Chude Qian Project 1 Report

Code Design:

The code design mainly follows OOP design convention. There is a class eight_puzzle under file puzzle.py, there is a helper class frank_function which is under frank_generic, and lastly there is main.py which hosts the search algorithm, as well as the runner code. There is also a stresstest.py file which hosts all the different test files.

0. Overall:

Overall, the program is chosen the python language for its convenience in syntax as well as a lot of methods are simplified in python. As for the data structure, instead of some complicated data structure, I choose to use a simple string to represent the problem. Doing so has these advantage: 1. In python, string building and rebuilding is relatively simple, compared to Java. A lot of methods are already there for us to use. 2. Using string, I can simply append string into a list, instead of performing some nested list operation. With that being said, one might argue that string takes more memory. However, in these case since I are not creating a super long string, therefore I would argue it is more like a constant to the system memory. **Note: all test has been done using intel-4790k, 16GB ram.**

0.5 Frank_function:

In this class it is more like my personal library. There is customPrint which allows to print with different status pre-fix in ROS style. With tag includes [DEBUG], [INFO],[Warning],[Error],[Critical],[Fatal]. With setting Debug and Minimal, you can configure whether to see DEBUG printout or INFO printout. By default, both DEBUG and INFO printout are not displayed.

Under Frank_function there is also find index function which allows us to quickly find the index of a specific puzzle board. This is adapted from my other class find component. I was a bit lazy to combine it into puzzle.py and worried about reusability

1. Eight_puzzle class:

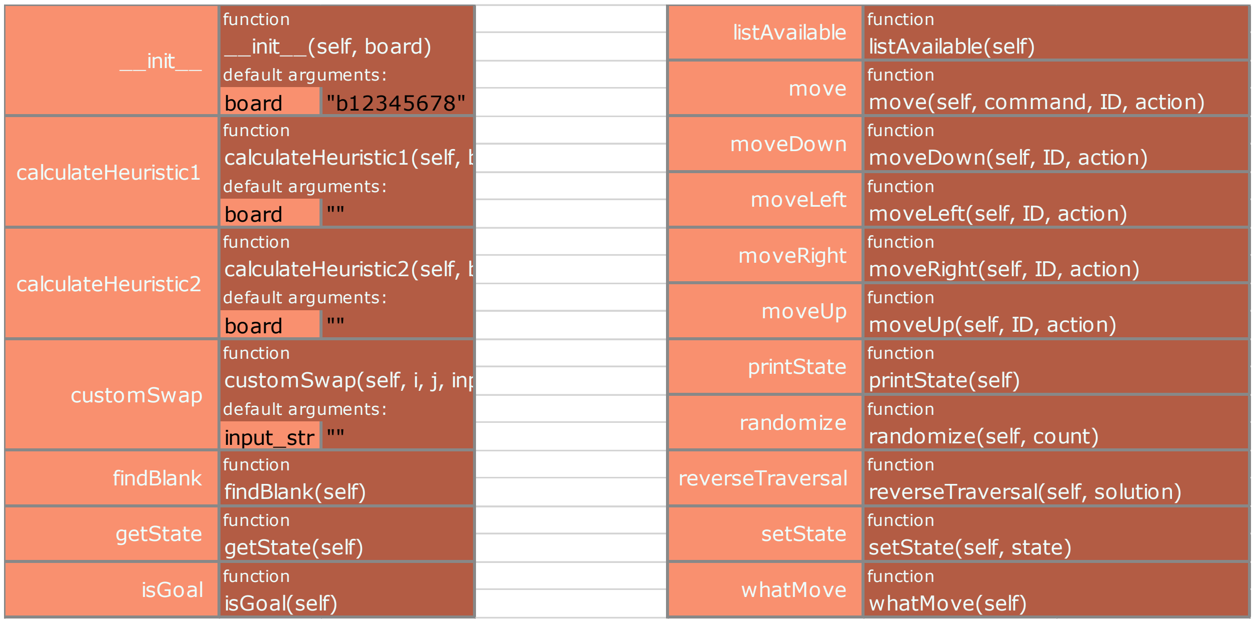
Under eight_puzzle class, there is __init__ which serve as the constructor. I can split the code into several function regions: There is the helper method for search algorithm, there is motion methods, and there is I/O methods.

For helper methods for search algorithm, I have: CalculateHeuristic1 and CalculateHeuristic2 which is responsible for calculating the h function according to the book. Function 1 is simply doing the difference and function 2 is more involved using city block distance. listAvailable function helps list all the available motion in an array, with a specific ordering. If a specific motion is not valid, it will show as empty content. This allows us to keep track of the motion. ReverseTraversal is a recursive function allow us to basically operate a linked list traversal from the goal state all the way back to the initial state. With the help of the function whatMove allows us to also keep track of each individual movements.

For motion methods there is the findBlank function which allows us to find where the blank is on the board. There are a total of 5 move functions, 4 of them controls movement individually in 4 directions, and one is the “master move” command. Each of the move command takes in an ID, which will be provided by findBlank function, and an action parameter. This allows us to separate whether I want to check validity or actually manipulate the board. customSwap function is a function we wrote helping the move functions. It basically takes in 2 ID and swaps them.

For generic helper function, there is getState function which allows to return the state directly. However eventually I realized we can directly access the state… So that function is not really used. Another helper function is randomize. Randomize function basically takes in a integer representing how many randomization step we need to perform, and then perform randomization to the board. printState function is responsible for printing the result in a nice manner. Set State is responsible for setting the board. It is capable of two styles of board input: “b12345678” or “b12 345 678”. Again, by using python’s split and joint function.

Below is the function extraction diagram



__init__	function __init__(self, board) default arguments:	board "b12345678"
calculateHeuristic1	function calculateHeuristic1(self, b default arguments:	board ""
calculateHeuristic2	function calculateHeuristic2(self, b default arguments:	board ""
customSwap	function customSwap(self, i, j, inp default arguments:	input_str ""
findBlank	function findBlank(self)	
getState	function getState(self)	
isGoal	function isGoal(self)	

listAvailable	function listAvailable(self)
move	function move(self, command, ID, action)
moveDown	function moveDown(self, ID, action)
moveLeft	function moveLeft(self, ID, action)
moveRight	function moveRight(self, ID, action)
moveUp	function moveUp(self, ID, action)
printState	function printState(self)
randomize	function randomize(self, count)
reverseTraversal	function reverseTraversal(self, solution)
setState	function setState(self, state)
whatMove	function whatMove(self)

2. Main.py:

Main python file is responsible for two functions:

Function ingest basically is a file opener. It is directly copied from my EECS 397 Python class homeowork.

Function checkinit basically checks the initialization status. If user did not do setState in command, it will warn you that you have not done that.

Then it’s the fun part, astar function. The basic design is: It first obtain (create) all the child state from the first of the openlist (which is initialized with the passed in state), calculate the heuristics and get the function value, populate them. It then compare with the open list and close list to make sure that if this is a state we have seen or not. If we have never seen this state, we append it to the open list. If we have seen this in the openlist, we then populate the heuristics in the openlist. If we have seen this in the closedlist, we then update its corresponding value. Once all of the child possibility of a given state in openlist is checked, we move this list into the close list, and then we sort the openlist to obtain the best state one and start the whole process again. If we hit the goal state, within our goal state, we should have a linked list like structure as we keep track of the parent child relationship, and then move backwards all the way to the start list. We then forward traversal one more time, so that we can output the result in the form of user consumable.

As for beam search, it is similar to the a* search, but the difference is we focus on actually getting all the childs available. Note that beamsearch has k limitation. Every child expansion completion, we sort the whole child list, and then we trim off those who are outside the best k ones. Once completed, the process is the exactly the same as a* search, where we traverse back and spit out the information via terminal.

Code Correctness:

It is generally hard to discuss code correctness without going into the details line by line. In here, we will proof by experiments and code interpretation.

For testing the eight_puzzle class's motion section, we can use brute force testing. I wrote a function simply asking for randomizing state(100) and repeat this process for 10000 times. We were able to: 1. Continually run this program without exception. 2. The generated result makes sense. 3. During small batch test, when we enable DEBUG tag and Minimal tag, we are able to see that it process the motion in a valid argument where it discusses where is valid where is invalid. If it is invalid, you will be able to see it respawn another request until fulfilled. Test code here:

```
def randomizeStressTest():
    current = eight_puzzle()
    with open('randomizeTest.csv', mode='w') as randomizeTest_File:
        RandomizeTest_writer = csv.writer(randomizeTest_File, delimiter=',', quot
echar='"', quoting=csv.QUOTE_MINIMAL)
        for i in range(0,10000):
            current.randomize(100)
            RandomizeTest_writer.writerow([current.getState()])
```

Test result can be found in randomizeTest.csv.

To test the key function of producing all possible child, we will use the assumption that our motions are all correct, by testing with the case of simply moving blank across the board, in total of 8 cases, we are able to see that the generated child are all valid. Therefore we can conclude that our listAvailable function is valid too.

Another key feature to test out is the command parser. To proof that, we simply utilizes the command Dr. Lewikie provided in class, and we were able to successfully get 2 run result. In addition, to further test randomization, the experiment is run constantly with me spam clicking the run button in vs code, no exception were thrown so I think it is good. See below:

```
[WARNING]setState Operation called
[WARNING]printState operation called
+-----------+
| b | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
[WARNING]Moving right!
[WARNING]printState operation called
+-----------+
| 1 | b | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
[WARNING]randomizeState command called
[WARNING]Request: 2 steps
[WARNING]printState operation called
+-----------+
| 1 | b | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
[WARNING]solve A-star command called
[WARNING]using h1 heuristic
====================Solution====================
Starting state
+-----------+
| 1 | b | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
move left
+-----------+
| b | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
[WARNING]Solution path:['left']
[WARNING]Solution length:1
[WARNING]randomizeState command called
[WARNING]Request: 10 steps
[WARNING]solve A-star command called
[WARNING]using h2 heuristic
====================Solution====================
Starting state
+-----------+
| 1 | 4 | 2 |
| 3 | 5 | b |
| 6 | 7 | 8 |
+-----------+
move left
+-----------+
| 1 | 4 | 2 |
| 3 | b | 5 |
| 6 | 7 | 8 |
+-----------+
move up
+-----------+
| 1 | b | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
move left
+-----------+
| b | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
[WARNING]Solution path:['left', 'up', 'left']
[WARNING]Solution length:3
[WARNING]solve beam command called
====================Solution====================
Starting state
+-----------+
| 1 | 4 | 2 |
| 3 | 5 | b |
| 6 | 7 | 8 |
+-----------+
move left
+-----------+
| 1 | 4 | 2 |
| 3 | b | 5 |
| 6 | 7 | 8 |
+-----------+
move up
+-----------+
| 1 | b | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
+-----------+
move left
+-----------+
| b | 1 | 2 |          | 6 | 7 | 8 |          [WARNING]Solution      [WARNING]Solution
                                              length:4               path:['left', 'up',
| 3 | 4 | 5 |          +-----------+                                 'left']
```

In addition, for the next part experiments, we are required to generate a batch of datas. In that case, we were able to see also that with a total of nearly 5k test samples, we had no exception crash. All unsolved are related to not enough maxNodes which I would then conclude that our algorithm are correct. In addition, as will illustrate in the next part, all 3 algorithm's solving are relatively close result. They will not be the same due to the algorithm design, but they are close.

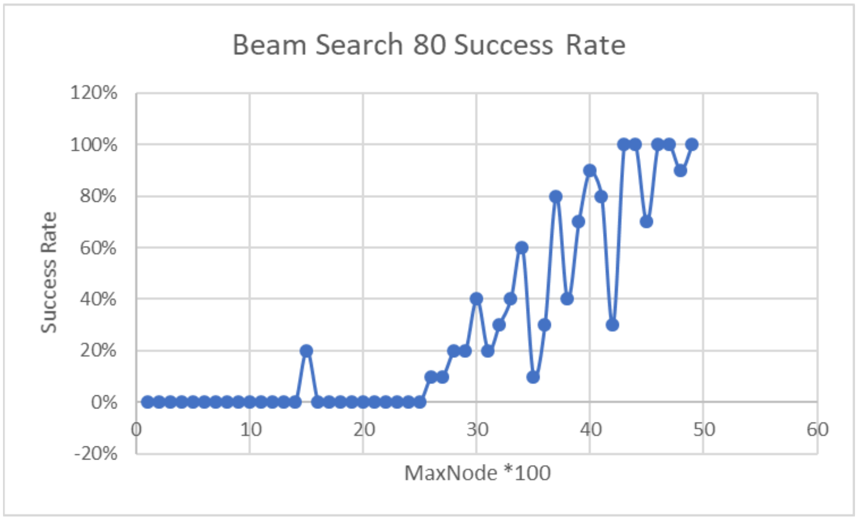
Experiments:

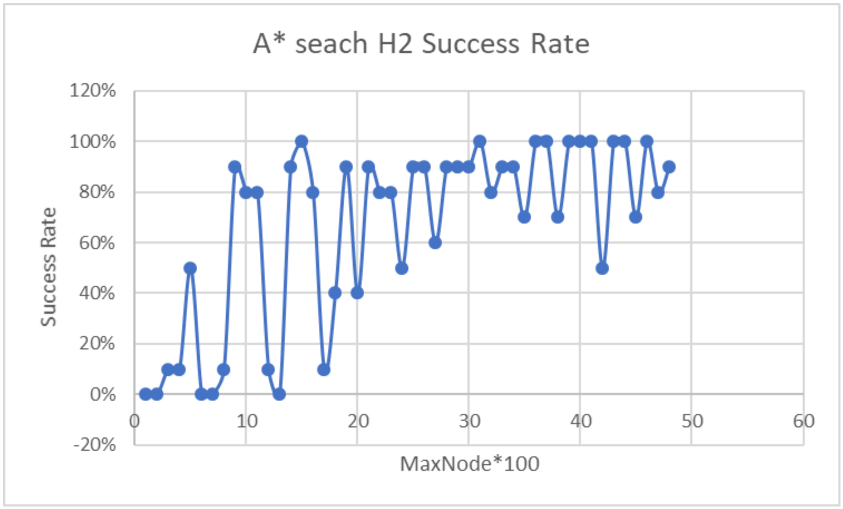
A. How does fraction of solvable puzzles from random initial states vary with the maxNode Limit?

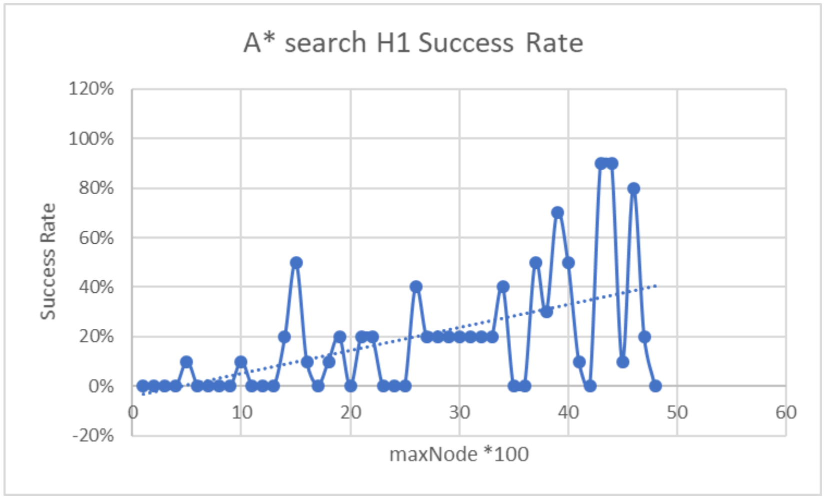
To test this, we wrote our test function:

```python
def maxNodesTest():
    with open('maxNodeTest.csv', mode='w') as maxNodeTest_file:
        current = eight_puzzle()
        for i in range(0, 100):
            for j in range(0,10):
                current.randomize(20)
                resulth1,steps1 = astar(current,"h1",i*100)
                resulth2,steps2 = astar(current,"h2",i*100)
                resultB,steps3 = beam(current,80,i*100)
                maxNodeTest_writer = csv.writer(maxNodeTest_file, delimiter=',',
quotechar='"', quoting=csv.QUOTE_MINIMAL)
                maxNodeTest_writer.writerow(['A-
Star h1',(i*100), resulth1, steps1])
                maxNodeTest_writer.writerow(['A-
Star h2',(i*100), resulth2, steps2])
                maxNodeTest_writer.writerow(['Beam 80',(i*100), resultB, steps3])
```

What it does is for each maxNode (0,100,200….1000) we do 10 test runs with 10 randomized board. We would like to see the result.







As illustrated above, you can clearly see that with the max node increasing, overall all the search algorithms are able to provide us a solution better. From the above trend we can conclude that H2 is able to give us the most solution even when maxNode amount is set to a relatively small value, and just from this graph, we can also conclude that under small maxNOde setting, H2 behave the best.

b. For A* search, which heuristic is better?

For this experiment, we stressed the solving capability of Astar with H1 and H2 algorithm solving the same randomized result. Code is follow:

```
def heruisticsOptionCompare():
    with open('heruisticsCompare.csv', mode='w') as heruisticsCompare_file:
        heruisticsCompare_writer = csv.writer(heruisticsCompare_file, delimiter='
,', quotechar='"', quoting=csv.QUOTE_MINIMAL)
        current = eight_puzzle()
        for i in range(0, 100):
            current.randomize(20)
            resulth1,steps1 = astar(current,"h1",10000)
            resulth2,steps2 = astar(current,"h2",10000)
            heruisticsCompare_writer.writerow(['A-Star h1',resulth1, steps1])
            heruisticsCompare_writer.writerow(['A-Star h2',resulth2, steps2])
```

Data can be found in HeruisticsCompare.

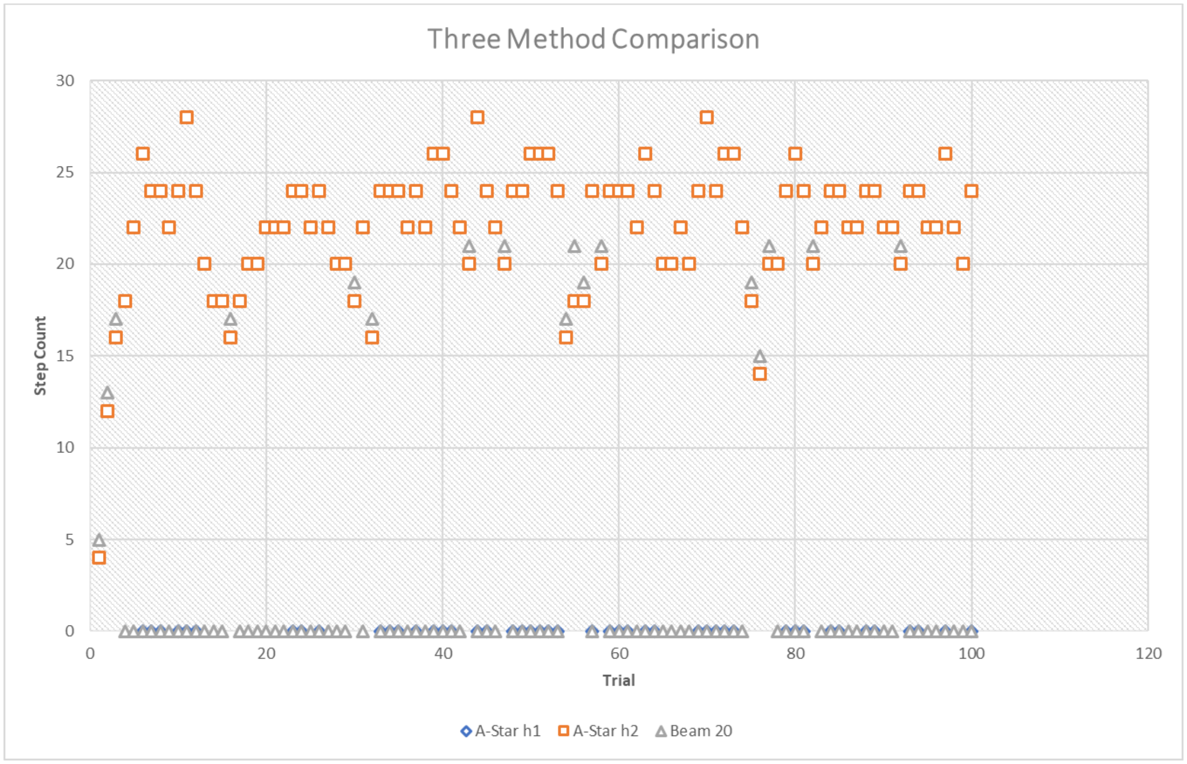
We observe that with the same condition: H1 has 24 unsolved, and H2 solved all successfully. This already says a lot about superiority. Now lets give H1 some advantage by clearing out all 0 entries. With that being said, the average for H1 is 20.5087 which is still larger than H2's 19.76. therefore we can say that H2 is better than H1

3. How does the solution length vary across the three search methods?

In this experiment we setup the experiment to stress test all three method under a same given randomized board. Code:

```
def solutionLengthCompare():
    with open('LengthCompare.csv', mode='w') as LengthCompare_file:
        LengthCompare_writer = csv.writer(LengthCompare_file, delimiter=',', quot
echar='"', quoting=csv.QUOTE_MINIMAL)
        current = eight_puzzle()
        for i in range(0, 100):
            current.randomize(20)
            resulth1,steps1 = astar(current,"h1",10000)
            resulth2,steps2 = astar(current,"h2",10000)
            result3,steps3 = beam(current,80,10000)
            result4,steps4 = beam(current,20,10000)
            LengthCompare_writer.writerow(['A-Star h1',resulth1, steps1])
            LengthCompare_writer.writerow(['A-Star h2',resulth2, steps2])
            LengthCompare_writer.writerow(['Beam 20',result4, steps4])
```

The maxNode are set to the same to control the environment. A total of 10000 test are conducted.



As illustrated above, you can see, A-Start with H2 is always under other methods. Which shows superiority. In addition, note how there are a lot of points where beam and h1 just simply won't be able to produce a valid result. Comparitively, we can see that H2 was able to successfully find a valid solution.

4. For each of the three search methods, what fraction of your generated problems were solvable.

This is a relative tricky question to answer. Obviously if you don't randomize the question too much the success rate will be high, but that does not represent anything. In this case, I propose a randomization of 20 steps, which falls under typical solvable step of under 30.

Using data produced by previous question we have:

A* search H1	54%
A* search H2	100%
Beam search	40%

Note that for the purpose of generating result quickly, maxNode was not set to a very high value but rather 10000.

Discussion:

a. Based on your experiments, which algorithm is better suited for this problem? Which finds shorter routes? Which algorithm seems superior in terms of time and space?

I think that A* search algorithm is generally better than beam search. Although beam search theoretically is able to produce a more comprehensive result. However, when you are physically space limited, beam search is not able to fully extend hence causing potential incompleteness. As illustrated above, H2 heuristics is able to produce the shortest route in use with A* search. Time complexity wise, I think A* search is also better.

b. Discuss any other observation you made.

During development, I misunderstand A* search to only look for the next stage. Causing problem since that effectively become a greedy algorithm. It was then corrected. It is interesting since the greedy algorithm always leads me to a local minimal and at that point, the success rate is ridiculously low and the path length is super super long.