Adrian Guzman
EECS 391

Project 1 Writeup

1. Code Design

My program can be divided into three main sections, two of which are classes. The first section is the main method, which opens a file specified by the command line argument and passes it into a new instance of a PuzzleSolve object.

The next section is the class PuzzleSolve, but before I explain this I will explain the Node class, which is the third and final section. A node represents an instance of an 8-puzzle. It contains the state (the order of the tiles stored as a 2d array. Default set to the solved puzzle), the heuristic value of the puzzle, the parent of the puzzle (a reference to the Node the current Node was derived from, if any. Used only in search algorithms), and the path direction (the direction the parent moved to reach the current Node state. This is for ease of tracking the directional path and is not memory efficient. To make memory efficient, directional path should be derived from the actual node path which is calculated using the parent value).

```
def __init__(self,statestring=[["b", "1", "2"], ["3", "4", "5"], ["6", "7", "8"]],hval=0):
    self.state = self.setState(statestring)
    self.hval = hval
    self.parent = None
    self.path_direction = None
```

A Node has only the ability to manipulate its own state through the defined methods setState(), move(), and randomizeState().

I chose to make an object representative of a single puzzle state for a few reasons. Primarily, each node can store a reference to its parent, which makes finding the path from the node found by the search algorithm to the start node trivial. The following method shows the simplicity of such a search:

```
def generate_path(self, node):
    """
    Generates the path of board states following a nodes parent
    """

    path = [node]
    while node.parent != None:
       path.append(node.parent)
       node = node.parent
    path.reverse()
    return path
```

Additionally, when searching through a state space, instead of representing each puzzle state as a 2d array and trying to keep track of heuristic values and paths, it is easy and intuitive to instead search through nodes that can store the additional information upon creation. Then it becomes only a matter of tracking nodes, with access to the associated information coming much quicker and easier when needed.

The PuzzleSolve class is where the fun begins. It begins by creating a master Node upon instantiation, which will serve as the single Node that it manipulates throughout the duration of the program's runtime. It contains the run_commands() method which parses a text file and executes the commands read in line by line. Depending on the command, a state manipulation method is called on the master Node object which modifies it in place, or a search algorithm is called with the master Node as input, after which the master Node is set to the output Node of the search algorithm if one is returned. The in place Node modifications are defined in the Node class itself and simply deal with modifying a 2d array (the Node state). The search algorithms however are unique to the PuzzleSolve class.

A wrapper method first determines which type of search to execute: beam or A-star.

```
if search == "A-star":
        final = self.Astar(arg)
    elif search == "beam":
        final = self.local_beam(arg)
```

The node returned by the specified search is then stored so that the path can be derived recursively from the parent attribute using the generate_path_directions() and generate_path() methods. The derived paths are printed and the master Node is set to reference the locally stored node (the node returned by the search algorithm).

For my version of A-star search, everything is implemented in the same way as normal A-star search, except for the priority queue. Instead of using a priority queue, at the end of each search iteration, I sorted the list of expandable Nodes by increasing heuristic value, which results in the best heuristic valued Node at the front of the list, and in the next iteration the algorithm just selects the Node at the front of the list.

```
searchable.sort(key = lambda x:x.hval ,reverse=False)
```

I chose this implementation because initially I could not find a native python PriorityQueue object to use, so I resorted to sorting as such. I have since found that the python Queue module allows creation of a PriorityQueue which I would definitely use if I wanted to make the algorithm as efficient as possible, since the python sort is O(nlogn) and PriorityQueue retrieval is O(logn). I'm not implementing the PQ however because I have done all my testing with the sort

method and don't want to repeat all of it to make sure the PQ functions correctly, since that isn't what's being examined here. When the search terminates it returns the final Node (which will be in the solved state with a reference to its parent) if a solution is found or None if there is no solution. Otherwise the A-star search is straightforward in its implementation

For my version of local beam search, it keeps track of a maximum of k potential nodes in a list. For each Node in this list, generate the Node's successors and store all of these successor Nodes in another temporary list. Then we sort the list by the heuristic value of the Node (I am using heuristic 2 because it provided short paths in A-star (and because I don't have time to implement another)). We sort the successors by heuristic value so that we can iterate over the list sequentially, adding each successor to the new list of nodes to be searched until the new list contains the maximum number of states specified, at which point the loop will break. This ensures each Node we add has the best heuristic value in the remaining set of successors, so that we ultimately add the best k successors.

2. Code Correctness

I will test each of the required methods using a separate text file

a. testSetState.txt

Output:

```
setState b12 345 678
printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

setState 287 b54 136

printState
['2', '8', '7']
['b', '5', '4']
['1', '3', '6']
```

The state is set to the goal state as well as another random valid state, and is represented clearly and accurately in both instances

b. testMove.txt

Output:

```
setState b12 345 678

printState
```

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

move right

printState

['1', 'b', '2']
['3', '4', '5']
['6', '7', '8']

move down

printState

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

move left

printState

['1', '4', '2']
['b', '3', '5']
['6', '7', '8']

move left

printState
['1', '4', '2']
['b', '3', '5']
['6', '7', '8']

You can see that each of the first 3 moves moves the blank tile in the desired position with the desired tile swap. The last move tries to move the blank tile to an out of bounds index, so it does not change the state, as the move is invalid.

c. testRandomizeState.txt
Output:

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

randomizeState 2

printState

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

randomizeState 100

printState
['b', '1', '2']
['3', '6', '5']
['7', '4', '8']

The state starts off in the solved position, then is randomized by 2 moves, which you can see are ‘right’ then ‘down’. The state is then randomized by 100 moves, resulting in a pseudo-random solvable state. I’m not sure if I understood the prompt correctly, because it specifies to set the seed so that the same sequence of ‘random’ moves are generated whenever the method is called, the only variable is how many moves are made. This results in the same pseudo-random state being generated, only dependent on n, which doesn’t really seem random to me, but it seems like this must be the case for testing methods. If not then I messed up and would just remove the seed so a truly random move is chosen every time.

d. testA-star.txt
Output:

setState 142 358 67b

solve A-star h1

Solution Length: 4
Solution (Directions): ['up', 'left', 'up', 'left']
Solution (Puzzle Path)
Start Node:
['1', '4', '2']
['3', '5', '8']

['6', '7', 'b']

['1', '4', '2']
['3', '5', 'b']
['6', '7', '8']

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

['1', 'b', '2']
['3', '4', '5']
['6', '7', '8']

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

setState 142 358 67b

solve A-star h2

Solution Length: 4
Solution (Directions): ['up', 'left', 'up', 'left']
Solution (Puzzle Path)
Start Node:
['1', '4', '2']
['3', '5', '8']
['6', '7', 'b']

['1', '4', '2']
['3', '5', 'b']
['6', '7', '8']

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

['1', 'b', '2']
['3', '4', '5']
['6', '7', '8']

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

printState
['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

*(Output is broken up into segments separated by explanations below. The rest of the state paths in the output are summarized to begin and end nodes for the sake of readability)

From the initial state, we can make a sequence of moves by hand, for example let's make the moves [right, down, right, down] and we get the state

['1', '4', '2']
['3', '5', 8']
['6', '7', 'b']

Since we know the shortest path to the goal from this state, we can feed it to A-star and we should get a result path of length 4 using heuristic2, however we may get a longer path using heuristic1 as it is not as accurate.
Upon running the test we see that both heuristics generated the path [up, left, up, left] which we know is indeed the shortest path to the goal state, so from this simple example we can assume with a high probability the algorithm works correctly. Of course this is only one test, so we don't know if it actually works correctly, but to increase our confidence we can perform more simple tests and verify the results by hand. The following is another test that can be solved by hand showing the algorithm functioning correctly.

Initial State:
['3', '1', '2']
['6', 'b', '5']
['7', '4', '8']

setState 312 6b5 748

Solve A-star h2

Solution Length: 4

Solution (Directions): ['down', 'left', 'up', 'up']
This is the inverse of the path I used to reach the initial state, so this is correct

We can also randomize the state by n moves, then solve it and make sure the solution length is less than or equal to n.

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

randomizeState 10

solve A-star h2

Solution Length: 4
Solution (Directions): ['left', 'up', 'up', 'left']
Solution (Puzzle Path)
Start Node:
['1', '4', '2']
['3', '7', '5']
['6', '8', 'b']

...

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

The solution length is less than the number of random moves made and checking by eye it looks like the correct solution.

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

randomizeState 478

solve A-star h2

Solution Length: 24

Solution (Directions): ['left', 'up', 'left', 'down', 'right', 'up', 'up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'right', 'up', 'left', 'down', 'left', 'up']
Solution (Puzzle Path)
Start Node:
['2', '6', '7']
['8', '4', '5']
['3', '1', 'b']

…

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

We can't exactly know for sure if this is the shortest path without assuming the algorithm is correct, but the solution length is much lower than the number of random moves made, so we can predict with a high probability that this is accurate. To ensure even further, we can also make sure that all results from beam search for the same puzzle are equal to or higher than this value, since beam search can not find a better solution than this, since it is optimal. We could run beam search with an increasing k, and the result of the path returned by beam search should approach the number from A-star search: 24, as k reaches infinity and turns beam into BFS, exhausting every possible state. This is tested in the beamSearch test file.

With the unsolvable state:
*I did not include the unsolvable state in the test for the graders sake, as it blows up the terminal.

['1', 'b', '3']
['2', '4', '5']
['6', '7', '8']

Is run, the program will run until all possibilities are exhausted, unless maxNodes is set.

After running these tests we can say that the algorithm is correct with a relatively high confidence. Obviously more tests should be run and analyzed, but for the sake of this report we want to keep it short.

Since A-star stores all expandable states, it should never find a suboptimal solution, and should only fail when no solution exists.

e. testBeam.txt

For beam search we will use the same test states as we did in A-star, to check if solutions found by beam search are the global optimum or just local.

Output:

```
setState 142 358 67b

solve beam 10

Solution Length:  4
Solution (Directions):  ['up', 'left', 'up', 'left']
Solution (Puzzle Path)
Start Node:
['1', '4', '2']
['3', '5', '8']
['6', '7', 'b']

['1', '4', '2']
['3', '5', 'b']
['6', '7', '8']

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

['1', 'b', '2']
['3', '4', '5']
['6', '7', '8']

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

setState 142 358 67b

solve beam 1
```

Solution Length: 4
Solution (Directions): ['up', 'left', 'up', 'left']
Solution (Puzzle Path)
Start Node:
['1', '4', '2']
['3', '5', '8']
['6', '7', 'b']

['1', '4', '2']
['3', '5', 'b']
['6', '7', '8']

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

['1', 'b', '2']
['3', '4', '5']
['6', '7', '8']

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

setState 312 6b5 748

solve beam 10

Solution Length: 4
Solution (Directions): ['down', 'left', 'up', 'up']
Solution (Puzzle Path)
Start Node:
['3', '1', '2']
['6', 'b', '5']
['7', '4', '8']

['3', '1', '2']
['6', '4', '5']
['7', 'b', '8']

['3', '1', '2']
['6', '4', '5']
['b', '7', '8']

['3', '1', '2']
['b', '4', '5']
['6', '7', '8']

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

randomizeState 10

solve beam 10

Solution Length: 4
Solution (Directions): ['left', 'up', 'up', 'left']
Solution (Puzzle Path)
Start Node:
['1', '4', '2']
['3', '7', '5']
['6', '8', 'b']

['1', '4', '2']
['3', '7', '5']
['6', 'b', '8']

['1', '4', '2']
['3', 'b', '5']
['6', '7', '8']

['1', 'b', '2']

```
['3', '4', '5']
['6', '7', '8']

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

printState
['b', '1', '2']
['3', '4', '5']
['6', '7', '8']
```

The results are exactly the same as for A-star. When we initially set the state to

```
['1', '4', '2']
['3', '5', '8']
['6', '7', 'b']
```

Beam search with k=10 returns the same path ['up', 'left', 'up', 'left'] that was returned by A-star search. In fact beam search with k=1 on the same state also returns the same path, meaning our heuristic is a good one. Again, we know this is the optimal path, as was described in the A-star tests, and from deriving the solution by hand.

Then we set the path to

```
['3', '1', '2']
['6', 'b', '5']
['7', '4', '8']
```

And again we get the same optimal path ['down', 'left', 'up', 'up'].

Moving 10 pseudo-random moves from the goal again gives state

```
['1', '4', '2']
['3', '7', '5']
['6', '8', 'b']
```

And when we solve with beam search we get the same solution path ['left', 'up', 'up', 'left'] as before.

Now let’s look at a more randomized state:

```
printState

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

randomizeState 478

printState
```

```
['2', '6', '7']
['8', '4', '5']
['3', '1', 'b']

solve beam 10

Solution Length:  34
Solution (Directions):  ['up', 'left', 'left', 'down', 'right', 'up', 'up', 'left', 'down',
'right', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'left', 'up', 'up', 'right',
'down', 'down', 'left', 'up', 'right', 'up', 'left', 'down', 'down', 'right', 'up', 'up',
'left']
Solution (Puzzle Path)
Start Node:
['2', '6', '7']
['8', '4', '5']
['3', '1', 'b']

…

['b', '1', '2']
['3', '4', '5']
['6', '7', '8']
```

The result is now slightly different from A-star search on the same starting state. In A-star we got a solution length of 23, here we get 34. This means that beam search got stuck on a local optima and returned that, never finding the global optima returned by A-star. If we run the search with k=100 instead, we get a path of length 24, and with k=200 path length = 24 as well.

```
randomizeState 478

printState

['2', '6', '7']
['8', '4', '5']
['3', '1', 'b']

solve beam 200

Solution Length:  24
Solution (Directions):  ['left', 'up', 'left', 'down', 'right', 'up', 'up', 'right',
'down', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'right', 'up',
'left', 'down', 'left', 'up']
```

Solution (Puzzle Path)
Start Node:
['2', '6', '7']
['8', '4', '5']
['3', '1', 'b']
…
['b', '1', '2']
['3', '4', '5']
['6', '7', '8']

So as stated earlier in A-star testing, as k increases the solution length should approach the global optima found by A-star search. Here as k increases length approaches 24 which is what was returned by A-star, so our confidence in the performance of these algorithms increases tremendously as a result of these findings and comparisons.

Now lets see what happens for the unsolvable state:

setState 1b3 245 678

solve beam 2
…

Again, an infinite loop is entered (unless maxNodes is set) because successor states are always added, and none of these successor states have a solution, so the algorithm continues forever.

After running these tests we can say that the algorithm is correct with a very high confidence. Obviously more tests should be run and analyzed, but for the sake of this report we want to keep it short.

Beam search fails to find the optimal path when it encounters a local maxima and mistakes it for the global maxima. This is more likely the lower the value of k nodes to explore. It also fails on a puzzle with no solution, entering an infinite loop.

3. Experiments
    a.

    Here I generated limits from 10 to 500 in increments of 10 and ran the following simulation for each limit: iterate 100 times and each iteration create a random puzzle by making 100 random moves away from the solved position. Try to solve the puzzle using Astar with heuristic 2. If it is solved, increment solved_puzzles, if

it is not solved increment unsolved puzzles. After the 100 iterations are complete, store both variables in a df. I gathered the data below:

maxN	solved	unsolved
10	0	100
20	6	94
30	13	87
40	19	81
50	24	76
60	27	73
70	28	72
80	34	66
90	36	64
100	39	61
110	29	71
120	43	57
130	43	57
140	47	53
150	38	62
160	45	55
170	52	48
180	52	48
190	49	51
200	53	47
210	50	50
220	51	49
230	57	43
240	56	44
250	63	37
260	62	38
270	59	41
280	66	34
290	61	39
300	64	36
310	71	29
320	63	37
330	66	34
340	65	35
350	74	26
360	68	32
370	67	33
380	66	34
390	76	24
400	65	35

410	82	18
420	70	30
430	78	22
440	78	22
450	73	27
460	70	30
470	72	28
480	77	23
490	81	19

Clearly, as the number of maxNodes increases, so does the ratio of solved to unsolved puzzles. The ratio increased from a 6% solve rate at 20 maxNodes to an 81% solve rate at 490 maxNodes. As maxNodes increases, the search space is increased, making it more likely that the search will find the optimal solution. To ensure you find a solution or determine one does not exist, you want to set maxNodes to infinity, so the algorithm will explore the whole search space.

b. For the following two questions I will use the below data set. To generate this set I used the following code

```
df2 = pd.DataFrame(columns = ["h1NodeCount", "h2NodeCount", "beamCount", "H1 Time", "H2 Time", "Beam Time", "H1 H2 Time Diff"])
solver = PuzzleSolve()
for i in range(5):
    puzzle_AstarH1 = Node()
    puzzle_AstarH1.randomizeStateNoSeed(80)
    puzzle_AstarH2 = copy.deepcopy(puzzle_AstarH1)
    puzzle_beam = copy.deepcopy(puzzle_AstarH1)

    A1time = time.time()
    solver.master_node = puzzle_AstarH1
    final = solver.Astar("h1")
    A1time = time.time() - A1time
    h1length = len(solver.generate_path_directions(final))

    A2time = time.time()
    solver.master_node = puzzle_AstarH2
    final = solver.Astar("h2")
    A2time = time.time() - A2time
    h2length = len(solver.generate_path_directions(final))

    beamTime = time.time()
    solver.master_node = puzzle_beam
    final = solver.local_beam("15")
```

```
    beamTime = time.time() - beamTime
    beamlength = len(solver.generate_path_directions(final))

    df2.loc[i] = [h1length, h2length, beamlength, A1time, A2time, beamTime, A1time-A2time]
```

Dataset:

Trial	H1	h2	beam	H1 Time	H2 Time	Beam Time	H1 H2 Time Diff
1	13.0	13.0	13.0	2.057544	0.540978	0.125078	1.516566
2	7.0	7.0	7.0	0.016346	0.012048	0.027469	0.004298
3	15.0	15.0	15.0	1.746945	0.335510	0.152715	1.411435
4	8.0	8.0	8.0	0.020562	0.015701	0.035652	0.004861
5	14.0	14.0	14.0	0.446303	0.218518	0.133878	0.227785
6	12.0	12.0	12.0	0.266384	0.116176	0.094709	0.150208
7	14.0	14.0	14.0	0.700353	0.603761	0.130995	0.096592
8	14.0	14.0	70.0	1.884036	0.572640	4.389016	1.311396
9	16.0	16.0	16.0	3.893648	1.399813	0.269244	2.493835
10	8.0	8.0	8.0	0.029391	0.024670	0.057163	0.004721
11	10.0	10.0	10.0	0.120179	0.046476	0.084509	0.073703
12	9.0	9.0	9.0	0.038497	0.028267	0.048863	0.010230
13	5.0	5.0	5.0	0.004274	0.003371	0.008048	0.000903
14	16.0	16.0	26.0	2.283196	1.134764	0.500911	1.148432
15	22.0	22.0	22.0	5.731065	2.395727	0.354918	3.335338
16	18.0	18.0	26.0	5.262945	2.847553	0.598115	2.415392
17	6.0	6.0	6.0	0.011321	0.008891	0.017345	0.002430
18	8.0	8.0	8.0	0.022654	0.017872	0.035936	0.004782
19	20.0	20.0	30.0	48.953500	7.592402	0.681341	41.361098
20	18.0	18.0	18.0	16.432523	2.362710	0.239568	14.069813

Heuristic 2 is better than Heuristic 1. From an analytical standpoint, H2 is basically an improved version of H1. Whereas H1 identifies a tile and classifies it as misplaced, H2 does the same thing, but additionally gives the tile a ‘wrongness’ weight which gives more information on how close the puzzle is to complete. Doing some data analysis we can back up this claim. To analytically compare the heuristics,we need to compare the search time of A-star algorithm on a puzzle state using each heuristic. From the data above you can see that using H2, A-star usually completes faster for the same puzzle state than using H1. Specifically, A-star using H2 is on average 3.48 seconds faster than A-star using H1 (average of last column). We can definitively say that H2 is better than H1 in terms of speed. The only time H1 could possibly be better than H2 is when

the puzzle is in a nearly solved state, since calculating H1 is less time consuming than calculating H2.

c.

Since A-star always finds the optimal solution, it will always have the same solution length, because the same solution will be found regardless of heuristic used. What differs is the time taken to find the solution.

A-star using H1 and H2 have an average solution length of ~13 moves

A-star using H1 has an average time of ~4.5 seconds

A-star using H2 has an average time of ~1 second

Beam search using k=15 has an average solution length of ~17 moves

Since Beam search has a relatively small k, sometimes it will need to expand a large number of nodes to find the solution, as is seen in the data set

d. This data set contains the results of all searches (as lengths of solution) for a n=100 randomized puzzle. When a value is 0 that means a solution was not found. MaxNodes was set to 1000 here.

Idx	h1len	h2len	beamlen
0	13	13	13
1	0	0	23
2	0	15	15
3	0	0	50
4	0	0	21
5	10	10	10
6	0	15	15
7	0	0	23
8	0	0	54
9	8	8	8
10	9	9	9
11	8	8	8
12	0	0	24
13	8	8	8
14	15	15	15
15	14	14	14
16	0	0	53
17	0	14	14
18	13	13	13
19	14	12	12
20	13	13	13
21	10	10	10
22	6	6	6
23	11	11	11
24	0	0	48
25	0	0	16

26	0	0	0
27	0	0	34
28	0	0	23
29	0	0	24
...	...	...	...
70	0	16	16
71	0	0	21
72	0	0	58
73	13	13	47
74	0	0	21
75	0	0	39
76	6	6	6
77	0	0	17
78	5	5	5
79	0	0	20
80	9	9	9
81	0	14	14
82	14	14	14
83	8	8	8
84	0	0	25
85	8	8	8
86	0	0	23
87	0	0	22
88	0	0	29
89	14	14	14
90	0	0	61
91	0	0	33
92	0	0	24
93	10	10	10
94	12	12	12
95	12	12	12
96	0	0	19
97	0	0	18
98	0	0	22
99	0	14	14

A-star using H1 was able to solve 44/100 puzzles
A-star using H2 was able to solve 52/100 puzzles
Beam using k=15 was able to solve 98 puzzles

4. Discussion
    a. The aim is to find the shortest path to the solution from a given puzzle state. A-star is guaranteed to find the optimal solution if one exists, while beam has a

probable chance of finding a slightly suboptimal path rather than the global optimal path. The issue is that since A-star is guaranteed to find the optimal path, it must store more nodes and consequently iterate through those nodes. This leads to more space and time required than for beam search. Since beam search only stores k nodes at once, we can control the space requirement and therefore time requirement of the search, but again it is not guaranteed to find the optimal solution. It is really a tradeoff of space/time v consistency. From my understanding, we value consistency over space/time, so A-star is better for this problem. As I described earlier, H2 is better than H1, so we want A-star with H2.

b. Implementing A-star was relatively easy, since it is a pre defined algorithm. I ran into an issue where I wasn't storing the recursive heuristic value of a node, I was storing only a single nodes heuristic value, which wasn't giving me the global optimal solution, but it was a quick fix. Implementing beam was also relatively easy since the algorithm itself is simple and parts of the algorithm are similar to A-star. The most important process I think was developing the program structure, especially the Node class which made my life much easier when writing the algorithms and other methods.

   I was uncertain how to test A-star to make sure that the solution returned really is the optimal, and I'm still not 100% sure how to be 100% confident. What I did seems like a good heuristic, where we evaluate very simple states and check them. And adding one complex state, and comparing the value to the value returned by beam search with increasing k for the same state actually helped me find the above mentioned bug. Ensuring that beam never finds a more optimal path than A-star was my heuristic for determining the correctness of my algorithm.

1. Extra Credit

**To run the test file for a Cube you must change the line of code in the main function which is at the bottom of the python file. Replace "node" on the second to last line with "cube"

```
if __name__ == "__main__":
    try:
        filename = sys.argv[1]
        #filename = 'testA-star.txt'
    except:
        print("Please enter the name of a file with program commands")

    PuzzleSolver = PuzzleSolve("node")
    PuzzleSolver.run_commands(filename)
```

So I partially completed the extra credit. It took longer than I thought so I didn't have enough time to implement it fully. I created a Cube class that resembles the Node class in function, but instead of representing a state of the 8-puzzle, it represents a state of a 2x2 Rubik's cube. The state is represented by a 6x4 matrix, where each row represents a cube face, and each column represents a tile on the face. Look at the Cube class comments to see more specifics. The functions in the class are move, randomizeState, and some rotation helpers. The functions work correctly as is demonstrated in the test file I provided.

I also added some conditionals to the 8-puzzle code to check if a Cube node is being modified to handle it appropriately. Everything technically works, I can call beam search and A-star search on a Cube node. But if the cube begins in a state that is greater than n=2 random moves away from the solved state, the state space blows up and I don't have time to wait for the algorithms to finish. This is because I used a horrible heuristic that simply counts the number of unique tiles on each face and sums them up. The more unique tiles on each face, the farther it is from solved, so initially I thought this would be okay. Turns out not. When the state is greater than n=2 random moves away from solved state, my program runs out of memory or it runs for what I would guess is an extremely long time. If I had more time to implement a better heuristic, I would use the manhattan distance heuristic, which would decrease the size of the state space and result in quicker searches for beam and A-star search.