1. Code Design:

The code starts off by defining a class called EightPuzzle that has methods to set the initial state of the puzzle, move the tiles in the puzzle, randomize the puzzle, and solve the puzzle using different heuristics. The two heuristics implemented in the code are h1 and h2. h1 calculates the number of misplaced tiles in the current state, and h2 calculates the Manhattan distance between the current state and the goal state.

Initializing the initial state of the puzzle:

```
class EightPuzzle:
    def __init__(self):
        self.state = None
        self.goal_state = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
        self.counter = 0
        self.blank_row = None
        self.blank_col = None
        self.direction = None
        self.moves = []
        self.max_nodes = int(1000000000)
```

The EightPuzzle object has attributes:

- state - stores the current state of the object
- goal_state - the goal state (always the same)
- blank_row - the row that has the blank tile
- blank_col - the column that has the blank tile
- direction - stores the direction that it takes to get to that current state. Will be used later to store direction for solves
- Moves - stores the directions that the state needs to move in order to solve it
- Max_nodes - specifies the maximum number of nodes to be considered during a search

Note: represented the blank tile with the integer 0

Setting the state of the puzzle when given an argument string:

```
def setState(self, state):
    self.state = [[int(x) if x != 'b' else 0 for x in row]
                  for row in state.split()]

    for row in range(3):
        for col in range(3):
            if self.state[row][col] == 0:
                self.blank_row = row
                self.blank_col = col
```

- Sets the state attribute of the eight puzzle to be a 3x3 grid using a 2D array.
- This is where blank_row and blank_col is set

Prints the state of the puzzle:

```
def printState(self):
    state_str = ''
    for row in self.state:
        for col in row:
            if col == 0:
                state_str += 'b'
            else:
                state_str += str(col)
        state_str += ' '
    print(state_str[:-1])
    return state_str[:-1]
```

- Goes through the state of the puzzle (2D array) and stores it in a string with the format "### ### ###"

Converts the state into String format:

```
def convertIntoString(self, state):
    state_str = ''
    for row in state:
        for col in row:
            if col == 0:
                state_str += 'b'
            else:
                state_str += str(col)
        state_str += ' '
    return state_str[:-1]
```

- Goes through the state (2D array) and converts the state into the string format

Move function that moves the blank tile ‘up’, ‘down’, ‘left’, or ‘right’:

```
def move(self, direction):
    if direction == 'up' and self.blank_row > 0 and self.state[self.blank_row-1][self.blank_col] != 0:
        # Move up
        self.state[self.blank_row][self.blank_col] = self.state[self.blank_row-1][self.blank_col]
        self.state[self.blank_row-1][self.blank_col] = 0
        self.blank_row -= 1
        return True
    elif direction == 'down' and self.blank_row < 2 and self.state[self.blank_row+1][self.blank_col] != 0:
        # Move down
        self.state[self.blank_row][self.blank_col] = self.state[self.blank_row+1][self.blank_col]
        self.state[self.blank_row+1][self.blank_col] = 0
        self.blank_row += 1
        return True
    elif direction == 'left' and self.blank_col > 0 and self.state[self.blank_row][self.blank_col-1] != 0:
        # Move left
        self.state[self.blank_row][self.blank_col] = self.state[self.blank_row][self.blank_col-1]
        self.state[self.blank_row][self.blank_col-1] = 0
        self.blank_col -= 1
        return True
    elif direction == 'right' and self.blank_col < 2 and self.state[self.blank_row][self.blank_col+1] != 0:
        # Move right
        self.state[self.blank_row][self.blank_col] = self.state[self.blank_row][self.blank_col+1]
        self.state[self.blank_row][self.blank_col+1] = 0
        self.blank_col += 1
        return True
    else:
        # Invalid move
        return False
```

- Argument is the string of the direction
- The if statements check if the move is possible
  For example if the state is “b12 345 678”, it’s not possible to move the blank tile up or to the left.
- If the move is not possible, the state doesn’t change
- Returns True if the move is possible and False if its not (used later on searches)

Make n random moves from the goal state:

```
def randomizeState(self, n):
    self.setState("b12 345 678")
    directions = ['up', 'down', 'left', 'right']
    i = 0
    while i < n:
        direction = random.choice(directions)
        if self.move(direction):
            i += 1
        else:
            continue
```

- Sets the state first to the goal state
- Iterate through n times and moves the state in each time (note that variable i does not increment if the move method returns false)

h1: function that calculates the heuristic value of the input state using misplaced tiles as the heuristic:

```
def h1(self, state):
    misplaced = 0
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    for row in range(3):
        for col in range(3):
            if self.state[row][col] != goal_state[row][col]:
                misplaced += 1
    return misplaced
```

- Compares each tile with the goal state and return the number misplaced tiles

h2: function that calculates the heuristic value of the input state using manhattan distance as the heuristic:

```
def h2(self, state):
    manhattan_distance = 0
    for row in range(3):
        for col in range(3):
            value = state[row][col]
            if value != 0:
                goal_row = (value - 1) // 3
                goal_col = (value - 1) % 3
                distance = abs(goal_row - row) + abs(goal_col - col)
                manhattan_distance += distance
    return manhattan_distance
```

- The function iterates over each element in the state, except for the blank tile (which is represented by the value 0)
- For each non-blank tile, the function calculates the goal position for that tile based on its value (i.e., the goal position for the tile with value 1 is (0,0), the goal position for the tile with value 2 is (0,1), etc.).
- Then calculates the Manhattan distance between the current position and the goal position for the tile, and adds that distance to the total Manhattan distance for the state.

Same as the h2 method but doesn't take an input state. Just uses the object's state to compute heuristic value:

```
def getEvaluation(self):
    manhattan_distance = 0
    for row in range(3):
        for col in range(3):
            value = self.state[row][col]
            if value != 0:
                goal_row = (value - 1) // 3
                goal_col = (value - 1) % 3
                distance = abs(goal_row - row) + abs(goal_col - col)
                manhattan_distance += distance
    return manhattan_distance
```

Returns the opposite direction of the input direction: (used for the search function)

```
def opposite_direction(self, direction):
    if direction == 'up':
        return 'down'
    elif direction == 'down':
        return 'up'
    elif direction == 'left':
        return 'right'
    elif direction == 'right':
        return 'left'
```

Returns the successors:

```
def getSuccessors(self):
    successors = []

    # Check if the current state is the goal state
    if self.state == self.goal_state:
        return successors

    # Otherwise, get the successors of the current state
    for direction in ['up', 'down', 'left', 'right']:
        if self.move(direction):
            successor = EightPuzzle()
            successor.state = [row[:] for row in self.state]
            stringOfState = self.convertIntoString(successor.state)
            successor.setState(stringOfState)
            successor.direction = direction
            successors.append(successor)

            # Move the blank tile back to its original position
            self.move(self.opposite_direction(direction))
    return successors
```

- It first initializes an empty list to hold the successors
- Then checks if the current state is already the goal state, in which case there are no successors.
- Otherwise, it tries to move the blank tile in all four possible directions, and if a move is valid, it creates a new instance of the EightPuzzle class to represent the resulting state
- Then sets the direction of the move that led to this state and appends the new instance to the list of successors
- Finally, it moves the blank tile back to its original position so that the current state is not modified
- Returns the list of successor states.

solve A star search h1:

```
def solve(self, heuristic):
    if (heuristic == "h1"):
        # Initialize the search
        frontier = []
        heapq.heappush(frontier, (self.h1(self.state), 0, self.state, []))
        stringOfFirstState = self.convertIntoString(self.state)
        reached = {stringOfFirstState: 0}

        # Initialize the number of nodes reached
        num_nodes_reached = 1

        while frontier:
            # Pop the state with the lowest estimated cost
            _, num_moves, state, moves = heapq.heappop(frontier)
            stringOfState = self.convertIntoString(state)
            self.setState(stringOfState)

            # Check if the goal has been reached
            if self.state == self.goal_state:
                print(
                    f"Number of nodes Reached for h1: {num_nodes_reached}")
                print(f"Number of tile moves needed for h1: {len(moves)}")
                print("Sequence of moves for h1: " + " -> ".join(moves))
                return True

            # Expand the state
            for direction in ['up', "down", "right", "left"]:
                if self.move(direction):
                    new_puzzle = EightPuzzle()
                    new_puzzle.state = [row[:] for row in self.state]
                    new_puzzle.direction = direction
                    string_new_state = new_puzzle.convertIntoString(
                        new_puzzle.state)
                    new_cost = num_moves + 1
                    if (string_new_state not in reached) or (new_cost < reached[string_new_state]):
                        reached[string_new_state] = new_cost
                        # Add the new move to the list of moves
                        new_moves = moves.copy()
                        new_moves.append(direction)
                        heapq.heappush(frontier, (new_puzzle.h1(
                            new_puzzle.state) + new_cost, new_cost, new_puzzle.state, new_moves))
                    self.move(self.opposite_direction(direction))

            # Check if the maximum number of nodes has been reached
            num_nodes_reached = len(reached)
            if num_nodes_reached > self.max_nodes:
                print(
                    f"Number of nodes Reached for h1: {num_nodes_reached}")
                print(
                    f"Maximum number of nodes ({self.max_nodes}) reached for h1.")
                return False

        # No solution found
        return False
```

- Separates the solve into two h1 and h2 (hence the if statement)
- First, the initial state is added to the frontier as a tuple containing the estimated cost (using the specified heuristic), the number of moves to reach the current state, the current state, and an empty list of moves.
- The reached dictionary is also initialized with the string representation of the initial state and its cost.

- The algorithm then enters a while loop, popping the state with the lowest estimated cost from the frontier, and checking if it is the goal state. If it is, the number of nodes reached, number of moves, and sequence of moves are printed, and the function returns True.

- if the current state is not the goal state, the algorithm expands it by trying all possible moves. For each move, a new puzzle object is created, and the cost of reaching this new state is calculated as the sum of the cost to reach the current state and the cost of the move. If this new state has not been reached before, or the new cost is lower than the previous cost of reaching this state, it is added to the frontier with its cost, number of moves, state, and the list of moves leading to this state.

- If the maximum number of nodes is reached, the function returns False. If the frontier is empty and no solution is found, the function also returns False.

Solve A Star h2:

- Has the same code as the code above but changes h1 function to h2

Solve Beam k:

```
def solveBeam(self, k):
    # Initialize the starting state
    current_puzzle = EightPuzzle()
    current_String = self.convertIntoString(self.state)
    current_puzzle.setState(current_String)
    states = [current_puzzle]

    # Initialize the number of nodes reached
    num_nodes_reached = 1

    # check to see if the state is the goal state
    if (self.state == self.goal_state):
        print(f"Number of nodes Reached for beam: {num_nodes_reached}")
        print(f"Number of moves for beam: 0")
        print(f"Sequence of moves for beam: ")
        return True

    # Initialize the beam with the current state
    beam = [(current_puzzle.getEvaluation(), current_puzzle)]

    # Initialize the reached set with the cost of reaching the current state
    reached = {}
    reached[self.convertIntoString(current_puzzle.state)] = 0

    while beam:
        # Select the k best states from the expanded states
        beam = heapq.nsmallest(k, beam, key=lambda x: x[0])

        # Check if the goal state has been reached
        if beam[0][1].state == self.goal_state:
            stringOfState = self.convertIntoString(beam[0][1].state)
            self.setState(stringOfState)
            self.moves = beam[0][1].moves
            print(f"Number of node Reached for beam: {num_nodes_reached}")
            print(f"Number of moves for beam: {len(self.moves)}")
            print(f"Sequence of moves for beam: {' -> '.join(self.moves)}")
            return True

        # Expand the states in the beam
        new_beam = []
        for state in beam:
            successors = state[1].getSuccessors()
            for successor in successors:
                # Calculate the new cost to reach the successor state
                new_cost = len(state[1].moves) + 1
                if self.convertIntoString(successor.state) not in reached or new_cost < reached[self.convertIntoString(successor.state)]:
                    # Set the moves and add the state to the new beam
                    successor.moves = state[1].moves + [successor.direction]
                    new_beam.append((successor.getEvaluation(), successor))

                    # Add the state to the reached set with the new cost
                    reached[self.convertIntoString(
                        successor.state)] = new_cost

        num_nodes_reached = len(reached)

        # Update the beam with the expanded states
        beam = new_beam

        # Check if maximum number of nodes has been reached
        if num_nodes_reached > self.max_nodes:
            print(f"Number of nodes Reached for beam: {num_nodes_reached}")
            print(
                f"Maximum number of nodes ({self.max_nodes}) reached for beam.")
            return True

    # No solution found
    return False
```

- The solveBeam function takes a parameter k which determines the width of the beam. The larger the value of k, the wider the beam and the more states will be expanded.

- Begins by initializing the starting state and the beam with the current state. It then initializes the reached set with the cost of reaching the current state.

- Enters a loop that continues until the beam is empty or the goal state is reached. In each iteration, the k best states from the expanded states in the beam are selected. If the goal state is found, the function returns the solution. Otherwise, the states in the beam are expanded by generating all possible successors and calculating the new cost to reach each successor state.

- If a successor state has not been reached before or the new cost is less than the cost to reach it previously, it is added to the new beam with its corresponding moves. The algorithm updates the reached set with the new cost of reaching the successor state.

- The loop continues until the maximum number of nodes is reached or a solution is found. If the maximum number of nodes is reached, the algorithm prints a message and returns. If no solution is found, the algorithm returns False.

Main method to read the command.txt file and perform commands:

```
if __name__ == '__main__':
    # Parse the input filename from the command line arguments
    if len(sys.argv) != 2:
        print("Usage: python eight_puzzle.py input_file")
        sys.exit(1)
    input_file = sys.argv[1]

    # Read the commands from the input file
    with open(input_file, 'r') as f:
        commands = f.read().splitlines()

    # Create the puzzle object and execute the commands
    puzzle = EightPuzzle()
    for command in commands:
        parts = command.split()
        method = parts[0]
        args = parts[1:]

        if method == 'setState':
            state_str = " ".join(args)
            puzzle.setState(state_str)
        elif method == 'move':
            puzzle.move(*args)
        elif method == 'printState':
            puzzle.printState()
        elif method == 'randomizeState':
            if len(args) == 0:
                puzzle.randomizeState()
            elif len(args) == 1:
                n = int(args[0])
                puzzle.randomizeState(n)
        elif method == 'solve':
            if args[0] == 'A-star':
                input_str = args[-1]
                puzzle.solve(input_str)
            else:
                input_str = int(args[-1])
                puzzle.solveBeam(input_str)
        elif method == 'maxNodes':
            n = int(args[0])
            puzzle.max_nodes = n
```

- The first part of the code checks if the correct number of command-line arguments have been provided. If the correct number of arguments have not been provided, the program prints a usage message and exits.

- The code then reads in the commands from the input file, which is the second command-line argument. The commands are split into a list of strings. Next, an instance of the EightPuzzle class is created.

- The program then loops through each command in the commands list. Each command is split into a method and its arguments. The method is used to call the corresponding function in the EightPuzzle class.

- The available methods are:
    - setState: sets the state of the puzzle to the input string.
    - move: moves a tile in the puzzle in a given direction.
    - printState: prints the current state of the puzzle.
    - randomizeState: randomizes the state of the puzzle.
    - solve: solves the puzzle using either the A* algorithm or beam search.
    - maxNodes: sets the maximum number of nodes that can be expanded during a search.
- If the solve method is called with 'A-star' as the first argument, it calls the EightPuzzle solve method using the A* algorithm. If it is called with any other argument, it calls the EightPuzzle solveBeam method using beam search.

Command Line Interface:
python3 EightPuzzle.py commands.txt

Note: needs the text file at the end or else it gives an error

2. Code Correctness

The next part of the writeup should demonstrate your code on examples with the goal of proving to the grader that your search algorithms function correctly.

This should explain and illustrate how the search algorithms work and how they can fail. Choose examples that are concise and easy to verify.

These examples should be created by reading commands from a file, which you should include your submission. Be sure to set the seed of your random number generator so that your code generates the same random examples across multiple runs. When we run your code, we should get the same output that you did.

Not everything you include in your writeup needs to have been generated from the com-and file. For example, you will need to write methods for the experiments below. The purpose of the command file is to allow the grader to check basic functionality and correctness.

Methods:

- **setState <state> :**
    - If state hasn't been set yet, should just return no state:

      Command text:
      printState

      Output:
      No State

    - If the state is set, the state should just be the current state:

      Command text:
      setState 647 85b 321
      printState

      Output:
      647 85b 321

- **printState :**
    - Prints the state after set:

      Command text:
      setState 647 85b 321
      printState

      Output:

647 85b 321

- Prints the state after a solve (should be just goal state)

Command text:

solve beam 10
printState

Output:

Number of nodes Reached for h2: 986
Number of tile moves needed for h2: 12
Sequence of moves for h2: down -> right -> right -> down -> left -> up -> right -> up -> left -> down -> left -> up
b12 345 678

- **move <direction> :**
  - Moves the blank tile in the direction indicated if possible

Command text:

setState b12 345 678
printState
move down
printState
move right
printState
move up
printState
move left
printState

Output:

b12 345 678
312 b45 678
312 4b5 678
3b2 415 678
b32 415 678

Output states match with the command text.

- **randomizeState <n> :**
  - Randomizes n moves away from the goal state:

Command Text:

randomizeState 150
printState
solve A-star h2

Output:

b25 347 681
Number of nodes Reached for h2: 986
Number of tile moves needed for h2: 12
Sequence of moves for h2: down -> right -> right -> down -> left -> up -> right -> up -> left -> down -> left -> up

- **solve A-star <heuristic> :**
  - Solve the puzzle using A-star search using heuristics h1 and h2 but the start state is already the goal state

    Command Text:

    setState b12 345 678
    solve A-star h1
    setState b12 345 678
    solve A-star h2

    Output:

    Number of nodes Reached for h1: 1
    Number of tile moves needed for h1: 0
    Sequence of moves for h1:
    Number of nodes Reached for h2: 1
    Number of tile moves needed for h2: 0
    Sequence of moves for h2:

  - Solve the puzzle using A-star search using heuristics h1 and h2:

    Command Text:

    setState 647 85b 321
    printState
    solve A-star h1
    printState
    setState 647 85b 321
    solve A-star h2
    printState

Output:

647 85b 321
Number of nodes Reached for h1: 174241
Number of tile moves needed for h1: 25
Sequence of moves for h1: left -> down -> left -> up -> up -> right -> down -> down -> right -> up -> up -> left -> down -> down -> left -> up -> up -> right -> down -> right -> down -> left -> up -> left -> up
b12 345 678
Number of nodes Reached for h2: 148082
Number of tile moves needed for h2: 25
Sequence of moves for h2: left -> down -> left -> up -> up -> right -> down -> down -> right -> up -> up -> left -> down -> down -> left -> up -> up -> right -> down -> right -> down -> left -> up -> left -> up
b12 345 678

- **solve beam <k> :**
  - Solve the puzzle using Local Beam search using heuristic h2 as the evaluation function but the start state is already the goal state

    Command Text:

    setState b12 345 678
    solve beam 10
    printState

    Output:

    Number of nodes Reached for beam: 1
    Number of moves for beam: 0
    Sequence of moves for beam:
    b12 345 678

  - Solve the puzzle using Local Beam search using heuristic h2 as the evaluation function

    Command Text:

    setState 647 85b 321
    printState
    solve beam 10
    printState

    Output:

    647 85b 321

Number of nodes reached for beam: 532
Number of moves for beam: 33
Sequence of moves for beam: down -> left -> up -> up -> right -> down -> down -> left -> up -> left -> down -> right -> up -> up -> right -> down -> down -> left -> up -> left -> down -> right -> up -> left -> up -> right -> down -> down -> left -> up -> right -> up -> left
b12 345 678

- **maxNodes <n> :**
  - Set the maxNodes for local beam search
    - Without setting max nodes

      Command Text:

      setState 647 85b 321
      solve beam 10

      Output:

      Number of nodes reached for beam: 532
      Number of moves for beam: 33
      Sequence of moves for beam: down -> left -> up -> up -> right -> down -> down -> left -> up -> left -> down -> right -> up -> up -> right -> down -> down -> left -> up -> left -> down -> right -> up -> left -> up -> right -> down -> down -> left -> up -> right -> up -> left

    - With max nodes be less than number of nodes reached

      Command Text:

      maxNodes 531
      setState 647 85b 321
      printState
      solve beam 10

      Output:

      Number of nodes Reached for beam: 532
      Maximum number of nodes (531) reached for beam.

  - Set the maxNodes for A* search (since the code is basically the same for both heuristics just going to demonstrate it works for h2)
    - Without setting max nodes

      Command Text:

      setState 647 85b 321
      solve A-star h2

Output:

Number of nodes Reached for h2: 148082
Number of tile moves needed for h2: 25
Sequence of moves for h2: left -> down -> left -> up -> up -> right -> down -> down -> right -> up -> up -> left -> down -> down -> left -> up -> up -> right -> down -> right -> down -> left -> up -> left -> up

- With max nodes be less than number of nodes reached

Command Text:

maxNodes 1000
setState 647 85b 321
solve A-star h2

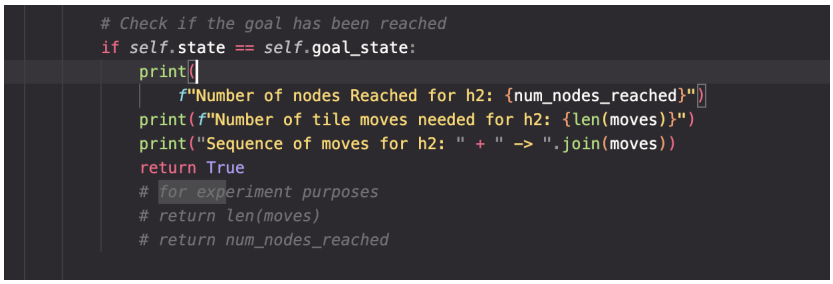
Output:

Number of nodes Reached for h2: 1001
Maximum number of nodes (1000) reached for h2.

Note: Sorry the commands in the text file are jumbled up. All the commands mentioned in the above section should be in it though

3. Experiments

Note: I had comments to the main code that indicated when its for the experiment.
Example:



```
# Check if the goal has been reached
if self.state == self.goal_state:
    print(
        f"Number of nodes Reached for h2: {num_nodes_reached}")
    print(f"Number of tile moves needed for h2: {len(moves)}")
    print("Sequence of moves for h2: " + " -> ".join(moves))
    return True
    # for experiment purposes
    # return len(moves)
    # return num_nodes_reached
```

For each part of the experiment I commented one return and uncommented another.
The experiment file to start a comment to start. As you go through each step of the experiment, uncomment each specific part necessary. Sorry for the confusion.

1. **How does the fraction of solvable puzzles from random initial states vary with the maxNodes limit?**

   Ran a test which checked the percentage of problems solved for a number of max node values. For each max node value I ran a test on 300 of the same generated states. This code is contained in the experiments.py file.

Max Nodes	A-Star: h1	A-Star: h2	Local Beam: k=15
75	0.11	0.14	0.16
112	0.16	0.206	0.34
168	0.17	0.26	0.50
252	0.28	0.36	0.86
378	0.34	0.42	0.95
567	9.43	0.5	0.98
850	0.50	0.56	1
1275	0.53	0.7	1
1912	0.72	0.76	1
2868	0.73	0.83	1
4302	0.85	0.87	1
6453	0.86	0.91	1

2. **For A* search, which heuristic is better, i.e., generates lower number of nodes?**
   For h1: Out of 300 randomly generated states, average number of nodes generated: 4221
   For h2: Out of 300 randomly generated states, average number of nodes generated: 2598

3. **How does the solution length (number of moves needed to reach the goal state) vary across the three search methods?**
   For - h1: Out of 300 randomly generated states, average solution length: 8.88
   For - h2: Out of 300 randomly generated states, average solution length: 8.88
   For - beam: Out of 300 randomly generated states, average solution length: 9.23

   A* search is the optimal solution and both h1 and h2 are admissible heuristics hence why both have smaller solution length than beam search.

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

Max Nodes	A-Star: h1	A-Star: h2	Local Beam: k=15
75	0.11	0.14	0.16
112	0.16	0.206	0.34
168	0.17	0.26	0.50
252	0.28	0.36	0.86
378	0.34	0.42	0.95
567	9.43	0.5	0.98
850	0.50	0.56	1
1275	0.53	0.7	1
1912	0.72	0.76	1

2868	0.73	0.83	1
4302	0.85	0.87	1
6453	0.86	0.91	1

Referenced Question 1.
According to my experiment, with the max nodes being 6453, each of the three search methods generated relatively high fractions for solvability. The fractions were 0.86, 0.91, 1.0 for A* with h1, A* with h2, local beam search respectively. Even though these have max node constraints, they are quite representative of the general case. As the maxNodes go higher and higher, eventually all the search methods will have 100% solvability because of the way randomizeState() is designed.

4. Discussion

1. **Based on your experiments, which search algorithm is better suited for this problem? Which one finds shorter solutions (solutions with less number of moves)? Which algorithm seems superior in terms of time and space?**

   Based on my experiments, the best search algorithm most suited for this problem is A* search with h2 as the heuristic function. A* with h2 heuristic finds the solution with least number of moves as explained in a question above. The algorithm that seems superior in terms of time and space is local beam search because it generates the least amount of nodes and because of that takes the least amount of time to complete.

2. **Discuss any other observations you made, such as the difficulty of implementing and testing each of these algorithms.**
   I like how we had the freedom to make our own command file and make our own tests for our methods. It was fun creating the experiment part of the assignment and getting data regarding the solution length, number of nodes generated, etc. During the process of implementing the searches, I had to go back and change the data structures because at first, I implemented them with the wrong data structure. I just dove into the code and realized that it would have been better if I created an outline and went step by step ensuring that I had everything correct. I had to do a lot of new parts that I haven't done before like reading from a text file (have down before but didn't really know how), using python, implementing A* and local beam search, priority heaps, and a lot more. I also struggled with the basics and that cost me a lot of time. I spent so long just because of a type error or small errors like that. Overall though, I had a great experience with this assignment and really enjoyed writing the code for it and seeing what my code did in my experiments.