8-puzzle Solver

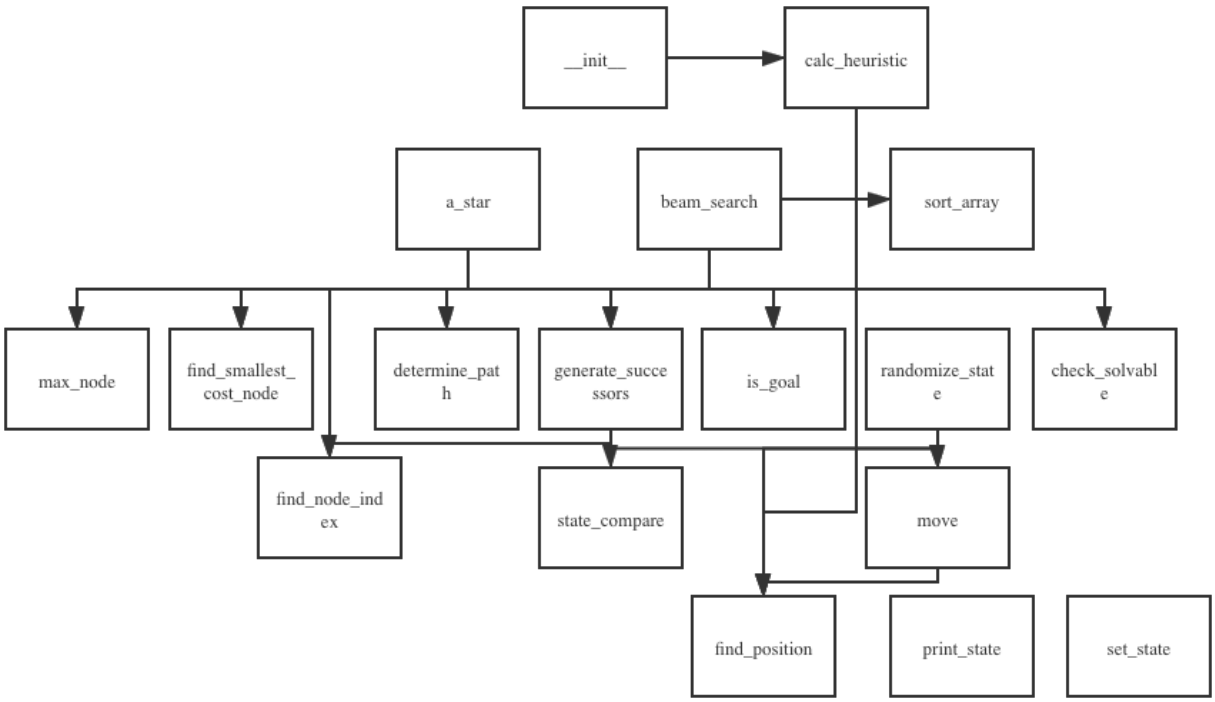
Ningjia Huang

The goal of this program is to solve 8-puzzle using two informed searching algorithms: A-star search and local beam search.

1. Code Design

1.1 Overview

I consider each state as a node with attributes current_state, max(maximum nodes that can be expanded which is specified by the user), gCost, parent, goal, and hCost. I write these two searching algorithms as two methods in each node. Commands will be read from a text file and executed line by line. A global list variable *expanded* is used to keep track of the expanded nodes so that visited node won't be visited again. The following figure describes the relationship between each function. The arrow between two functions means function 1 "uses" function 2.



The following subsections of section 1 briefly pointed out the essential functions in this program.

1.2 Read File from a Text File: set_state(puzzle)

The commands in the text file("text.txt") will be read in the main method by using the if statement. The basic idea is to read from the text file line by line and seperate each command by whitespace. Since each line consists of method name and parameters, string[0] is used to identify the method, and the rests are considered as parameters. **Note that in the text file, the "B" should be in uppercase.**

```
for character in line:
    file = open('text.txt','r')
    lines = file.readlines()
    for line in lines:
        line = line.strip()
        string = line.split(' ')
        if string[0] == "setState":
            node.setstate(string[1],string[2],string[3])
            print(node.currentstate)
        elif string[0] == "printState":
            node.printstate()
        elif string[0] == "move":
            direction = ''
            if string[1] == "up":
                direction = 'u'
            elif string[1] == "down":
                direction = 'd'
            elif string[1] == "left":
                direction = 'l'
            elif string[1] == "right":
                direction = 'r'
            else:
                print("Wrong direction.")
            node.currentstate = node.move(direction,node.currentstate)
        elif string[0] == "randomizeState":
            randomizestate(int(string[1]),node)
        elif string[0] == "solve":
            method = string[1]
            if string[2] == "h1":
                heuristic = 1
            if string[2] == "h1":
                heuristic = 1
            elif string[2] == "h2":
                heuristic = 2
            else:
                heuristic = 0
            if method == "A-star":
                node.hCost = node.calcheuristic(heuristic)
                node.astar(100000000)
            elif method == "beam":
                node.beam_search(int(string[2]), 10000000)
        elif string[0] == "maxNode":
            node.max = int(string[1])
```

1.3 Unsolvable Cases: check_solvable()

According to our textbook, only about half of the states may reach the goal state. In order to handle the unsolvable state and prevent to go into , a method called check_solvable() is implemented in order to check whether a puzzle is solvable or not. If it is not solvable, an error message will be printed. The check_solvable() method uses the fact that if the parity of inversions in a state is odd, then the puzzle is unsolvable and vice versa. The essential logic of this implementation is as following:

```
for i in range(len(originrep)):
    for j in range(len(goalrep)):
         if(origin_rep[i] == goal_rep[j]):
             for i_pointer in range(i+1,len(origin_rep)):
                if i < len(origin_rep)-1:
                    if(origin_rep[i_pointer] in goal_rep[:j]):
                        num_inversion += 1
parity = num_inversion % 2
if(parity == 0):
    return True
else:
    print("The puzzle given is unsolvable.")
    return False
```

1.4 Move the Tile: move(direction)

This function moves the tile by 1 place(either up, down, left, or right). It first uses find_position() method to find the position of the tile, then perform the moving:

```
if direction == 'u':
    copy2[row][col] = copy2[row-1][col]
    copy2[row-1][col] = 'B'
elif direction == 'd':
    copy2[row][col] = copy2[row+1][col]
    copy2[row+1][col] = 'B'
elif direction == 'l':
    copy2[row][col] = copy2[row][col-1]
    copy2[row][col-1] = 'B'
elif direction == 'r':
    copy2[row][col] = copy2[row][col+1]
    copy2[row][col+1] = 'B'
else:
    raise Exception("Undefined action.")
```

1.5 Generate Successors of a Node: generate_successors()

This method generates the successors of the current node by moving the tile to all possible neighboring places. If the tile is on the edges, there will be some restrictions on the direction that it moves.

```
successors = []
i,j = findposition('B',self.currentstate)
    if i > 0:
successors.append(Node(self.move('u',self.currentstate),self.gCost+1,self
))
        if i < 2:             successors.append(Node(self.move('d',self.c
urrentstate),self.gCost+1,self))
        if j > 0:             successors.append(Node(self.move('l',self.c
urrentstate),self.gCost+1,self))
        if j < 2:             successors.append(Node(self.move('r',self.c
urrentstate),self.gCost+1,self))
        if self.parent == None:
            return successors
        else:
            for successor in successors:
                if statecompare(successor.currentstate, successor.parent.
parent.currentstate):
                    successors.pop(findnode_index(successor, successors))
            return successors
```

1.6: Calculate Heuristic h1 and h2: calc_heuristic(heuristic)

This method contains two heuristics: **h1**, which counts the number of misplaced tiles; and **h2**, which counts the sum of the distances of the tiles from their goal position.

```
if heuristic==2:
    if self.isgoal():
        return 0
    else:
        for i in range(len(self.currentstate)):
        for j in range(len(self.currentstate[i])):
            if self.currentstate[i][j] != self.goal[i][j]:
                    rep = self.currentstate[i][j]
                    igoal, jgoal = findposition(rep, self.goal)
                    hCost += abs(igoal - i) + abs(jgoal - j)
    return hCost
elif (heuristic == 1):
    for i in range(len(self.currentstate)):
        for j in range(len(self.currentstate[i])):
        if (self.current_state[i][j] != self.goal[i][j]):
            hCost += 1
    return hCost
```

1.7 Randomize State: randomize_state(n)

This function takes *n* random moves from the goal state in order to assure the state that is generated is solvable. We take random move by randomly generating an integer between 0 and 3, representing "up", "down", "left", and "right". If the final state is the same as the state of input node, then regenerate it.

```
def randomizestate(n,node):
    node.currentstate = node.goal
    movementlist = ['u','d','l','r']
    movementhistory = []
    for k in range(n):
        integer = random.randint(0,3)
        i,j = findposition('B',node.currentstate)
        while (i == 0 and integer == 0) or (i == 2 and integer == 1) or (
j == 0 and integer == 2) or (j == 2 and integer == 3):
            integer = random.randint(0,3)
        node.currentstate = node.move(movementlist[integer],node.currents
tate)
        movementhistory.append(movementlist[integer])
    if(node.currentstate == node.goal):
        return randomizestate(n,node)
    else:
        print("Generated state: ", node.currentstate)
        print("The movements from goal been taken is: ", movementhistory)
        return node.currentstate
```

2. Code Correctness

1. A* Search

A* search always expands the one with the lowest evaluation function in the queue. It will examine the queue until the goal state is reached or there is no element in the queue. Note that if the current node has the same state as a node that is already expanded, we need to keep the one with the lower evaluation function. Same for the queue.
For expanded:

```
for successor in successors:
    if successor in expanded:
        for node in expanded:
            if successor == node:
                if successor.gCost + successor.hCost < node.gCost + node.
hCost:
                    node.gCost = successor.gCost
                    node.hCost = successor.hCost
                    node.parent = successor.parent
                    queue.insert(0,node)
                    expanded.pop(findnodeindex(node,expanded))
                    deal = True
```

For Queue:

```
if successor in queue:
    for element in queue: #condition: when the current node is in queue
        if successor.currentstate == element.currentstate: #if unexpanded
 nodes have the same current state, keep the one with the lower cost
            if successor.gCost + successor.hCost < element.gCost + elemen
t.hCost:
                element.gCost = successor.gCost
                element.hCost = successor.hCost
                element.parent = successor.parent
                deal = True
```

The A* search will always find the solution to a solvable problem unless the number of expanded nodes exceeds the maxNode.

1.1 Normal Short Case(h2):

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

The solution of this input should be short: down -> left -> up -> up.

```
Searching using A* search...
[['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']]
Goal state reached, # of moves:  4
```

1.2 Normal Long Case(h2):

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

Based on my research on the internet, the solution of this input should have more than 20 steps.

```
Searching using A* search...
[['8', '6', '7'], ['2', '5', '4'], ['3', 'B', '1']]
[['8', '6', '7'], ['2', 'B', '4'], ['3', '5', '1']]
[['8', '6', '7'], ['2', '4', 'B'], ['3', '5', '1']]
[['8', '6', '7'], ['2', '4', '1'], ['3', '5', 'B']]
[['8', '6', '7'], ['2', '4', '1'], ['3', 'B', '5']]
[['8', '6', '7'], ['2', 'B', '1'], ['3', '4', '5']]
[['8', 'B', '7'], ['2', '6', '1'], ['3', '4', '5']]
[['B', '8', '7'], ['2', '6', '1'], ['3', '4', '5']]
[['2', '8', '7'], ['B', '6', '1'], ['3', '4', '5']]
[['2', '8', '7'], ['3', '6', '1'], ['B', '4', '5']]
[['2', '8', '7'], ['3', '6', '1'], ['4', 'B', '5']]
[['2', '8', '7'], ['3', 'B', '1'], ['4', '6', '5']]
[['2', 'B', '7'], ['3', '8', '1'], ['4', '6', '5']]
[['B', '2', '7'], ['3', '8', '1'], ['4', '6', '5']]
[['3', '2', '7'], ['B', '8', '1'], ['4', '6', '5']]
[['3', '2', '7'], ['4', '8', '1'], ['B', '6', '5']]
[['3', '2', '7'], ['4', '8', '1'], ['6', 'B', '5']]
[['3', '2', '7'], ['4', 'B', '1'], ['6', '8', '5']]
[['3', '2', '7'], ['4', '1', 'B'], ['6', '8', '5']]
[['3', '2', 'B'], ['4', '1', '7'], ['6', '8', '5']]
[['3', 'B', '2'], ['4', '1', '7'], ['6', '8', '5']]
[['3', '1', '2'], ['4', 'B', '7'], ['6', '8', '5']]
[['3', '1', '2'], ['4', '7', 'B'], ['6', '8', '5']]
[['3', '1', '2'], ['4', '7', '5'], ['6', '8', 'B']]
[['3', '1', '2'], ['4', '7', '5'], ['6', 'B', '8']]
[['3', '1', '2'], ['4', 'B', '5'], ['6', '7', '8']]
[['3', '1', '2'], ['B', '4', '5'], ['6', '7', '8']]
[['B', '1', '2'], ['3', '4', '5'], ['6', '7', '8']]
Goal state reached, # of moves:  27
```

1.3 Exceed Maximum node(h2):

With the test case in section 1.2, we specify the maximum node(e.g. 10) as a parameter in a_star() method. The output will be:

```
Searching using A* search...
Visited nodes exceed the maximum limit. Max limit: 10
```

1.4 Not Solvable Case(h2):

The check_solvable() method assures the method will not run forever. If the test case is unsolvable, for example:

```
Searching using A* search...
[['8','1','2'],['B','4','3'],['7','6','5']]
```

The output will be:

```
The puzzle given is unsolvable.
```

2. Local Beam Search

For local beam search, the evaluation function I used is h2, which counts the sum of the distances of the tiles from their goal positions. A dictionary is used to keep track of the expanded nodes in order to avoid being visited again. The keys for the dictionary are the tuple forms of the state of each node(e.g. if the state of node is `[['1','2','3'],['B','4','5'],['6','7','8']]`, then it will be converted into `(('1','2','3'),('B','4','5'),('6','7','8'))`) because list cannot be used as key in dictionary.
The implementation of this method can be divided into two parts:
1) add k non-explored successors into the queue:

```
while(len(queue) != 0 and found == False):
    while(len(queue) != 0):

        if(maxnode(n,numcounter)):
            return
          current_node = queue.pop()
          if tuple(map(tuple,current_node.current_state)) not in explored
_map.keys():
            exploredmap[tuple(map(tuple,current\node.current_state))] = c
urrent_node
          else:
            continue
          if(currentnode.is_goal()):
            found = True
                print(*current_node.determine_path(),sep="\n")
                print("Goal state reached, # of moves: ", len(current_nod
e.determine_path()))
                    return
          else:
            successors = current_node.generate_successors()
                for successor in successors:
                    tup = tuple(map(tuple,successor.currentstate))
                    if explored_map.get(tup) != None:
                    successors.pop(find_node_index(successor,successors))
                        num_counter += 1
                        children_k.extend(successors)
            children_k = sort_array(children_k)
            children_k = children_k[:k]
```

2) check whether there is a goal state in the k successors:

```
for j in range(len(children_k)):
    if(children_k[j].current_state == children_k[j].goal):
        found = True
        print(*children_k[j].determine_path(),sep="\n")
        print("Goal state reached, # of moves: ",       len(current_node.
determine_path()))
        return
```

There are two reasons that the local beam search may not find the solution: 1) the choice of *k* is not proper so the solution may be cut off; 2) the maxNode is exceeded.

2.1 Normal Short Case

Using the same input as in section 1.1, we get:

```
Searching using local beam search...
[['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']]
Goal state reached, # of moves:  4
```

2.2 Normal Long Case

Using the same input as in section 1.2 with k = 800:

```
Searching using local beam search...
[['8', '6', '7'], ['2', '5', '4'], ['3', 'B', '1']]
[['8', '6', '7'], ['2', 'B', '4'], ['3', '5', '1']]
[['8', '6', '7'], ['2', '4', 'B'], ['3', '5', '1']]
[['8', '6', '7'], ['2', '4', '1'], ['3', '5', 'B']]
[['8', '6', '7'], ['2', '4', '1'], ['3', 'B', '5']]
[['8', '6', '7'], ['2', 'B', '1'], ['3', '4', '5']]
[['8', '6', '7'], ['B', '2', '1'], ['3', '4', '5']]
[['8', '6', '7'], ['3', '2', '1'], ['B', '4', '5']]
[['8', '6', '7'], ['3', '2', '1'], ['4', 'B', '5']]
[['8', '6', '7'], ['3', '2', '1'], ['4', '5', 'B']]
[['8', '6', '7'], ['3', '2', 'B'], ['4', '5', '1']]
[['8', '6', 'B'], ['3', '2', '7'], ['4', '5', '1']]
[['8', 'B', '6'], ['3', '2', '7'], ['4', '5', '1']]
[['B', '8', '6'], ['3', '2', '7'], ['4', '5', '1']]
[['3', '8', '6'], ['B', '2', '7'], ['4', '5', '1']]
[['3', '8', '6'], ['2', 'B', '7'], ['4', '5', '1']]
[['3', 'B', '6'], ['2', '8', '7'], ['4', '5', '1']]
[['3', '6', 'B'], ['2', '8', '7'], ['4', '5', '1']]
[['3', '6', '7'], ['2', '8', 'B'], ['4', '5', '1']]
[['3', '6', '7'], ['2', '8', '1'], ['4', '5', 'B']]
[['3', '6', '7'], ['2', '8', '1'], ['4', 'B', '5']]
[['3', '6', '7'], ['2', 'B', '1'], ['4', '8', '5']]
[['3', 'B', '7'], ['2', '6', '1'], ['4', '8', '5']]
[['B', '3', '7'], ['2', '6', '1'], ['4', '8', '5']]
[['2', '3', '7'], ['B', '6', '1'], ['4', '8', '5']]
[['2', '3', '7'], ['4', '6', '1'], ['B', '8', '5']]
[['2', '3', '7'], ['4', '6', '1'], ['8', 'B', '5']]
[['2', '3', '7'], ['4', 'B', '1'], ['8', '6', '5']]
[['2', 'B', '7'], ['4', '3', '1'], ['8', '6', '5']]
```

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

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

```
[['3', '5', 'B'], ['1', '2', '8'], ['6', '7', '4']]
[['3', 'B', '5'], ['1', '2', '8'], ['6', '7', '4']]
[['3', '2', '5'], ['1', 'B', '8'], ['6', '7', '4']]
[['3', '2', '5'], ['B', '1', '8'], ['6', '7', '4']]
[['3', '2', '5'], ['6', '1', '8'], ['B', '7', '4']]
[['3', '2', '5'], ['6', '1', '8'], ['7', 'B', '4']]
[['3', '2', '5'], ['6', '1', '8'], ['7', '4', 'B']]
[['3', '2', '5'], ['6', '1', 'B'], ['7', '4', '8']]
[['3', '2', 'B'], ['6', '1', '5'], ['7', '4', '8']]
[['3', 'B', '2'], ['6', '1', '5'], ['7', '4', '8']]
[['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']]
Goal state reached, # of moves:  133
```

2.3 Exceed Maximum node

Using the same input as in section 1.3, we get:

```
Searching using local beam search...
Visited nodes exceed the maximum limit. Max limit: 10
```

2.4 Not Solvable Case

Using the same input as in section 1.4, we get:

```
The puzzle given is unsolvable.
```

2.5 Randomly Generated Case With n = 20

3. Experiments

Puzzle	Shortest Path/ random move	A* with h1	A* with h2	Local Beam Search(k=30)	Local Beam Search(k=500)
[[1,2,B],[3,4,5],[6,7,8]]	2	2	2	2	2
[[1,2,5],[3,4,8],[6,7,B]]	4	4	4	4	4
[[1,2,5],[3,B,8],[6,4,7]]	6	6	6	6	6
[[1,2,5],[3,8,7],[6,4,B]]	8	8	8	8	8
[[1,2,5],[3,8,7],[B,6,4]]	10	10	10	388	10
[[2,8,5],[1,7,B],[3,6,4]]	15	15	15	252	21
[[1,2,8],[B,7,5],[3,6,4]]	17	17	17	2337	335
[[1,2,8],[7,6,5],[3,4,B]]	18	18	18	—	26
[[1,4,2],[B,7,5],[3,6,8]]	10	5	5	5	5
[[1,4,2],[3,7,5],[6,B,8]]	30	3	3	3	3
[[1,7,3],[B,6,2],[4,8,5]]	60	17	17	2343	57
[[1,2,5],[B,7,3],[6,4,8]]	90	15	15	2577	221
[[2,7,8],[1,6,3],[4,B,5]]	120	21	21	53	31
[[6,1,3],[2,8,5],[4,B,7]]	150	Run for too long time	23	—	327

a)

The larger the maxNode is, the more problems will be solvable. This is reasonable since according to the a_star() and beam_search() methods, if the number of expanded nodes exceeds maxNode, then the method will start to return. The smaller the maxNode is, the less the number of nodes will be examined.

b)

A* search with h2 is better than A* search with h1.

c)

Since the value of *k* will affect the number of nodes expanded, we choose *k=30* and *k=500* for comparison. The following figure shows the number of nodes expanded for these three searching algorithms. The black parts are the states chosen by myself. The blue parts are the states generated by randomize_state() method. According to this figure, if the solution is small, the number of nodes expanded are pretty much the same. If the solution is relatively long, A* search will be more efficient than local beam search. There is no extreme distinction between A* search with h1 and A* search with h2, but for the last row, A* with h1 runs for a really long time. For local beam search, the one with larger value of *k* will be more efficient.

d)

The answer to this question depends on the *maxNode* we choose. If the *maxNode* is under the

actual number of nodes we need to explore, then the node will be counted as "unsolvable". By choosing a large value for *maxNode*, we can enlarge the probability that a puzzle is solvable. To test what fraction of generated problems are solvable, we generate 50 random states as a sample and run these three searching algorithms on them. All the test cases are generated by using the randomize_state(n) method with n = 30. The test cases are also stored in the *3d.txt* file.

Test Method Example:

```
def test3d(node):
    file = open('3d.txt','a')
    num_count = 0
    exist_map = {}
    i = 0
    for i in range(51):
        test_case = randomize_state(30,node)
        if tuple(map(tuple,testcase)) not in exist_map:
            exist_map[tuple(map(tuple,test_case))] = True
            file.write(str(test_case))
            file.write("\n")
            current_node = Node(test_case,0)
            if current_node.a_star(100000):
                num_count += 1
        else:
            i -= 1
    return num_count
```

d.1) A* with heuristic h1:

Since this approach takes too much time, I shrink the sample size to 20 with n = 30.
Informative Portion of Output:

```
Successful # of trials:  20 out of 20 trials,  fraction:  1.0
```

d.2) A* with heuristic h2:

Informative Portion of Output:

```
Successful # of trials:  50 out of 50 trials,  fraction:  1.0
```

Therefore, all of the 50 random generated test cases can be solved.

d.3) Local Beam Search with heuristic h2

By using k = 20, the informative portion of output I get is:

```
Successful # of trials:  50 out of 50 trials,  fraction:  1.0
```

If we use a larger *k* value, the solution will be shorter:

```
Successful # of trials:  41 out of 50 trials,  fraction:  0.82
```

Note that in this case, note all of the randomized state can reach the goal state. There might be 2 reasons: 1) the value of *k* is not well chosen; 2) the number of nodes we expand exceeds the maxNode limit. To test the first hypothesis, we enlarge the value of *k*. The output is as following:

```
Successful # of trials:  43 out of 50 trials,  fraction:  0.86
```

As we noted, the number of successful trials increases 2, which means the better selection of *k* value will improve the probability of solving the problems. It is true that if we do not select the proper value for *k*, we may miss the solution.

4. Discussion

a)

Based on the experiments, A* search with h2 is better suited for this problem. A* search with h1 requires more memory and more time to find the solution. Local beam search without well-chosen *k* usually expands more nodes than the other two searching algorithms, takes more memory, and usually finds longer solutions. A* with h2 usually finds shorter solutions. All in all, A* search with h2 seems to be superior in terms of time and space.

b)

- When the solutions of the problems are short(that is, close to the goal state), these three searching algorithms usually expand the same number of nodes(if the k value for local beam search is well-chosen).
- Typically, the larger the value of *k* is, the less the nodes are needed to be expanded. This is reasonable since if the value of *k* is too small, the solution might be cut off and it will take more time to find the solution in the future.
- A* search with h1 takes much longer time as the length of solution grows.