

# 8 Puzzle using A-star and Local Beam Search

Mehlam Saifudeen

## **1. Code Design:**

The code contains 2 files which are named `Project1_final.py` and `text_input_8puzzle.py`. The first python file contains the code for the methods that are specified in the assignment which are the `setState`, `printState`, `move`, `maxNodes`, `randomizedState`, `solve A-star`, and `solve beam`. Besides these there are also other helper methods that are present in the `Project1_final.py` file such as the heuristics and total cost which will be explained later in the report. This code of solving the 8 star puzzle using A-star and local beam search is one of the fundamental problems in artificial intelligence that is the optimal solution path. The A-star search method was tested using 2 different heuristics and the local beam search with different sizes of beam width. This sort of an algorithm can be used in google maps and other applications that help us find the shortest path between 2 places. This entire project was coded in python and a part of it in jupyter notebook for testing. The `Project1_final.py` was coded as a class to represent the board of the 8 piece puzzle. The solution to the eight piece puzzle is a sequence of blank tile moves that eventually lead to the goal state being reached. For both the A-star and the local beam search algorithm, I have python dictionaries that keep a track of the states that have been reached in the puzzle. Here each node of the path keeps a track of the parent state, the child state and the path that takes from the parent to the child state. When the goal state of the 8 puzzle is reached we can traverse back through the nodes to the original state by moving the blank tile accordingly. The path of the solution can be displayed along with the length of the path taken to reach the goal state.

In addition to the 2 python files there are also some `.txt` files that have a sequence of commands that can be inputted and parsed into the code through the command prompt. At the same time I have included a `testing_code.ipynb` file which has the same `.txt` files and other commands that have been run in the command prompt to test for the set and search methods.

The code needs to take in `.txt` files as an input into the command prompt and parse the commands from in there. For this the `text_input_8puzzle.py` file which contains an argument parser library of python.

The following are the methods that have been included as a part of the code in order:

- a. **def \_\_init\_\_(self):** This acts as the constructor that helps initialize the seed, the goal state and the current state for the puzzle.
- b. **goalCheck (self, state):** This method checks if the current state of the puzzle is the completed goal state and if so will print that.
- c. **getAvailableActions (self, state):** This method is used to list all the available moves and the location of the blank tile. At the end of the method I randomly shuffle the available actions to avoid any bias.
- d. **setState (self, state):** This method is self explanatory in that it takes a state as an input which can be any legal configuration that exists in the domain of the 8 puzzle. Once it is confirmed that the state is a set of legal string characters the set is a state for further operations to be carried out.

```
def setState(self, state):
    # This method sets the state of the puzzle to a string in the format "b12 345 678"
    if len(state) != 11:
        print("The length of the state is incorrect")
    #list to keep track of the elements that have been added to board
    elements_added = []

    #Loop through all possible positions of state
    for row_index, row in enumerate (state.split(" ")):
        for column_index, element in enumerate(row):

            if element not in ['b', '1','2', '3', '4', '5', '6', '7', '8']:
                print("Character is not valid", element)
                break
            else:
                if element == "b":
                    if element in elements_added:
                        print("blank should only be added once")
                    else:
                        self.state[row_index][column_index] = 0
                        elements_added.append("b")
                else:
                    if int (element) in elements_added:
                        print("Tile {} has been added twice".format(element))
                        break
                    else:
                        self.state[row_index][column_index] = int(element)
                        elements_added.append(int(element))
```

- e. **printState(self, state), table\_print\_stat(self, state), and table\_print\_solution:** These 3 methods essentially have the same idea as in the first 2 are used to print the current state of the puzzle board. Once the puzzle has been set we can print the state using the printState method which prints the puzzle as a string. The table\_print\_state method is an additional method added to the code which helps present the state of the puzzle in a better way. Similarly the table\_print\_solution prints the solution path where each node is presented as a puzzle format.

```

def printState(self):
    final_state = [] #Display the final state of the board in the format b12 345 678
    # Iterate through the tiles
    for row in self.state:
        for element in row:
            if element == 0:
                final_state.append("b")
            else:
                final_state.append(str(element))
    print("".join(final_state[0:3]), "".join(final_state[3:6]), "".join(final_state[6:9]))

def table_print_state(self, state):
    print("\nCurrent State")
    for row in state:
        print("-" * 10)
        print("| {} | {} | {} |".format(*row))

def table_print_solution(self, solution_path):
    # Display the solution path in an puzzle table format
    try:
        # Solution path is in reverse order
        for depth, state in enumerate(solution_path[::-1]):
            if depth == 0:
                print("\nStarting State")

            elif depth == (len(solution_path) - 2):
                print("We have reached the goal!")
                for row_num, row in enumerate(state[0]):
                    print("-" * 13)

```

- f. **move (self, state, action):** This method is used to move the blank tile up, down, left or right. Although if the move is not a legal move an appropriate message is displayed. The move is made by a series of conditional statements and appended to a list.

```

def move(self, state, action):
    # This method moves the tile up, down, left, or right
    available_actions, blank_row, blank_column = self.getAvailableActions(state)

    next_state = copy.deepcopy(state)

    if action not in available_actions:
        print("This move is not allowed")
        return False

    else:
        if action == "up":
            tile_moved = state[blank_row - 1][blank_column]
            next_state[blank_row][blank_column] = tile_moved
            next_state[blank_row - 1][blank_column] = 0
        elif action == "down":
            tile_moved = state[blank_row + 1][blank_column]
            next_state[blank_row][blank_column] = tile_moved
            next_state[blank_row + 1][blank_column] = 0
        elif action == "right":
            tile_moved = state[blank_row][blank_column + 1]
            next_state[blank_row][blank_column] = tile_moved
            next_state[blank_row][blank_column + 1] = 0
        elif action == "left":
            tile_moved = state[blank_row][blank_column - 1]
            next_state[blank_row][blank_column] = tile_moved
            next_state[blank_row][blank_column - 1] = 0
    return next_state

```

- g. **randomizedState (self, n):** This method takes a random sequence of n legal moves and moves backwards from the goal state. The method will first set the state of the puzzle to the final goal state followed by stating the n random moves and picking one of them each time and adding it to the list. The state is then set to the result of the next move.

```

def randomizedState(self, n):
    # This method takes a random series of moves backwards from the goal state
    present_state = (self.goal_state)

    for i in range(n):
        available_actions, _, _ = self.getAvailableActions(present_state)
        random_move = random.choice(available_actions)
        present_state = self.move(present_state, random_move)

    # Set the state of the puzzle to the random state
    self.state = present_state

```

h. **calculate\_h1\_heuristic (self, state), calculate\_h2\_heuristic (self, state), calculate\_total\_cost (self, node\_depth, state, heuristic):**

Here the h1 heuristic refers to the method that calculates the number of tiles that are out of their position. We also convert the puzzle back to a list for comparison. The h2 heuristic refers to the Manhattan distance of all the tiles which is defined as the calculation of the absolute value of x and y difference of the current tile position from the goal tile position.

The total cost is the final helper method that returns the total cost and depth of state using the depth and the heuristic.

```
def calculate_h2_heuristic(self, state):
    # Helper method that calculates and returns the h2 heuristic for a given state
    # The h2 heuristic for the eight puzzle is defined as the sum of the Manhattan distances of all the tiles
    # Manhattan distance is calculated by absolute value of the x and y difference of the current tile position from the goal tile position

    current_tile = {}
    goal_tile = {}
    heuristic = 0

    # Create dictionaries of the current state and goal state
    for row_index, row in enumerate(state):
        for column_index, element in enumerate(row):
            current_tile[element] = (row_index, column_index)

    for row_index, row in enumerate(self.goal_state):
        for column_index, element in enumerate(row):
            goal_tile[element] = (row_index, column_index)

    for tile, position in current_tile.items():
        # Do not count the distance of the blank
        if tile == 0:
            pass
        else:
            # Calculate heuristic as the Manhattan distance
            goal_position = goal_tile[tile]
            heuristic += (abs(position[0] - goal_position[0]) + abs(position[1] - goal_position[1]))

    return heuristic

def calculate_total_cost(self, node_depth, state, heuristic):
    # Helper method that returns the total cost of a state using the depth and the heuristic
    if heuristic == "h2":
        return node_depth + self.calculate_h2_heuristic(state)
    elif heuristic == "h1":
        return node_depth + self.calculate_h1_heuristic(state)
```

## 2. Code Correctness

As mentioned in the introduction, this code uses 2 search methods to solve the 8 puzzles which are the A-star method and the local beam search. The snippets below display the crux of both the algorithms.

```

Astar_search(self, heuristic = "h2", max_nodes = 99, print_solution = True):
#This method does the A star search and returns the list of the solution and the length

frontier = {}
expanded_nodes = {}

self.starting_state = copy.deepcopy(self.state)
present_state = copy.deepcopy(self.state)
node_index = 0

expanded_nodes[node_index] = {"state": present_state, "parent": "root", "action": "start",
                              "total_cost": self.calculate_total_cost(0, present_state, heuristic), "depth": 0}

frontier[node_index] = {"state": present_state, "parent": "root", "action": "start",
                        "total_cost": self.calculate_total_cost(0, present_state, heuristic), "depth": 0}

invalid = False

# all_nodes keeps track of all nodes on the frontier and is the priority queue. Each element in the list is a tuple consisting of node index and total cost
all_frontier = [(0, frontier[0]["total_cost"])]

while not invalid:
    current_depth = 0
    for node_num, node in expanded_nodes.items():
        if node["state"] == present_state:
            current_depth = node["depth"]

    available_actions, _, _ = self.getAvailableActions(present_state)

    for action in available_actions:
        repeat = False

        if node_index >= max_nodes:
            invalid = True
            print("This solution could not be found in {} nodes".format(max_nodes))
            self.num_nodes_generated = max_nodes

solve_beam(self, k=1, max_nodes = 99, print_solution = True):
#This method performs the local beam search where k is the number of successors states to consider on each iteration. The evaluation function is the sum of h1 and h2 heuristics.

self.starting_state = copy.deepcopy(self.state)
starting_state = copy.deepcopy(self.state)

if starting_state == self.goal_state:
    self.success(node_dict = {}, num_nodes_generated = 0)

all_nodes = {}
node_index = 0

all_nodes [node_index] = {"state": starting_state, "parent": "root", "action": "start"}

starting_score = self.calculate_h1_heuristic(starting_state) + self.calculate_h2_heuristic(starting_state)

# Available nodes is all the possible states that can be accessed from the current state stored as an (index, score) tuple
available_nodes = [(node_index, starting_score)]

invalid = False
valid = False

while not invalid:
    if node_index >= max_nodes:
        invalid = True
        print("The solution was not found in the first {} generated nodes".format(max_nodes))
        break

    successor_nodes = [] # These can be reached from all the available states

    for node in available_nodes:
        repeat = False
        present_state = all_nodes[node[0]]["state"]

        available_actions, _, _ = self.getAvailableActions(present_state)

```

Below are some of the .txt files that were run in the command prompt of the code that generated the outputs for the set methods as well as the search methods. These were run in jupyter notebook where the ipynb and the txt files were present in the same folder as the 2 python code files.

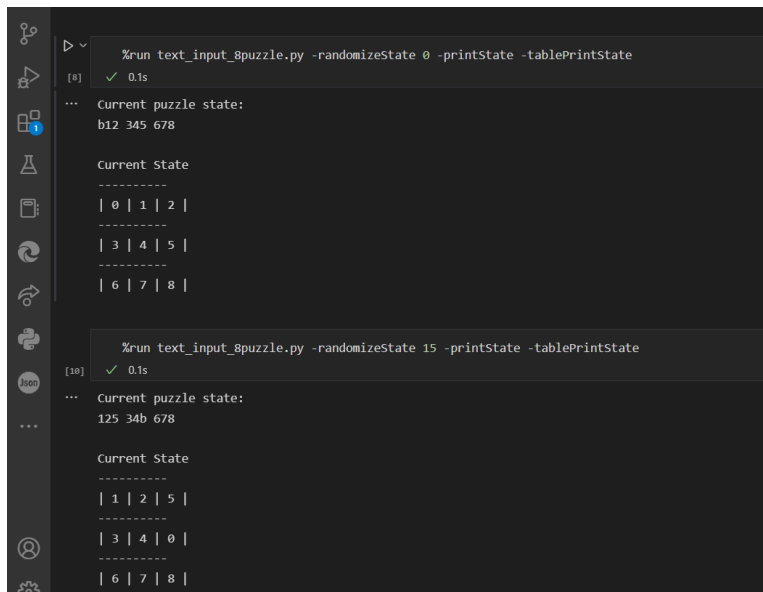
**Astar\_search:** is a method that runs a loop from the start to the end that is until the right state is found or the maximum number of nodes have been reached. On each iteration the total number of available moves are searched through and also checks which states have already been reached and does not expand those ones again. The value of the state based on the heuristic is calculated and the best state is added to the priority

queue, which is sorted by cost. Either one of two heuristics can be used for the A star search method. The first one is calculating the moves of the blank tile and the second one is to calculate the sum of Manhattan distance which can be calculated by the absolute value of the x and y difference of the current file position from its goal state position. The best state is then taken out from the front of the priority queue and expanded. The state is then added to the dictionary. The states that have not been expanded are added to the frontier to keep track of those states which have been added but not expanded. Lastly the goal state is checked when expanded and not added as there may be a path with a lower cost to reach that goal state. The total number of nodes expanded is the sum of frontier and the expanded nodes.

**solve\_beam:** Here too the loop runs till the goal state is reached or the maximum number of nodes have been expanded. The successor states are evaluated and added to a queue using the local beam search algorithm. By the end of the search the k best neighbor states are retained from the list of all states. These k best states then become the current states which are then expanded and these states are added to the dictionary as well. Here the check for the goal state occurs when the state is generated because when the goal state is found it ends the local beam search.

**Below are some of the snippets of the code that was run in the ipynb file to test the methods and run some of the experiments for part 3 as well.**

The method below shows the operation of the set\_state method and the table\_print\_state which displays the current state of the board in the form of a puzzle.



```
%run text_input_8puzzle.py -randomizeState 0 -printState -tablePrintState
[8] ✓ 0.1s
... Current puzzle state:
b12 345 678

Current State
-----
| 0 | 1 | 2 |
-----
| 3 | 4 | 5 |
-----
| 6 | 7 | 8 |

%run text_input_8puzzle.py -randomizeState 15 -printState -tablePrintState
[10] ✓ 0.1s
... Current puzzle state:
125 34b 678

Current State
-----
| 1 | 2 | 5 |
-----
| 3 | 4 | 0 |
-----
| 6 | 7 | 8 |
```

The below snippets show the test cases of the Astar\_search method for both the h1 and the h2 heuristics. We can see a drastic difference in the total nodes generated.

```
%run text_input_8puzzle.py -setState "321 574 688" -solveAStar h1 -tablePrintSolution
[14] ✓ 0.5s

... Solution is below
Solution length: 13
Solution Path ['start', 'up', 'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'goal']
Total nodes generated: 524

Starting State
-----
| 3 | 2 | 1 |
-----
| 5 | 7 | 4 |
-----
| 6 | 0 | 8 |
-----

The current depth is: 1
-----
| 3 | 2 | 1 |
-----
| 5 | 0 | 4 |
-----
| 6 | 7 | 8 |
-----

The current depth is: 2
-----
| 3 | 0 | 1 |
-----
| 5 | 2 | 4 |
-----
```

```
%run text_input_8puzzle.py -setState "321 574 688" -solveAStar h2 -tablePrintSolution
[15] ✓ 0.2s

... Solution is below
Solution length: 13
Solution Path ['start', 'up', 'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'goal']
Total nodes generated: 213

Starting State
-----
| 3 | 2 | 1 |
-----
| 5 | 7 | 4 |
-----
| 6 | 0 | 8 |
-----

The current depth is: 1
-----
| 3 | 2 | 1 |
-----
| 5 | 0 | 4 |
-----
| 6 | 7 | 8 |
-----

The current depth is: 2
-----
| 3 | 0 | 1 |
-----
| 5 | 2 | 4 |
-----
```

```
-----
| 1 | 4 | 2 |
-----
| 3 | 5 | 0 |
-----
| 6 | 7 | 8 |
-----

The current depth is: 11
-----
| 1 | 4 | 2 |
-----
| 3 | 0 | 5 |
-----
| 6 | 7 | 8 |
-----

The current depth is: 12
-----
| 1 | 0 | 2 |
-----
| 3 | 4 | 5 |
-----
| 6 | 7 | 8 |
-----

We have reached the goal
-----
| 0 | 1 | 2 |
-----
| 3 | 4 | 5 |
-----
| 6 | 7 | 8 |
-----
```

The snippets below show examples of the local beam search with 2 different k values and the output for the same. We can see the drastic difference in the number of nodes generated and the solution length for both the k values.

```

%run text_input_8puzzle.py -setState "321 574 6b8" -solveBeam 10 -tablePrintSolution
✓ 0.2s

Solution is below
Solution Length: 31
Solution Path ['start', 'up', 'right', 'up', 'left', 'down', 'left', 'up', 'right', 'down', 'right', 'up', 'left', 'down', 'right', 'up', 'left', 'left', 'down', 'right', 'up',
'right', 'down', 'left', 'left', 'up', 'right', 'down', 'right', 'up', 'left', 'left', 'goal']
Total nodes generated: 418

Starting State
-----
| 3 | 2 | 1 |
-----
| 5 | 7 | 4 |
-----
| 6 | 0 | 8 |
-----

The current depth is: 1
-----
| 3 | 2 | 1 |
-----
| 5 | 0 | 4 |
-----
| 6 | 7 | 8 |
-----

The current depth is: 2
-----
| 3 | 2 | 1 |
-----
| 5 | 4 | 0 |
-----

```

```

%run text_input_8puzzle.py -setState "321 574 6b8" -solveBeam 1000 -tablePrintSolution
✓ 7.3s

Solution is below
Solution Length: 13
Solution Path ['start', 'up', 'up', 'right', 'down', 'left', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'goal']
Total nodes generated: 3453

Starting State
-----
| 3 | 2 | 1 |
-----
| 5 | 7 | 4 |
-----
| 6 | 0 | 8 |
-----

The current depth is: 1
-----
| 3 | 2 | 1 |
-----
| 5 | 0 | 4 |
-----
| 6 | 7 | 8 |
-----

The current depth is: 2
-----
| 3 | 0 | 1 |
-----
| 5 | 2 | 4 |
-----

```

Below are some examples of commands added in .txt files which are parsed into the python code. The command line statement and the output is shown. These text file are attached into the same zip folder as the python and the ipynb files as well:

- a. setState  
"3b2 615 748"  
printStats  
solveBeam 20  
maxNodes 2000



```
▷ %run text_input_8puzzle.py -readCommands "randomize_solve_beam_2.txt"
[9] ✓ 0.1s
... Current puzzle state:
3b2 615 748
Solution is below
Solution Length: 5
Solution Path ['start', 'down', 'down', 'left', 'up', 'up', 'goal']
Total nodes generated: 50
```

- b. randomizeState 16  
solveAStar h2  
tablePrintSolution

```
▷ %run text_input_8puzzle.py -readCommands "randomize_solve_Astar.txt"
[14] ✓ 0.1s
... Solution is below
Solution Length: 2
Solution Path ['start', 'left', 'left', 'goal']
Total nodes generated: 5
```

- c. setState  
"b12 345 678"  
printState  
move "up"  
move "left"  
setState  
printState  
"1b2 345 678"  
move "up"

```
▷ %run text_input_8puzzle.py -readCommands "P1-test.txt"
[16] ✓ 0.1s
... Current puzzle state:
b12 345 678
This move is not allowed
This move is not allowed
The length of the state is incorrect
Character is not valid p
Character is not valid "
Current puzzle state:
b12 345 345
This move is not allowed
```

- d. randomizeState 16  
printState  
tablePrintState

```
▶ %run text_input_8puzzle.py -readCommands "randomize_print_table.txt"
[19] ✓ 0.1s
... Current puzzle state:
12b 345 678
```

### 3. Experiments

- a. The fraction of solutions from random initial states that can vary with the maxNodes limit can be large as the greater the number of random moves that are made to state, the greater number of nodes that need to be visited in order to reach the goal. This leads to maxNodes being larger in order for the algorithm to find the correct solution. This mainly happens because as the randomization is increased the initial state grows away from the intended goal state. This causes the algorithm to expand deeper and visit more of the expanded nodes to check if that is the final solution. If the maxNodes is a small value then it limits the number of nodes that can be expanded by the algorithm and if the randomization value is extremely large then the solution will most likely not be found. If the maxNodes value is large then it does not matter if the randomization value is small or large as there is enough room to find the solution even then. Even if the maxNodes limit is at or within a few standard deviations of the randomization value, the chances of finding the solution remain high. Upon randomizing 10000 times and keeping the maxNodes as 1000 no solution was found using the h2 heuristic of the A-star algorithm. The result was the same for h1 and the solveBeam with  $k = 10$  and  $100$  as well. But when the value of maxNodes was increased to 10000, the solution was found in 22 lengths for the h2 heuristic. The total nodes generated were 1988. The result was found in 22 lengths for the h1 heuristic as well except the number of nodes expanded was 9879. The solution was found with solveBeam  $k = 100$  in 34 lengths with 3975 nodes being expanded and 36 lengths and 491 nodes expanded when  $k = 10$  as less neighbors are added to the queue for this one. A part of this is included in the ipynb file.
- b. For the A star search the h2 Manhattan distance heuristic is better in all cases than the h1 heuristic that calculates the number of files that are out of place from their final position. This is seen in one of the example cases above. The h1 heuristic has a larger space complexity than the h2 heuristic. The solution length for both h1 and the h2 heuristic is the same, that is 13 for the state "321 574 6b8". But the number of nodes expanded for the h1 heuristic is 524 whereas for the h2 heuristic is 213 which is half the number of expanded nodes. This makes the h2 heuristic more efficient than the h1 heuristic.

- c. Across the 3 searches the number of solution lengths only varies between A star search as a whole and the local beam search. From the example in part B we can see that the solution length is the same for both the h1 and the h2 heuristic although the h2 heuristic was still more efficient than the h1 because it expanded less total number of nodes than the h1 heuristic. Although for the local beam search the solution length varied according to the value of k which is the number of neighboring states. In an example above for a setState "321 574 6b8" where the local beam search was applied with  $k = 10$  the solution length was 31 whereas when  $k = 1000$  the solution length was 13. Although this does not mean that a higher k value is necessarily better because the total number of nodes expanded for when  $k = 10$  is 418 and for when  $k = 1000$  is 3453. This is because in the first case less number of neighboring states are expanded whereas in the second case the number of neighboring states added to the queue is 100 times more which exponentially increases the number of states added to the queue making the search algorithm less efficient and more complex in terms of space.
- d. A\* is a complete algorithm and is guaranteed to find the solution to the problem. It is ensured that only legal moves are made during the randomization of the puzzle state. This leads to the solution always being found because it is within a set of legal moves. There is a full guarantee that the solution will be found to a problem when using A star search,  
When there is a maxNodes limit set the h2 heuristic is more probable to find the solution than the h1 heuristic because it is more efficient in terms of space complexity. For a sample of 500 randomized states when the maxNodes limit is set to half the number of randomized states the results observed are that the solution is found 25 percent of the times in the h1 heuristic and 60 percent of the times using the h2 heuristic.

#### 4. Discussion

- a. Based on my experiments the algorithm that would better be suited for this problem would be the h2 (Manhattan) distance heuristic over the h1 heuristic. This is because it has the least number of nodes generated as compared to h1 heuristic and the local beam search when both  $k = 1000$  and  $k = 10$ . This leads to a shorter and more efficient solution to the goal state. It also leads to a faster run time and more optimal space complexity as well making the h heuristic which is the Manhattan distance of the A star search the best algorithm for the 8 puzzle amongst the ones in the code.
- b. There was definitely a lot of difficulty in testing and running every part of the search algorithm as it required use of simple cases amongst the harder ones from the start as some of the test cases had a lot of nodes generated which led to an extended run time. The runtime of most of the search and looping parts of the code is  $O(n)$  which leads to a linear increase in time complexity depending on

the number of nodes generated for each of the search algorithms. In addition to this implementing the `randomizeState` involves some complexity as it depends on the input of the number of moves from the goal state which has an effect on the branching factor of the algorithm. In addition to this the `maxNodes` that can be expanded also is a factor that determines if a certain number of randomized moves can take place or not.