Sliding Puzzles Part 3 Report

I did explorations A, B, D, and G.

**Exploration A:**

I used the first 40 puzzles in the 15 puzzles file to test the effect of a multiplier value.

**Results:**

As the multiplier value increased above 1, the runtimes increased as well, just as the instructions said they would.

As the multiplier decreased from 1, however, it got interesting. I started at 1 and decreased it by 0.1 each time. The lower the multiplier value was, the faster A\* found a solution. In fact, there was an exponential decrease in search time. However, these solutions were not always the shortest possible solution. When the value was anywhere between 0.1 and 0.6, the results were completely inaccurate and would almost always give solution lengths way bigger than the shortest solution. When the value was 0.6 or 0.7, there started being some semblance of accuracy, with A\* returning the correct length most of the time. At 0.8, there was only one incorrect value out of all 40 of the tested puzzles. At 0.9, it was perfectly accurate for those 40 puzzles.

So, in conclusion, lower multiplier values meant faster runtimes but less accurate results. Values of 0.9 and perhaps 0.8 seem usable, while still being significantly faster than having no multiplier or a multiplier with a value of 1.

Figure 1: A graph of the time to solve the first 40 puzzles given multiplier values from 0.1-1.0

**Exploration B:**

I used this puzzle: “BDHCOFAJIGNKM.EL” with a correct solution length of 40.

I ran my weighted A\* on this puzzle 100 times with modifier values of 0.5, 0.55, 0.6, and 0.65.

**Results:**

At a value of 0.5, A\* always returned either 42 or 44, both of which are incorrect.

At 0.6 and 0.65, A\* always returned 40, so the tiebreaking had no noticeable effect.

At 0.55, however, something interesting happened. The result would alternate between 40 and 42 randomly, so the tiebreaking actually had a noticeable effect. Without tiebreaking, a modifier value of 0.55 would always return 42, the wrong answer. However, with tiebreaking, it would sometimes return 40, meaning the chance of a correct answer increased from 0% to something higher than 0%. I ran a couple trials, and the chance of returning 40 rather than 42 averaged out to about 65%.

The advantage of adding random tiebreaking is two-fold. First, it gives situations in which modified A\* returns an incorrect value a chance of returning the correct value. Something is always better than nothing. Second, it allows you to use lower modifier values (as a result of the first reason) and therefore achieve faster runtimes.

**Exploration D:**

I tested BFS, IDDFS, and A\* using different puzzles for each.

For BFS, I used the 22-move puzzle “IABDNECJ.FKHMOGL”, which took 153.85 seconds.

For IDDFS, I used the 20-move puzzle “ABCFEJGD.MLHNIKO”, which took 16.76 seconds.

For A\*, I used the 39-move puzzle “EICDJGLHBAK.NMOF”, which took 31.22 seconds.

I also tested some shorter puzzles with all three algorithms to see how total node count compared.

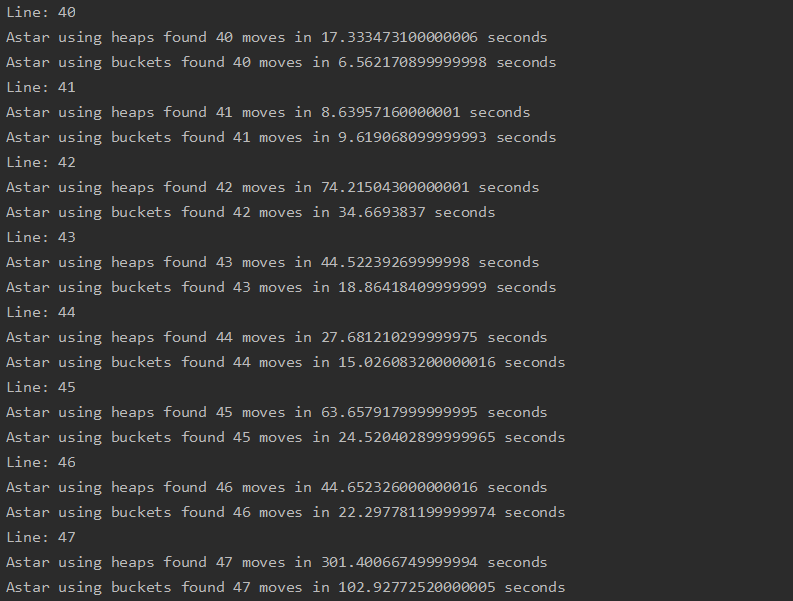
**Results:**

The results were somewhat unexpected overall.

In terms of total nodes processed, A\* almost always had the least, although the margin by which it had the least varied. This was likely dependent on how accurate the heuristic estimate was, just like run time. BFS was in the middle, and IDDFS almost always had the most, except in special cases in which total moves was very low and IDDFS just so happened to process the correct path first. This part was pretty much what I expected.

What I didn’t quite expect were the results for nodes per second. It actually lined up with the order for total nodes; A\* had the least per second, then BFS, then IDDFS had the most. I initially was confused by these results, thinking that since A\* was the fastest, it would have the most nodes per second, and so on. This ended up making more sense the more I thought about it, though. The advantage of A\* over BFS comes in a lower amount of nodes processed, but time to process a single node would be higher since each node also had to added into a heap. IDDFS, on the other hand, processed nodes much faster because it wasn’t taking every single child, so each individual node had a lower processing time.

**Exploration G:**

I implemented heaps as buckets by creating a dictionary of f(x) values with the key as a list of tuples. Doing so decreased my runtime by 2 to 3 times (usually). Here are the runtimes of some longer puzzles:

(Note that when it prints “using heaps”, it means using lists to represent heaps)

As we can see, buckets usually made A\* noticeably faster for puzzles with significant runtimes. However, I found that bucketing actually made puzzles that already could be solved quickly, say in 0.001 seconds, a bit slower. I think that this might be because of the overhead of creating all the buckets, which is inefficient for the short run. However, it is evident that buckets are better (in terms of runtime) for longer puzzles.