CS205, Artificial Intelligence, Dr. Eamonn Keogh
The Eight Puzzle

Md Rayhanul Masud (SID: 862317259)
Email mmasu012@ucr.edu

May 13 2022

References The references that I have followed to complete the assignment

- The lecture contents related to the Blind Search and Heuristic Search topics
- Documentation of Python 3.10 https://docs.python.org/3
- The description of the project
- The attached sample report of the project description

Code Implementation The python code implementation of this project is solely original except the library/packages from python

- **heapq** to maintain a priority queue for the list of Eight Puzzle board states based on 3 different heuristics
- **deepcopy** to generate new board state
- **time** to calculate the execution time needed to run the code
- **matplotlib** to generate plots/figures to describe the comparisons identified among the 3 different heuristics to search for a solution of the given problem

Outline of the report

- Cover Page (Page 1)
- The description of 3 different heuristics (Page 2 to Page 6)
- The sample trace of a search (over the test cases) with the source code implementation (At the end of the report)
- The source code is also upload in https://github.com/mmasu012/Eight-Puzzle

1 Introduction

1.1 What is the Eight Puzzle

The Eight Puzzle is a puzzle where an orientation of 8 tiles numbering from 1-8 placed in a 3*3 matrix is given, and the puzzle solver needs to turn the orientation into a specific/given orientation called Goal state to solve the puzzle, by a series of sliding the tiles to the next possible empty blank position in the matrix.

This project is aimed to implement 3 known heuristics: Uniform Cost Search, A^* search with Misplaced Tiles and A^* search with Manhattan Distance to solve the puzzle. To implement them, Python (version 3) is chosen. The source code is attached with the this report. To further run the source code, the whole project is uploaded in github. The following sections of this report describe the basic understanding of the heuristics used, and provide comparative analysis based on the results found running the source code on several test cases.

2 Description of 3 heuristics used

2.1 Uniform Cost Search

When the algorithm attempts to choose which one of the current nodes in the queue is to be expanded, according to A^* search, the algorithm calculates a cost function adding the current cost $g(n)$ and the heuristic cost $h(n)$. $h(n)$ is considered to be zero for Uniform Cost Search approach.

2.2 A^* search with Misplaced Tiles

When the algorithm attempts to choose which one of the current nodes in the queue is to be expanded, according to A^* search, the algorithm calculates a cost function adding the current cost $g(n)$ and the heuristic cost $h(n)$. $h(n)$ is considered to be the total count of the misplaced tiles in the current orientation excluding the tile numbered zero as it is the blank grid of the given matrix.

Misplaced Tiles Heuristic Example

Goal State

1	2	3
4	5	6
7	8	0

Current State

1	2	3
8	**7**	6
5	**4**	0

In the given example above, the heuristic will find that the current state has $h(n) = 4$ because the four bold tiles in the current state are not matched with the goal state.

2.3 A^* search with Manhattan Distance

When the algorithm attempts to choose which one of the current nodes in the queue is to be expanded, according to A^* search, the algorithm calculates a cost function adding the current cost $g(n)$ and the heuristic cost $h(n)$ based on the Manhattan Distance between the tiles which are not in order with the goal state. Manhattan Distance between two grid in a matrix is the sum of the absolute value of the difference of the row indices and column indices excluding the tile numbered zero as it is the blank grid of the given matrix.

In the given example below, the heuristic calculates the Manhattan distance for the Misplaced tiles (bold faced in the example). $h(n) = 2(tilenumber\ 8) + 2(7) + 1(5) + 1(4) = 8$

Misplaced Tiles Heuristic Example

Goal State

1	2	3
4	5	6
7	8	0

Current State

1	2	3
8	**7**	6
5	**4**	0

3 Comparisons among 3 heuristics used

3.1 Test cases used for the experiment results

The project implementation considers the test cases provided in the project description.

Depth of solution to test cases {0, 2, 4, 8, 12, 16, 20, 24} left to right

1	2	3
4	5	6
7	8	0

1	2	3
4	5	6
0	7	8

1	2	3
5	0	6
4	7	8

1	3	6
5	0	2
4	7	8

1	3	6
5	0	7
4	8	2

1	6	7
5	0	3
4	8	2

7	1	2
4	8	5
6	3	0

0	7	2
4	6	1
3	5	8

3.2 Execution Time vs Depth

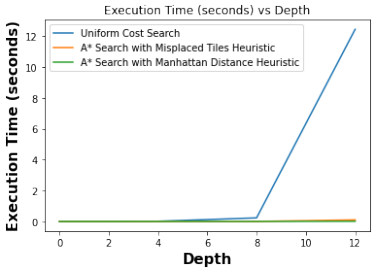


Figure 1: A comparison of 3 heuristics on the Eight Puzzle Problem increasing solution depth

- As Figure 1 suggests that for uniform cost search, the execution time is exponentially increasing as the depth of the solution increases.
- In case of other two heuristics, A^* search with Manhattan Distance is faster than A^* search with Misplaced tiles.

3.3 Node count (expanded) vs Depth

- As Figure 2 suggests that for uniform cost search, the number of node count expanded is exponentially increasing as the depth of the solution increases.
- In case of other two heuristics, A^* search with Manhattan Distance does not usually expand as much as nodes expanded in case of A^* search with Misplaced tiles.

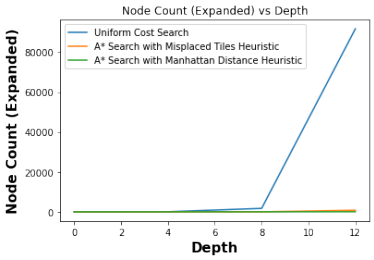


Figure 2: A comparison of 3 heuristics on the Eight Puzzle Problem for increasing solution depth

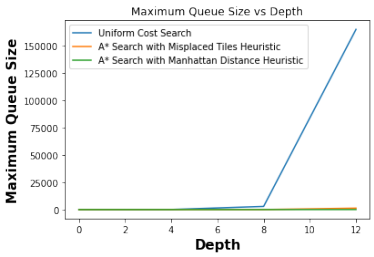


Figure 3: A comparison of 3 heuristics on the Eight Puzzle Problem for increasing solution depth

3.4 Maximum queue size vs Depth

- As Figure 3 suggests that for uniform cost search, the maximum queue size during the search at any time is exponentially increasing as the depth of the solution increases.
- In case of other two heuristics, A^* search with Manhattan Distance does not usually have as much as nodes in the queue in case of A^* search with Misplaced tiles.

3.5 Execution time vs Node count (expanded)

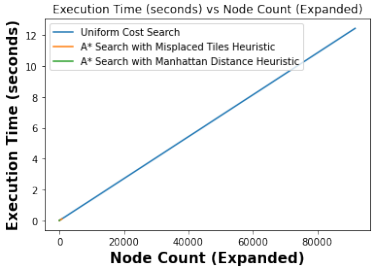


Figure 4: A comparison of 3 heuristics on the Eight Puzzle Problem for increasing node count

- As Figure 4 suggests that for uniform cost search, the execution time is linearly increasing as the number of node count expanded during the search increases.
- In case of other two heuristics, A^* search with Manhattan Distance needs less execution time than A^* search with Misplaced tiles, since A^* search with Manhattan Distance does not usually have as much as nodes expanded as that of A^* search with Misplaced tiles because of heuristic cost calculated.

3.6 Maximum queue size vs Node count

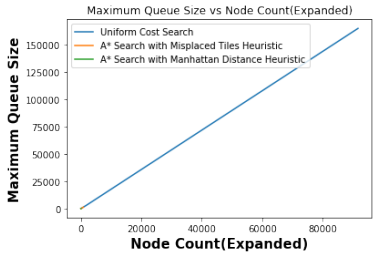


Figure 5: A comparison of 3 heuristics on the Eight Puzzle Problem for increasing node count expansion

- As Figure 5 suggests that for uniform cost search, the maximum queue size at any instant of the search is linearly increasing as the number of node count expanded during the search increases.
- In case of other two heuristics, A^* search with Manhattan Distance finds smaller queue size than A^* search with Misplaced tiles at any instant, since A^* search with Manhattan Distance does not usually have as much as nodes expanded as that of A^* search with Misplaced tiles because of heuristic cost calculated.

4 Instructions to run the source code

The source is uploaded https://github.com/mmasu012/Eight-Puzzle. There is a jupyter notebook which can be run to test the implementation. There is another file to generate the plots used for analysis across 3 heuristics discussed in the report.

5 Results of the experiment

		Max Queue Size	Node Expaned	Running Time
Depth = 8				
	Uniform	2918	1743	0.2
	Misplaced	29	18	0
	Manhattan	21	12	0
		Max Queue Size	Node Expaned	Running Time
Depth = 12				
	Uniform	165004	91661	12.4
	Misplaced	1325	818	0.1
	Manhattan	82	54	0
		Max Queue Size	Node Expaned	Running Time
Depth = 16				
	Uniform			
	Misplaced	65035	37856	5
	Manhattan	1274	770	0.1

Figure 6: Results for depth 0, 2, 4

6 Conclusion

- Eight puzzle problem needs huge computation power when the solution depth is bigger.
- Manhattan distance heuristic performs better than the Misplaced Tiles heuristics in respect of space and time complexities.
- Uniform cost search needs a huge amount of execution time for solution depth 16 20 and 24.

		Max Queue Size	Node Expaned	Running Time
Depth = 0				
	Uniform	1	0	0
	Misplaced	1	0	0
	Manhattan	1	0	0
		Max Queue Size	Node Expaned	Running Time
Depth = 2				
	Uniform	4	3	0
	Misplaced	3	2	0
	Manhattan	3	2	0
		Max Queue Size	Node Expaned	Running Time
Depth = 4				
	Uniform	82	57	0.01
	Misplaced	8	4	0
	Manhattan	8	4	0

Figure 7: Results for depth 8, 12, 16

		Max Queue Size	Node Expaned	Running Time
Depth = 20				
	Uniform			
	Misplaced	1556989	853624	147.7
	Manhattan	36070	22298	3.1
		Max Queue Size	Node Expaned	Running Time
Depth = 24				
	Uniform			
	Misplaced			
	Manhattan	487095	311068	52.3

Figure 8: Results for depth 20, 24

- Manhattan distance can work for all the test cases in a very short execution time, where as when the depth is 24, even misplaced tiles needs huge amount of time.
- Due to taking more amount of execution time, the comparative analysis presents for the testcases with solution depth upto 12.
- All the statistics experimented from solution depth 0 to depth 24 have been attached in the report.

In [1]:

```python
import time
import heapq
import copy
```

In [2]:

```python
UNIFORM_COST = 'Uniform cost'
MISPLACED_TILES = 'Misplaced tiles heuristic'
MANHATTAND_DIST = 'Manhattan distance heuristic'
heuristic_choices = {
    '1' : UNIFORM_COST,
    '2' : MISPLACED_TILES,
    '3' : MANHATTAND_DIST
}

test_cases = [
                [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]],

                [[1, 2, 3],
                 [4, 5, 6],
                 [0, 7, 8]],

                [[1, 2, 3],
                 [5, 0, 6],
                 [4, 7, 8]],

#                 [[1, 3, 6],
#                  [5, 0, 2],
#                  [4, 7, 8]],

#                 [[1, 3, 6],
#                  [5, 0, 7],
#                  [4, 8, 2]],

#                 [[1, 6, 7],
#                  [5, 0, 3],
#                  [4, 8, 2]],

#                 [[7, 1, 2],
#                  [4, 8, 5],
#                  [6, 3, 0]],

#                 [[0, 7, 2],
#                  [4, 6, 1],
#                  [3, 5, 8]],
]
```

In [3]:

```python
print(test_cases)
```

```
[[[1, 2, 3], [4, 5, 6], [7, 8, 0]], [[1, 2, 3], [4, 5, 6], [0, 7, 8]], [[1, 2, 3], [5, 0, 6], [4, 7, 8]]]
```

In [4]:

```python
# The class is used to maintain the state of any orientation during the Eight Puzzle simulation
class EightPuzzleNode:

    def __init__(self, board, heuristic, depth):

        self._board = board
        self._heuristic = heuristic
        self._goal_state = \
                [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]]
        self._goal_value_position_mapping = self._getValuePositionMapping()
        self._blankCellIndex = self._setBlankCellIndex()
        self._depth = depth
        self._calculateTotalCost()


    def getBoard(self):
        return copy.deepcopy(self._board)

    # maps the goal position of a tile with a number from 1-8
    def _getValuePositionMapping(self):

        value_position_mapping = {}
        for row_indx in range(3):
            for col_indx in range(3):

                value_position_mapping[self._goal_state[row_indx][col_indx]] = \
                    (row_indx, col_indx)

        return value_position_mapping

    # returns the depth of the state
    def getDepth(self):

        return self._depth

    # returns the heuristic used
    def getHeuristic(self):

        return self._heuristic

    # sets the blank cell index of the state
    def _setBlankCellIndex(self):

        for row_indx in range(3):

            for col_indx in range(3):

                if self._board[row_indx][col_indx] == 0:
                    return row_indx, col_indx

    # returns the blank cell index of the state
    def getBlankCellIndex(self):

        return self._blankCellIndex

    # checks if the state is in a solved/goal orientation
    def isSolved(self):

        for row_indx in range(3):

            if self._board[row_indx] != self._goal_state[row_indx]:
                return False

        return True

    # prints the current state
    def printCurrentState(self):

        for row_indx in range(3):
            print(self._board[row_indx])

    # calculates the number of misplaced tiles
    def _findMisplacedTilesCount(self):

        misplaced_tiles_count = 0
        for row_indx in range(3):
            for col_indx in range(3):

                if self._board[row_indx][col_indx] and \
                    self._board[row_indx][col_indx] != self._goal_state[row_indx][col_indx]:
                        misplaced_tiles_count+= 1

#         print(misplaced_tiles_count)
        return misplaced_tiles_count

    # calculates the manhattan distance of the current state
    def _findManhattanDistance(self):

        manhattan_distance = 0
        for row_indx in range(3):
            for col_indx in range(3):

                if self._board[row_indx][col_indx] and \
                    self._board[row_indx][col_indx] != self._goal_state[row_indx][col_indx]:

                    goal_row_indx, goal_col_indx = \
                        self._goal_value_position_mapping[ self._board[row_indx][col_indx] ]

                    distance = abs(row_indx - goal_row_indx) + abs(col_indx - goal_col_indx)
                    manhattan_distance += distance

#         print(manhattan_distance)
        return manhattan_distance

    # calculates the total cost according to the heuristic used for the search
    def _calculateTotalCost(self):


        if self._heuristic == UNIFORM_COST:
            self._total_cost = self._depth

        elif self._heuristic ==  MISPLACED_TILES:
            self._total_cost = self._depth + self._findMisplacedTilesCount()

        else:
            self._total_cost = self._depth + self._findManhattanDistance()

    # returns the total cost calculated
    def getTotalCost(self):

        return self._total_cost

    # returns the current cost and heuristic cost
    def getCurrentAndHeuristicCost(self):

        if self._heuristic == UNIFORM_COST:
            return self._depth, 0

        elif self._heuristic ==  MISPLACED_TILES:
            return  self._depth, self._findMisplacedTilesCount()

        else:
            return self._depth, self._findManhattanDistance()

    # comparator function used for the priority queue
    def __lt__(self, nxt):

        if self._heuristic != UNIFORM_COST:
            if self.getTotalCost() == nxt.getTotalCost():
                return self.getDepth() < nxt.getDepth()

        return self.getTotalCost() < nxt.getTotalCost()
```

In [5]:

```python
# gets selection of heuristic to be used from the user
def select_heuristic_search_approach():

    while True:
        heuristic_choice = input('Press 1 to select Uniform Cost Search\n' + \
                                 'Press 2 to select Misplaced Tile Heuristic search\n' + \
                                 'Press 3 to select Manhattan Distance Search\n').strip()

        if heuristic_choice not in heuristic_choices:
            print('Wrong selection')
            continue
        else:
            print(heuristic_choices[heuristic_choice], ' selected')

            return heuristic_choices[heuristic_choice]

# gets input of a new puzzle from the user
def get_new_puzzle():

    print('You will be prompted to enter each row of three rows of the puzzle\n' + \
          'You need to use a zero to represent the blank in your puzzle \n' + \
          'Each of the row can contain three digits delimited by spaces\n' + \
          'Please enter a valid puzzle')

    puzzle = []
    for row_index in range(3):
        row_values = input('Enter row ' + str(row_index + 1) + '\t')
        puzzle.append(list(map(int, row_values.split())))

    return puzzle

# converts the board state to a string to mantain the repeated states of the solution
# search space
def getPuzzleString(board):

    puzzle_string = ''
    for row_indx in range(3):
        for col_indx in range(3):
            puzzle_string += str(board[row_indx][col_indx])

    return puzzle_string

# creates a priority queue
def makeQueue(puzzle):

    nodes = [puzzle]
    heapq.heapify(nodes)

    return nodes

# finds the expanded nodes from the given node
def getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice, depth,
            new_blank_row_idx, new_blank_col_idx, repeated_states):

    if 0<=new_blank_row_idx<=2 and 0<=new_blank_col_idx<=2:

        new_board = copy.deepcopy(board)
        new_board[blank_row_idx][blank_col_idx], \
            new_board[new_blank_row_idx][new_blank_col_idx] = \
                board[new_blank_row_idx][new_blank_col_idx], \
                    board[blank_row_idx][blank_col_idx]

        new_puzzle_string = getPuzzleString(new_board)
        if new_puzzle_string not in repeated_states:
            return EightPuzzleNode(new_board, heuristic_choice, depth)


    return None

# returns the expanded nodes from the given node
def getNewNodes(puzzle_node, repeated_states):

    blank_row_idx, blank_col_idx = puzzle_node.getBlankCellIndex()
    board = puzzle_node.getBoard()
    heuristic_choice = puzzle_node.getHeuristic()
    depth = puzzle_node.getDepth()

    new_nodes = []

    # left
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx, blank_col_idx - 1,
                         repeated_states)
    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)

    # right
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx, blank_col_idx + 1,
                          repeated_states)
    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)

    # top
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx - 1, blank_col_idx,
                          repeated_states)
    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)

    # bottom
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx + 1, blank_col_idx,
                          repeated_states)

    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)


    return new_nodes

# returns the solution for the given puzzle node
def solvePuzzle(puzzle_node):

    max_queue_size = 0
    depth = -1
    expanded_node_count = 0
    start_time = time.time()

    repeated_states = {}
    nodes = makeQueue(puzzle_node)
    repeated_states[getPuzzleString(puzzle_node.getBoard())] = True

    while True:

        if len(nodes) > max_queue_size:
            max_queue_size = len(nodes)

        if len(nodes) == 0:
            is_solved = False
            break

        node = heapq.heappop(nodes)
        g_n, h_n = node.getCurrentAndHeuristicCost()
        print('The best state to expand with a g(n) = ', g_n, ' and h(n) = ', h_n)
        node.printCurrentState()

        if node.isSolved():
            depth = node.getDepth()
            is_solved = True
            break
        else:
            new_nodes = getNewNodes(node, repeated_states)
            if len(new_nodes) > 0:
                expanded_node_count += 1

            for new_node in new_nodes:
                heapq.heappush(nodes, new_node)
#                 print(new_node.printCurrentState())


    end_time = time.time()
    execution_time_in_seconds = end_time - start_time

    return (is_solved, max_queue_size, depth,
            expanded_node_count, "{:.1f}s".format(execution_time_in_seconds))
```

In [6]:

```python
# driver function
if __name__ == "__main__":

    while True:

        heuristic_choice = select_heuristic_search_approach()

        input_case = input('Press 1 to run eight puzzle simulation with default test cases\n' + \
                           'Press 2 to enter a new puzzle\n' + \
                           'Press any key to exit\n').strip()

        if input_case == '1':

            counter = 0
            for board in test_cases:
                puzzle = EightPuzzleNode(board, heuristic_choice, 0)

                print()
                print()
                print("Puzzle Test Case: ", counter + 1)
                puzzle.printCurrentState()
                print(heuristic_choice)

                is_solved, max_queue_size, depth, \
                    expanded_node_count, execution_time_in_seconds = \
                                solvePuzzle(puzzle)

                print('is_solved = ', is_solved, ' max_queue_size = ', max_queue_size,
                      ' depth = ', depth, ' expanded_node_count = ', expanded_node_count,
                      ' execution_time_in_seconds = ', execution_time_in_seconds)

                counter += 1


        elif input_case == '2':

            board = get_new_puzzle()
            puzzle = EightPuzzleNode(board, heuristic_choice, 0)

            print("Puzzle Test Case: ", counter + 1)
            puzzle.printCurrentState()
            print(heuristic_choice)

            is_solved, max_queue_size, depth, \
                    expanded_node_count, execution_time_in_seconds = \
                                solvePuzzle(puzzle)

            print('is_solved = ', is_solved, ' max_queue_size = ', max_queue_size,
                  ' depth = ', depth, ' expanded_node_count = ', expanded_node_count,
                  ' execution_time_in_seconds = ', execution_time_in_seconds)

        else:
            print('Thank you')
            break
```

```
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
3
Manhattan distance heuristic  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
1


Puzzle Test Case:  1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Manhattan distance heuristic
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  1  depth =  0  expanded_node_count =  0  execution_time_in_seconds =  0.0s


Puzzle Test Case:  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Manhattan distance heuristic
The best state to expand with a g(n) =  0  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  3  depth =  2  expanded_node_count =  2  execution_time_in_seconds =  0.0s


Puzzle Test Case:  3
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Manhattan distance heuristic
The best state to expand with a g(n) =  0  and h(n) =  4
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  3
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  8  depth =  4  expanded_node_count =  4  execution_time_in_seconds =  0.0s
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
2
Misplaced tiles heuristic  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
1


Puzzle Test Case:  1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Misplaced tiles heuristic
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  1  depth =  0  expanded_node_count =  0  execution_time_in_seconds =  0.0s


Puzzle Test Case:  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Misplaced tiles heuristic
The best state to expand with a g(n) =  0  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  3  depth =  2  expanded_node_count =  2  execution_time_in_seconds =  0.0s


Puzzle Test Case:  3
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Misplaced tiles heuristic
The best state to expand with a g(n) =  0  and h(n) =  4
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  3
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  8  depth =  4  expanded_node_count =  4  execution_time_in_seconds =  0.0s
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
1
Uniform cost  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
1


Puzzle Test Case:  1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Uniform cost
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  1  depth =  0  expanded_node_count =  0  execution_time_in_seconds =  0.0s


Puzzle Test Case:  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Uniform cost
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  4  depth =  2  expanded_node_count =  3  execution_time_in_seconds =  0.0s


Puzzle Test Case:  3
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Uniform cost
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 0, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[0, 4, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 0]
[5, 6, 3]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  2  and h(n) =  0
[0, 1, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[0, 2, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[2, 0, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 3, 6]
[5, 2, 0]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[0, 7, 6]
[5, 4, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 0, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 0, 7]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[5, 1, 3]
[0, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 0, 2]
[5, 6, 3]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 0, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 7, 0]
[4, 8, 6]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[0, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[5, 1, 3]
[2, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 0]
[5, 6, 3]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[0, 4, 7]
The best state to expand with a g(n) =  4  and h(n) =  0
[2, 5, 3]
[1, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 6]
[5, 2, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[0, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 1, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 2, 3]
[1, 7, 6]
[5, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[7, 0, 6]
[5, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 0, 8]
[4, 6, 7]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 6]
[5, 0, 2]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 2, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[2, 3, 0]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 2, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 6, 2]
[5, 0, 3]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  82  depth =  4  expanded_node_count =  57  execution_time_in_seconds =  0.
0s
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
2
Misplaced tiles heuristic  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
k
Thank you
```

In [7]:

```python
#
```

In [ ]:

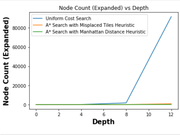
```python
```

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

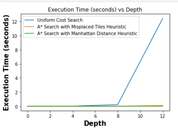
In [2]:
```
def generateLineplot(uniform_x, misplaced_x, manhattan_x, uniform_y,
                     misplaced_y, manhattan_y, x_label, y_label, title):

    fig, ax = plt.subplots()

    # Plot a simple line chart
    ax.plot(uniform_x, uniform_y , label='Uniform Cost Search')
    ax.plot(misplaced_x, misplaced_y, label='A* Search with Misplaced Tiles Heuristic')
    ax.plot(manhattan_x, manhattan_y, label='A* Search with Manhattan Distance Heuristic')

    ax.set_xlabel(x_label, fontweight ='bold', fontsize = 15)
    ax.set_ylabel(y_label, fontweight ='bold', fontsize = 15)

    plt.legend()
    plt.title(title)
    plt.show()
```

In [3]:
```
def generateBarplot(x, uniform_y, misplaced_y, manhattan_y,
                    x_label, y_label, title):

    # set width of bar
    barWidth = 0.25
    fig = plt.subplots(figsize =(12, 8))

    # set height of bar
#     uniform_y = [12, 30, 1, 8, 22]
#     misplaced_y = [29, 6, 16, 5, 10]
#     manhattan_y = [29, 3, 24, 25, 17]

    # Set position of bar on X axis
    br1 = np.arange(len(x))
    br2 = [p + barWidth for p in br1]
    br3 = [p + barWidth for p in br2]

    # Make the plot
    plt.bar(br1, uniform_y, color ='r', width = barWidth,
            edgecolor ='grey', label ='Uniform Cost Search')
    plt.bar(br2, misplaced_y, color ='g', width = barWidth,
            edgecolor ='grey', label ='A* Search with Misplaced Tiles Heuristic')
    plt.bar(br3, manhattan_y, color ='b', width = barWidth,
            edgecolor ='grey', label ='A* Search with Manhattan Distance Heuristic')

    # Adding Xticks
    plt.xlabel(x_label, fontweight ='bold', fontsize = 15)
    plt.ylabel(y_label, fontweight ='bold', fontsize = 15)
    plt.xticks([r + barWidth for r in range(len(x))],
               x)

    plt.title(title)
    plt.legend()
    plt.show()
```

In [4]:
```
# node expand vs depth
uniform_x = [0, 2, 4, 8, 12]
uniform_y = [0, 3, 57, 1743, 91641]

manhattan_x = [0, 2, 4, 8, 12]
manhattan_y = [0, 2, 4, 12, 54]

misplaced_x = [0, 2, 4, 8, 12]
misplaced_y = [0, 2, 4, 18, 818]

generateLineplot(uniform_x, misplaced_x, manhattan_x, uniform_y,
                 misplaced_y, manhattan_y, 'Depth', 'Node Count (Expanded)', 'Node Count (Expanded) vs Depth')
```

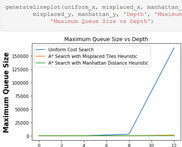


In [5]:
```
# time in seconds vs depth
uniform_x = [0, 2, 4, 8, 12]
uniform_y = [0, 0, 0.01, 0.23, 12.44]

manhattan_x = [0, 2, 4, 8, 12]
manhattan_y = [0, 0, 0, 0, 0.01]

misplaced_x = [0, 2, 4, 8, 12]
misplaced_y = [0, 0, 0, 0, 0.1]

generateLineplot(uniform_x, misplaced_x, manhattan_x, uniform_y,
                 misplaced_y, manhattan_y, 'Depth', 'Execution Time (seconds)', 'Execution Time (seconds) vs Depth')
```

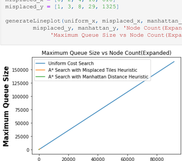


In [6]:
```
# time in seconds vs node expand
uniform_x = [0, 3, 57, 1743, 91641]
uniform_y = [0, 0, 0.01, 0.23, 12.44]

manhattan_x = [0, 2, 4, 12, 54]
manhattan_y = [0, 0, 0, 0, 0.01]

misplaced_x = [0, 2, 4, 18, 818]
misplaced_y = [0, 0, 0, 0, 0.1]

generateLineplot(uniform_x, misplaced_x, manhattan_x, uniform_y,
                 misplaced_y, manhattan_y, 'Node Count (Expanded)', 'Execution Time (seconds)', \
                 'Execution Time (seconds) vs Node Count (Expanded)')
```

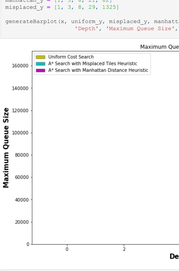
In [7]:
```
# max queue size vs depth
uniform_x = [0, 2, 4, 8, 12]
uniform_y = [1, 4, 82, 2918, 165004]

manhattan_x = [0, 2, 4, 8, 12]
manhattan_y = [1, 3, 8, 21, 82]

misplaced_x = [0, 2, 4, 8, 12]
misplaced_y = [1, 3, 8, 29, 1325]

generateLineplot(uniform_x, misplaced_x, manhattan_x, uniform_y,
                 misplaced_y, manhattan_y, 'Depth', 'Maximum Queue Size', \
                  'Maximum Queue Size vs Depth')
```



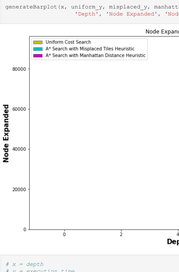
In [8]:
```
# max queue size vs node expand
uniform_x = [0, 3, 57, 1743, 91641]
uniform_y = [1, 4, 82, 2918, 165004]

manhattan_x = [0, 2, 4, 12, 54]
manhattan_y = [1, 3, 8, 21, 82]

misplaced_x = [0, 2, 4, 18, 818]
misplaced_y = [1, 3, 8, 29, 1325]

generateLineplot(uniform_x, misplaced_x, manhattan_x, uniform_y,
                 misplaced_y, manhattan_y, 'Node Count(Expanded)', 'Maximum Queue Size', \
                  'Maximum Queue Size vs Node Count(Expanded)')
```



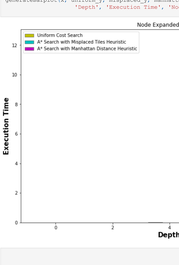
In [9]:
```
# x = depth
# y = max_queue_size

x = [0, 2, 4, 8, 12]
uniform_y = [1, 4, 82, 2918, 165004]
manhattan_y = [1, 3, 8, 21, 82]
misplaced_y = [1, 3, 8, 29, 1325]

generateBarplot(x, uniform_y, misplaced_y, manhattan_y,
                'Depth', 'Maximum Queue Size', 'Maximum Queue Size Bar Plot')
```



In [10]:
```
# x = depth
# y = node_expanded

x = [0, 2, 4, 8, 12]
uniform_y = [0, 3, 57, 1743, 91641]
manhattan_y = [0, 2, 4, 12, 54]
misplaced_y = [0, 2, 4, 18, 818]

generateBarplot(x, uniform_y, misplaced_y, manhattan_y,
                'Depth', 'Node Expanded', 'Node Expanded Bar Plot')
```



In [11]:
```
# x = depth
# y = execution_time

x = [0, 2, 4, 8, 12]
uniform_y = [0, 0, 0.01, 0.23, 12.44]
manhattan_y = [0, 0, 0, 0, 0.01]
misplaced_y = [0, 0, 0, 0, 0.1]

generateBarplot(x, uniform_y, misplaced_y, manhattan_y,
                'Depth', 'Execution Time', 'Node Expanded Bar Plot')
```



In [ ]:
```
```