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

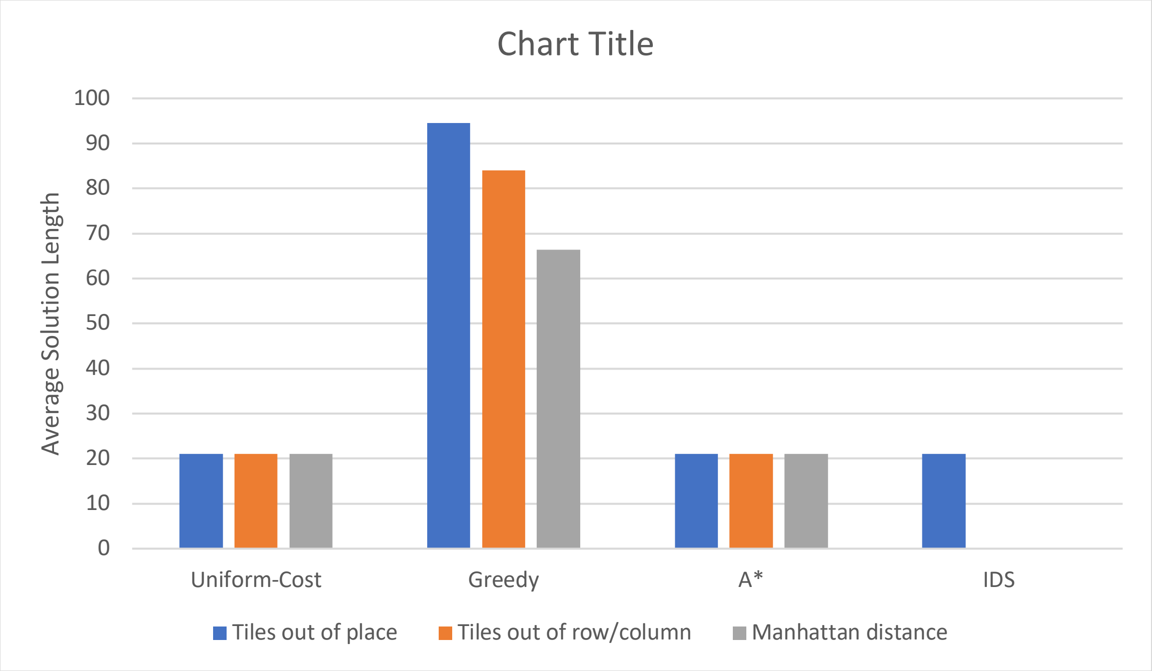


Chart Title
Average Solution Length
100
90
80
70
60
50
40
30
20
10
0
Uniform-Cost
Greedy
A*
IDS
Tiles out of place
Tiles out of row/column
Manhattan distance

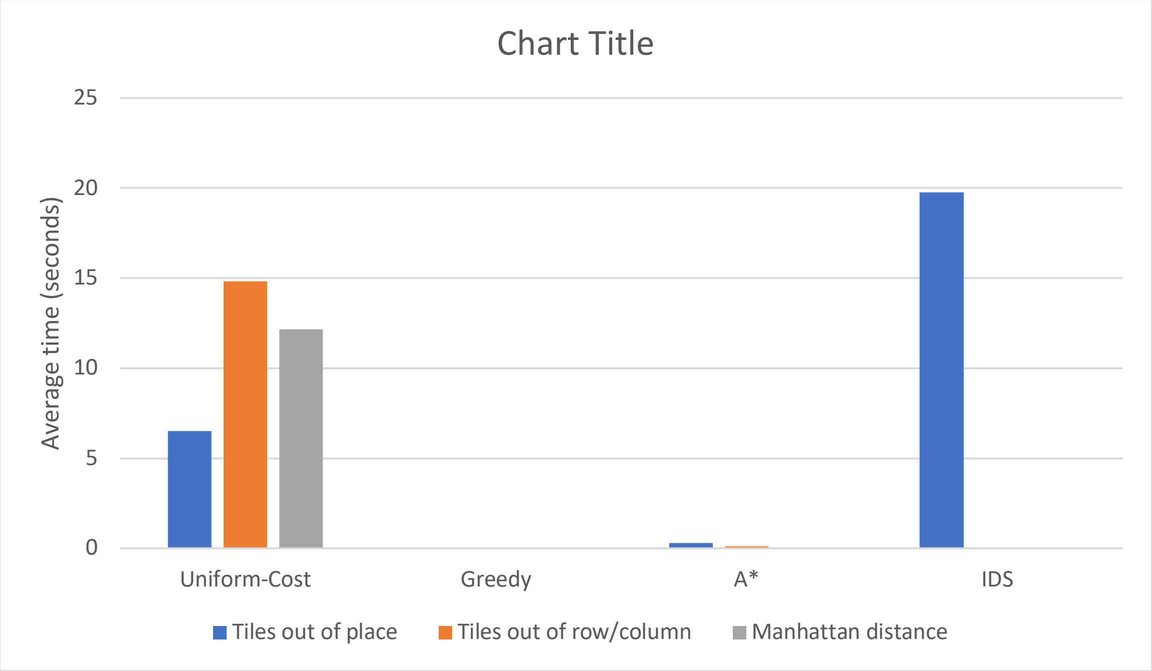
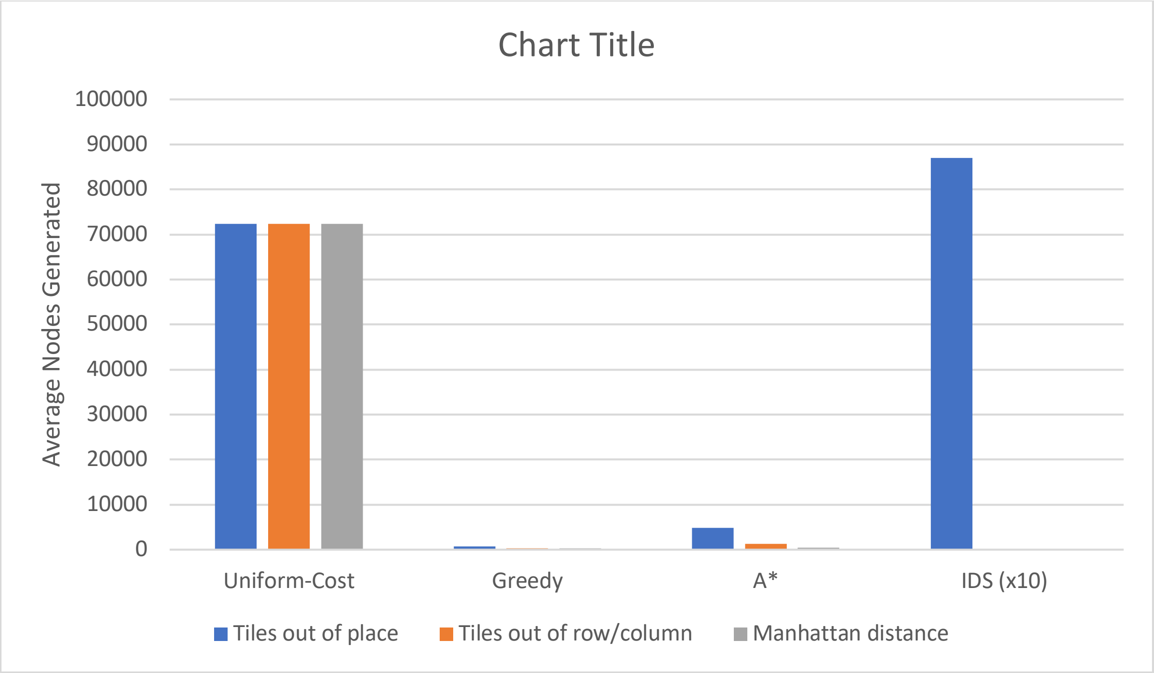


Chart Title
Average time (seconds)
25
20
15
10
5
0
Uniform-Cost
Greedy
A*
IDS
Tiles out of place
Tiles out of row/column
Manhattan distance

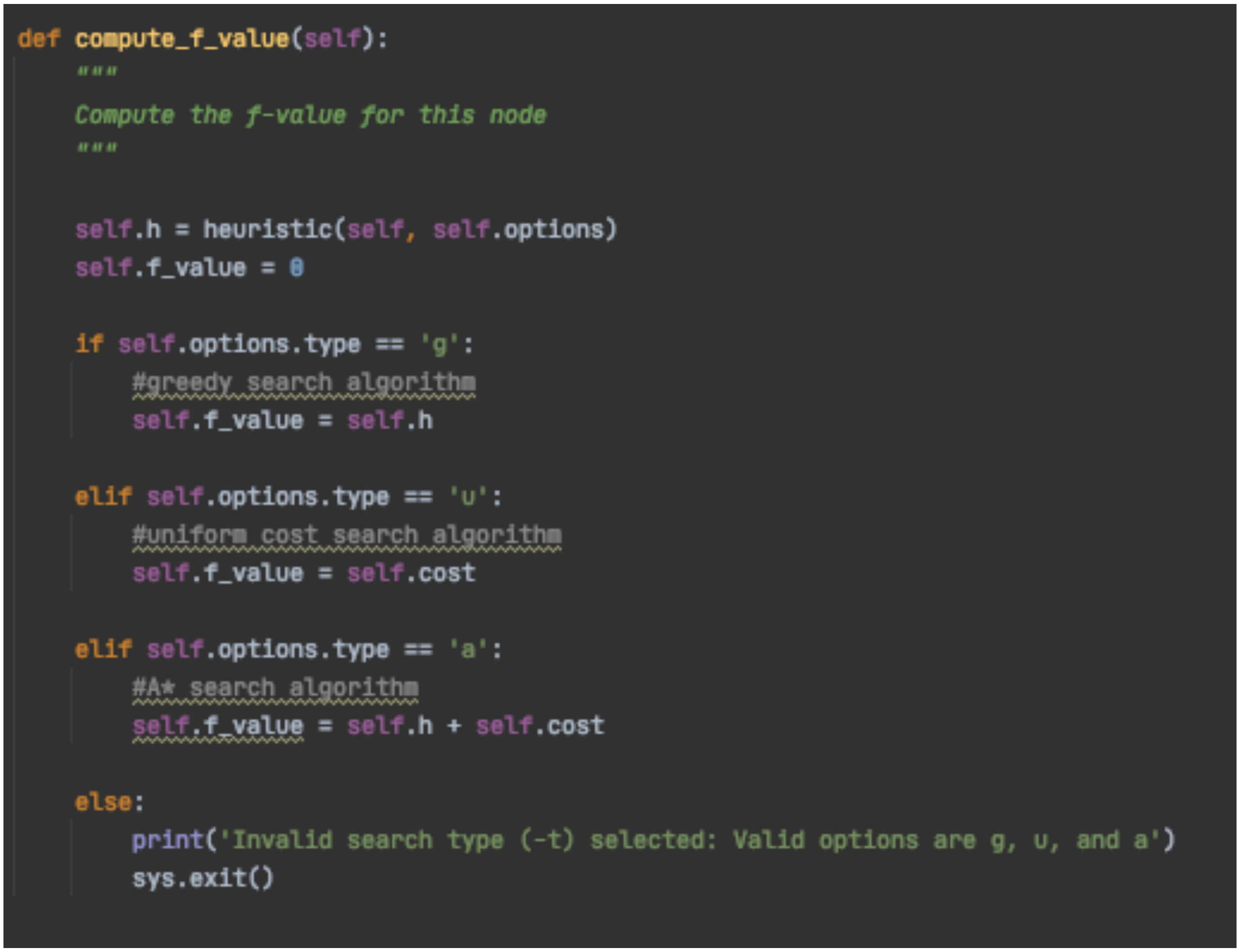


I think the biggest thing that I learned here were two things: One is how different search functions really optimize in different ways, and two how important your heuristic is when it comes to optimization. Both of these things were really fascinating to see with the data provided from running my code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

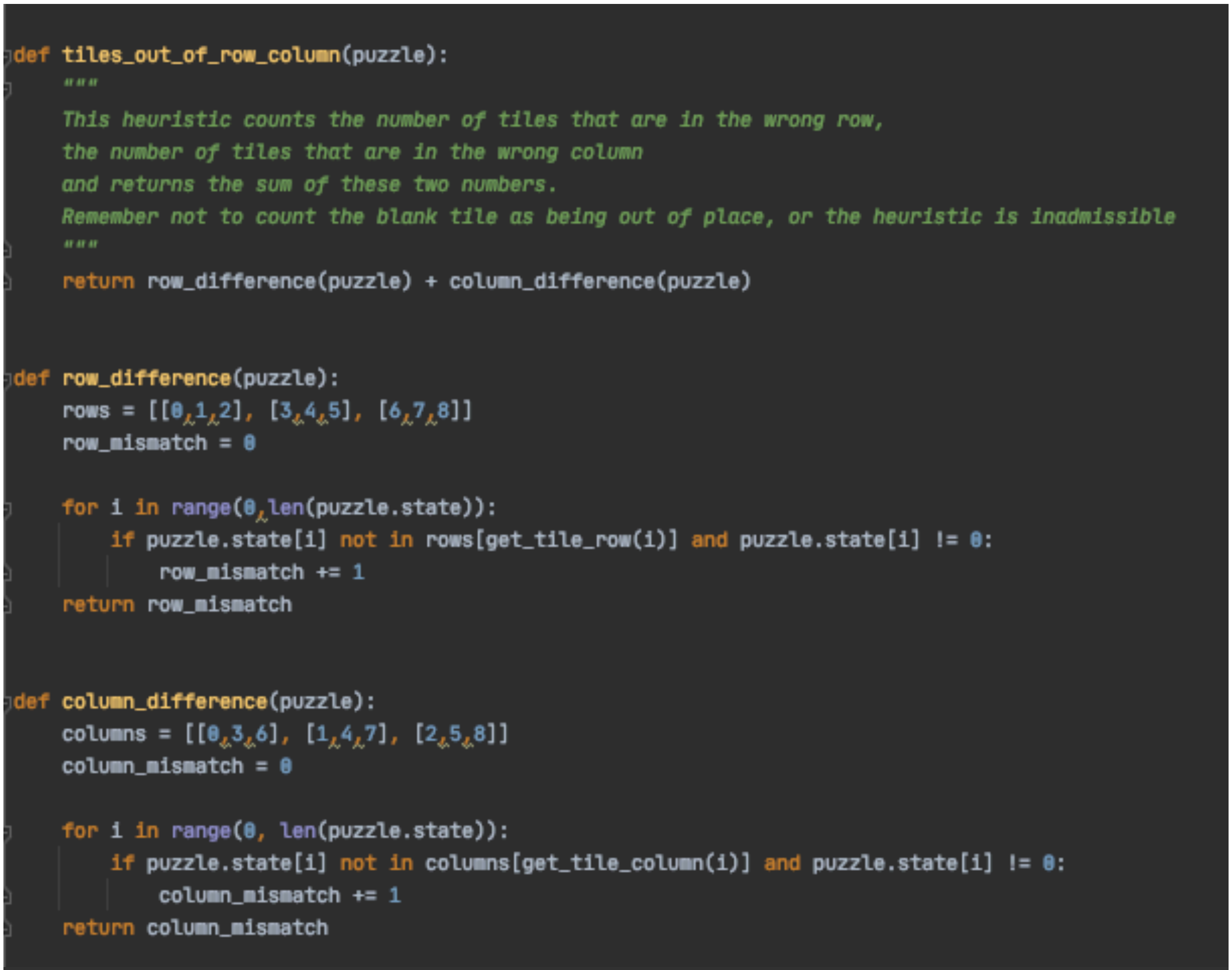
The different search functions really have tradeoffs. As shown, the greedy search was significantly faster not only than the other search functions, but even much faster than A*. This being said, it was nowhere close to the optimized solution, and quite frankly it was the only one that could not get 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.

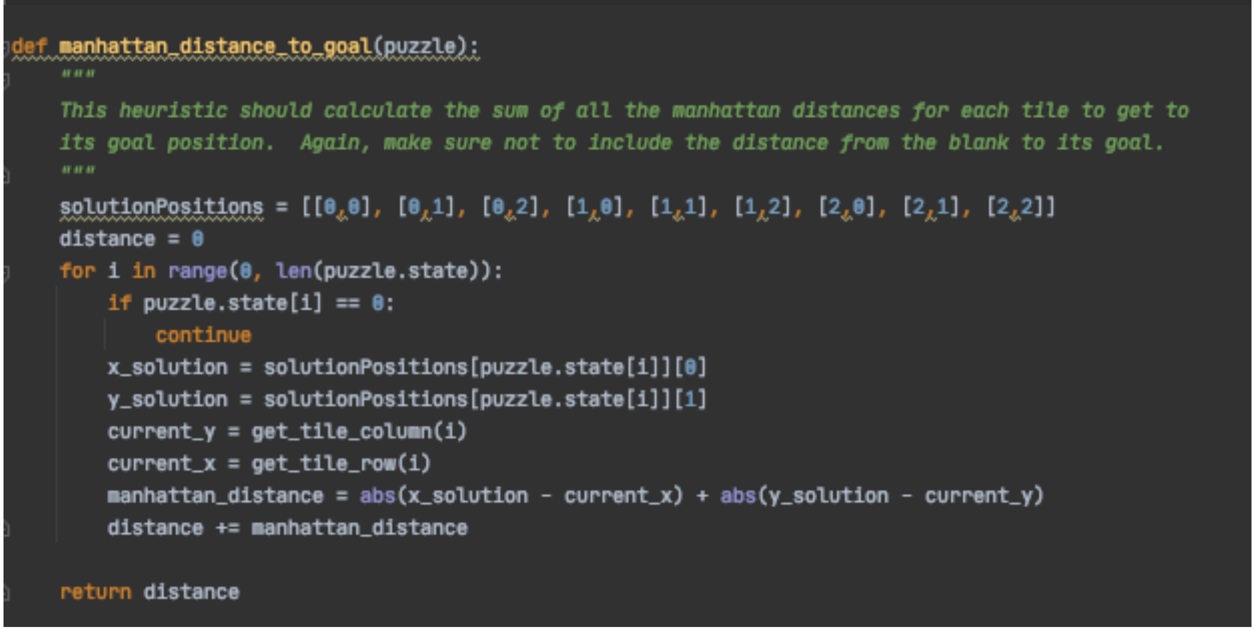
Code Part 1:



```python
def compute_f_value(self):
    """
    Compute the f-value for this node
    """

    self.h = heuristic(self, self.options)
    self.f_value = 0

    if self.options.type == 'g':
        #greedy search algorithm
        self.f_value = self.h

    elif self.options.type == 'u':
        #uniform cost search algorithm
        self.f_value = self.cost

    elif self.options.type == 'a':
        #A* search algorithm
        self.f_value = self.h + self.cost

    else:
        print('Invalid search type (-t) selected: Valid options are g, u, and a')
        sys.exit()
```



```python
def tiles_out_of_row_column(puzzle):
    """
    This heuristic counts the number of tiles that are in the wrong row,
    the number of tiles that are in the wrong column
    and returns the sum of these two numbers.
    Remember not to count the blank tile as being out of place, or the heuristic is inadmissible
    """
    return row_difference(puzzle) + column_difference(puzzle)


def row_difference(puzzle):
    rows = [[0,1,2], [3,4,5], [6,7,8]]
    row_mismatch = 0

    for i in range(0,len(puzzle.state)):
        if puzzle.state[i] not in rows[get_tile_row(i)] and puzzle.state[i] != 0:
            row_mismatch += 1
    return row_mismatch


def column_difference(puzzle):
    columns = [[0,3,6], [1,4,7], [2,5,8]]
    column_mismatch = 0

    for i in range(0, len(puzzle.state)):
        if puzzle.state[i] not in columns[get_tile_column(i)] and puzzle.state[i] != 0:
            column_mismatch += 1
    return column_mismatch
```



```python
def manhattan_distance_to_goal(puzzle):
    """
    This heuristic should calculate the sum of all the manhattan distances for each tile to get to
    its goal position.  Again, make sure not to include the distance from the blank to its goal.
    """
    solutionPositions = [[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]]
    distance = 0
    for i in range(0, len(puzzle.state)):
        if puzzle.state[i] == 0:
            continue
        x_solution = solutionPositions[puzzle.state[i]][0]
        y_solution = solutionPositions[puzzle.state[i]][1]
        current_y = get_tile_column(i)
        current_x = get_tile_row(i)
        manhattan_distance = abs(x_solution - current_x) + abs(y_solution - current_y)
        distance += manhattan_distance

    return distance
```

Part 2:

2:

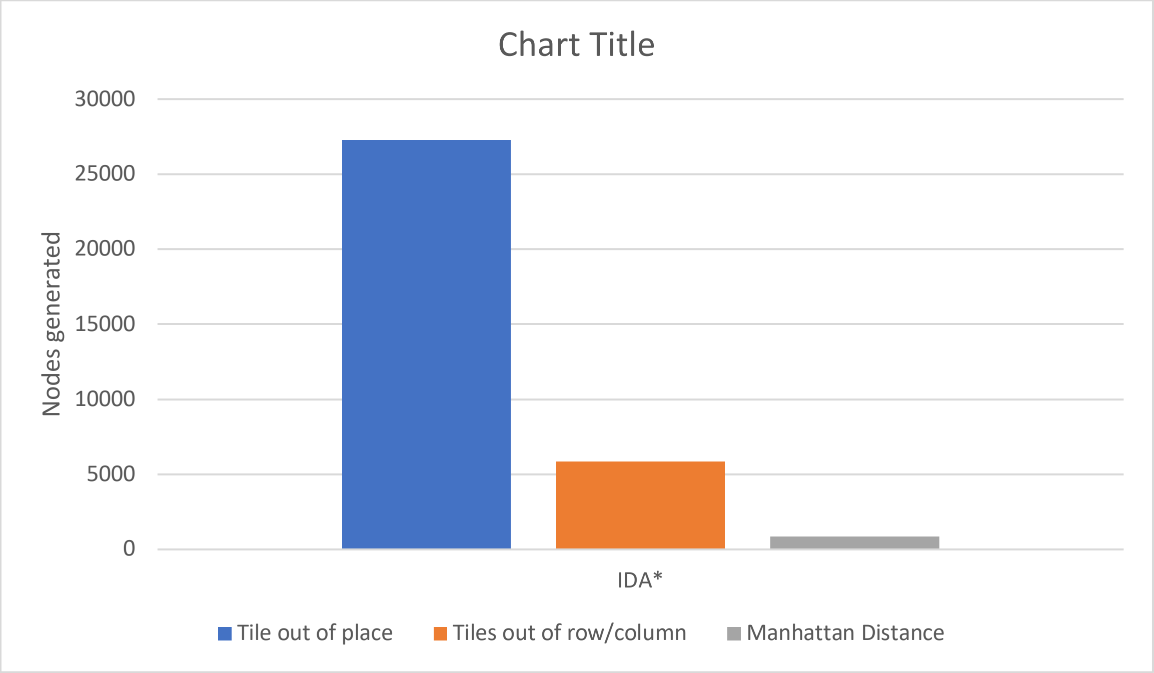


Chart Title
30000
25000
20000
15000
10000
5000
0
Nodes generated
IDA*
Tile out of place
Tiles out of row/column
Manhattan Distance

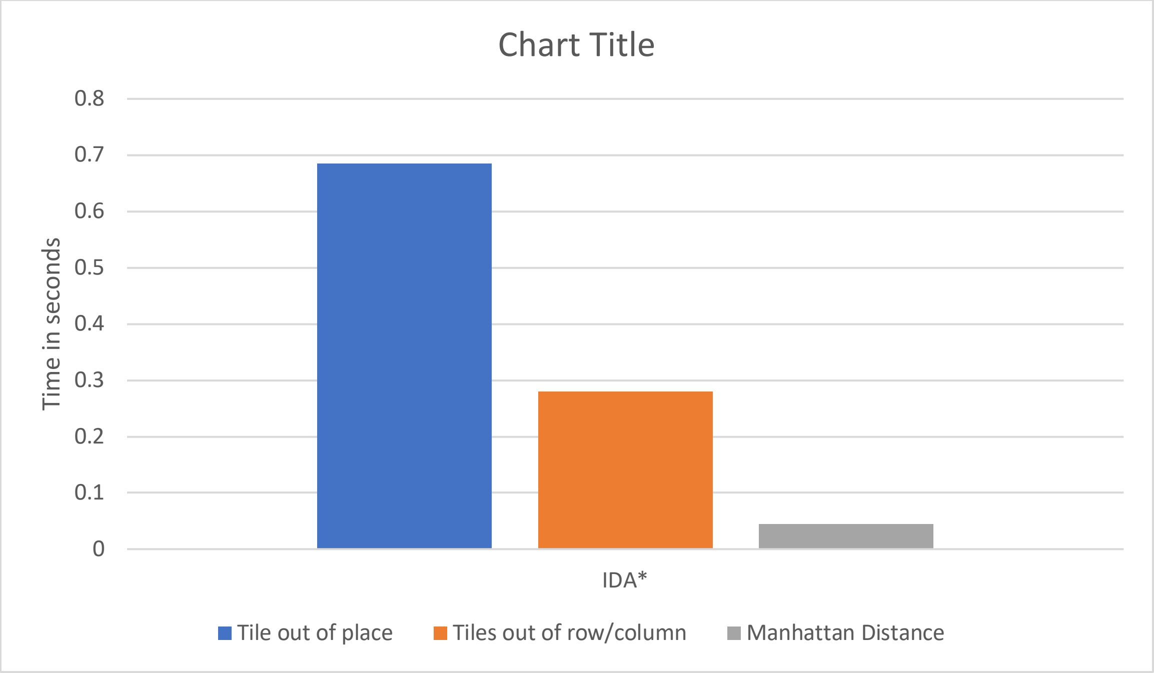


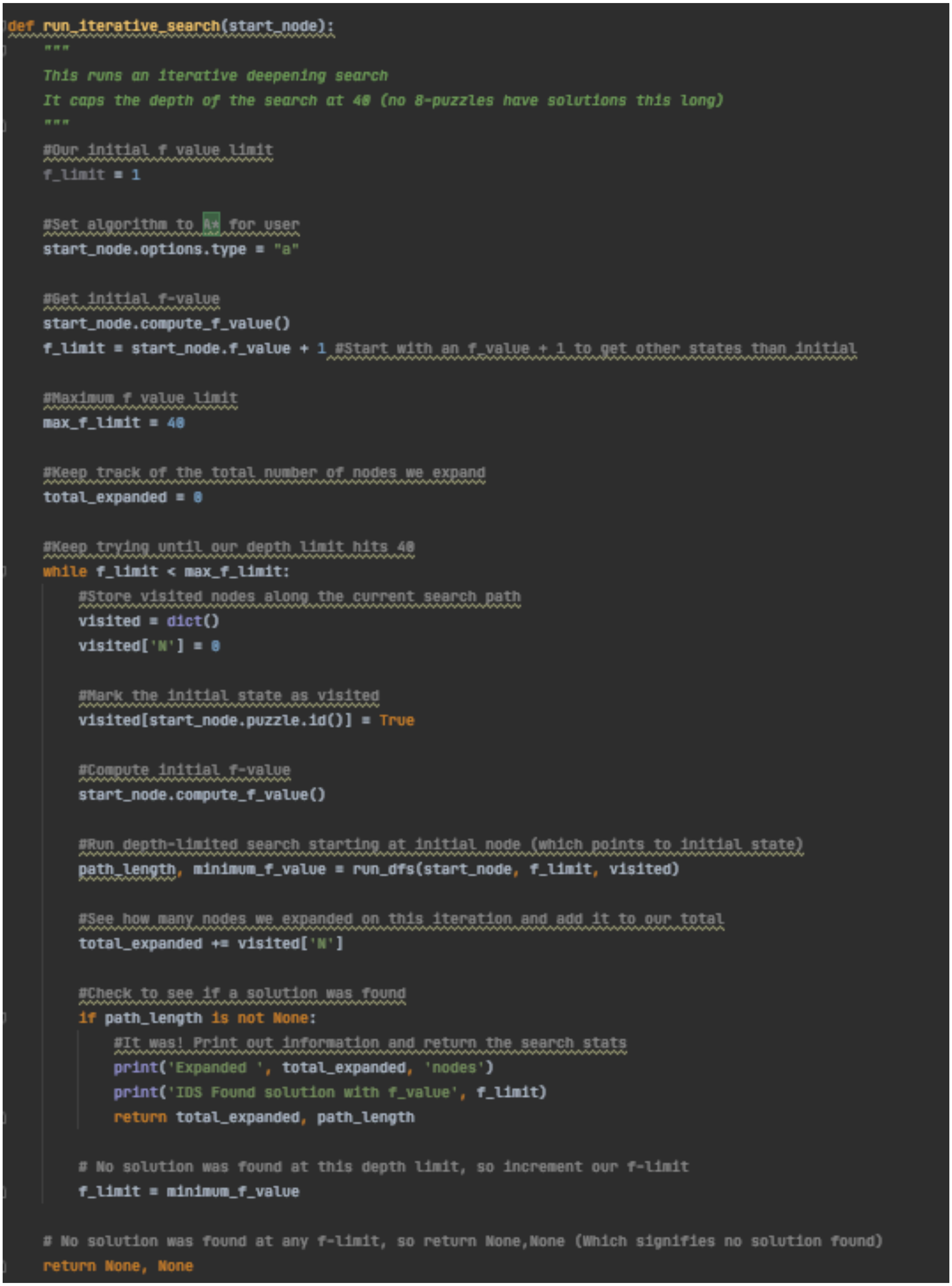
Chart Title
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time in seconds
IDA*
Tile out of place
Tiles out of row/column
Manhattan Distance

IDA* ANALYSIS:

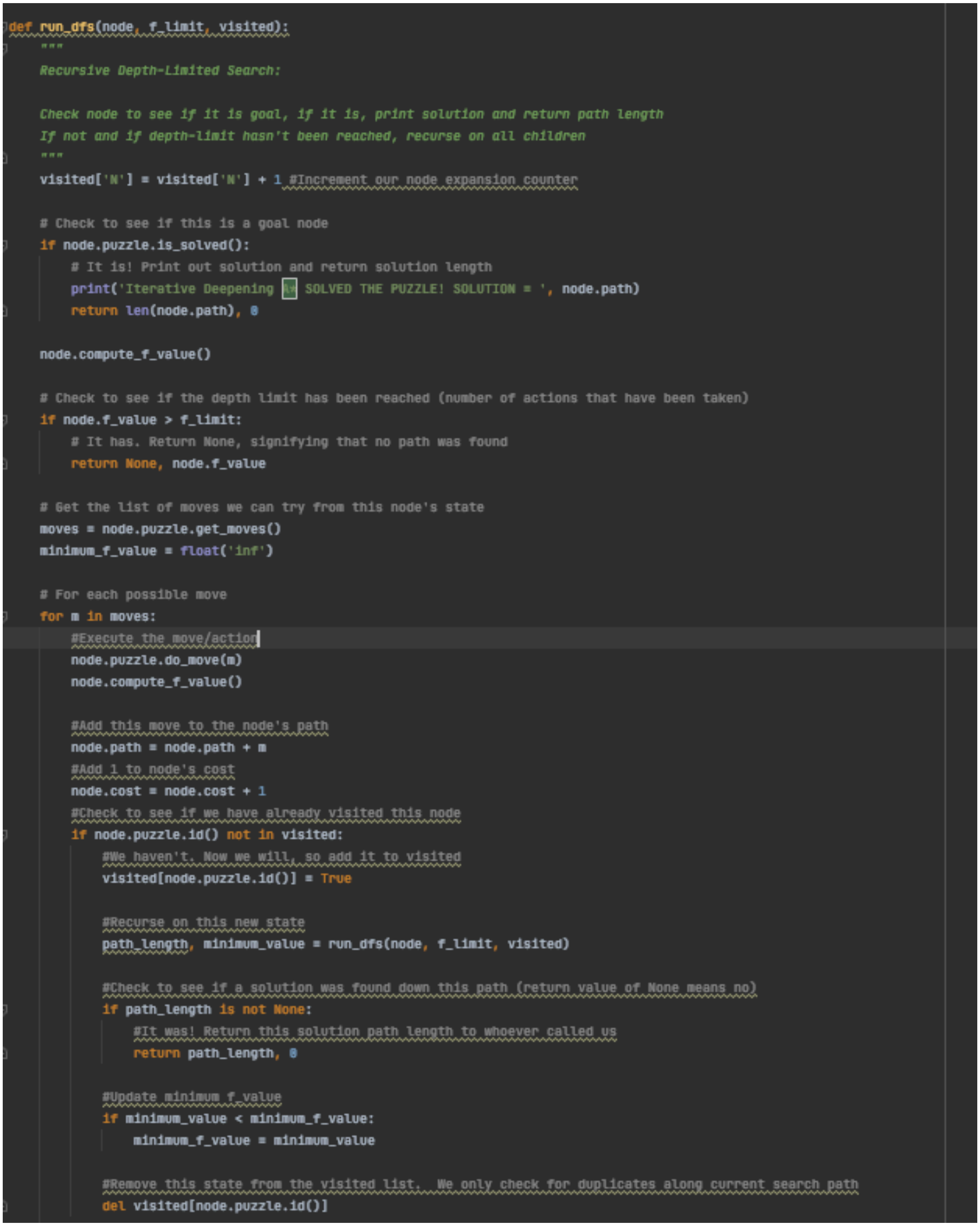
Through my implementation of IDA*, I learned a lot about the different search options you have when finding a solution. It was really interesting to figure out how to combine both the depth first search aspect of IDS with the heuristic and f value of A*. It was interesting to see how we can use and combine 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.

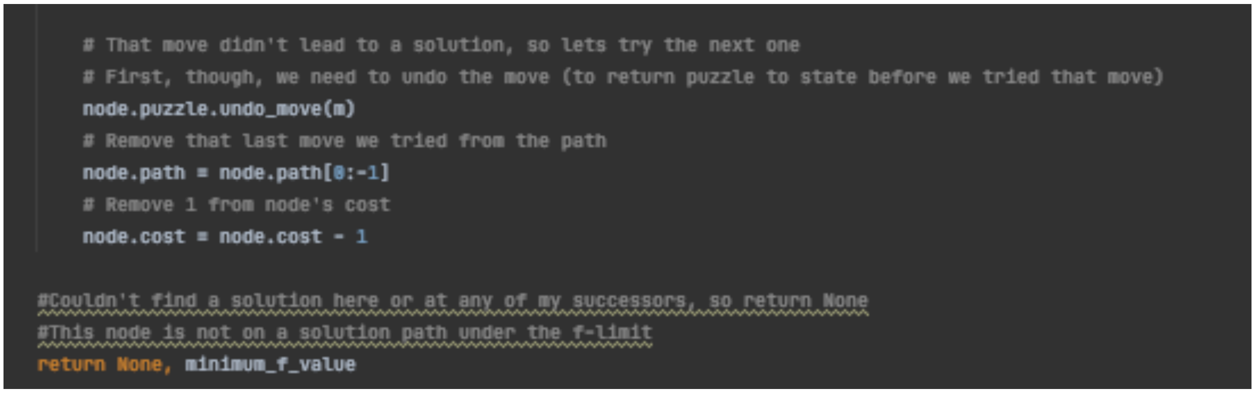
Code for Part 2:



```
def run_iterative_search(start_node):
    """
    This runs an iterative deepening search
    It caps the depth of the search at 40 (no 8-puzzles have solutions this long)
    """
    #Our initial f value limit
    f_limit = 1

    #Set algorithm to A* for user
    start_node.options.type = "a"

    #Get initial f-value
    start_node.compute_f_value()
    f_limit = start_node.f_value + 1 #Start with an f_value + 1 to get other states than initial

    #Maximum f value limit
    max_f_limit = 40

    #Keep track of the total number of nodes we expand
    total_expanded = 0

    #Keep trying until our depth limit hits 40
    while f_limit < max_f_limit:
        #Store visited nodes along the current search path
        visited = dict()
        visited['N'] = 0

        #Mark the initial state as visited
        visited[start_node.puzzle.id()] = True

        #Compute initial f-value
        start_node.compute_f_value()

        #Run depth-limited search starting at initial node (which points to initial state)
        path_length, minimum_f_value = run_dfs(start_node, f_limit, visited)

        #See how many nodes we expanded on this iteration and add it to our total
        total_expanded += visited['N']

        #Check to see if a solution was found
        if path_length is not None:
            #It was! Print out information and return the search stats
            print('Expanded ', total_expanded, 'nodes')
            print('IDS Found solution with f_value', f_limit)
            return total_expanded, path_length

        # No solution was found at this depth limit, so increment our f-limit
        f_limit = minimum_f_value

    # No solution was found at any f-limit, so return None,None (Which signifies no solution found)
    return None, None
```



```python
def run_dfs(node, f_limit, visited):
    """
    Recursive Depth-Limited Search:

    Check node to see if it is goal, if it is, print solution and return path length
    If not and if depth-limit hasn't been reached, recurse on all children
    """
    visited['N'] = visited['N'] + 1 #Increment our node expansion counter

    # Check to see if this is a goal node
    if node.puzzle.is_solved():
        # It is! Print out solution and return solution length
        print('Iterative Deepening A* SOLVED THE PUZZLE! SOLUTION = ', node.path)
        return len(node.path), 0

    node.compute_f_value()

    # Check to see if the depth limit has been reached (number of actions that have been taken)
    if node.f_value > f_limit:
        # It has. Return None, signifying that no path was found
        return None, node.f_value

    # Get the list of moves we can try from this node's state
    moves = node.puzzle.get_moves()
    minimum_f_value = float('inf')

    # For each possible move
    for m in moves:
        #Execute the move/action
        node.puzzle.do_move(m)
        node.compute_f_value()

        #Add this move to the node's path
        node.path = node.path + m
        #Add 1 to node's cost
        node.cost = node.cost + 1
        #Check to see if we have already visited this node
        if node.puzzle.id() not in visited:
            #We haven't. Now we will, so add it to visited
            visited[node.puzzle.id()] = True

            #Recurse on this new state
            path_length, minimum_value = run_dfs(node, f_limit, visited)

            #Check to see if a solution was found down this path (return value of None means no)
            if path_length is not None:
                #It was! Return this solution path length to whoever called us
                return path_length, 0

            #Update minimum f_value
            if minimum_value < minimum_f_value:
                minimum_f_value = minimum_value

            #Remove this state from the visited list.  We only check for duplicates along current search path
            del visited[node.puzzle.id()]
```



```
    # That move didn't lead to a solution, so lets try the next one
    # First, though, we need to undo the move (to return puzzle to state before we tried that move)
    node.puzzle.undo_move(m)
    # Remove that last move we tried from the path
    node.path = node.path[0:-1]
    # Remove 1 from node's cost
    node.cost = node.cost - 1

#Couldn't find a solution here or at any of my successors, so return None
#This node is not on a solution path under the f-limit
return None, minimum_f_value
```