Project 1 CS170: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Jeng-Rung Tu

jtu016@ucr.edu

SID: 862055356

10/30/2022

In completing this assignment, I consulted:

- Blind search and heuristics search lecture provided by Dr Eamonn Keogh

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

- All array used from numpy, to handle matrix (2d-array) structure and shorter the array runtime.

Outline of this report:

CS 170 Project 1: The Eight Puzzle

Jeng-Rung Tu, SID: 862055356, Oct 30 2022

Introduction

The Eight Puzzle game is a simple sliding puzzle toy. The purpose of this game is to slide each tile until all the tiles are in a particular pattern. This game is constructed in a 3x3 grid, with 8 tiles and 1 blank. Player can move any tiles into the blank spot. Once all the tiles are in the correct spot, the player win the game. As Figure 1 shows, the goal state of the puzzle. Normally it would be set as all tiles in ascending order, and the blank spot in the bottom right. The same game can also shift into a 4x4 grid with 15 tiles, which is also known as the Fifteen Puzzle game.

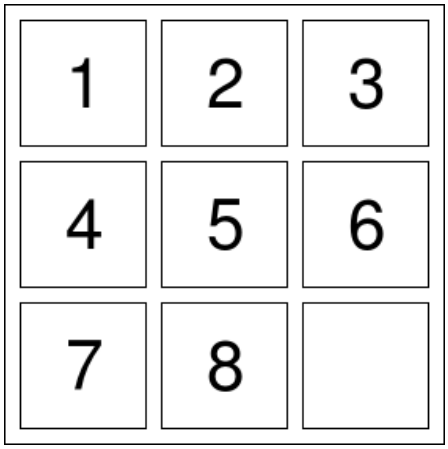

Figure 1: Goal State
Source: https://medium.com/@sairamankumar2/8-puzzle-problem-aa578104ae15

This is the first project at UCR CS170 (Introduction to Artificial Intelligence) teach by Dr. Eamonn Keogh. The purpose of the project is to write an AI program that can solve the Eight Puzzle with any given initial state. The program is constructed into 3 different algorithms, each is running independently. The user will have the ability to test out the efficiency of each algorithm. The program is written in python3 (version 3.10.6), with package “time” and “numpy”.

Comparison of Algorithm

The 3 algorithms I implemented are Uniform cost search, A* with misplaced tiles and A* with Manhattan distance. The next section will describe more in detail.

Uniform Cost Search:

As mentioned in the project prompt, the uniform cost search can also be understood as A* search but with the h(n) = 0. This this game, it will be the same format as breadth first search. The logic of the algorithm is basically iterating through all the node one by one, until we find a node that match the goal state. Because the cost of each node is all 1, so we don't need to sort the queue and the node will be explored as breadth first search order.

A* with Misplaced Tiles

This algorithm is similar to the uniform cost search, but instead of having the cost to be 1, now we are having the cost as the amount of misplaced tiles. As the figure 2 shows, comparing the initial state and the goal state, now tile 4, 7 and 8 are misplaced from the goal state, so the cost of this initial state will be 3. In the program, after we iterate through the child and add all the state into the queue with the corresponding cost. We will need to sort the queue based on the cost, then run the state with the smallest amount of cost.

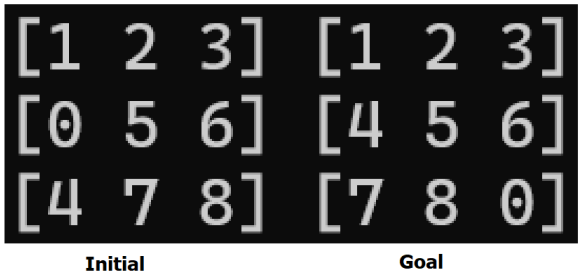

Figure 2

A* with Manhattan Distance

This is another A* search algorithm but now we are using a different method to calculate the cost of each state. For this algorithm, we calculate the cost based on the amount of movements that require for each tile to be moved into the goal state. As the figure 3 shows, we know that tiles 4, 7 and 8 are misplaced. And now we calculate that tile 4 takes 1 movement (go top) to move into the correct spot. Same as tile 7 and 8 both

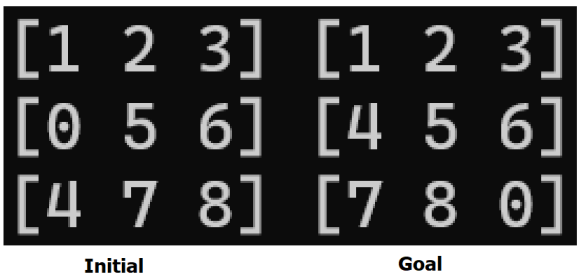

Figure 3

takes 1 movement. Therefore, we know the total amount of movement for the initial state to move into goal state is 3, we can mark this state has a cost of 3. Then the rest is the same as A* with Misplaced Tiles, we add all the child into the queue, sort the queue based on the cost and run the smallest first.

Comparison of Algorithms on Sample Puzzles

As show in the table 1 along with figure 4, I used the sample puzzles that Dr Eamonn Keogh provides to test out all 7 depth and record the expended node for each. As the table 1 shows, the x axis is the name of 3 algorithms I tested, the y axis is the depth of the puzzle and the number in each cell is the amount of expended node that takes to solve the puzzle for each algorithm. when it is running an easy puzzle (depth 2 and 4), the expended node does not have such a huge difference between 3 algorithms. However, when the puzzle got harder, the amount of expended node for uniform cost search

	uniform cost search	A* with misplaced tiles	A* with manhattan distance
0	0	0	0
2	3	2	2
4	20	4	4
8	182	18	12
16	9621	696	96
20	53059	3618	520
24	203413	22757	1868

Table 1

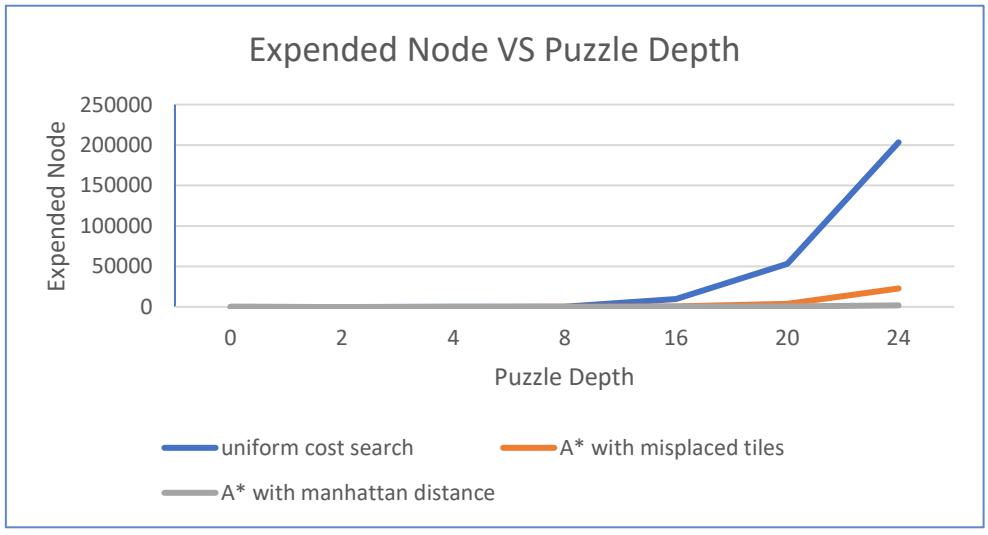

Figure 4

become really huge. At this point, we can see a clear difference between the algorithms that with and without A*.

Moreover, when it comes to solve a hard puzzle (depth 20 and 24), there is also a huge difference between A* with misplaced tile and A* with manhattan distance. When solving a depth 24 puzzle, manhattan distance only makes 1868 expended node, while misplaced tile makes 22757 expended node. If we compare more additional information between these two algorithms, when solving a depth 24 puzzle, misplaced tile takes 20 seconds and have 12494 as the maximun size of queue. While manhattan distance only takes 0.2 seconds and have 1098 as the maximun size of queue.

Conclusion

After testing all 3 algorithms with different depth of puzzles, and comparing all their expended node, maximum size of queue and time required to finish, it can be considered that:

1. The performance of both time and space, A* with manhattan distance would be the best algorithm, A* with misplaced tile would be the second and uniform cost search would be the worst.
2. Even though A* improve the proformance of the algorithm by a lot, there is still a significant difference between A* with misplaced tile and A* with manhattan distance. When using the manhattan distance, it calcuate the distance with all pieces of tiles, which improve the accuracy of algorithm.

The following is a trackback of an EASY puzzle (Depth 4) using A* with Manhattan Distance

```
Welcome to my 8-Puzzle Solver:
Type '1' to use a default puzzle
Type '2' to create your own.
1

You select to use the default puzzle.
Type '1' for Depth 0
Type '2' for Depth 2
Type '3' for Depth 4
Type '4' for Depth 8
Type '5' for Depth 16
Type '6' for Depth 20
Type '7' for Depth 24.
**Error input will automatically use Depth 16.

3
You select Depth 4
[1 2 3]
[5 0 6]
[4 7 8]


Enter your choice of algorithm:
Type '1' for Uniform Cost Search
Type '2' for A* with the Misplaced Tile heuristic
Type '3' for A* with the Manhattan distance heuristic.
3

You select to use A* with the Manhattan distance heuristic
The best state to expand with a g(n): 0 h(n): 0 is:
[1 2 3]
[5 0 6]
[4 7 8]

The best state to expand with a g(n): 1 h(n): 1 is:
[1 2 3]
[0 5 6]
[4 7 8]

The best state to expand with a g(n): 2 h(n): 2 is:
[1 2 3]
[4 5 6]
[0 7 8]

The best state to expand with a g(n): 3 h(n): 3 is:
[1 2 3]
[4 5 6]
[7 0 8]

The best state to expand with a g(n): 4 h(n): 4 is:
[1 2 3]
[4 5 6]
[7 8 0]

Goal State!

Solution path: 4
Number of nodes expanded: 4
Max queue size: 6
Algorithm finish! Taking 0.0010006427764892578 seconds
```

The following is a trackback of a MEDIUM puzzle (Depth 16) using Uniform Cost Search

** I delete 7 records (g(n) = 3-7) to save space

```
Welcome to my 8-Puzzle Solver:
Type '1' to use a default puzzle
Type '2' to create your own.
1

You select to use the default puzzle.
Type '1' for Depth 0
Type '2' for Depth 2
Type '3' for Depth 4
Type '4' for Depth 8
Type '5' for Depth 16
Type '6' for Depth 20
Type '7' for Depth 24.
**Error input will automatically use Depth 16.

5
You select Depth 16
[1 6 7]
[5 0 3]
[4 8 2]


Enter your choice of algorithm:
Type '1' for Uniform Cost Search
Type '2' for A* with the Misplaced Tile heuristic
Type '3' for A* with the Manhattan distance heuristic.
1

You select to use Uniform Cost Search
Initial State:
The best state to expand with a g(n): 0 h(n): 0 is:
[1 6 7]
[5 0 3]
[4 8 2]

The best state to expand with a g(n): 1 h(n): 0 is:
[1 6 7]
[5 3 0]
[4 8 2]

The best state to expand with a g(n): 2 h(n): 0 is:
[1 6 0]
[5 3 7]
[4 8 2]

The best state to expand with a g(n): 10        h(n): 0 is:
[1 3 0]
[5 2 6]
[4 7 8]

The best state to expand with a g(n): 11        h(n): 0 is:
[1 0 3]
[5 2 6]
[4 7 8]

The best state to expand with a g(n): 12        h(n): 0 is:
[1 2 3]
[5 0 6]
[4 7 8]

The best state to expand with a g(n): 13        h(n): 0 is:
[1 2 3]
[0 5 6]
[4 7 8]

The best state to expand with a g(n): 14        h(n): 0 is:
[1 2 3]
[4 5 6]
[0 7 8]

The best state to expand with a g(n): 15        h(n): 0 is:
[1 2 3]
[4 5 6]
[7 0 8]

The best state to expand with a g(n): 16        h(n): 0 is:
[1 2 3]
[4 5 6]
[7 8 0]

Goal State!

Solution path: 16
Number of nodes expanded: 9621
Max queue size: 8789
Algorithm finish! Taking 5.528748273849487 seconds
```

The following is a trackback of a HARD puzzle (Depth 24) using A* with Misplaced Tile

** I delete 15 records (g(n) = 3-17) to save space

```
Welcome to my 8-Puzzle Solver:
Type '1' to use a default puzzle
Type '2' to create your own.
1

You select to use the default puzzle.
Type '1' for Depth 0
Type '2' for Depth 2
Type '3' for Depth 4
Type '4' for Depth 8
Type '5' for Depth 16
Type '6' for Depth 20
Type '7' for Depth 24.
**Error input will automatically use Depth 16.

7
You select Depth 24
[0 7 2]
[4 6 1]
[3 5 8]


Enter your choice of algorithm:
Type '1' for Uniform Cost Search
Type '2' for A* with the Misplaced Tile heuristic
Type '3' for A* with the Manhattan distance heuristic.
2

You select to use A* with the Misplaced Tile heuristic
The best state to expand with a g(n): 0 h(n): 0 is:
[0 7 2]
[4 6 1]
[3 5 8]

The best state to expand with a g(n): 1 h(n): 1 is:
[7 0 2]
[4 6 1]
[3 5 8]

The best state to expand with a g(n): 2 h(n): 2 is:
[7 6 2]
[4 0 1]
[3 5 8]

The best state to expand with a g(n): 18        h(n): 8946 is:
[1 2 3]
[7 0 6]
[5 4 8]

The best state to expand with a g(n): 19        h(n): 8981 is:
[1 2 3]
[7 4 6]
[5 0 8]

The best state to expand with a g(n): 20        h(n): 13450 is:
[1 2 3]
[7 4 6]
[0 5 8]

The best state to expand with a g(n): 21        h(n): 20269 is:
[1 2 3]
[0 4 6]
[7 5 8]

The best state to expand with a g(n): 22        h(n): 22299 is:
[1 2 3]
[4 0 6]
[7 5 8]

The best state to expand with a g(n): 23        h(n): 22686 is:
[1 2 3]
[4 5 6]
[7 0 8]

The best state to expand with a g(n): 24        h(n): 22757 is:
[1 2 3]
[4 5 6]
[7 8 0]

Goal State!

Solution path: 24
Number of nodes expanded: 22757
Max queue size: 12494
Algorithm finish! Taking 27.455851793289185 seconds
```

Demo of the code

URL to my code GitHub: https://github.com/JengRung/CS170_The-Eight-Puzzle

```
import time
import numpy    # Help to handle array

class puzzle:
    def __init__(self, customMatrix):
        self.matrix = customMatrix
        self.parent = None
        self.height = 0
        self.cost = 0
        self.expended = 0   # Number of expended node
        self.maxQueue = 0   # Max size of queue

    def printMatrix(self):
        for row in self.matrix:
            print(row)

    # Return an array of all the possible child from the current matrix
    def getChild(self):

        # Set x and y for 0's position
        for i in range(len(self.matrix)):
            for j in range(len(self.matrix[i])):
                if self.matrix[i][j] == 0:
                    x, y = i, j
                    break

        # Swap 2 number then return a new matrix
        def swap(x1,y1,x2,y2):
            temp = self.matrix.copy()
            temp[x1][y1], temp[x2][y2] = temp[x2][y2], temp[x1][y1]
            return temp

        # Check all 4 childs and add to the array if psosible
        child = []

        # Right
        if x+1 < len(self.matrix):
            newChild = puzzle(swap(x, y, x+1, y))

            # Update path and cost for new child
            newChild.parent = self
            newChild.height = self.height + 1

            child.append(newChild)

        # Left
        if x-1 >= 0:
            newChild = puzzle(swap(x, y, x-1, y))

            # Update path and cost for new child
            newChild.parent = self
            newChild.height = self.height + 1

            child.append(newChild)

        # Down
        if y+1 < len(self.matrix[x]):
            newChild = puzzle(swap(x, y, x, y+1))

            # Update path and cost for new child
            newChild.parent = self
            newChild.height = self.height + 1
```

```
            child.append(newChild)

        # Up
        if y-1 >= 0:
            newChild = puzzle(swap(x, y, x, y-1))

            # Update path and cost for new child
            newChild.parent = self
            newChild.height = self.height + 1

            child.append(newChild)

        return child

    # Check if the current matrix is the same as goal
    def checkGoalState(self, goal):
        if numpy.array_equal(self.matrix, goal):
            return True
        else:
            return False

    # Get matrix
    def getMatrix(self):
        return self.matrix

# Searching function: uniform_cost_search, misplaced_tiles, manhattan_distance

def uniform_cost_search(start, goal):

    # Check if the start and goal are the same
    if start.checkGoalState(goal):
        return start

    # Create a queue to store all the possible path
    queue = []
    queue.append(start)

    # An array to keep track of all the visited matrix
    seen = []

    # Count the expended node, increment 1 when a node is pop from the queue
    expendCnt = 0

    # Record the max size of the queue
    maxQueue = len(queue)

    # Loop until the queue is empty
    while(True):
        # Check if queue is empty, then return failure
        if len(queue) == 0:
            return False

        # Get the first element in the queue
        current = queue.pop(0)
        expendCnt += 1

        # Check for the goal state
        if current.checkGoalState(goal):
            # Store the max size of the queue to the result Node
            current.maxQueue = maxQueue
            return current

        # Add current to seen, convert numpy array to list to be able to compare
        seen.append(current.getMatrix().tolist())

        # Get all the child of the current matrix
        child = current.getChild()
```

```
        # Loop through all the child
        for c in child:
            # If child not in seen, Add the child to the queue
            if c.getMatrix().tolist() not in seen:
                # Update expend node count
                c.expended = expendCnt
                queue.append(c)

        # Update max queue size
        if len(queue) > maxQueue:
            maxQueue = len(queue)

# Helper function to calculate the misplaced tiles
# Logic: Calculate the total number of misplaced tiles
def misplaced_tiles(matrix, goal):
    count = 0
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            if matrix[i][j] != goal[i][j] and matrix[i][j] != 0:
                count += 1
    return count

def a_star_with_misplaced_tiles(start, goal):

    # Check if the start and goal are the same
    if start.checkGoalState(goal):
        return start

    # Create a queue to store all the possible path
    queue = []
    queue.append(start)

    # An array to keep track of all the visited matrix
    seen = []

    # Count the expended node, increment 1 when a node is pop from the queue
    expendCnt = 0

    # Record the max size of the queue
    maxQueue = len(queue)

    # Loop until the queue is empty
    while(True):
        # Check if queue is empty, then return failure
        if len(queue) == 0:
            return False

        # Get the first element in the queue
        current = queue.pop(0)
        expendCnt += 1

        # Check for the goal state
        if current.checkGoalState(goal):
            # Store the max size of the queue to the result Node
            current.maxQueue = maxQueue
            return current

        # Add current to seen, convert numpy array to list to be able to compare
        seen.append(current.getMatrix().tolist())

        # Get all the child of the current matrix
        child = current.getChild()

        # Loop through all the child
        for c in child:
            # If child not in seen, Add the child to the queue with the cost
            if c.getMatrix().tolist() not in seen:
                # Calculate the cost of the child base on misplaced tiles
                c.cost = misplaced_tiles(c.getMatrix(), goal) + c.height
```

```
                # Update expend node count
                c.expended = expendCnt
                queue.append(c)

        # Update max queue size
        if len(queue) > maxQueue:
            maxQueue = len(queue)

        # Sort the queue by cost
        queue = sorted(queue, key=lambda x: x.cost, reverse=False)


# Logic: Iterate through the matrix and find the number that is not in the right position, calcyulate the
distance between the current position and the goal position, then add the distance to the total count
def manhattan_distance(matrix, goal):
    count = 0
    for x1 in range(len(matrix)):
        for y1 in range(len(matrix[x1])):
            # Ignore 0 in the matrix
            if matrix[x1][y1] != 0:
                # Find the position of the number in the goal
                for x2 in range(len(goal)):
                    for y2 in range(len(goal[x2])):
                        if goal[x2][y2] == matrix[x1][y1]:
                            count += abs(x1-x2) + abs(y1-y2)
    return count


def a_star_with_manhattan_distance(start, goal):

    # Check if the start and goal are the same
    if start.checkGoalState(goal):
        return start

    # Create a queue to store all the possible path
    queue = []
    queue.append(start)

    # An array to keep track of all the visited matrix
    seen = []

    # Count the expended node, increment 1 when a node is pop from the queue
    expendCnt = 0

    # Record the max size of the queue
    maxQueue = len(queue)

    # Loop until the queue is empty
    while(True):
        # Check if queue is empty, then return failure
        if len(queue) == 0:
            return False

        # Get the first element in the queue
        current = queue.pop(0)
        expendCnt += 1

        # Check for the goal state
        if current.checkGoalState(goal):
            # Store the max size of the queue to the result Node
            current.maxQueue = maxQueue
            return current

        # Add current to seen, convert numpy array to list to be able to compare
        seen.append(current.getMatrix().tolist())

        # Get all the child of the current matrix
        child = current.getChild()
```

```
        # Loop through all the child
        for c in child:
            # If child not in seen, Add the child to the queue with the cost
            if c.getMatrix().tolist() not in seen:
                # Calculate the cost of the child base on misplaced tiles
                c.cost = manhattan_distance(c.getMatrix(), goal) + c.height
                # Update expend node count
                c.expended = expendCnt
                queue.append(c)

        # Update max queue size
        if len(queue) > maxQueue:
            maxQueue = len(queue)

        # Sort the queue by cost
        queue = sorted(queue, key=lambda x: x.cost, reverse=False)

# A helper function to print the matrix
def printMatrix(matrix):
    for m in matrix:
        print(m)
    print()

# Main function start here
def main():
    # Default puzzle set
    # Goal state
    goal=numpy.array(
            [[1,2,3],
            [4,5,6],
            [7,8,0]])

    # Depth 0
    puzzle0=numpy.array(
            [[1,2,3],
            [4,5,6],
            [7,8,0]])

    # Depth 2
    puzzle2=numpy.array(
            [[1,2,3],
            [4,5,6],
            [0,7,8]])

    # Depth 4
    puzzle4=numpy.array(
            [[1,2,3],
            [5,0,6],
            [4,7,8]])

    # Depth 8
    puzzle8=numpy.array(
            [[1,3,6],
            [5,0,2],
            [4,7,8]])

    # Depth 16
    puzzle16=numpy.array(
            [[1,6,7],
            [5,0,3],
            [4,8,2]])

    # Depth 20
    puzzle20=numpy.array(
            [[7,1,2],
            [4,8,5],
            [6,3,0]])

    # Depth 24
```

```
    puzzle24=numpy.array(
            [[0,7,2],
            [4,6,1],
            [3,5,8]])

    print("Welcome to my 8-Puzzle Solver: \nType '1' to use a default puzzle  \nType '2' to create your
own.")
    choice = int(input())

    if choice == 1:
        print("\nYou select to use the default puzzle. \nType '1' for Depth 0  \nType '2' for Depth 2
\nType '3' for Depth 4 \nType '4' for Depth 8 \nType '5' for Depth 16 \nType '6' for Depth 20 \nType '7'
for Depth 24. \n**Error input will automatically use Depth 16.\n")
        choicePuz = int(input())
        if choicePuz == 1:
            print("You select Depth 0")
            puzzleChoice = puzzle0
        elif choicePuz == 2:
            print("You select Depth 2")
            puzzleChoice = puzzle2
        elif choicePuz == 3:
            print("You select Depth 4")
            puzzleChoice = puzzle4
        elif choicePuz == 4:
            print("You select Depth 8")
            puzzleChoice = puzzle8
        elif choicePuz == 5:
            print("You select Depth 16")
            puzzleChoice = puzzle16
        elif choicePuz == 6:
            print("You select Depth 20")
            puzzleChoice = puzzle20
        elif choicePuz == 7:
            print("You select Depth 24")
            puzzleChoice = puzzle24
        else:
            print("Using default Depth 16")
            puzzleChoice = puzzle16

        printMatrix(puzzleChoice)

    if choice == 2:
        print("\nYou select to create your own puzzle. Please enter your puzzle, use a zero to represent
the blank. Enter each row, use space between numbers. Example: 1 2 3")
        print("please enter the first row")
        firstRow = input()
        print("please enter the second row")
        secondRow = input()
        print("please enter the third row")
        thirdRow = input()

        puzzleChoice = numpy.array(
                    [[int(firstRow[0]),int(firstRow[2]),int(firstRow[4])],
                    [int(secondRow[0]),int(secondRow[2]),int(secondRow[4])],
                    [int(thirdRow[0]),int(thirdRow[2]),int(thirdRow[4])]])

    start = puzzle(puzzleChoice)

    print("\nEnter your choice of algorithm: \nType '1' for Uniform Cost Search \nType '2' for A* with the
Misplaced Tile heuristic \nType '3' for A* with the Manhattan distance heuristic.")
    choice = int(input())
    print()

    if choice == 1:
        print("You select to use Uniform Cost Search")
        # Calculate the time to run the algorithm
        startTime = time.time()
        result = uniform_cost_search(start, goal)
        endTime = time.time()
```

```
        # Store result nodes expand
        nodeExpand = result.expended
        depth = result.height
        maxQueue = result.maxQueue

        # Store the solution path to an array, then print it out with the corresponding height
        path = []
        while(result.parent != None):
            path.append((result.getMatrix(), result.height, result.cost))
            result = result.parent
        path.append((result.getMatrix(), result.height, result.cost))
        path.reverse()

        print("Initial State: ")
        for p in path:
            print("The best state to expand with a g(n): " + str(p[1]) + "\th(n): " + str(p[2]) + " is:")
            printMatrix(p[0])

        print("Goal State! \n")
        print("Solution path: " + str(depth) + "\nNumber of nodes expanded: " + str(nodeExpand) + "\nMax
queue size: " + str(maxQueue))

    if choice == 2:
        print("You select to use A* with the Misplaced Tile heuristic")
        # Calculate the time to run the algorithm
        startTime = time.time()
        result = a_star_with_misplaced_tiles(start, goal)
        endTime = time.time()

        # Store result nodes expand
        nodeExpand = result.expended
        depth = result.height
        maxQueue = result.maxQueue

        # Store the solution path to an array, then print it out with the corresponding height
        path = []
        while(result.parent != None):
            path.append((result.getMatrix(), result.height, result.expended))
            result = result.parent
        path.append((result.getMatrix(), result.height, result.expended))
        path.reverse()

        for p in path:
            print("The best state to expand with a g(n): " + str(p[1]) + "\th(n): " + str(p[2]) + " is:")
            printMatrix(p[0])

        print("Goal State! \n")
        print("Solution path: " + str(depth) + "\nNumber of nodes expanded: " + str(nodeExpand) + "\nMax
queue size: " + str(maxQueue))

    if choice == 3:
        print("You select to use A* with the Manhattan distance heuristic")
        # Calculate the time to run the algorithm
        startTime = time.time()
        result = a_star_with_manhattan_distance(start, goal)
        endTime = time.time()

        # Store result nodes expand
        nodeExpand = result.expended
        depth = result.height
        maxQueue = result.maxQueue

        # Store the solution path to an array, then print it out with the corresponding height
        path = []
        while(result.parent != None):
            path.append((result.getMatrix(), result.height, result.expended))
            result = result.parent
        path.append((result.getMatrix(), result.height, result.expended))
```

```
        path.reverse()

        for p in path:
            print("The best state to expand with a g(n): " + str(p[1]) + "\th(n): " + str(p[2]) + " is:")
            printMatrix(p[0])

        print("Goal State! \n")
        print("Solution path: " + str(depth) + "\nNumber of nodes expanded: " + str(nodeExpand) + "\nMax
queue size: " + str(maxQueue))

    print(f"Algorithm finish! Taking {endTime - startTime} seconds \n \n")

if __name__ == "__main__":

    main()
```