Name: Nurlan Nazaraliyev
CS205: Introduction to Artificial Intelligence
SID: 862393740
Email: nnaza008@ucr.edu

In completing this assignment, I consulted:

- Lecture nodes for search algorithms and references in section 6

All-important code is original. Some built-in functions of the python (deepcopy, queue) have been used

Outline of this report:

- Cover page: (this page)
- My report: pages 2 to 4
- Sample results shown on page 3 and 4
- My code pages 6 to 21

# CS205: Project 1: The 8-Puzzle

## 1. Introduction

In this project, we are supposed to solve some puzzles, such as 8-puzzle which is a 3x3 matrix with numbers from 1 to 8, and a blank cell. The challenge is to solve this puzzle, I have implemented 3 search algorithms, namely, Uniform Cost Search, A* with the misplaced tile heuristic and A* with the Manhattan distance heuristic. The interface allows the user to input the size of the puzzle, for example, the input puzzle size can be 3 for 8-puzzle, 4 for 15-puzzle, or 5 for 25-puzzle. Also, the user has a chance to input the initial state of the puzzle and the goal state. The solution should find the path from the initial state to the goal state according to the 3 different algorithms. There is the option to enter the default goal state, which is the sorted puzzle. The project definition given to us has several input states to compare the results of the algorithms. The next section discusses the implemented algorithms design and implementation. Then comes the result and evaluation section.

## 2. Algorithms Implemented

For this project, 3 different algorithms are implemented, namely, Uniform Cost Search, A* with the misplaced tile heuristic and A* with the Manhattan distance heuristic. The code has been implemented in python and the search algorithm part is very similar to the pseudo-code in the project manual. In general, the search algorithm takes into consideration the cost functions: g(n) and h(n), the depth and heuristic cost functions, respectively.

### 2.1. Uniform Cost Search

Uniform Cost Search algorithm only considers the depth of the node from the root node and ignores the heuristic cost. Simply, this algorithm expands the root node (initial state of the puzzle); in other words, it creates all the possible movements from the initial state; blank position can move to up/down/left/right positions. Then the algorithm queues all these expanded states and goes over them one-by-one. Actually, the depth of all the expanded states from the root state (g(n) =1) is the same, being equal to 1. Because of this reason, in UCS algorithm, we cannot sort the queue and UCS performs the worst out of all the 3 algorithms. UCS behaves like breadth first search.

### 2.2. A* with the Misplaced Tile Heuristic

Similarly, the A* search algorithm also expands all the possible states from the initial state. However, the A* search algorithm also considers the heuristic cost function. For A* with misplaced tile heuristic, the heuristic cost function is the number of different cell entries except the blank symbol, in other words, the number of the different tiles which are not in the proper entries in the goal state. The queue of all the nodes that have been expanded in the previous round is sorted according to the summation of heuristic cost and depth cost functions: g(n)+h(n). The best node in the queue which has the lowest cost function is the next node to be expanded. This heuristic can be verified to be admissible because it never overestimates the cost of the current state. Since in the misplaced tiles (MT), we also know the number

of different/misplaced tiles, an additional piece of information, MT search algorithm performs better than UCS.

### 2.3. A* with the Manhattan Distance Heuristic (MD)

MD version of A* search algorithm implements heuristic cost function differently. MD heuristic cost function calculates the difference in the tile position. For example, if the state look like the below instance:

1 # 3

4 5 6

7 8 2

The row difference is 2, and column difference is 1 in position. So, the MD heuristic for this state is 3. This heuristic cost function can also be verified to be admissible since the function never overestimates the true cost of the state. As the heuristic for MD calculates the positional difference of the state from the goal state, it performs better than the other two algorithms.

## 3. Implementation Details

The interface of the project is designed to be user-friendly. Firstly, the user is asked to enter the input; the puzzle size, initial puzzle state and the goal state. The last input that needs to be entered is the algorithm. The default goal state is the one in which the puzzle is already solved. The project is implemented using two class definitions: Node and Problem. Node is the class with variables like state, depth from the initial state, heuristic function, and the Boolean value which tells us if the node has been expanded or not. The Problem class defines the problem; contains the variables like the initial state, goal state, the initial node (though can be neglected since we already have the initial state or initial state can be neglected), and the puzzle size.

Some other utility functions are also implemented to print the state matrix and move the blank symbol to other possible neighboring positions; up, down, left, or right. To do this, I check the border conditions, to see if the movement is possible. Also, we need to check if the puzzle (8- or 15-puzzle) is solvable or not, since we do not know how to check the solvability of 25-puzzle. Specifically, for 8-puzzle, there are 9!=362,880 possible states of which solely the half can be solved, the others are not solvable. Basic idea is to calculate the number of inversions; if it is even, then the puzzle is solvable; otherwise, unsolvable. This is only applicable to the cases where the goal state is the default goal state, otherwise there is the time limit of 1 hour.

## 4. Results and Evalution

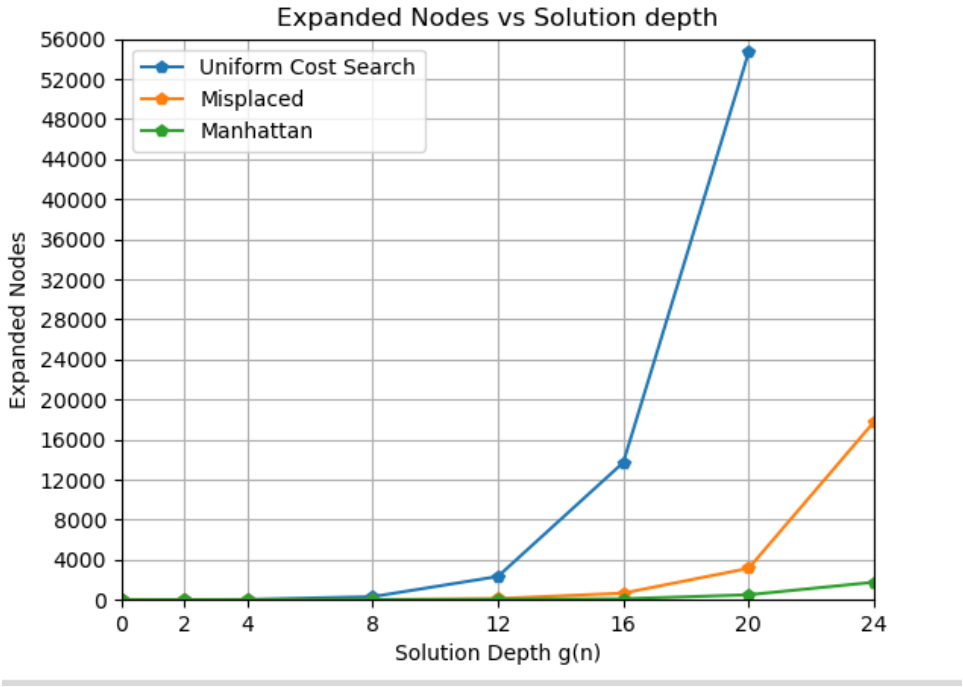


Figure 1: Expanded Nodes vs Solution depth for 8-Puzzle

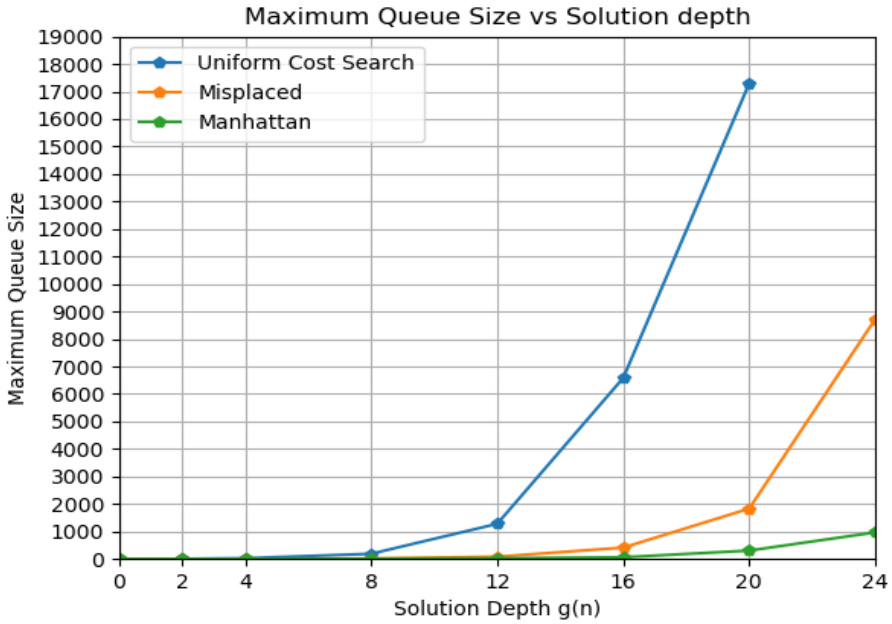


Figure 2. Max Queue Size vs Solution Depth

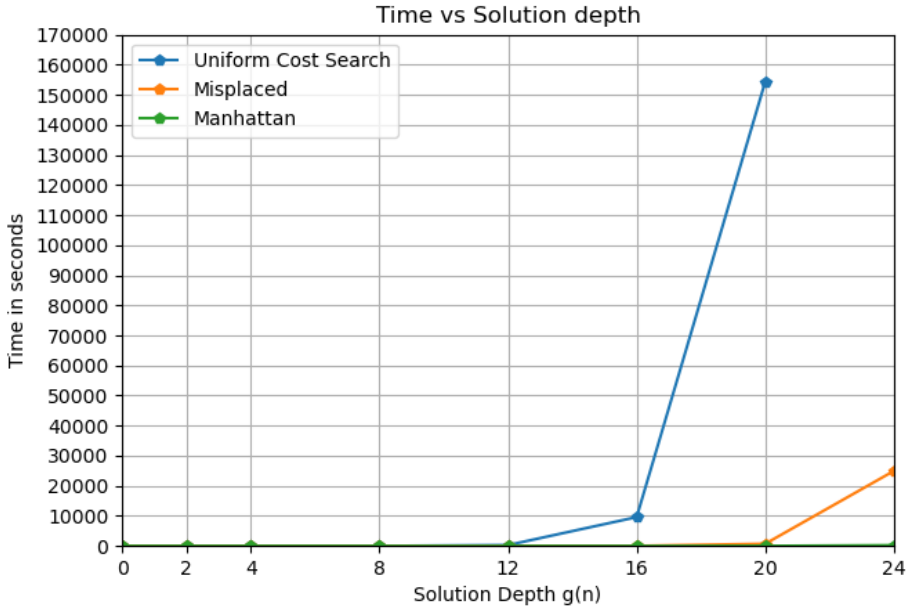


Figure 3. Time vs Solution Depth

As can be seen from Figure 1, UCS expands considerably more nodes to get to the goal state. However, A* with the MD and MT heuristic search algorithms expands comparably a smaller number of nodes. This is because UCS considers only the depth from the parent node which is 1 for all the children no matter

how much difference they have. The reason why MT heuristic has more expanded nodes is that it does not look at the levels of the differences (no information related to position).

A similar conclusion can be given to Figure 2 and 3.

5. Conclusion

In this project, we have implemented three different search algorithms, namely, Uniform Cost Search, A* with Misplaced Tile Heuristic and A* with Manhattan Distance Heuristic search algorithms. Looking at the results, it is very obvious that A* with MD Heuristic beats the other two algorithms in performance. This observation is very powerful if the solution depth is more than 12.

6. References


1. https://matplotlib.org/stable/index.html
2. https://en.wikipedia.org/wiki/Eight_queens_puzzle
3. https://en.wikipedia.org/wiki/15_puzzle
4. Lecture Slides for Search Algorithms
5. https://www.youtube.com/watch?v=bhmCmbj9VAg&t=338s&ab_channel=ECJunction
6. https://docs.python.org/3/library/copy.html
7. https://stackoverflow.com/questions/3899980/how-to-change-the-font-size-on-a-matplotlib-plot


Interface:

```
Enter number of rows (=columns) in input or press Enter for 8-puzzle:

The selected number of rows is:3
Now, let's create the input matrix or state
Enter the 1-th row with space in between numbers and 0 for blank:
1 2 3
Enter the 2-th row with space in between numbers and 0 for blank:
4 5 0
Enter the 3-th row with space in between numbers and 0 for blank:
7 8 6
initial state is as below:
-------
|1|2|3|
-------
|4|5|0|
-------
|7|8|6|
-------
Do you want to have default goal states: y/n, press Enter for y
Default goal state looks like:
-------
|1|2|3|
-------
|4|5|6|
-------
|7|8|0|
-------

The puzzle is solvable.
Now, you need to select the algorithm:
1: Uniform Cost Search.
2: A* with the Misplaced Tile heuristic.
3: A* with the Manhattan Distance heuristic.
Your choice: 1
Search Started
Expanding node:
-------
|1|2|3|
-------
|4|5|0|
-------
|7|8|6|
-------
Best node chosen to expand with g(n) = 1 and h(n) = 0:
-------
|1|2|0|
-------
|4|5|3|
-------
|7|8|6|
-------
```

```
Best node chosen to expand with g(n) = 1 and h(n) = 0:
-------
|1|2|3|
-------
|4|5|6|
-------
|7|8|0|
-------
Success:)
The total of 2 nodes have been expanded
The max number of nodes/states in the queue in one time instance is 3
Time taken is 0.5779266357421875 seconds
-------
|1|2|3|
-------
|4|5|6|
-------
|7|8|0|
-------
```

```
# this file contains utility function implementations

# in order to solve the problem

# first define the initial state: input from the user mostly
```

```
# goal state


import time
import copy
import matplotlib.pyplot as plt
import sys


#goal state:
goalState = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]


#lets create Node class which will save input node
# with the following properties:
class Node:
def __init__(self, node_state, node_depth=0, expanded=False):
self.parent_nodes = []
self.node_state = node_state # state of the node
self.node_depth = node_depth # depth from depth 0; parent node depth + 1
self.heuristic = 0 # heuristic cost of the node
self.expanded = expanded # is this node expanded before?


def get_state(self):
return self.node_state


class Problem:
def __init__(self, initial_state, goal_state, node=Node, length=3):
self.initial_state = initial_state
```

```
self.goal_state = goal_state
self.node = node
self.puzzle_length = length


#def initial_state(self):
# return self.initial_state
def goal_state(self):
return self.goal_state
def set_init_state(self, state):
self.initial_state = state



def print_state(current_state):
for i in current_state:
print( "-------")
string = "|"
for k in i:
string += str(k) + "|"
print(string)
print( "-------")
def move(current_state, action=0):
# action = 0: move blank cell up
# action = 1: move blank cell down
# action = 2: move blank cell left
# action = 3: move blank cell right
for i in current_state:
for k in i:
if k == 0:
```

```
index = (current_state.index(i), i.index(k))
# up movement
if action == 0:
if index[0] != 0:
current_state[index[0]][index[1]] = current_state[index[0] - 1][index[1]]
current_state[index[0] - 1][index[1]] = 0
return current_state
else:
return None
# down movement
if action == 1:
if index[0] != len(current_state)-1:
current_state[index[0]][index[1]] = current_state[index[0] + 1][index[1]]
current_state[index[0] + 1][index[1]] = 0
return current_state
else:
return None
# left movement
if action == 2:
if index[1] != 0:
current_state[index[0]][index[1]] = current_state[index[0]][index[1] - 1]
current_state[index[0]][index[1] - 1] = 0
return current_state
else:
return None
# right movement
if action == 3:
if index[1] != len(current_state)-1:
current_state[index[0]][index[1]] = current_state[index[0]][index[1] + 1]
```

```
current_state[index[0]][index[1] + 1] = 0
return current_state
else:
return None


# a function to create all possible nodes
def possible_states(state_matrix):
expansions_matrix = []

up = copy.deepcopy(state_matrix)
down = copy.deepcopy(state_matrix)
left = copy.deepcopy(state_matrix)
right = copy.deepcopy(state_matrix)

expansions_matrix.append(move(up, 0))
expansions_matrix.append(move(down, 1))
expansions_matrix.append(move(left, 2))
expansions_matrix.append(move(right, 3))

return expansions_matrix


# check if there is blank in the input matrix
def is_state_valid(matrix, row):
```

```
for i in range(row):
for k in range(row):
if matrix[i][k] == 0:
return True
return False


def get_algo():
print("Now, you need to select the algorithm: ")
print("1: Uniform Cost Search.")
print("2: A* with the Misplaced Tile heuristic.")
print("3: A* with the Manhattan Distance heuristic.")
algo = input("Your choice: ")
try:
algo = int(algo)
except ValueError:
print("Invalid Input: " + algo)
if algo < 1 or algo > 3:
print("Enter 1, 2, or 3!")
return algo


# function shown in the project pdf
def general_search(problem=Problem, function=1):
time_started = time.time()


my_queue = []
visited_states = []
n_visited = 0
```

```
queue_size = 0
max_queue_size = 0
goal_state = problem.goal_state

if function == 1:
heuristic = 0 # heuristic is zero for ucs
elif function == 2:
heuristic = heuristic_misplaced(problem.initial_state, goal_state)
elif function == 3:
heuristic = heuristic_manhattan(problem.initial_state, goal_state)

node = Node(problem.initial_state, 0, False)
node.heuristic = heuristic
my_queue.append(node)
visited_states.append(node.node_state)
queue_size += 1
max_queue_size += 1

while True:
if function != 1:
my_queue = sorted(my_queue, key=lambda x: (x.node_depth + x.heuristic, x.node_depth))

if len(my_queue) == 0:
print("Failure! No solution found within an hour!")
return None, None, None
next_node = my_queue.pop(0)
```

```
queue_size -= 1


if next_node.expanded == False:
n_visited += 1
next_node.expanded = True
if n_visited != 1:
print("Best node chosen to expand with g(n) = " + str(next_node.node_depth) + \
" and h(n) = " + str(next_node.heuristic) + ":")
print_state(next_node.get_state())
else:
print("Expanding node: ")
print_state(next_node.get_state())


if next_node.get_state() == goal_state:
print("Success:)")
print("The total of "+str(n_visited-1)+" nodes have been expanded\
\nThe max number of nodes/states in the queue in one time instance is "\
+ str(max_queue_size) + "\nTime taken is " + \
str((time.time()-time_started)*1000) + " seconds")
print_state(next_node.get_state())
return n_visited-1, max_queue_size, (time.time()-time_started)*1000 #, child_node.parent_nodes


expansion = possible_states(next_node.get_state())
for child_state in expansion:
if is_state_visited(child_state, visited_states):
#print_state(child_state)
child_node = Node(child_state)
```

```
if function == 1:
child_node.node_depth = next_node.node_depth + 1
child_node.heuristic = 0
elif function == 2:
child_node.node_depth = next_node.node_depth + 1
child_node.heuristic = heuristic_misplaced(child_node.get_state(), goal_state)
elif function == 3:
child_node.node_depth = next_node.node_depth + 1
child_node.heuristic = heuristic_manhattan(child_node.get_state(), goal_state)
my_queue.append(child_node)
visited_states.append(child_node.get_state())
queue_size += 1
if queue_size > max_queue_size:
max_queue_size = queue_size


if time.time() - time_started > 1*60*60*1000:
print("An hour passed but no solution has been found. Exit!!!")


def is_state_visited(state, array):
if state is None:
return False
for i in range(len(array)):
if state == array[i]:
return False
return True
```

```
def heuristic_misplaced(current_state, goal_state):
#goal_state = problem.goal_state()
#current_state = problem.initial_state()
h = 0
length = len(current_state)


for i in range(length):
for j in range(length):
if int(current_state[i][j]) != goal_state[i][j] and int(current_state[i][j]) != 0:
h += 1
return h


def heuristic_manhattan(current_state, goal_state):
#goal_state = problem.goal_state()
#current_state = problem.initial_state()
h = 0
length = len(current_state)
gr, gc, r, c = 0, 0, 0, 0


for l in range(1, length*length):
for i in range(length):
for j in range(length):
if int(current_state[i][j]) == l:
r = i
c = j
if goal_state[i][j] == l:
gr = i
```

```
gc = j
h += abs(gr-r) + abs(gc-c)


return h


# a function to create a custom input and goal state
def create_states():
print("Enter number of rows (=columns) in input or press Enter for 8-puzzle:")
try:
row_number = input() or int(3)
row_number = int(row_number)
if row_number <= 0:
print("Invalid number for rows")
raise ValueError
except ValueError:
print("Error: input a valid number for the number of rows in input")
exit()


print("The selected number of rows is:" + str(row_number))
print("Now, let's create the input matrix or state")


initial_state = []
for i in range(row_number):
print("Enter the " + str(i + 1) +
"-th row with space in between numbers and 0 for blank:")
input_row = input()
```

```
try:
input_row = [int(a) for a in input_row.split()]
except ValueError:
print("Invalid input!")
exit()


initial_state.append(input_row)


if not is_state_valid(initial_state, row_number):
print("There is no blank symbol; 0 in the state matrix")
return None, None
print("initial state is as below:")
print_state(initial_state)
#print("There is an error in initial state. Check ")
#let's create the default goal states:
goal_state = [[(k+1) + (row_number*m) for k in range(row_number)] for m in range(row_number)]
goal_state[row_number-1][row_number-1] = 0 # blank symbol
#global goalState
#goalState = goal_state


print("Do you want to have default goal states: y/n, press Enter for y")
print("Default goal state looks like:")
print_state(goal_state)


mode = input() or "y"
if mode != "y":
```

```
for i in range(row_number):
print("Enter the " + str(i + 1) +
"-th row with space in between numbers and 0 for blank:")
input_row = input()
try:
input_row = [int(a) for a in input_row.split()]
except ValueError:
print("Invalid input!")
exit()


goal_state.append(input_row)


if not is_state_valid(initial_state, row_number):
print("There is no blank symbol; 0 in the state matrix")
return None, None
return initial_state, goal_state


def check_solvability(state):
length = len(state)
total_inversion = 0
for row in range(length):
for column in range(length):
total_inversion += find_inversion(state, row, column)
if total_inversion % 2 == 1:
return False
return True
def find_inversion(state, row, column):
```

```
length = len(state)
inversion = 0
for i in range(length):
for k in range(length):
if i >= row and k >= column:
if state[row][column] > state[i][k] and state[i][k] != 0:
inversion += 1
return inversion


def plot():
depth0 = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
depth2 = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
depth4 = [[1, 2, 3], [5, 0, 6], [4, 7, 8]]
depth8 = [[1, 3, 6], [5, 0, 2], [4, 7, 8]]
depth12 = [[1, 3, 6], [5, 0, 7], [4, 8, 2]]
depth16 = [[1, 6, 7], [5, 0, 3], [4, 8, 2]]
depth20 = [[7, 1, 2], [4, 8, 5], [6, 3, 0]]
depth24 = [[0, 7, 2], [4, 6, 1], [3, 5, 8]]
depth = [[[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]]]
depth_array = [0, 2, 4, 8, 12, 16, 20, 24]
nodes_array1 = [] #contains #ofnodes expanded for each input for algo 1
```

```
max_qs_array1 = [] #contains max queue size for each input for algo 1
time1 = []
nodes_array2 = [] #contains #ofnodes expanded for each input for algo 2
max_qs_array2 = [] #contains max queue size for each input for algo 2
time2 = []
nodes_array3 = [] #contains #ofnodes expanded for each input for algo 3
max_qs_array3 = [] #contains max queue size for each input for algo 3
time3 = []
problem = Problem(depth0, goalState, len(goalState))
print_state(problem.goal_state)
for i in range(len(depth_array)-1):
problem.set_init_state(depth[i])
number_nodes_expanded, max_queue_size, time = general_search(problem, 1)
nodes_array1.append(number_nodes_expanded)
max_qs_array1.append(max_queue_size)
time1.append(time)


for i in range(len(depth_array)):
problem.set_init_state(depth[i])
number_nodes_expanded, max_queue_size, time = general_search(problem, 2)
nodes_array2.append(number_nodes_expanded)
max_qs_array2.append(max_queue_size)
time2.append(time)


for i in range(len(depth_array)):
problem.set_init_state(depth[i])
number_nodes_expanded, max_queue_size, time = general_search(problem, 3)
nodes_array3.append(number_nodes_expanded)
```

```
max_qs_array3.append(max_queue_size)
time3.append(time)
print("nodes array1:")
print(nodes_array1)
print("max_queue array1")
print(max_qs_array1)

print("nodes array2:")
print(nodes_array2)
print("max_queue array2")
print(max_qs_array2)

print("nodes array3:")
print(nodes_array3)
print("max_queue array3")
print(max_qs_array3)
print("time3")
print(time3)

plt.plot(depth_array[0:-1], time1, marker='p', label='Uniform Cost Search')
plt.plot(depth_array, time2, marker='p', label='Misplaced')
plt.plot(depth_array, time3, marker='p', label='Manhattan')
plt.xlabel("Solution Depth g(n)")
#plt.ylabel("Expanded Nodes")
#plt.title("Expanded Nodes vs Solution depth")
#plt.ylabel("Maximum Queue Size")
#plt.title("Maximum Queue Size vs Solution depth")
```

```
plt.ylabel("Time in seconds")
plt.title("Time vs Solution depth")
plt.legend(loc="upper left")
plt.grid()
plt.margins(x=0, y=0)
plt.xticks(depth_array)
plt.yticks([*range(0, 180000, 10000)])
plt.rc('xtick', labelsize=20)
plt.rc('ytick', labelsize=20)


plt.show()




def main():
initial_state, goal_state = create_states()
if goal_state == goalState:
if check_solvability(initial_state):
print("The puzzle is solvable.")
else:
print("The puzzle is unsolvable.")
sys.exit()
else: print("Since the goal state is different"
"the solvability check is not done. \n"
"If the puzzle is unsolvable, the system will exit after some 1 hour:)")
```

```
algo = get_algo()
#print_state(initial_state)
#print_state(goal_state)
problem = Problem(initial_state, goal_state, len(initial_state))


print("Search Started")
number_nodes_expanded, max_queue_size, time = general_search(problem, algo)
#backtrace = input("Do you want to print the states in the solution path: y/[n]") or "n"


#if backtrace == 'y':
# for i in parent_nodes:
# print_state(i.get_state())



if __name__ == "__main__":
main()
```