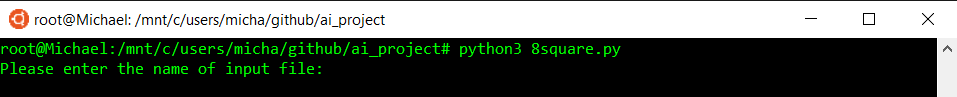
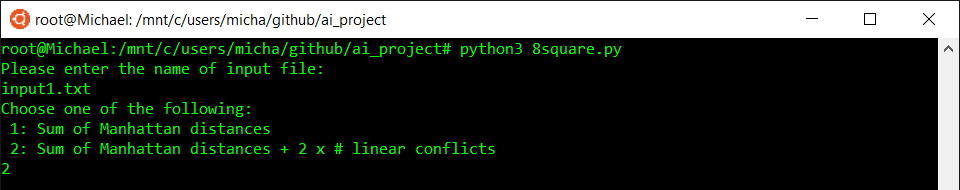
**How to use the 8square Program**

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

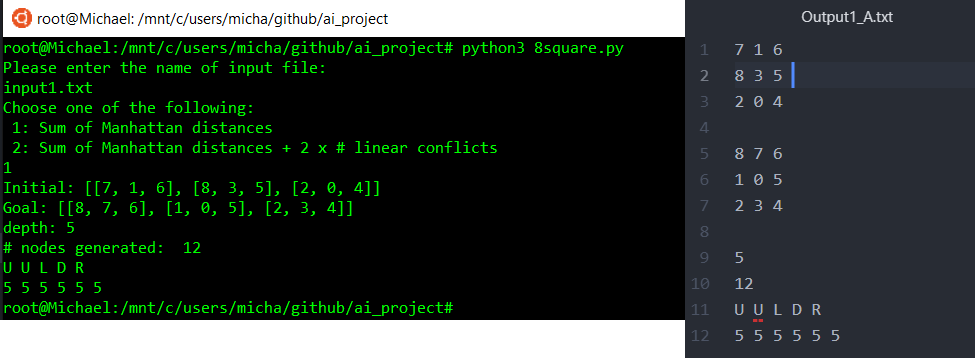
2. Run 8square.py. You can do this through double clicking the file or through cmd. It will prompt you to enter the name of the input file. **\*\*\*PYTHON3 IS NECESSARY\*\*\***



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.



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



**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.visited = [] #list of all visited states

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))

self.visited.append(self.curr\_state.state)

#if not empty, get lowest cost and expand

to\_expand = self.pq.get()

poss\_expansions = {"U","D","L","R"}

for moves in poss\_expansions:

new\_node = Node(self.move(moves),goal,moves, to\_expand[2], self.option)

if (new\_node.state != None): #if it's a valid move

if(new\_node.state not in self.visited): #if not visited, put Node in pq

self.pq.put((new\_node.cost, self.node\_count, new\_node))

#print(new\_node.depth, new\_node.cost, new\_node.move, new\_node.state)

self.node\_count+=1

# else: #valid move, but already reached before in frontier

# #need to update cost and node if cheaper node found

# for i in range(len(self.pq.queue)):

# existing\_node = self.pq.queue[i] #is actually a tuple (cost, # generated, node)

# if (existing\_node[2].state == new\_node.state and existing\_node[0] > new\_node.cost): #cheaper node found

# new\_pq = queue.PriorityQueue()

# #update pq of expandable nodes, update with cheaper node

# while (not self.pq.empty()):

# tup = self.pq.get()

# if (tup[0] > new\_node.cost):

# new\_pq.put((new\_node.cost, self.node\_count, new\_node))

# if (tup[2].state != new\_node.state):

# new\_pq.put(tup)

# self.pq = new\_pq

# self.node\_count += 1

# return

#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]

self.visited.append(self.curr\_state.state)

#print(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

12

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

12

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

27

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

24

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

2200

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

914

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

1245

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

616

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