Anthony Dupont

CSCI 4202 – Program 1 State Space Search

Sep 9th 2019

Introduction:

A sliding-tile puzzle is a rectangular grid of tile with one empty space. Given a n x n sliding puzzle grid where n > 1, write a series of program that will take a start state and solve it to reach the goal state by moving the empty tile.

Problem Description:

Given an n x n grid and a starting state, write a series of program that will take the start state and works towards solving itself to the goal state, one step at a time, applying the rules of the game. The first algorithm will be a backtracking algorithm that will be limited by a depth bound. It will be a depth first search algorithm and keep applying the first rule applicable until it find the solution or it reaches the limit, if the limit is reached, we backtrack once and we apply the next rule instead. The second algorithm is a graph search which will be a breadth-first search. Given a state, we expand the state by creating all successors of this state and utilize an open and closed queue to determine which states to look at. The third algorithm is called A* and uses graph search with a function f(n) = g(n) + h(n). This function will utilize a priority queue where when we have to choose a node to expand, we choose the cheapest node available given by the f(n) function.

Solution:

The rest of the code can be found at the bottom of the pdf, (Classes and functions)

Backtrack1:

```
       def backtrack1(datalist, ruleObj, stateObj, rulesApplied, stateExamined):
    number = stateExamined[0]   # I'm using an array to simulate a pass by reference with my
number variable
    number += 1
    stateExamined[0] = number
    # Grab the newest data, which will always be at the 0th index
    data = datalist[0]  # This grabs a grid
    # Check data against the entire list to make sure it's not repeating an earlier data
    for i in range(len(datalist))[1:]:  # This will start at i = 1
        if data == datalist[i]:
            return stateObj.FAIL
    # Check if we have a match with the goal
    if data == GOALSTATE:
        return stateObj.NIL
    # if the length of the datalist is greater than the depth bound limit, return fail
    if len(datalist) > BOUND:
        return stateObj.FAIL
    # Grab the rules that we can apply to the current grid
    ruleList = ruleObj.applicableRules(data)
    # Loop here
    loopOver = False
    while loopOver is False:
        # if we have no rules from above, return fail
        if not ruleList:    # this checks if the list is null/empty
```

```
            return stateObj.FAIL
        # else grab the first rule from the list of rules
        R = ruleList[0]     # 0th index is always going to be the first
        ruleList.pop(0)
        # Create the new grid by applying the rules
        newGrid = ruleObj.applyRule(data,R)
        rulesApplied.insert(0, R)
        # Add the new grid into the new list of grids
        newDatalist = datalist.copy()
        newDatalist.insert(0, newGrid)
        # call backtrack1 and store the return value
        returnValue = backtrack1(newDatalist, ruleObj, stateObj, rulesApplied, stateExamined)
        # if variable == fail, go loop
        if returnValue is not stateObj.FAIL:
            loopOver = True
        # else it was a fail and we remove the rule we applied
        else:
            rulesApplied.pop(0)

    return rulesApplied    # We want to return a list of all the rules that were applied
    # technically it's always updated and we don't need to return but oh well
    # End loop here
```

With backtrack1 I used a global variable for the Depth to constantly check against to see if I needed to backtrack or not. I also check to see if there are any rules left applicable and if there aren't any, we backtrack. If we're not backtracking then we are getting the rules that are applicable for our current state and applying the very first rule and repeating until a backtrack happens or the goal state is reached. If everything fails, we return a fail and the global variable for the depth is increased by one and everything repeats.

Graph-Search:

```
def graphSearch(currentState, rules, state):
    # When you create the successors, a shortcut is to add all of those nodes to open
    # It will take longer to solve but it will still solve the solution
    # Create first node
    nodeS = GraphNode()
    nodeS.currentState = copy.deepcopy(currentState)     # we use a deep copy to not affect the
current state
    name = 0
    nodeS.name = name
    name += 1
    nodeS.next1 = None
    # Create open queue and fill it
    open = queue.Queue()
    open.put(nodeS) # we place the first state in the queue
    # Create closed queue
    closed = queue.Queue()
    # Create variables for the loop
    path = []
    loopOver = False
    stateToAdd = GraphNode()
    currentState = GraphNode()
    m = []  # this will be the list of all successors of the current state
    statesGenerated = 0
    statesExplored = 0
    while loopOver == False:
        if open.empty():    # this means that there are no more open states left and we failed
the find a solution
            print("Failed :(")
            return state.FAIL
```

```
        # Retrieve a node and expand it
        n = open.get(0) # retrieve a state from open and get rid of it from open
        statesExplored += 1
        closed.put(n)   # add the state we retrieved to closed
        path.append(n.currentState)
        if n.currentState == GOALSTATE:  # we found a solution if this is true
            # Display length of path and then return
            pathCost = 0
            while n.next1 != None:
                pathCost += 1
                if pathCost == 1:
                    n.printGrid(False)  # The false is for an arrow, I know this is misleading,
sorry
                else:
                    n.printGrid(True)
                print()
                n = n.next1
            nodeS.printGrid(True)
            print()
            print("Total cost of path: ", pathCost)
            print("States generated: ", statesGenerated)
            print("States explored: ", statesExplored)
            return state.NIL
        ruleList = rules.applicableRules(n.currentState)
        # Creating all successors for expansion
        while len(ruleList) > 0:
            r = ruleList[0]
            ruleList.pop(0)
            successor = rules.applyRule(n.currentState, r)
            m.append(successor)
            statesGenerated += 1
        # m is a set of all successors of n currently
        while len(m) > 0:
            # if the successor already exists in closed, don't add it
            if m[0] not in path:    # using path because Queue is not iterable
                # The successor is not in closed so we do want to add it to open
                currentState.currentState = copy.deepcopy(m[0])
                # make it point back to n
                currentState.next1 = n
                currentState.name = name
                name += 1
                stateToAdd = copy.deepcopy(currentState)
                open.put(stateToAdd)
                m.pop(0)
            else:
                m.pop(0)
```

For graph search, I used a class to hold details about my node, the current state, the name, the pointer, the heuristic even though it's not used for this algorithm. I went ahead and created the very first node, initialized it with the current state, pointer to null, and placed it in the queue called open since it needs to be evaluated. I go ahead and start the loop and check if the queue is empty or not. If it is empty than we have failed, if it's not empty I retrieve what is inside the queue and compare it against the goal state to check if we've succeeded with this node, if not, I create all successors and make them all point to the node that is responsible for their creation, and I add them all into open if they are not currently in the closed queue. I'm not allowed to iterate over a queue since that's a property of a queue, so I created a list called path to hold the same values as the queue and compared the successors against the path instead. And then the loop repeats.

A*:

```
def aStarAlg(currentState, rules, state):
# very similar to graph search except we will be using a priority queue and H(n)
# H(n) = H1(n) + H2(n) which will be the amount of wrong tiles and the manhattan distance
name = 0    # will increment, this will be used for my priority queue to never have a duplicate
# h1 = 0
h2 = 0
nodeS = GraphNode()
nodeS.currentState = copy.deepcopy(currentState)  # we use a deep copy to not affect the current
state
nodeS.name = name
# h1 = wrongTilesAmount(currentState)
h2 = manhattanDistance(currentState)
nodeS.heuristic = h2# h1 + h2
nodeS.next1 = None
name += 1

# Create open priority queue and fill it
open = PriorityQueue()
open.put((nodeS.heuristic, nodeS))  # we place the first state in the queue with h(n) being the
priority
# Create closed queue
closed = queue.Queue()

# Create variables for the loop
path = []
possibleNodes = []
loopOver = False
heuristicDuplicates = True
stateToAdd = GraphNode()
currentState = GraphNode()
m = []  # this will be the list of all successors of the current state
statesGenerated = 0
statesExplored = 0

while loopOver == False:
    if open.empty():  # this means that there are no more open states left and we failed the find
a solution
        print("Failed :(")
        return state.FAIL
    # Retrieve a node and expand it
    a = open.get()
    # since I copied graph search and everything uses n, this was easier to to instead of
replacing every n with n[1]
    n = a[1]
    statesExplored += 1
    closed.put(n)  # add the state we retrieved to closed
    path.append(n.currentState)
    if n.currentState == GOALSTATE:  # we found a solution if this is true
        # Display length of path and then return
        pathCost = 0
        moves = 0
        while n.next1 != None:
            pathCost += n.heuristic
            moves += 1
            if pathCost == n.heuristic: # This will only be false for the very first drawing
                n.printGrid(False)  # The false is for an arrow, I know this is misleading, sorry
            else:
                n.printGrid(True)
            print()
            n = n.next1
        pathCost += nodeS.heuristic # this is needed since the first heustic of the loop was 0
(goal)
        nodeS.printGrid(True)
        print()
        print("Total heuristic cost of path: ", pathCost)
        print("Total moves: ", moves)
```

```
        print("States generated: ", statesGenerated)
        print("States explored: ", statesExplored)
        return state.NIL
    ruleList = rules.applicableRules(n.currentState)
    # Creating all successors for expansion
    while len(ruleList) > 0:
        r = ruleList[0]
        ruleList.pop(0)
        successor = rules.applyRule(n.currentState, r)
        m.append(successor)
        statesGenerated += 1
    # m is a set of all successors of n currently
    while len(m) > 0:
        # if the successor already exists in closed, check if the current one has a smaller h(n)
        if m[0] not in path:  # using path because Queue is not iterable
            # The successor is not in closed so we do want to add it to open
            currentState.currentState = copy.deepcopy(m[0])
            # make it point back to n
            currentState.next1 = n
            currentState.name = name
            name += 1
            # h1 = wrongTilesAmount(currentState.currentState)
            h2 = manhattanDistance(currentState.currentState)
            currentState.heuristic = h2 # h1 + h2
            stateToAdd = copy.deepcopy(currentState)
            open.put((stateToAdd.heuristic, stateToAdd))
            m.pop(0)
        else:
            # check against already existing node
            m.pop(0)
```

A* used the same exact logic as graph search except that open was replaced to be a priority queue. The priority is based on the heuristic of that node which is calculated by using a function that returns to us how many wrong tile a current state has, or another function that returns the Manhattan distance of that state. A* is called twice, one with the Manhattan distance, and one with the wrong tile amount. Inside of the priority queue, if two nodes have the same h(n), then the priority will be picked based on the name which the name is a simple integer that starts at 0 and increments every time a node is created. If a node has a low number, it was created earlier. This allows me to chose which node should be chosen when a tie happens.

Results:

Backtrack1:

```
Current grid:                 Goal State:

[3, 1, 2]                      [0, 1, 2]

[4, 7, 0]                      [3, 4, 5]

[6, 8, 5]                      [6, 7, 8]


Backtrack1 is being performed...

Failed to find a solution for Depth Bound:  1

Visited  4  states. Increasing bound...

Failed to find a solution for Depth Bound:  2
```

```
Visited  16  states. Increasing bound...
Failed to find a solution for Depth Bound:  3
Visited  43  states. Increasing bound...
Failed to find a solution for Depth Bound:  4
Visited  94  states. Increasing bound...

Solution:
1 . down
2 . left
3 . up
4 . left
5 . up
Examined  291  total amount of states!
```

Graph Search:

```
Current grid:                 Goal State:
[3, 1, 2]                      [0, 1, 2]
[4, 7, 0]                      [3, 4, 5]
[6, 8, 5]                      [6, 7, 8]

Graph search is being performed...
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]

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

      /\
      |
```

[3, 1, 2]

[4, 0, 5]

[6, 7, 8]

/\

|

[3, 1, 2]

[4, 7, 5]

[6, 0, 8]

/\

|

[3, 1, 2]

[4, 7, 5]

[6, 8, 0]

/\

|

[3, 1, 2]

[4, 7, 0]

[6, 8, 5]

Total cost of path: 5

States generated: 118

States explored: 46

A*:

A* is being performed with H1...

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

[3, 1, 2]

```
[0, 4, 5]
[6, 7, 8]

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

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

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

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

Total heuristic cost of path:  20
Total moves:  5
States generated:  15
States explored:  6
```

A* is being performed with H2...

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

[3, 1, 2]

[0, 4, 5]

[6, 7, 8]

/\

|

[3, 1, 2]

[4, 0, 5]

[6, 7, 8]

/\

|

[3, 1, 2]

[4, 7, 5]

[6, 0, 8]

/\

|

[3, 1, 2]

[4, 7, 5]

[6, 8, 0]

/\

|

[3, 1, 2]

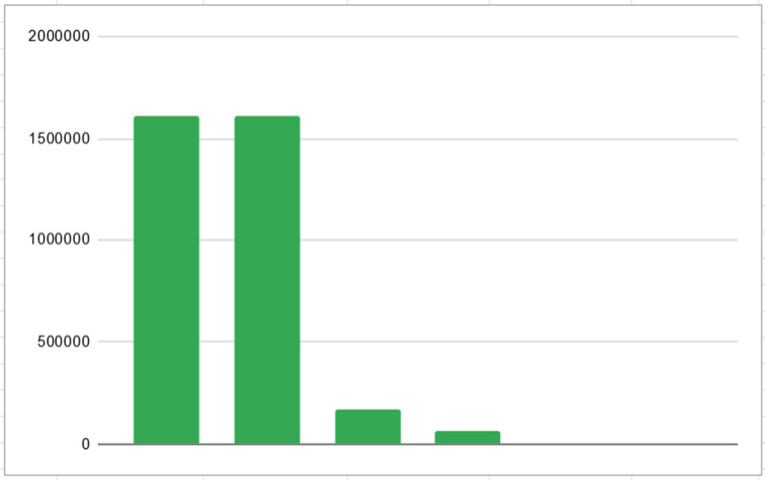
[4, 7, 0]

[6, 8, 5]

```
Total heuristic cost of path:  23

Total moves:  5

States generated:  17

States explored:  7
```

Conclusion:

	A	B	C	D	E	F	G	H
1	Backtrack		Graph Search		A*			
2	Nodes Generated	Nodes examined	Nodes Generated	Nodes examined	Nodes Generated	Nodes examined		
3	1609887	1609887	173324	63771	231	84		20-moves.json
4	127247	127247	23822	8879	212	77		15-moves.json



The graph displays the results of each algorithm for the 20-moves.json file. A* also doesn't show up on the graph because of how many little nodes it generated. A* is by far the best algorithm out of the three, it generates less states and also explores less than backtrack1 and graph search. Graph search I only tested up to 20-moves.json. It works for everything from 20 and under. 20-moves.json takes about 20-30 minutes to solve, I've tested 25-moves.json for about an hour and still didn't get a result, I can only assume that it works since it works for everything less than or equal to 20 moves. In the results section above, I used the 5-moves.json input file which is pretty small but I didn't want to make this report more than 20 pages with a longer input file. If I used the 10-moves.json, it would lengthen this report by quite a lot. Keep in mind that my A* is not completely working. It's more of an A algorithm than an A* because it doesn't get the most optimized path but it finds a solution very quickly (not a bad solution either) and generated barely any nodes and explores even less so my algorithm specifically is still good at getting answers. Also my code is very messy for this assignment and I would like to give a few reasons as to why. This was my very first time with python, so I still have some habits from other languages like declaring my variables above a loop or creating my variable before I use it. In python I can create the variable the exact line that I need it. I went with python because I wanted a language that would have a garbage collector to not deal with memory leak. I quickly realized that I didn't like the object oriented side of python. I was hoping to apply a lot of object oriented in this assignment but it seems like although python does support it, it's not a language you want to go to when you want to use

object oriented programming. I can't give clear reasons as to why I don't like it, I know a lot of others agree with me, whether our opinion is valid or not. This led me to make functions for everything and pass in classes into it instead, there's a lot of functions and the functions themselves are very lengthy and can be very hard to read so I apologize if you get lost in my code.

Additional code:

(Not necessarily in order)

```
# This is what I use to replicate an ENUM for now
class State():
    FAIL = 0
    NIL = 1

class GraphNode():
    def __init__(self):
        self.currentState = [[]]
        self.heuristic = 0
        self.next1 = None
        self.next2 = None
        self.next3 = None
        self.next4 = None
        self.name = 0
        # max connections is 4 connections

    def printGrid(self, drawArrow):
        # This one is different since I don't have a size variable
        if drawArrow == True:
            print("\t/\\")
            print("\t|")
        for i in range(len(self.currentState)):
            print(self.currentState[i])

    def __lt__(self, other):
        return self.name < other.name


class Puzzle:
    def __init__(self):
        self.movesToSolve = 0
        self.size = 0
        self.currentState = []   #[0,0,0,
                                 # 0,0,0,
                                 # 0,0,0]
        self.rules = Rules()
        self.returnTypes = State()

    def readJsonFile(self, fileName):
        # https://stackabuse.com/reading-and-writing-json-to-a-file-in-python/
        with open(fileName) as json_file:
            data = json.load(json_file)
        return data

    # This is the function that will make sure the json file is a valid one
    def setUp(self, jsonData):
        # First error checking, check if ALL the fields exists in the first place, then check
value, if it all passed, store it
        if "n" not in jsonData:
            print("Json file does not have a field for n")
            return False
        if "start" not in jsonData:
```

```
            print("Json file does not have a field for start")
            return False
        if "goal" not in jsonData:
            print("Json file does not have a field for goal")
            return False
        # Second error checking, the actual values inside of the json file
        # We know the fields exist right now
        if jsonData["n"] < 1: # If this is true, there is an error with the value
            print("Json file does not have a valid number for n")
            return False
        else:
            self.size = jsonData["n"];

        # ------------------------------------------------------------------------------
        # at this point we know that all fields have a value in them
        global GOALSTATE
        #global BOUND
        #if self.size == 3:
        #    BOUND = 31
        #elif self.size == 4:
        #    BOUND = 80
        gridList = []
        gridInt = []
        current = jsonData["start"]
        goal = jsonData["goal"]
        if validateGrid1(current, self.size): # if this is true, the grid is valid
            self.currentState = jsonData["start"]
            #for i in range(self.size):
            #    list1 = current[i]
            #    gridList += list1
            # Let's convert the list into ints to be able to comparegoal = jsonData["goal"]
            #for i in range(self.size * self.size):
            #    self.currentState.append(int(gridList[i]))
        else:
            print("Json file does not have a valid grid for start")
            return False
        if validateGrid1(goal, self.size):
            GOALSTATE = jsonData["goal"]
            #gridList.clear()
            #for i in range(self.size):
            #    list1 = goal[i]
            #    gridList += list1
            # Let's convert the list into ints to be able to compare
            #for i in range(self.size * self.size):
            #    GOALSTATE.append(int(gridList[i]))
        else:
            print("Json file does not have a valid grid for goal")
            return False
        # if we are at this point, we have past all the checks and the json file is valid
        return True

    def displayGrid(self):
        print("Current grid:\t\t\tGoal State:")
        for i in range(self.size):
            print(self.currentState[i], end='')
            print("\t\t\t\t", GOALSTATE[i])
        #for i in range(self.size*self.size):
        #    if i % self.size is 0:
        #        print("[", self.currentState[i], " ", end='')
        #    elif i % self.size is self.size - 1:
        #        print(self.currentState[i], "]")
        #    else:
        #        print(self.currentState[i], " ", end='')


class Rules:
```

```
    def __init__(self):
        self.up = "up"
        self.down = "down"
        self.right = "right"
        self.left = "left"

    def applicableRules(self, puzzleGrid):
        #Find the index of 0, that will tell us where our empty spot is
        #We know the grid is valid so we know there is a 0, we can just check for it instead of
checking if it exists
        indexI = getWhiteSpaceIndexI(puzzleGrid)
        indexJ = getWhiteSpaceIndexJ(puzzleGrid)
        #[4,1,2]
        #[3,0,5]
        #[6,7,8]
        rules = []
        size = len(puzzleGrid)

        if indexI - 1 >= 0:
            rules.append(self.up)
        if indexJ + 1 < size:
            rules.append(self.right)
        if indexI + 1 < size:
            rules.append(self.down)
        if indexJ - 1 >= 0:
            rules.append(self.left)
        return rules

    def applyRule(self, puzzleGrid, rule):
        if rule is self.up:
            return self.applyUp(puzzleGrid)
        if rule is self.right:
            return self.applyRight(puzzleGrid)
        if rule is self.down:
            return self.applyDown(puzzleGrid)
        if rule is self.left:
            return self.applyLeft(puzzleGrid)

    # When you apply a rule, make sure the grid is being coppied and you return the new grid
    def applyUp(self, puzzleGrid):
        size = len(puzzleGrid)
        indexI = getWhiteSpaceIndexI(puzzleGrid)
        indexJ = getWhiteSpaceIndexJ(puzzleGrid)
        newIndex = indexI - 1
        value = puzzleGrid[newIndex][indexJ]
        copiedGrid = copy.deepcopy(puzzleGrid)
        # copiedGrid = puzzleGrid.copy()
        copiedGrid[newIndex][indexJ] = 0
        copiedGrid[indexI][indexJ] = value
        return copiedGrid

    def applyDown(self, puzzleGrid):
        size = len(puzzleGrid)
        indexI = getWhiteSpaceIndexI(puzzleGrid)
        indexJ = getWhiteSpaceIndexJ(puzzleGrid)
        newIndex = indexI + 1
        value = puzzleGrid[newIndex][indexJ]
        copiedGrid = copy.deepcopy(puzzleGrid)
        # copiedGrid = puzzleGrid.copy()
        copiedGrid[newIndex][indexJ] = 0
        copiedGrid[indexI][indexJ] = value
        return copiedGrid

    def applyRight(self, puzzleGrid):
        size = len(puzzleGrid)
        indexI = getWhiteSpaceIndexI(puzzleGrid)
```

```
        indexJ = getWhiteSpaceIndexJ(puzzleGrid)
        newIndex = indexJ + 1
        value = puzzleGrid[indexI][newIndex]
        copiedGrid = copy.deepcopy(puzzleGrid)
        # copiedGrid = puzzleGrid.copy()
        copiedGrid[indexI][newIndex] = 0
        copiedGrid[indexI][indexJ] = value
        return copiedGrid

    def applyLeft(self, puzzleGrid):
        size = len(puzzleGrid)
        indexI = getWhiteSpaceIndexI(puzzleGrid)
        indexJ = getWhiteSpaceIndexJ(puzzleGrid)
        newIndex = indexJ - 1
        value = puzzleGrid[indexI][newIndex]
        copiedGrid = copy.deepcopy(puzzleGrid)
        # copiedGrid = puzzleGrid.copy()
        copiedGrid[indexI][newIndex] = 0
        copiedGrid[indexI][indexJ] = value
        return copiedGrid



def wrongTilesAmount(currentState):
    wrongTile = 0
    for i in range(len(currentState)):
        for j in range(len(currentState[i])):
            if currentState[i][j] != GOALSTATE[i][j]:
                wrongTile += 1
    return wrongTile


def manhattanDistance(currentState):
    rowSize = len(currentState)
    columnSize = len(currentState[0])
    manhattanNumber = 0
    for i in range(rowSize):
        for j in range(columnSize):
            if currentState[i][j] != GOALSTATE[i][j]:
                # calculate the manhattan distance of this tile
                if currentState[i][j] != 0:
                    currentIndexJ = j
                    currentIndexI = i
                    indexI = getIndexI(currentState, GOALSTATE[i][j])
                    indexJ = getIndexJ(currentState, GOALSTATE[i][j])
                    distanceI = abs(indexI - currentIndexI)
                    distanceJ = abs(indexJ - currentIndexJ)
                    manhattanNumber += distanceI + distanceJ
    return manhattanNumber



# Use this function since we are using a vector of vectors
def validateGrid1(grid,size):
    # The grid is recieved as a vector of vectors
    numberArray = []    # we will use this array to check if there is a duplicate number in the
grid or not
    for i in range(size):
        for j in range(size):
            if grid[i][j] >= 0 and grid[i][j] <= ((size * size) - 1):
                if grid[i][j] not in numberArray:
                    numberArray.append(grid[i][j]);
                else:
                    return False
            else:
                return False
```

```
    # if we're here, there was no duplicates and all the numbers were fine
    return True


# -----------------------------PROGRAM BEGINS HERE-----------------------------------
print("Hello I'm python and I look very simple which is the complicated part!")

# PART 1, read json file and make sure it's a proper file
validJson = False
obj = Puzzle()
while validJson != True:
    jsonFileName = input("Please type the json filename: ")
    jsonData = obj.readJsonFile(jsonFileName)
    if obj.setUp(jsonData) == True:
        print("")  # I'm going to use this for a new line
        obj.displayGrid()
        validJson = True
# END OF PART 1


# PART 2, create the rules for the sliding part
rules = Rules()
# END OF PART 2

# PART 3 backtracking
state = State()
datalist = []
rulesApplied = []
datalist.append(obj.currentState)
stateExamined = []
stateExamined.append(0)
loop = True
totalStatesVisited = 0
print()
print("Backtrack1 is being performed...")
while loop == True:
    if backtrack1(datalist, rules, state, rulesApplied, stateExamined) == state.FAIL:
        print("Failed to find a solution for Depth Bound: ", BOUND)
        print("Visited ", stateExamined[0], " states. Increasing bound...")
        BOUND += 1
        totalStatesVisited += stateExamined[0]
    else:
        totalStatesVisited += stateExamined[0]
        loop = False
printWinningPath(rulesApplied, totalStatesVisited)
obj.displayGrid()
print()
# END OF PART 3

# PART 4 graph search
print("Graph search is being performed...")
graphSearch(obj.currentState, rules, state)
# END OF PART 4

# PART 5 A* algorithm
print()
print("A* is being performed with H1...")
aStarAlgH1(obj.currentState, rules, state)
print()
print("A* is being performed with H2...")
aStarAlgH2(obj.currentState, rules, state)
# What would be better is to combine aStarAlgH1 and H2 to just one algorithm, I separated them
because I was using
# one of the copies to try to improve my algorithm, I understand how messy this is. I don't like
```

```
it either
print("I don't know why I struggle so much with python :)!")
```