How to use the 8square Program

1. Have 8square.py and input files in the same folder. You can find them all at https://github.com/ml5803/AI_Project

2. Run 8square.py. You can do this through double clicking the file or through cmd. It will prompt you to enter the name of the input file. ***PYTHON3 IS NECESSARY***

```
root@Michael: /mnt/c/users/micha/github/ai_project
root@Michael:/mnt/c/users/micha/github/ai_project# python3 8square.py
Please enter the name of input file:
```

3. Enter the name of the input file including the extension. Then enter 1 or 2 depending on which heuristic you would like to use.

```
root@Michael: /mnt/c/users/micha/github/ai_project
root@Michael:/mnt/c/users/micha/github/ai_project# python3 8square.py
Please enter the name of input file:
input1.txt
Choose one of the following:
 1: Sum of Manhattan distances
 2: Sum of Manhattan distances + 2 x # linear conflicts
2
```

4. Let it run and print to console the solution and generate the output file.

- Note: the state updates may be disabled by commenting out the print in Puzzle.next_state() method on line 118.

```
root@Michael: /mnt/c/users/micha/github/ai_project
root@Michael:/mnt/c/users/micha/github/ai_project# python3 8square.py
Please enter the name of input file:
input1.txt
Choose one of the following:
 1: Sum of Manhattan distances
 2: Sum of Manhattan distances + 2 x # linear conflicts
2
Initial: [[7, 1, 6], [8, 3, 5], [2, 0, 4]]
Goal: [[8, 7, 6], [1, 0, 5], [2, 3, 4]]
5 U 5 [[7, 1, 6], [8, 0, 5], [2, 3, 4]]
9 U 5 [[7, 0, 6], [8, 1, 5], [2, 3, 4]]
13 L 5 [[0, 7, 6], [8, 1, 5], [2, 3, 4]]
17 D 5 [[8, 7, 6], [0, 1, 5], [2, 3, 4]]
21 R 5 [[8, 7, 6], [1, 0, 5], [2, 3, 4]]
depth: 5
# nodes generated:  21
U U L D R
5 5 5 5 5 5
```

```
input1_B.txt
7 1 6
8 3 5
2 0 4

8 7 6
1 0 5
2 3 4

5
21
U U L D R
5 5 5 5 5 5
```

Source code:

```
#Michael Li

#CS 4613

#8Puzzle


#PYTHON3 ONLY PLEASE!

try:

    import queue

except ImportError:

    import Queue as queue

import math

import copy


class Node:

    def __init__(self, state, goal, move = None, parent = None, option = 1):

        self.depth = 0 if parent == None else parent.depth + 1

        self.state = state

        self.move = move

        self.cost = 9999999 if state == None else self.cost(goal, option) #state is None if invalid move returned - cost very
large so not expanded

        self.parent = parent


    #calculates cost for each node

    def cost(self,goal,option = 1):

        #g(n) = depth, h(n) = sum of manhattan_distance (+ 2 * linear_conflicts)

        cost = self.depth + manhattan_distance(self.state, goal)

        if option == 2:

            cost += 2 * num_linear_conflicts(self.state, goal)

        return cost
```

```
class Puzzle:

    def __init__(self,initial, goal, option = 1):

        self.curr_state = Node(initial,goal,option)

        self.goal = goal

        self.node_count = 1

        self.pq = queue.PriorityQueue()

        self.option = option

        self.solution_actions = [] #actions from initial node to goal node stored backwards

        self.solution_costs = [] #costs from initial node to goal node stored backwards


    #while the Puzzle instance hasn't been solved, decide next move and update the current states

    #also prints states expanded so user gets to see the program running - can comment out print in next state if desired

    #if solution found, print depth, # nodes generated, actions and costs along solution path

    def solve(self):

        while(not self.check_goal()):

            self.expand()

            self.next_state()


        print("depth:" , self.curr_state.depth)

        print("# nodes generated: ", self.node_count)


        ptr = self.curr_state

        while(ptr.parent != None):

            self.solution_actions.append(ptr.move)

            self.solution_costs.append(ptr.cost)

            ptr = ptr.parent

        #put root node cost into list

        self.solution_costs.append(ptr.cost)


        for i in range(len(self.solution_actions)-1,-1,-1):
```

```
            print(self.solution_actions[i],end = " ")
        print()
        for j in range(len(self.solution_costs)-1,-1,-1):
            print(self.solution_costs[j],end = " ")
        print()
        return self.curr_state #solution node

    #loops through 2d array to check in if curr_state is the goal
    def check_goal(self):
        #if all items match, found goal state - return true, else return false
        for i in range(len(self.curr_state.state)):
            for j in range(len(self.curr_state.state[0])):
                if (self.curr_state.state[i][j] != self.goal[i][j]):
                    return False
        return True

    #selects minimal cost from PriorityQueue, expands the node to 4 child nodes {L,R,U,D}
    #if move is invalid, node gets assigned a large constant to avoid being expanded.
    def expand(self):
        if self.pq.empty():
            #if pq is empty, put initial in pq and expand - only in very first run
            self.pq.put((self.curr_state.cost,1, self.curr_state))
        #if not empty, get lowest cost and expand
        to_expand = self.pq.get()

        poss_expansions = {"L","R","U","D"}

        for moves in poss_expansions:
            new_node = Node(self.move(moves),goal,moves, to_expand[2], self.option)
            self.pq.put((new_node.cost, self.node_count, new_node))
```

```
      self.node_count+=1


  #given a move, create new 2d list representing state if that move was done
  #if move is not valid, return None - happens on edge cases e.g. 0 at [0,0] and move would be L or U
  #if move valid, return 2d list representing new state (2d list)
  def move(self, move):
    state = copy.deepcopy(self.curr_state.state)
    dict_state = convert_dict(self.curr_state.state)
    zero = dict_state[0]

    #if zero located on edges and were to move out of bounds, return nothing
    if (zero[1] == 0 and move == "L" ) or (zero[1] == 2 and move == "R") or (zero[0] == 0 and move == "U") or (zero[0] == 
2 and move == "D"):
      return None
    if move == "L":
      state[zero[0]][zero[1]], state[zero[0]][zero[1]-1] = state[zero[0]][zero[1]-1],state[zero[0]][zero[1]]
    if move == "R":
      state[zero[0]][zero[1]], state[zero[0]][zero[1]+1] = state[zero[0]][zero[1]+1],state[zero[0]][zero[1]]
    if move == "U":
      state[zero[0]][zero[1]], state[zero[0]-1][zero[1]] = state[zero[0]-1][zero[1]],state[zero[0]][zero[1]]
    if move == "D":
      state[zero[0]][zero[1]], state[zero[0]+1][zero[1]] = state[zero[0]+1][zero[1]],state[zero[0]][zero[1]]
    return state


  #updates curr_state with next expanded node
  #prints here to show user that updates to curr_state, may disable if desired.
  def next_state(self):
    #update curr_state with next expanded node without removing from pq
    #update path records
    self.curr_state = self.pq.queue[0][2]
```

```
    print(self.node_count, self.curr_state.move,self.curr_state.cost, self.curr_state.state)


#generates output file
#lines 1 - 3 - initial state, lines 4-6 goal state, line 9 depth, line 10 total nodes generated (including invalid moves)
#line 11 - actions of solution path, #line 12 - costs of solution path
def make_output_file(self, filename, heuristic, initial, goal):
    filename = filename.split(".")
    filename[0] = filename[0].replace("Input","Output")
    if (heuristic == 1):
        filename[0] += "_A." #manhattan_distance
    elif (heuristic == 2):
        filename[0] += "_B." #linear_conflicts
    strfilename = ""
    strfilename = strfilename.join(filename)

    f= open(strfilename,"w+")
    row = len(initial)
    col = len(initial[0])

    for i in range(row):
        for j in range(col):
            f.write(str(initial[i][j])+ " ")
        f.write("\n")
    f.write("\n")
    for i in range(row):
        for j in range(col):
            f.write(str(goal[i][j]) + " ")
        f.write("\n")

    f.write("\n")
```

```
        f.write(str(self.curr_state.depth) + "\n")
        f.write(str(self.node_count )+ "\n")
        for i in range(len(self.solution_actions)-1,-1,-1):
            f.write(str(self.solution_actions[i]) + " ")

        f.write("\n")
        for j in range(len(self.solution_costs)-1,-1,-1):
            f.write(str(self.solution_costs[j]) + " ")
        return

#makes a 2d list of initial and goal states
#reads file prompted by user
#returns [initialstate(list), goalstate(list)]
def make_initial_goal(file):
    init = []
    goal = []

    i = 0
    for line in open(file, "r").readlines():
        if i < 3:
            init.append([ int(i) for i in line.split()])
        elif i > 3:
            goal.append([ int(i) for i in line.split()])
        i += 1
    return [init,goal]

#converts a list to a dictionary
def convert_dict(lst):
    dic = dict()
```

```
    for row in range(len(lst)):
        for col in range(len(lst[row])):
            dic[lst[row][col]] = [row, col]
    return dic

#converts a dictionary to a list/grid
def convert_list(dic):
    #set up grid
    lst = []
    temp = []
    root = int(math.sqrt(len(dic)))
    for i in range(root):
        temp.append("*")
    for j in range(root):
        lst.append(temp.copy())

    #lst = [["*","*","*"],["*","*","*"],["*","*","*"]]
    for num,rowcol in dic.items():
        lst[rowcol[0]][rowcol[1]] = num
    return lst

#given a state and a goal, return the Manhattan distances
#state - 2d list, goal - 2d list
def manhattan_distance(state, goal):
    sum = 0;
    state = convert_dict(state)
    goal = convert_dict(goal)
    for i in range(1,9,1):
        init_row, init_col = state[i][0], state[i][1]
        goal_row, goal_col = goal[i][0], goal[i][1]
```

```
        sum += abs(goal_row - init_row) + abs(goal_col - init_col)
    return sum


#given a state and a goal, return # of linear conflicts
#state - 2d list, goal - 2d list
def num_linear_conflicts(state,goal):
    state = convert_dict(state)
    goal = convert_dict(goal)
    sum = 0

    for i in range(1, 9):
        initial1_row, initial1_col = state[i][0], state[i][1]
        for j in range(1, 9):
            initial2_row, initial2_col = state[j][0], state[j][1]
            #check if on same row or col on state
            check_row = (initial2_row == initial1_row and initial2_col > initial1_col)
            check_col = (initial2_col == initial1_col and initial2_row > initial1_row)
            if check_row or check_col:
                goal_initial2_row, goal_initial2_col = goal[j][0], goal[j][1]
                goal_initial1_row, goal_initial1_col = goal[i][0], goal[i][1]
                #check if conflicts exist on goal state
                check_row_goal = (goal_initial2_row == goal_initial1_row and goal_initial2_col < goal_initial1_col) and
(initial2_row == goal_initial2_row)
                check_col_goal = (goal_initial2_col == goal_initial1_col and goal_initial2_row < goal_initial1_row) and
(initial2_col == goal_initial2_col)
                if check_row_goal or check_col_goal:
                    #print(i, "and",j," are conflicting")
                    sum += 1
    return sum


#main body of code
```

```
#takes in user input and runs code accordingly

#first prompt = input filename, second prompt = which heuristic 1: Manhattan distance 2: Manhattan + 2 * Linear conflict

#create Puzzle instance, solve and generate output file

if __name__ == "__main__":

    user_input = []

    user_input.append(input("Please enter the name of input file:\n"))

    user_input.append(int(input("Choose one of the following:\n 1: Sum of Manhattan distances\n 2: Sum of Manhattan distances + 2 x # linear conflicts\n")))


    rep = make_initial_goal(user_input[0])

    initial, goal = rep[0], rep[1]

    print("Initial:" ,initial)

    print("Goal:" , goal)


    p = Puzzle(initial, goal, user_input[1])

    # print(p.move("L"))

    p.solve()

    p.make_output_file(user_input[0],user_input[1],initial,goal)
```

Output files:

Input1 – Manhattan distance:

7 1 6

8 3 5

2 0 4

8 7 6

1 0 5

2 3 4

5

21

U U L D R

5 5 5 5 5 5

Input1 – Manhattan distance + 2 * Linear conflicts:

7 1 6

8 3 5

2 0 4

8 7 6

1 0 5

2 3 4

5

21

U U L D R

5 5 5 5 5 5

Input2 – Manhattan distance:

2 6 0

1 3 4

7 5 8

1 2 3

4 5 6

7 8 0

10

57

L D R U L L D R D R

10 10 10 10 10 10 10 10 10 10 10

Input2 – Manhattan distance + 2 * Linear conflicts:

2 6 0

1 3 4

7 5 8

1 2 3

4 5 6

7 8 0

10

53

L D R U L L D R D R

10 12 12 10 10 10 10 10 10 10 10

Input3 – Manhattan distance:

5 4 3

2 6 7

1 8 0

1 2 3

4 5 6

7 8 0

22

8037041

U L U L D D R U U L D D R R U L L D R U R D

12 12 12 12 12 12 12 14 16 16 16 16 18 18 18 20 22 22 22 22 22 22 22

Input3 – Manhattan distance + 2 * Linear conflicts:

5 4 3

2 6 7

1 8 0

1 2 3

4 5 6

7 8 0

22

263681

U L D R U U L L D D R U U R D L U L D R R D

12 14 14 14 16 18 20 22 22 22 22 20 20 20 20 20 22 22 22 22 22 22 22

Input4 – Manhattan distance:

8 7 3

0 4 5

6 2 1

1 2 3

4 5 6

7 8 0

23

84113

U R D D R U L D L U U R D R D L L U U R D R D

17 17 17 19 19 19 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23

Input4 – Manhattan distance + 2 * Linear conflicts:

8 7 3

0 4 5

6 2 1

1 2 3

4 5 6

7 8 0

23

29441

U R D D R U L D L U U R D R D L L U U R D R D

17 17 17 19 21 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23