**Task 1.2:**

Solved 40 puzzles from file: easy.txt

Average nodes expanded: 334.1

Average search time: 0.012200558185577392

Average solution length: 7.0

Solved 40 puzzles from file: medium.txt

Average nodes expanded: 28269.0

Average search time: 0.6403847694396972

Average solution length: 15.0

Solved 1 puzzles from file: hard.txt

Average nodes expanded: 870572.0

Average search time: 19.751783847808838

Average solution length: 21.0

Both random.txt and worst.txt puzzles took upwards of 30 minutes, so I chose to omit these for this part

**Task 1.3:**

**Uniform Cost Search:**

Solved 40 puzzles from file: easy.txt

Average nodes expanded: 143.7

Average search time: 0.00941963791847229

Average solution length: 7.0

Solved 40 puzzles from file: medium.txt

Average nodes expanded: 6407.25

Average search time: 0.5383153319358825

Average solution length: 15.0

Solved 40 puzzles from file: hard.txt

Average nodes expanded: 72320.5

Average search time: 6.834794729948044

Average solution length: 21.0

Solved 40 puzzles from file: worst.txt

Average nodes expanded: 181315.7

Average search time: 26.959357953071596

Average solution length: 30.05

Solved 40 puzzles from file: random.txt

Average nodes expanded: 58265.65

Average search time: 6.46006800532341

Average solution length: 16.775

**Greedy Best-First Search:**

Solved 40 puzzles from file: easy.txt

Average nodes expanded: 52.675

Average search time: 0.0028564453125

Average solution length: 10.7

Solved 40 puzzles from file: medium.txt

Average nodes expanded: 624.8

Average search time: 0.040770184993743894

Average solution length: 75.9

Solved 40 puzzles from file: hard.txt

Average nodes expanded: 716.2

Average search time: 0.061104482412338255

Average solution length: 94.55

Solved 40 puzzles from file: worst.txt

Average nodes expanded: 771.55

Average search time: 0.04984536170959473

Average solution length: 100.75

Solved 40 puzzles from file: random.txt

Average nodes expanded: 363.175

Average search time: 0.023558282852172853

Average solution length: 56.825

**A\* Search:**

Solved 40 puzzles from file: easy.txt

Average nodes expanded: 13.35

Average search time: 0.0014934778213500977

Average solution length: 7.0

Solved 40 puzzles from file: medium.txt

Average nodes expanded: 341.175

Average search time: 0.026452910900115967

Average solution length: 15.0

Solved 40 puzzles from file: hard.txt

Average nodes expanded: 4887.875

Average search time: 0.3487042486667633

Average solution length: 21.0

Solved 40 puzzles from file: worst.txt

Average nodes expanded: 100266.5

Average search time: 6.982451349496841

Average solution length: 30.05

Solved 40 puzzles from file: random.txt

Average nodes expanded: 11867.425

Average search time: 0.7757408559322357

Average solution length: 16.775

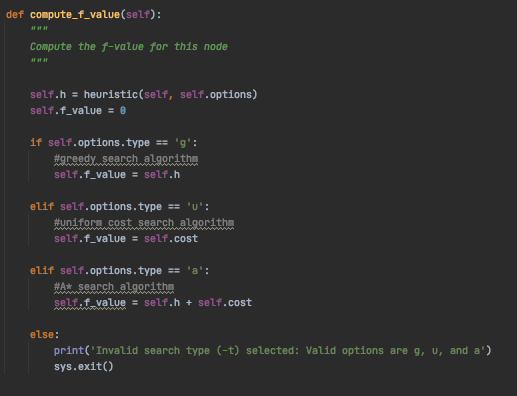
**Task 1.5: Statistics used with hard.txt**

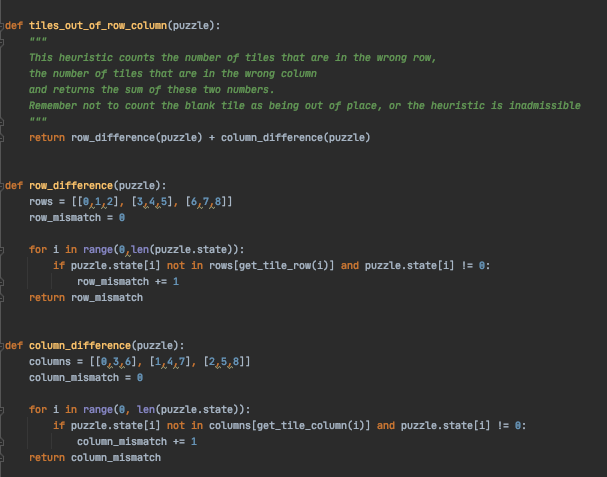
I think the biggest thing that I learned here were two things: One is how different search functions really optimize in different ways, and two how important your heuristic is when it comes to optimization. Both of these things were really fascinating to see with the data provided from running my code on solving an 8-number puzzle.

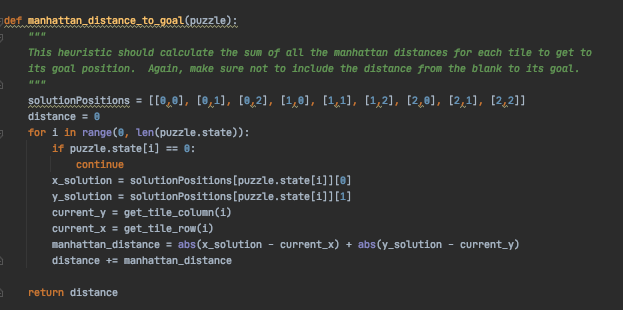
The different search functions really have tradeoffs. As shown, the greedy search was significantly faster not only than the other search functions, but even much faster than A\*. This being said, it was nowhere close to the optimized solution, and quite frankly it was the only one that could not get an optimized solution. You really make big sacrifices when you go with greedy, but also if you are more focused on finding a solution in a quick time, it isn’t a bad one to choose. Uniform cost was the worst of the improved algorithms, it generated a lot of nodes, was pretty slow, but still got an optimized path. A\* was significantly the fastest, as it used minimal nodes, was optimized, and also was super quick. It was only slower than greedy, and in real time it was not that much slower. Just the search algorithms alone greatly influenced the resulting data.

Heuristics played a much bigger part in this assignment then I originally thought. I was surprised at how on average the Manhattan distance was so much faster than the other two. It was interesting to see how simply by choosing to use a different approach to seeing how far you were from the goal state you were able to make the program run not only faster, but generate less nodes. It makes me realize that even for my own problems, breaking it down into which state is truly “closer” to the goal node is super important to finding an answer that requires less resources. This being said, I think that the heuristic only works depending on the algorithm used. The uniform cost search did not benefit at all from the Manhattan distance, while the others certainly did.

**Code Part 1:**

****

****



**Part 2:**

**2:**

**IDA\* ANALYSIS:**

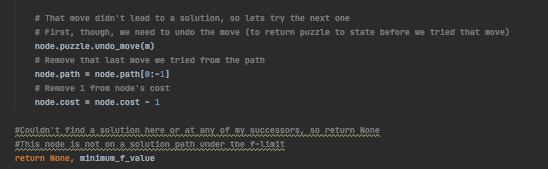
Through my implementation of IDA\*, I learned a lot about the different search options you have when finding a solution. It was really interesting to figure out how to combine both the depth first search aspect of IDS with the heuristic and f value of A\*. It was interesting to see how we can use and combine these different search algorithms and heuristics to get more optimized programs. IDA\* shows that not only can we get a good run time from a combination of depth first search and A\*, but that the different heuristics totally help us with optimization as well.

While it doesn’t seem like IDA\* works as fast as A\* with best first search, we can see in each iteration how it only generates nodes up until the minimum f value it sees. That way we minimize the distance we iterate down each time, and we only generate nodes up to the optimal solution cost. With IDA\* we never expand on nodes that have an f value that is greater than what the optimal cost would be. This way we are never expanding nodes we don’t need to, and we can find the optimal path a lot quicker than with normal IDS, and we generate much fewer nodes in total.

**Code for Part 2:**

****

****

****