

Assignment 1

CS 170: Introduction to Artificial Intelligence

Ariel Lira

Dr. Eamonn Keogh

alira008, 862125470

alira008@ucr.edu

12 February 2021

In completing this assignment, I consulted:

- The Blind Search and Heuristic Search lecture slides and notes annotated from lecture.
- The Sample Project Report provided to us by Dr. Eamonn Keogh.
- Python 2.7.14, 3.5, and 3.6 Documentation. This is the URL to the website for Python 2.7.14: <https://www.w3schools.com/python/>
- I consulted this website <https://www.d.umn.edu/~jrichar4/8puz.html> to see the definition of an 8-puzzle.
- Multiprocessing in Python Documentation. This URL leads to the website: <https://docs.python.org/3/library/multiprocessing.html>
- Time Access and Conversion for Python Documentation. Here is the URL to the website: <https://docs.python.org/3/library/time.html>

All important code is code is original. Unimportant subroutines that are not completely original are...

- Exit and argv subroutines from the sys to handle command line arguments and make sure program exits nicely.
- A bubble sort algorithm from <https://realpython.com/sorting-algorithms-python/#the-bubble-sort-algorithm-in-python> to easily sort the nodes from least cost to most cost.

- The subroutines sqrt, ceil, and fabs from the math module do help me do some computations.
- The subroutines time from the time module to help me calculate time it took to find solution.
- All subroutines from the multiprocessing module to help me speed up computation of different puzzles to solve.

Outline of this report:

- Cover page: (This page) and page 2.
- My report: Pages 3 to 9
- Sample trace on an easy problem, page 9
- Sample trace on a hard problem, page 10
- My code pages 10 to 23. Note that in case you would like to run my code, here is the URL to my GitHub repository so that you can download it to a local machine.

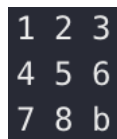
https://github.com/alira008/8-Puzzle_Solver_Using_AI

Eight Puzzle Project Report

Ariel Lira, SID 862125470, February 12, 2021

Introduction

For this project, I had to make a program that would attempt to solve any MxM puzzle that is given by the user. This project was an assignment for my CS 170: Introduction to Artificial Intelligence course at University of California, Riverside. This course was taught by Dr. Eamonn Keogh. For the purpose of this project, only 8-puzzles were tested but a user could very well try a 3-puzzle, 15-puzzle, 24-puzzle, and so on. According to Josh Richard, in his paper An Application Using Artificial Intelligence, an 8-puzzle is a game consisting of a 3x3 grid with a total 9 squares. The puzzle has squares that have a number 1-8, not repeating, and a blank space. The goal is to move the tiles into the goal state of the game, Figure 1.



1	2	3
4	5	6
7	8	b

Figure 1.

In my program, b represents a blank space. The program asks the user if they want to use a default puzzle that was hardcoded or if they want to input their own puzzles. The input would be stored into rows and columns and they a class called Puzzle that had all the necessary operations to manipulate the puzzle. I used the general search algorithm so that I could easily use a different queueing function if I wanted to. This is possible without having to modify the code much. When expanding the puzzle nodes, I made sure that we did not previously add the node onto the queue. This way, the program would not be stuck in an infinite loop expanding nodes that were already expanded previously. I gave the user the option to use 3 different queueing functions to find the

solution node of the puzzle. These three queuing functions are different kinds of search algorithms for finding the solution to a problem.

Comparison of Algorithms

These three algorithms are called Uniform Search, A* using the Misplaced Tile Heuristic, and finally A* using the Manhattan Distance Heuristic.

Uniform Search Algorithm

The uniform search algorithm is an algorithm that we check each node in a level before moving onto nodes that are deeper in the tree. This search algorithm is similar to the A* algorithms that I used in the program. The main difference is that the value for $h(n)$ is set to zero. This $h(n)$ value is the distance to get to the goal according to the slides of my professor, Dr. Eamonn Keogh. When determining the cost of the getting to the goal from a node for Uniform Search, we only look at the distance from the current node to the next, which happens to be 1. We can view this value $g(n)$, as the depth of the current node. The full cost equation that we will be using is $f(n) = g(n) + h(n)$.

A* using the Misplaced Tile Heuristic

The second search algorithm is called A* using the Misplaced Search Heuristic search algorithm. I will be referring to this algorithm as the Misplaced Tile search algorithm in my paper for shortness. This algorithm determines the cost of the nodes by adding number of misplaced numbers in the puzzle and the depth node. The only misplaced tile we do not count is the blank tile in the puzzle. This search algorithm makes sure that we expand the nodes that have the least misplaced values plus smallest depth first. This can be helpful with finding the solution faster instead of having to search every node in a depth before moving on to the next depth. I

A* using the Manhattan Distance Heuristic

The third search algorithm is A* using the Manhattan Distance Heuristic search algorithm. I will be referring to this algorithm as the Manhattan Distance search algorithm in my paper for shortness. This one determined the cost of the node by adding the distances from the current location to the correct location of each number of a puzzle node and the depth of the node. This search algorithm makes sure that we expand the nodes that have the smallest distance values plus smallest depth first. This search algorithm, like the other A* algorithm we talked about, can be helpful with finding the solution faster instead of having to search every node in a depth before moving on to the next depth.

Comparison of Algorithms on Sample Test Puzzles

To analyze the different efficiency of the search algorithms, I used 8 test puzzles. These 8 test puzzles are showing in Figure 2.

Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123 456 780	123 456 078	123 506 478	136 502 478	136 507 482	167 503 482	712 485 630	072 461 358

Figure 2.

I made a special function that would be called if the program has an argument passed to it. This special function would create the 8 test puzzle cases and output the data from each search algorithm into a file so that we could easily read data for analysis. After running all the test cases I made Figure 3. to view the data of each algorithm. I gave each test puzzle with a given queueing function 30 minutes to try to solve the puzzle.

Uniform Search	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8
total nodes expanded	0	13	59	499	3606	20292	71779	152937
max nodes in queue	0	8	28	190	1280	6584	17289	24048
depth of goal node	0	2	4	8	12	16	20	24
time taken	3.80E-06	0.0004201	0.010478	0.0742331	1.7866731	36.3042192	393.765762	1407.69111
Misplaced Tile Search	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8
total nodes expanded	0	4	9	49	241	1219	4245	8439
max nodes in queue	0	3	6	24	95	458	1552	3071
depth of goal node	0	2	4	8	12	16	20	NULL
time taken	2.90E-06	0.0002041	0.0004468	0.0116689	0.1478572	9.9941692	231.175529	1800.14816
Manhattan Distance Search	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8
total nodes expanded	0	4	9	24	66	412	1164	7344
max nodes in queue	0	3	6	12	30	157	433	2534
depth of goal node	0	2	4	8	12	18	20	26
time taken	3.10E-06	0.0007069	0.0013168	0.005151	0.0239229	0.5892072	6.0277941	1467.81109

Figure 3.

The Misplaced Tile Search algorithm ran out of time to find a solution for the 8th test puzzle. So, we will not use test case 8 when plotting. We will also not be plotting the values for test case 6, 7, and 8 in the uniform search because the values are far too big of a difference from the other search algorithms. Note that the seconds taken to solve test case 8 took a lot longer than test case 7 in the Manhattan Distance Search algorithm. I believe this is because the time to calculate the cost of using a node added up. For the Uniform Search, the time to get to the solution took a reasonable amount of time up until a solution depth greater than 16. The same thing can be said about the Misplaced Tile Search algorithm. The time to get to the solution of the puzzle took a reasonable amount of time for the Manhattan Distance Search algorithm up until a solution depth greater than 20. From this data, I wanted to see the relationship between the total nodes expanded and the depth of the goal node for each search algorithm. I plotted Figure 3. to see this relationship.

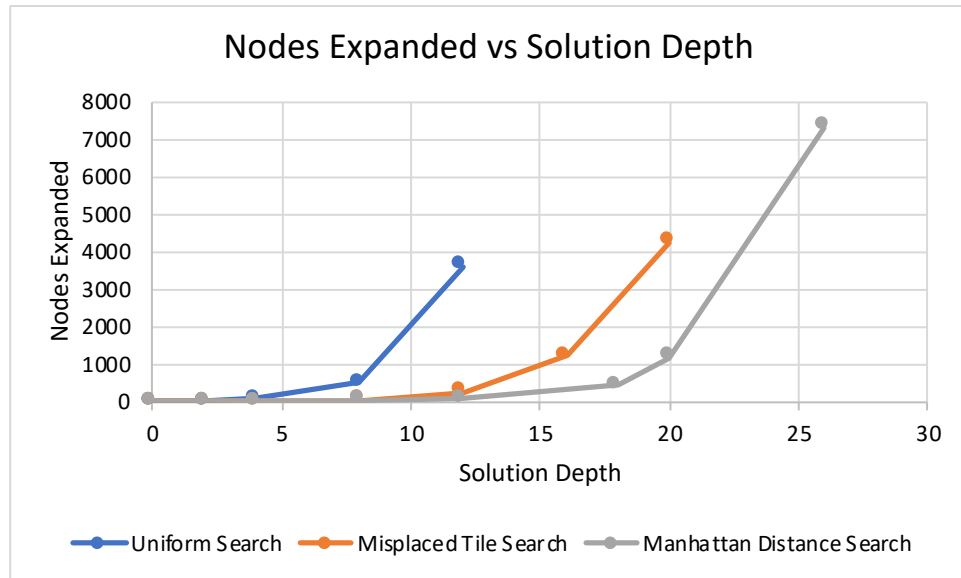


Figure 4.

From Figure 4., we can see that all three search algorithms had the same approximate depth goal up until depth 12. After that, the depths varied from each other. We can also see the number of nodes expanded for the Manhattan Distance Search algorithm was consistently smaller compared to the other two. I then finally plotted the relationship between the max number of nodes in the queue and the solution depth in Figure 5.

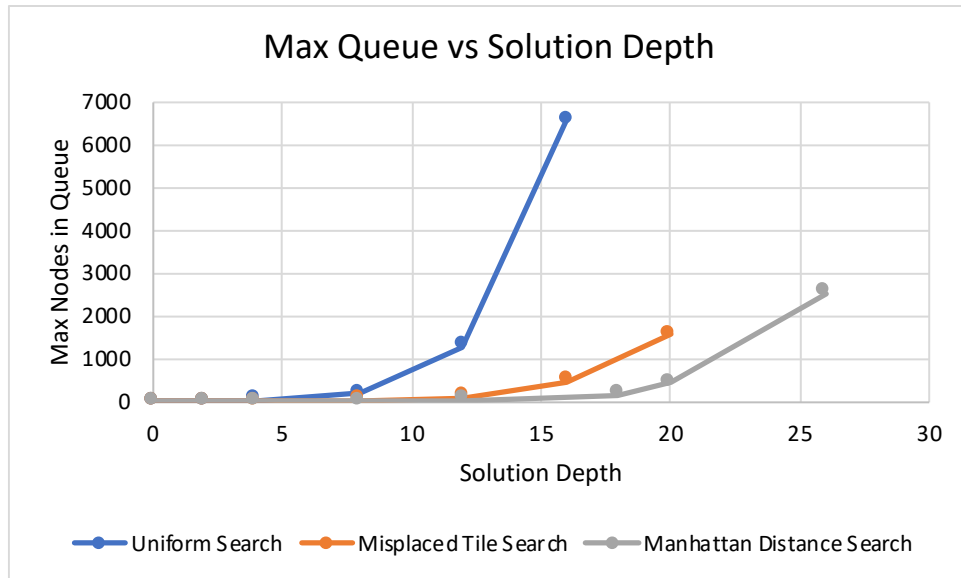


Figure 5.

Note that the Manhattan Distance Search algorithm consistently had less nodes in the queue. The max number of nodes in the queue went up steeply in the Uniform Search algorithm for problems with a solution depth higher than 12. Figure 5. is consistent with Figure 4. because the more nodes that were expanded for each search algorithm, the higher the max number of nodes there will be in a queue.

Conclusion

In conclusion, we can determine from this data that the Manhattan Distance Search algorithm performed the best for cases where the solution depth was between 0 and 20. After depth 20, it seemed that uniform search started to become the better algorithm for finding the solution the fastest. The Manhattan Distance Search algorithm also consistently used the least amount of space and nodes to determine the solution of the puzzle. The Uniform Search used the most amount of space and nodes and could lead to the program running out of space before reaching the solution to the puzzle. The Misplaced Tile Search algorithm seemed to be a nice

middle ground in performance and data usage between the Uniform Search algorithm and Manhattan Distance Search algorithm from solution depth 0 to solution depth 20.

The Following is a Traceback of an Easy Puzzle

```
► python3 project-1.py
Welcome to Ariel Lira's 8-puzzle solver for CS 170: Intro to AI
Type "1" to use a default puzzle, or "2" to enter your own puzzle. 2
Enter your puzzle, use a zero to represent the blank
    Enter row 1, use spaces or tabs between numbers          1 2 3
    Enter row 2, use spaces or tabs between numbers          5 0 6
    Enter row 3, use spaces or tabs between numbers          4 7 8
    Enter your choice of algorithm
        1. Uniform Cost Search
        2. A* with the Misplaced Tile heuristic.
        3. A* with the Manhattan distance heuristic.

3
Expanding state

1 2 3
5 b 6
4 7 8

The best state to expand with a  $g(n) = 1$  and  $h(n) = 3$  is...

1 2 3
b 5 6
4 7 8

The best state to expand with a  $g(n) = 2$  and  $h(n) = 3$  is...

1 2 3
4 5 6
b 7 8

Goal!!!
To solve this problem the search algorithm expanded a total of 9 nodes.
The maximum number of nodes in the queue at any one time was 6
The depth of the goal node was 4.
Puzzle took 0.000888 seconds to find solution.
```

The Following is a Traceback of a Hard Puzzle

```
► python3 project-1.py
Welcome to Ariel Lira's 8-puzzle solver for CS 170: Intro to AI
Type "1" to use a default puzzle, or "2" to enter your own puzzle. 2
Enter your puzzle, use a zero to represent the blank
    Enter row 1, use spaces or tabs between numbers          7 1 2
    Enter row 2, use spaces or tabs between numbers          4 8 5
    Enter row 3, use spaces or tabs between numbers          6 3 0
    Enter your choice of algorithm
        1. Uniform Cost Search
        2. A* with the Misplaced Tile heuristic.
        3. A* with the Manhattan distance heuristic.
3
Expanding state

7 1 2
4 8 5
6 3 b

The best state to expand with a  $g(n) = 1$  and  $h(n) = 11$  is...

7 1 2
4 8 5
6 b 3

The best state to expand with a  $g(n) = 2$  and  $h(n) = 11$  is...

7 1 2
4 8 5
b 6 3

Goal!!!
To solve this problem the search algorithm expanded a total of 1164 nodes.
The maximum number of nodes in the queue at any one time was 433
The depth of the goal node was 20.
Puzzle took 5.668020 seconds to find solution.
```

My Code

If you would like to download my code, here is the URL to my GitHub repo:

https://github.com/alira008/8-Puzzle_Solver_Using_AI

project-1.py

```
from math import sqrt, ceil, fabs
from sys import exit, argv
from puzzle import Puzzle
from sort import bubble_sort
from time import time
import multiprocessing

def main():
    #      Display the menu
    puzzle, algorithm_choice = menu()

    #      Find solution to the puzzle
    solution, _ = general_search(puzzle, algorithm_choice)

    if(type(solution) is not Puzzle):
        print (solution)

    return 0

def general_search(problem, queuing_function):
    puzzles_tried = set()
    node_list = []
    total_nodes_count = 0
    loop_count = 0
    prev_gn = 0
    max_queue = 0
    goal = solved_puzzle(problem)

    node_list.append(problem)
    print("Expanding state")
    print(problem)
    puzzles_tried.add(tuple(problem.get_grid_1d()))

    #      Start timer
    start = time()
    while (1):
        if (len(node_list) == 0):
            return "Failure: Solution was not found"

        else:
            node = node_list.pop(0)

            if (node.is_solved(goal)):
                #      End timer
```

```

        end = time()
        print ("Goal!!!")
        print ("To solve this problem the search algorithm expanded a total
of " + str(total_nodes_count) + " nodes.")
        print ("The maximum number of nodes in the queue at any one
time was " + str(max_queue))
        print ("The depth of the goal node was " + str(node.get_gn()) + ".")
        print("Puzzle took {:.6f} seconds to find solution.".format(end-
start))

        time_taken = end-start
        # print (puzzles_tried)
        return node, {"Total Nodes Expanded": total_nodes_count, "Max
Nodes in Queue": max_queue, "Depth": node.get_gn(), "Time taken": round((end-start), 7)}

    else:
        if(loop_count == 0):
            loop_count += 1
        elif (loop_count < 3):
            print("The best state to expand with a  $g(n) =$  " +
str(node.get_gn()) + " and  $h(n) =$  " + str(node.get_heuristic()) + " is...")
            print(node)
            loop_count += 1

        # Expand nodes
        temp_node_list, puzzles_tried = node.expand(puzzles_tried)
        # Total nodes expanded
        total_nodes_count += len(temp_node_list)

        if(queuing_function == "Misplaced"):
            # calculate the misplaced tiles of nodes
            for i in range(len(temp_node_list)):
                calc_misplaced(temp_node_list[i], goal)
            # Sort nodes by shortest  $f(n) = g(n) + h(n)$ 
            # temp_node_list = bubble_sort(temp_node_list)
        elif(queuing_function == "Manhattan"):
            # calculate the manhattan distances of nodes
            for i in range(len(temp_node_list)):
                calc_manhattan(temp_node_list[i], goal)
            # Sort nodes by shortest  $f(n) = g(n) + h(n)$ 
            # temp_node_list = bubble_sort(temp_node_list)

        # Append expanded nodes to the queue (node_list)
        for node in temp_node_list:
            node_list.append(node)
        # Set the max number of elements in a queue
        if(max_queue < len(node_list)):

```

```

        max_queue = len(node_list)

        #      Sort nodes by shortest  $f(n) = g(n) + h(n)$ 
        node_list = bubble_sort(node_list)

        #      Check if timer has gone over 15 minutes
        check_time = time() - start
        if (check_time > 1800.0):
            return "Failure: Went over 30 minutes.", {"Total Nodes
Expanded": total_nodes_count, "Max Nodes in Queue": max_queue, "Depth": "Failure: not
found", "Time taken": round((check_time), 7)}

```

```

def calc_manhattan(problem, goal):
    grid = problem.get_grid()
    n = problem.get_size()
    gn = problem.get_gn()
    total_nums = (n*n)-1
    x_distance = 0
    y_distance = 0

    for num in range(1, total_nums):
        #      locate prev in goal node
        #      j is the row
        for j in range(n):
            #      i is the column
            for i in range(n):
                if (str(num) == goal[j][i]):
                    goal_row = j
                    goal_col = i

        #      locate prev in user node
        #      j is the row
        for j in range(n):
            #      i is the column
            for i in range(n):
                if (str(num) == grid[j][i]):
                    row = j
                    col = i

        x_distance += fabs(col - goal_col)
        y_distance += fabs(row - goal_row)
    problem.set_heuristic(x_distance + y_distance + gn)

```

```

def calc_misplaced(problem, goal):
    grid = problem.get_grid()
    n = problem.get_size()

```

```

gn = problem.get_gn()
misplaced_count = 0

#    Check up until row n-1
#    row is the row
for row in range(n):
    #    col is the column
    for col in range(n):
        if (grid[row][col] != goal[row][col]):
            misplaced_count += 1

#    Check last row until before the blank space
for col in range(n-1):
    row = n-1
    if(grid[row][col] != goal[row][col]):
        misplaced_count += 1

problem.set_heuristic(misplaced_count + gn)

def solved_puzzle(problem):
    puzzle = []
    n = problem.get_size()

    i = 1
    for rows in range(n):
        row = []
        for cols in range(n):
            row.append(str(i))
            i += 1
        puzzle.append(row)
    puzzle[n-1][n-1] = 'b'

    return puzzle

def menu():
    print ("Welcome to Ariel Lira's 8-puzzle solver for CS 170: Intro to AI")
    puzzle_choice = input('Type "1" to use a default puzzle, or "2" to enter your own puzzle.
)
    puzzle = get_user_puzzle(int(puzzle_choice))

    print ("\nEnter your choice of algorithm")
    print ("\t1. Uniform Cost Search")
    print ("\t2. A* with the Misplaced Tile heuristic.")
    print ("\t3. A* with the Manhattan distance heuristic.")
    algorithm_choice = input()

```

```

if (algorithm_choice == '1'):
    algorithm_choice = "Uniform"
elif (algorithm_choice == '2'):
    algorithm_choice = "Misplaced"
else:
    algorithm_choice = "Manhattan"

return puzzle, algorithm_choice

def get_user_puzzle(choice):

    if (choice == 1):
        default_puzzle = [['1', '2', '3'], ['4', '8', 'b'], ['7', '6', '5']]

        ret = Puzzle(default_puzzle)
    else:
        #    grid arranged in rows and columns
        grid = []

        #    Get each row for the puzzle from the user
        print ("Enter your puzzle, use a zero to represent the blank")
        #    Ask user to input first row
        row = input("\nEnter row 1, use spaces or tabs between numbers\t\t")
        #    Check numbers from user
        row = row.split()

        #    Check the number of rows and columns needed for the puzzle
        n = len(row)
        #    Check the type of puzzle
        puzzle_type = (n * n) - 1

        #    Check if the row has a 0
        #    If one of the columns is a '0' change it to 'b'
        for i in range(n):
            if (row[i] == '0'):
                row[i] = 'b'
        #    Append the row to the puzzle grid
        grid.append(row)

        #    Ask the user to enter for the remaining of the puzzle
        for i in range(1, n):
            row = input("\nEnter row " + str(i+1) + ", use spaces or tabs between
numbers\t\t")

            row = row.split()

            #    Check if user input the correct amount of numbers for each row

```

```

        if (len(row) != n):
            print("\nWrong amount of numbers entered for row")
            print("For a " + str(puzzle_type) + "-puzzle, there are " + str(n) + "
rows and " + str(n) + " columns for each row.")
            exit(0)

        # Check if the row has a 0
        # If one of the columns is a '0' change it to 'b'
        for i in range(n):
            if (row[i] == '0'):
                row[i] = 'b'

        # Add row to the grid
        grid.append(row)

    ret = Puzzle(grid)

    return ret

# Function to rapidly get data from test cases and output the data into a txt file for easy
reading later
def testing_data():
    test_puzzles = [
        [['1', '2', '3'], ['4', '5', '6'], ['7', '8', 'b']],
        [['1', '2', '3'], ['4', '5', '6'], ['b', '7', '8']],
        [['1', '2', '3'], ['5', 'b', '6'], ['4', '7', '8']],
        [['1', '3', '6'], ['5', 'b', '2'], ['4', '7', '8']],
        [['1', '3', '6'], ['5', 'b', '7'], ['4', '8', '2']],
        [['1', '6', '7'], ['5', 'b', '3'], ['4', '8', '2']],
        [['7', '1', '2'], ['4', '8', '5'], ['6', '3', 'b']],
        [['b', '7', '2'], ['4', '6', '1'], ['3', '5', '8']]
    ]
    algorithm_choices = ["Uniform", "Misplaced", "Manhattan"]

    # Array of processes
    procs = []
    print_lock = multiprocessing.Lock()

    for algo_choice in algorithm_choices:
        for j in range(len(test_puzzles)):
            p = multiprocessing.Process(target=multi_proc_search,
args=(Puzzle(test_puzzles[j]), algo_choice, print_lock, ))
            procs.append(p)

    # Start each process
    for p in procs:

```



```

        p.start()

    # Wait until each process finishes
    for p in procs:
        p.join()

    return 0

# Function that is called by each process
def multi_proc_search(puzzle, algo_choice, lock):
    solution, data = general_search(puzzle, algo_choice)

    data["Algorithm"] = algo_choice + " algorithm"
    data = str(data)

    # Synchronize the processes when printing
    lock.acquire()
    fd = open("./analyze_data2.txt", "a")
    fd.write(data + "\n")
    fd.close()
    lock.release()

if __name__ == '__main__':
    if len(argv) < 2:
        main()
    else:
        testing_data()

    exit()

```

puzzle.py

```

class Puzzle:
    def __init__(self, grid):
        self.__grid = grid
        self.heuristic = 0
        self.gn = 0

    def __str__(self):
        str_grid = "\n"
        for row in self.__grid:
            for col in row:
                str_grid += str(col) + " "
            str_grid += "\n"

```

```

        return str_grid

'''
'''      Public functions '''
'''

def get_size(self):
    return len(self.__grid)

def is_solved(self, goal):
    return self.__grid == goal

def expand(self, puzzles_tried):
    nodes = []
    row, col = self.locate_b()

    if(self.__can_shift_up()):
        puzzle = Puzzle(self.__shift_up())
        puzzle.set_gn(self.gn + 1)

        #      Check if we already tried this puzzle or else we risk looping
forever on some puzzles
        puzzle_try = tuple(puzzle.get_grid_1d())
        if(puzzle_try not in puzzles_tried):
            nodes.append(puzzle)
            puzzles_tried.add(puzzle_try)

    if(self.__can_shift_down()):
        puzzle = Puzzle(self.__shift_down())
        puzzle.set_gn(self.gn + 1)

        #      Check if we already tried this puzzle or else we risk looping
forever on some puzzles
        puzzle_try = tuple(puzzle.get_grid_1d())
        if(puzzle_try not in puzzles_tried):
            nodes.append(puzzle)
            puzzles_tried.add(puzzle_try)

    if(self.__can_shift_left()):
        puzzle = Puzzle(self.__shift_left())
        puzzle.set_gn(self.gn + 1)

        #      Check if we already tried this puzzle or else we risk looping
forever on some puzzles
        puzzle_try = tuple(puzzle.get_grid_1d())

```

```

        if(puzzle_try not in puzzles_tried):
            nodes.append(puzzle)
            puzzles_tried.add(puzzle_try)

    if(self.__can_shift_right()):
        puzzle = Puzzle(self.__shift_right())
        puzzle.set_gn(self.gn + 1)

    #    Check if we already tried this puzzle or else we risk looping
forever on some puzzles
        puzzle_try = tuple(puzzle.get_grid_1d())
        if(puzzle_try not in puzzles_tried):
            nodes.append(puzzle)
            puzzles_tried.add(puzzle_try)

    return nodes, puzzles_tried

def is_near(self, number):
    #    Assume that point is not in our view distance (above, below, left, right)
    ret = 0
    #    Get points nearby
    pts = self.nearby_pts()
    #    Check if pts match with given number
    for point in pts:
        row = point[0]
        col = point[1]
        if (self.__grid[row][col] == str(number)):
            ret = 1

    return ret

def nearby_pts(self):
    n = len(self.__grid)
    #    Here we will store pts that we will return
    pts = []
    #    Get location of blank space
    row, col = self.locate_b()
    #    Make sure that pts are in our view distance (above, below, left, right)
    #    Make sure we are not giving pts that are outside of grid
    if (row != 0):
        pt_above = [row-1, col]
        pts.append(pt_above)

```

```

        if (row != n-1):
            pt_below = [row+1, col]
            pts.append(pt_below)

        if (col != 0):
            pt_left = [row, col-1]
            pts.append(pt_left)

        if (col != n-1):
            pt_right = [row, col+1]
            pts.append(pt_right)

    return pts

def get_grid(self):
    temp_grid = []

    for row in self.__grid:
        rows = []
        for col in row:
            rows.append(col)
        temp_grid.append(rows)

    return temp_grid

def get_grid_1d(self):
    temp_grid = []

    for row in self.__grid:
        for col in row:
            temp_grid.append(col)

    return temp_grid

def locate_b(self):
    n = len(self.__grid)
    for row in range(n):
        for col in range(n):
            if (self.__grid[row][col] == 'b'):
                return row, col

def get_heuristic(self):
    return int(self.heuristic)

def set_heuristic(self, h):

```

```

        self.heuristic = h

def get_gn(self):
    return int(self.gn)

def set_gn(self, gn):
    self.gn = gn

'''
'''      Private functions '''
'''
'''

def __can_shift_up(self):
    is_success = 0
    row, col = self.locate_b()

    #      Check if the location of the blank space is at the top of the grid
    if (row != 0):
        is_success = 1

    return is_success

def __can_shift_down(self):
    is_success = 0
    row, col = self.locate_b()
    n = len(self.__grid)

    #      Check if the location of the blank space is at the bottom of the grid
    if (row != n-1):
        is_success = 1

    return is_success

def __can_shift_left(self):
    is_success = 0
    row, col = self.locate_b()

    #      Check if the location of the blank space is at the left of the grid
    if (col != 0):
        is_success = 1

    return is_success

def __can_shift_right(self):
    is_success = 0
    row, col = self.locate_b()

```

```

n = len(self.__grid)

#      Check if the location of the blank space is at the right of the grid
if (col != n-1):
    is_success = 1

return is_success

def __shift_up(self):
    row, col = self.locate_b()
    temp_grid = self.get_grid()

    temp = temp_grid[row][col]
    temp_grid[row][col] = temp_grid[row-1][col]
    temp_grid[row-1][col] = temp

    return temp_grid

def __shift_down(self):
    row, col = self.locate_b()
    temp_grid = self.get_grid()

    temp = temp_grid[row][col]
    temp_grid[row][col] = temp_grid[row+1][col]
    temp_grid[row+1][col] = temp

    return temp_grid

def __shift_left(self):
    row, col = self.locate_b()
    temp_grid = self.get_grid()

    temp = temp_grid[row][col]
    temp_grid[row][col] = temp_grid[row][col-1]
    temp_grid[row][col-1] = temp

    return temp_grid

def __shift_right(self):
    row, col = self.locate_b()
    temp_grid = self.get_grid()

    temp = temp_grid[row][col]
    temp_grid[row][col] = temp_grid[row][col+1]
    temp_grid[row][col+1] = temp

```

```
return temp_grid
```

sort.py

```
# Sorting module from https://realpython.com/sorting-algorithms-python/#the-bubble-sort-algorithm-in-python
```

```
def bubble_sort(array):
    n = len(array)

    for i in range(n):
        # Create a flag that will allow the function to
        # terminate early if there's nothing left to sort
        already_sorted = True

        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value. With each
        # iteration, the portion of the array that you look at
        # shrinks because the remaining items have already been
        # sorted.

        for j in range(n - i - 1):
            if array[j].get_heuristic() > array[j + 1].get_heuristic():
                # If the item you're looking at is greater than its
                # adjacent value, then swap them
                array[j], array[j + 1] = array[j + 1], array[j]

                # Since you had to swap two elements,
                # set the `already_sorted` flag to `False` so the
                # algorithm doesn't finish prematurely
                already_sorted = False

        # If there were no swaps during the last iteration,
        # the array is already sorted, and you can terminate
        if already_sorted:
            break

    return array
```