **58** |
| 6 | 60 | 58 | 60 | 58 | 58 | **58** |
| 7 | 58 | 58 | 58 | 58 | 60 | **58** |
| 8 | 56 | 58 | 52 | 56 | 52 | **52** |
| 9 | 65 | 67 | 71 | 67 | 67 | **65** |
| 10 | 73 | 65 | 65 | 65 | 73 | **65** |

Since these were in the 60s and 70s and my results that had been in the 50s were about 7 off, I can say with decent confidence that the real lengths were about 7 to 10 less than the estimated values I found. Here’s my estimations:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **Estimate:** | **55** | **55** | **60** | **61** | **53** | **51** | **51** | **51** | **47** | **57** | **57** |

Exploration 4: Nodes Per Second

AFICDB0GEHJOLMKN BFS 19 28.46142 117423

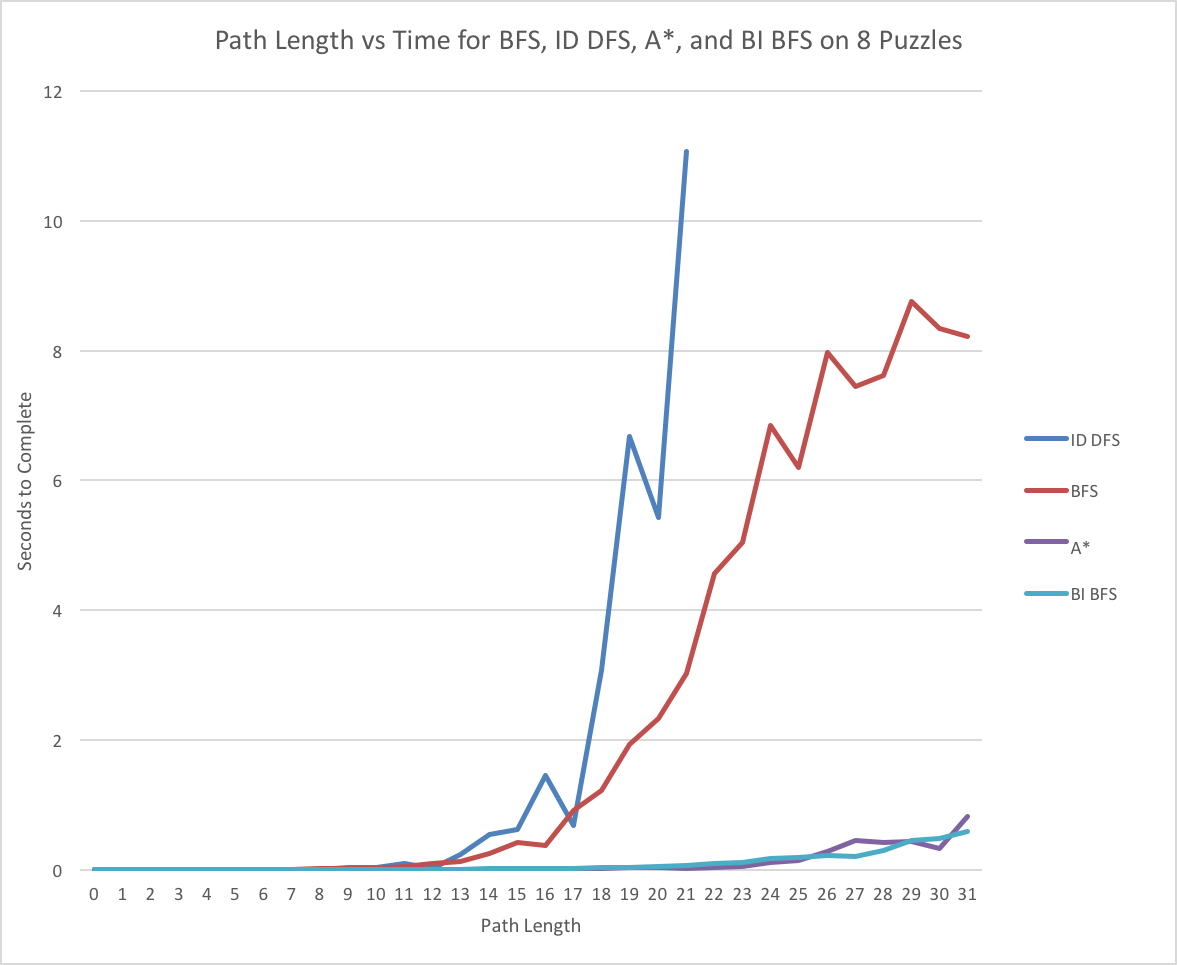
ABKCHDG0IFEJLMNO ID DFS 18 22.86628 125367

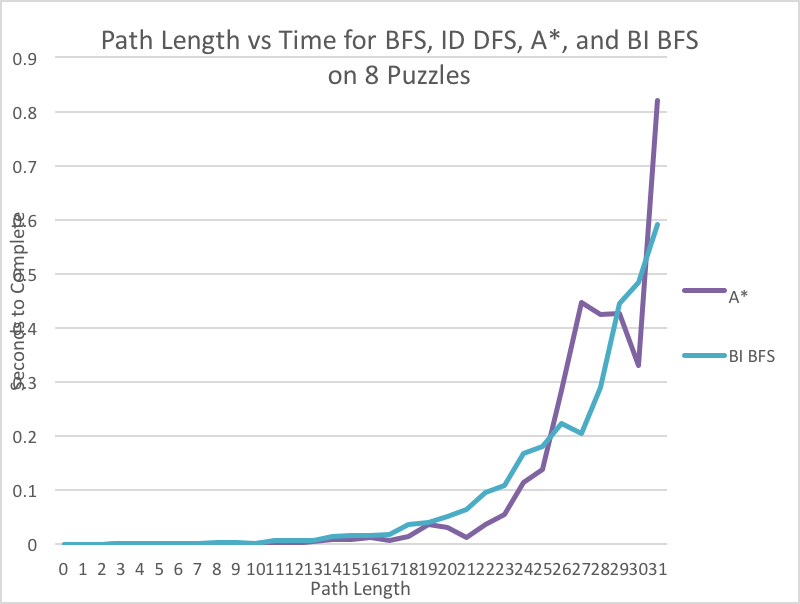
KDEB0AJFLIGNMHCO A-STAR 41 11.23793 25379

FIEBDA0CONKGHLMJ BI BFS 35 11.01297 170297

BFS and Bidirectional BFS were processes nodes at roughly the similar speeds, with Bi-BFS going faster. This makes sense because for each time it checks if it’s at the goal state, it processes 2 nodes instead of just 1 like BFS. Iterative Deepening DFS processed nodes at about the same speed as BFS which is reasonable because while ID DFS takes less memory, both ID DFS and BFS must check about the same number of nodes and take roughly the same time at smaller path lengths. A\* has the slowest nodes per second because for each node, it needs to recalculate the taxicab distance and decide based on that which nodes to go to next.

Exploration 5: 8-Puzzle

~~~~

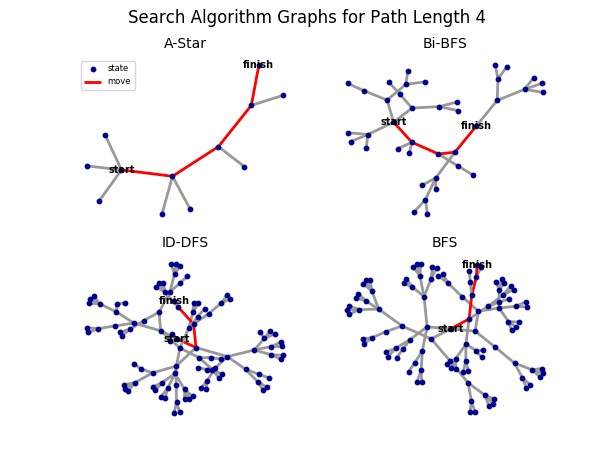
 After I generated 31 paths with incrementally increasing path lengths, I did BFS, ID DFS, A\*, and BI BFS on them, wrote them into an excel file, and plotted them (to the right)  
 I stopped at 20 for ID DFS because it took way to long (1301 seconds for path length 29) and if I kept that in the graph, you wouldn’t be able to see the other lines (more on ID DFS later). Below ID DFS is BFS, which rises exponentially for a bit, and then it appears to cap off. It does not really cap off, as we learned in 15 puzzle though, so this is likely due to a coincidence.

Down at the bottom are A\* and BI BFS which barely increase over the course of the 31 puzzles. At lower length solutions, the different traits of BI BFS (searching way more nodes) and A\* (taking much longer per node) balance each other out, and they run neck and neck through all the states.

When we take a closer look at BI BFS and A\* we see that BI BFS’s times are more predictable, whereas A\*’s are more erratic. This is due to the fact that BI BFS is more methodical and exhaustive in its search, so it’s time to run can be easily related to the path length it will find. On the other hand, A\*’s heuristic component creates a hit or miss situation where if by chance the heuristic is spot on, it will be much faster than BI BFS, but if the if it’s wrong a few times, it will take far longer (e.g. the last one).

Getting back to the ID DFS search, here’s what a more extensive graph of the ID DFS runtimes look like. We can actually find an exponential equation for this curve with an R2 of 0.99 (RS1!), so I figured there was no point in running ID DFS on 8 puzzles for a few more hours unless I wanted to make my computer suffer. Especially for situations like 8 puzzle where we have more than enough memory available, ID DFS is useless.

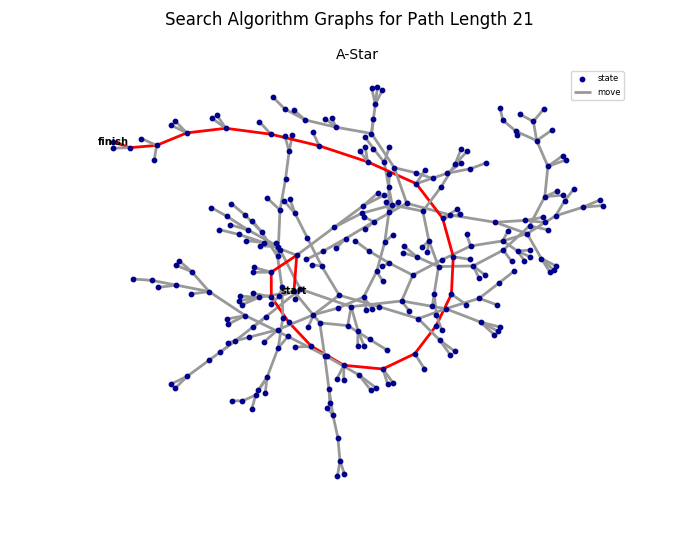


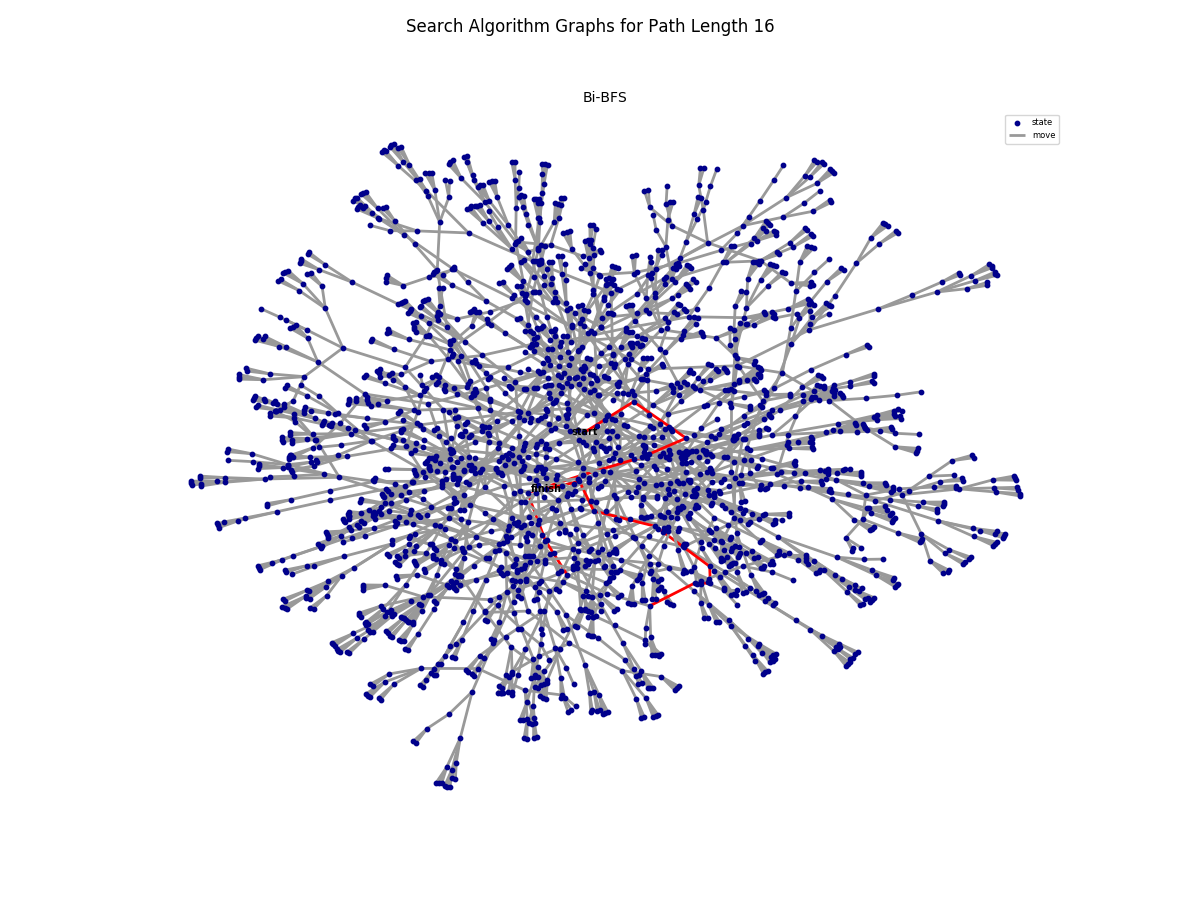
Exploration 6: Visualization!!

I spent way longer on this that I should have taking into consideration all the stuff I have in the next few weeks, but this was so much fun.

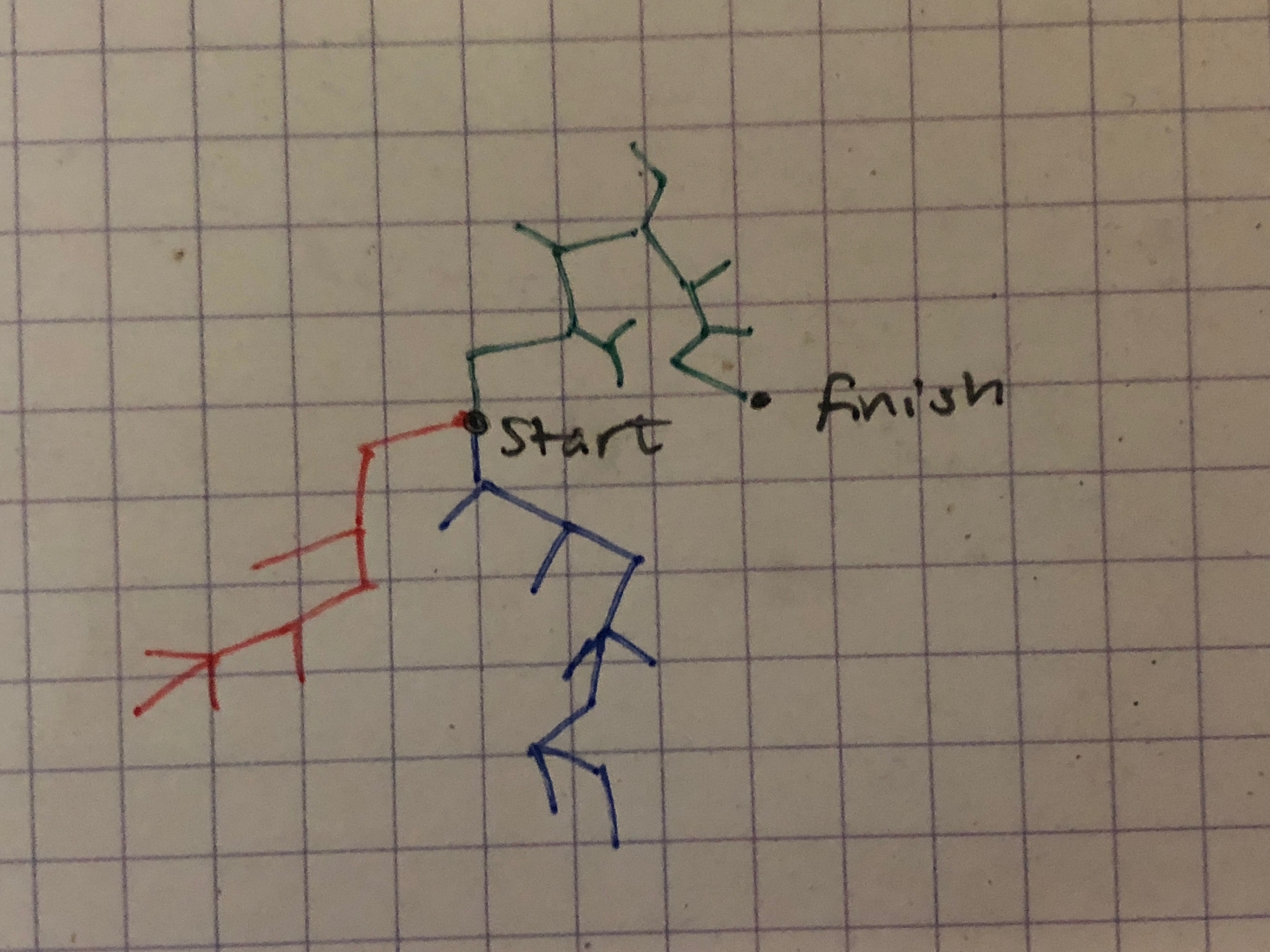
After some Googling, I found something called [graph-tool](https://graph-tool.skewed.de/) which had plenty of features that I could use to visualize my data. Once I got it all set up, it was just a matter of making it look logical and pleasing.

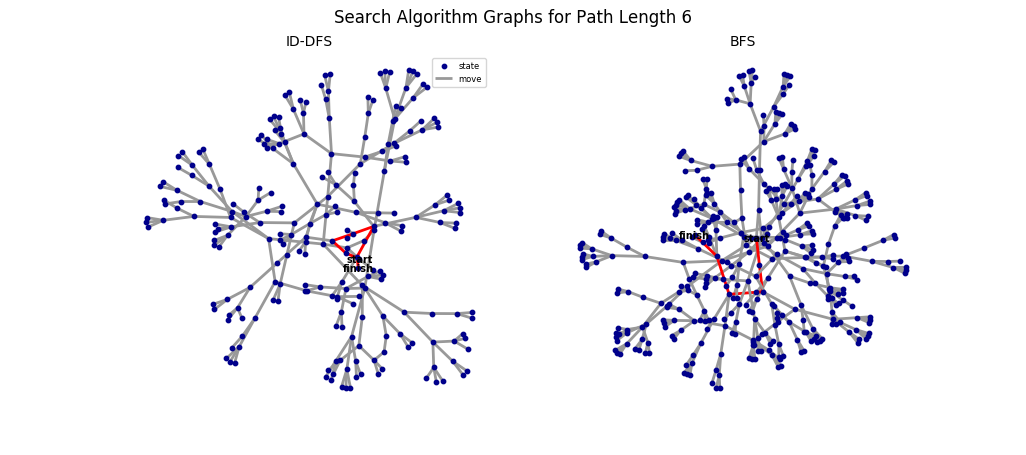
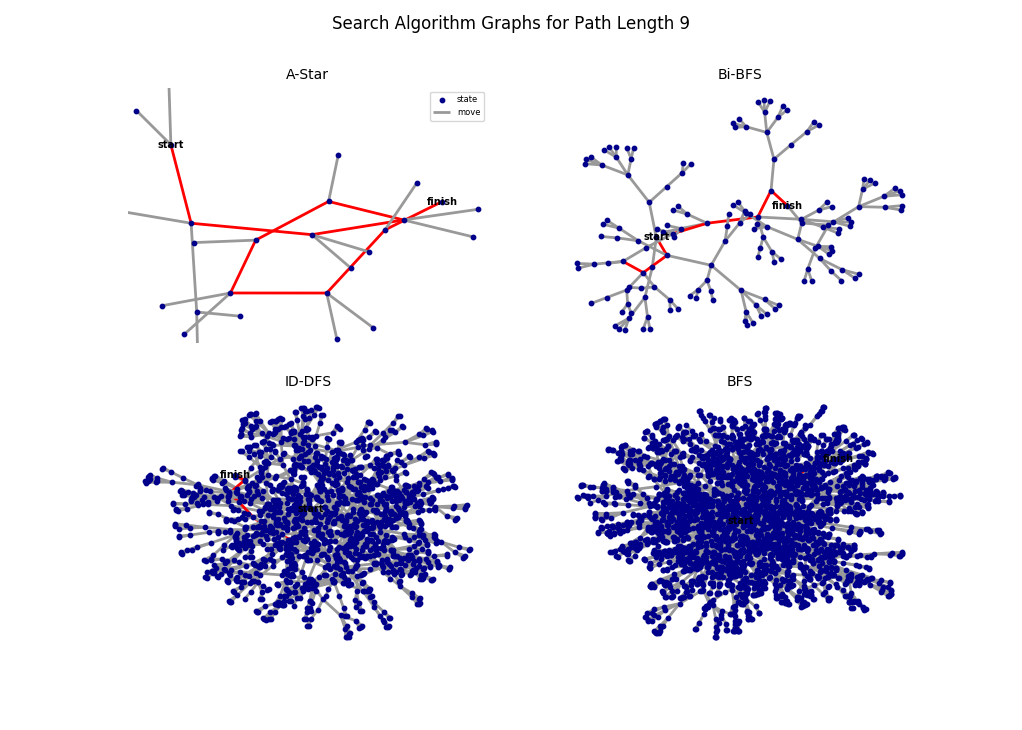
FYI: the blue dots are states (nodes), the lines are moves that connect the states (edges), and when lines are red, it means that they are the part of the shortest path. Follow the red lines from “start” to “finish” and you have yourself the path found.

 The first image shows all of the different searches for a path length of 4. Starting in the top right is A\* which is produces a much simpler graph compared to the others (more drastic as path lengths get longer). It’s very easy to follow the computers “thought process” here: it begins at “start” checks the 4 surrounding states and determines that the red one was the best. It went to the next, and it examined 3 children, and determined that the red one was best, and so on and so on until it found the end goal. On longer paths you’ll see it follow a random line for a while, and then stop and go back to what would eventually be the correct path (exactly like you showed on the board when introducing A\*).

The BI BFS’s number of nodes is always between A\* and ID DFS/BFS, and it takes a bit more imagination to see the computer’s though process. If you look closely, there’s always only one place where the “start” side and the “finish” side of the graph meet, and the shortest path always uses this. I’d like to color code the two sides when I get the chance to help visualize this better, but for now you can just imagine the computer starting at “start” and “finish” and simultaneously branching out, and as soon as they meet, they connect.

ID DFS and BFS are pretty similar in that they both look like a jumbled mess. When you look closely, you’ll see that while BI BFS always has nodes branching from both “start” and “finish”, that isn’t the case in ID DFS or BFS, because once it hits “finish”, it’s done. If I were to color code them (earlier processed nodes in a lighter color, later nodes darker) you would see that each ID DFS branch is basically the same color as it moves down, but two different branches will be different colors. That makes very little sense in words, so here’s a picture:

 Imagine that rather than blue, red, green, its light grey, grey, and dark grey, and that would be a very simplified version of what ID DFS would look like color coded. BFS on the other hand would be like a radial gradient, where lighter greys would be in the middle, and darker greys would be towards the ends, because that’s roughly the order its processed in. Without color coding, however, they look very similar.

 The complexity of the graphs also help to visualize the findings of the NPS extension.

P.S. I had to run the graphing thing (visualize2()) with python2.7 because I accidently installed graph-tool in 2.7 and by the time I realized it I just decided to live with my mistake.