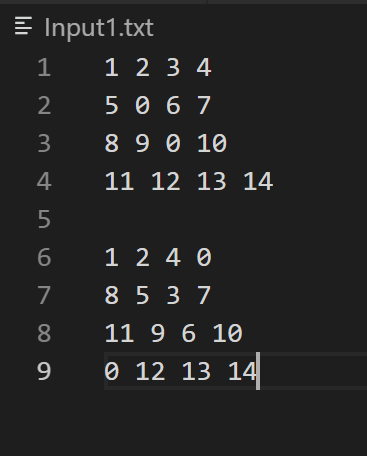
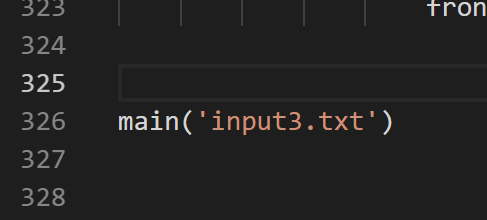
14 Puzzle- Instructions

1) Have a text file in the following format:



This input file should be in the same directory as project\_1.py (the code to solve the puzzle)

2) Write the name of the input file in line 326:



-In the above example, the input file is called input3.txt.

-Replace the ‘input3.txt’ with the name of your input file follow by the ‘.txt’

3) Run the Python file

Prerequisite: Python and a code compiler

How to run using VSCode:



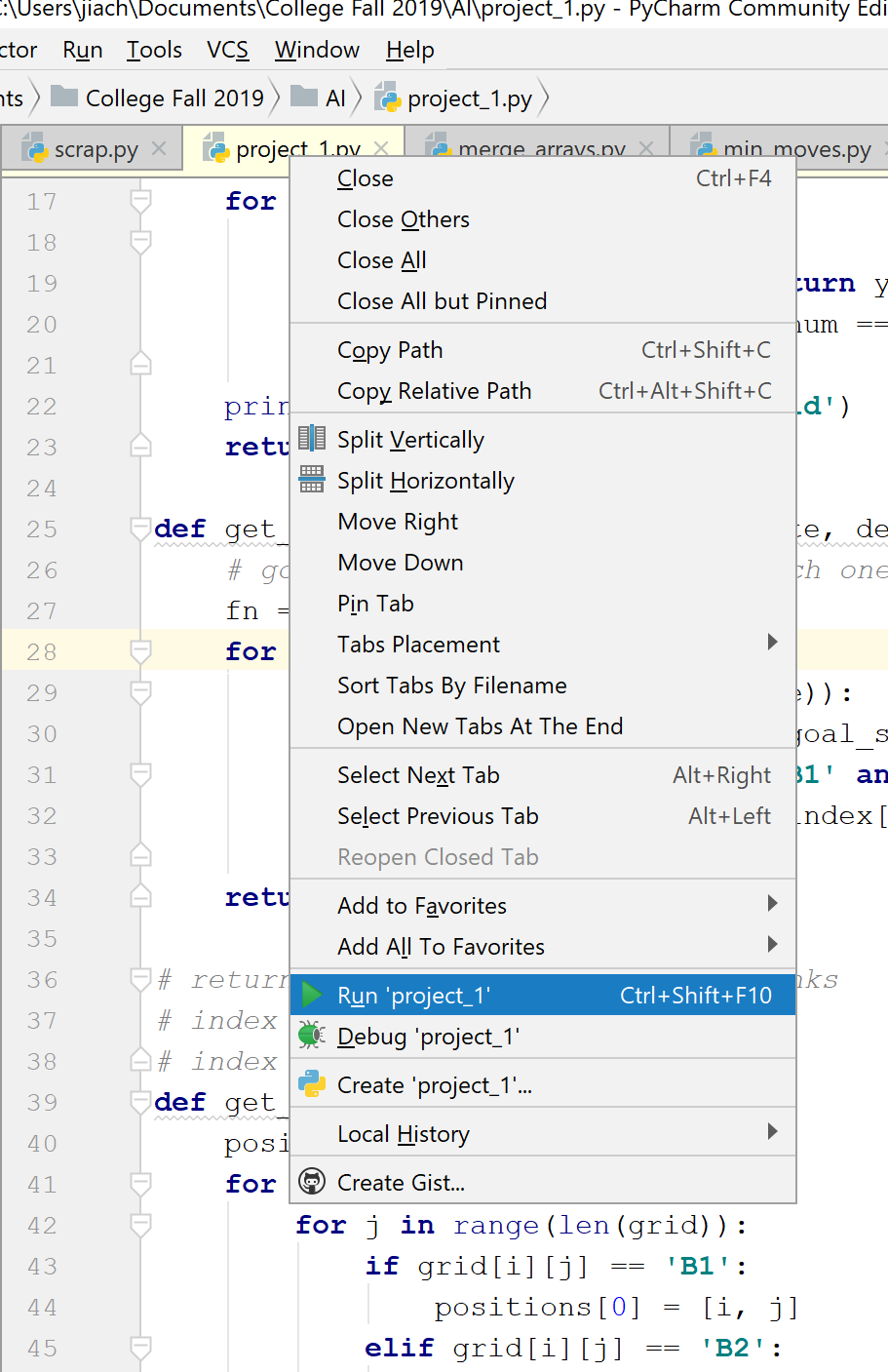
-Go to the directory containing the source code file; in my case it is …..\AI

-Type: python <name of the source code file> in terminal

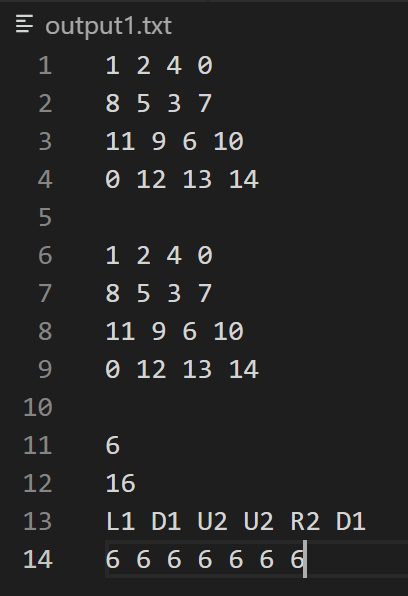
-In my case the name is project\_1.py

How to run using PyCharm:

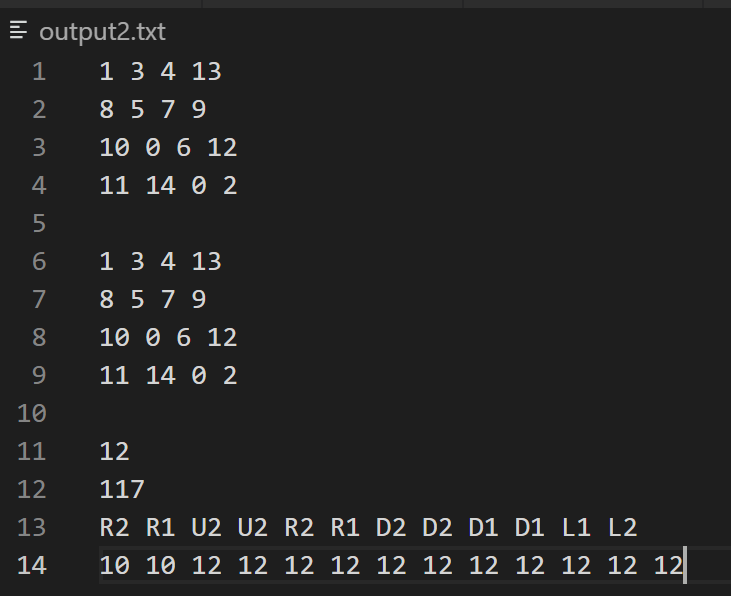
-Go to the file and press the run button in PyCharm:



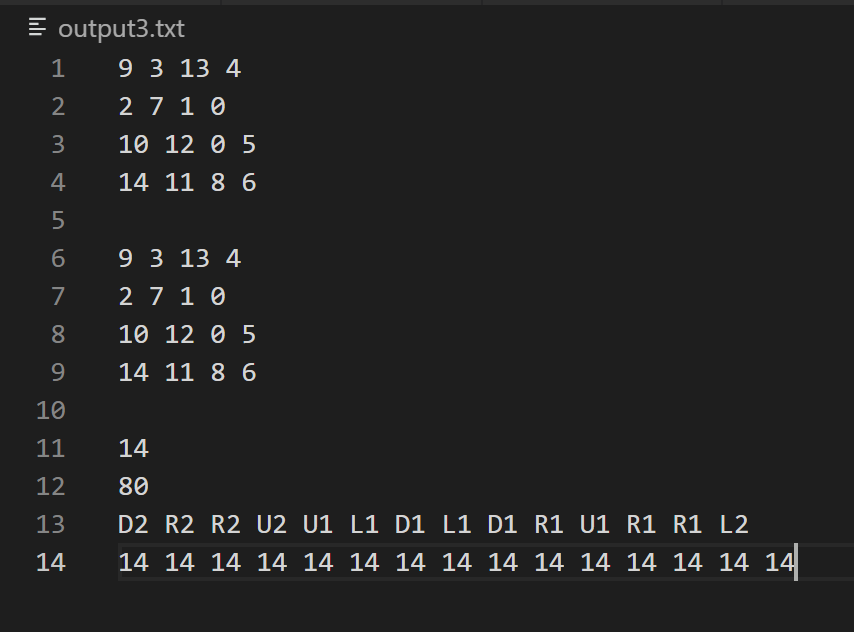
Output 1:



Output 2:



Output 3:



Code:

import math

from copy import copy, deepcopy

class Node:

    def \_\_init\_\_(self, state, parent, action, depth, fn):

        self.state = state

        self.parent = parent

        self.action = action

        self.depth = depth

        self.children = []

        self.fn = fn

# num could be B1 or B2

def find\_index(grid, num):

    for y in range(len(grid)):

        for x in range(len(grid)):

            if grid[y][x] == num: return y, x

            if grid[y][x] == 0 and (num == 'B1' or num == 'B2'):

                return y,x

    print('Error- num not in goal grid')

    return None

def get\_fn\_dist(curr\_state, goal\_state, depth):

    # go through each one and for each one check it's corresponding

    fn = 0

    for y in range(len(curr\_state)):

        for x in range(len(curr\_state)):

            goal\_index = find\_index(goal\_state, curr\_state[y][x])

            if curr\_state[y][x] != 'B1' and curr\_state[y][x] != 'B2':

                dist = abs(y - goal\_index[0]) + abs(x - goal\_index[1]) # dist = sum of abs diff of indexes

                fn += dist

    return fn + depth

# returns the positions for both blanks

# index 0 - position of first blank

# index 2 - position of second blank

def get\_blank\_positions(grid):

    positions = [[0,0], [0,0]]

    for i in range(len(grid)):

        for j in range(len(grid)):

            if grid[i][j] == 'B1':

                positions[0] = [i, j]

            elif grid[i][j] == 'B2':

                positions[1] = [i, j]

            # if len(positions)==2: break

    return positions

# new\_pos is good if the next step is not out of range

# next step can replace a blank only if the corressponding blank can move as well

# not out of range

def is\_new\_pos(new\_pos, state, new\_index):

    if new\_pos < 0 or new\_pos > 3:

        return False

    elif state[new\_index[0]][new\_index[1]] == 0:

        return False

    return True

# returns the new\_state

# i = action

# b = blank position

# swap states if valid; swap even if both are blanks

def apply\_actions(i, node, b, res\_state, blank\_name):

    not\_found = True

    if i == 'L':

        if b[1] - 1 >= 0:

            not\_found = False

            num = res\_state[b[0]][b[1] - 1]

            # move blank to left

            res\_state[b[0]][b[1]-1] = blank\_name

            # move num to right (curr\_pos)

            res\_state[b[0]][b[1]] = num

    elif i == 'R':

        if b[1] + 1 <= 3:

            not\_found = False

            num = res\_state[b[0]][b[1] + 1]

            # move blank to right

            res\_state[b[0]][b[1]+1] = blank\_name

            # move num to left (curr\_pos)

            res\_state[b[0]][b[1]] = num

    elif i == 'U':

        if b[0] - 1 >= 0:

            not\_found = False

            num = res\_state[b[0] - 1][b[1]]

            # move blank up

            res\_state[b[0]-1][b[1]] = blank\_name

            # move num down (curr\_pos)

            res\_state[b[0]][b[1]] = num

    elif i == 'D':

        if b[0] + 1 <= 3:

            not\_found = False

            num = res\_state[b[0] + 1][b[1]]

            # move blank down

            res\_state[b[0]+1][b[1]] = blank\_name

            # move num up (curr\_pos)

            res\_state[b[0]][b[1]] = num

    if not\_found: return None

    return res\_state

# returns a deep copy of the state

def get\_copy\_state(state):

    res\_state = []

    for row in state:

        res\_state.append(row[:])

    return res\_state

# return the children of all possible states

def get\_states(blank\_pos, node):

    b1\_x = blank\_pos[0][1]

    b1\_y = blank\_pos[0][0]

    b2\_x = blank\_pos[1][1]

    b2\_y = blank\_pos[1][0]

    actions = ['L', 'R', 'U', 'D']

    state\_lst = []

    node\_state = node.state

    # gest actions for moving first blank first

    for i in actions:

        # create deep copy of node state

        res\_state = get\_copy\_state(node\_state)

        res\_state = apply\_actions(i, node, (b1\_y, b1\_x), res\_state, 'B1')

        if res\_state != None:

            state\_lst.append([res\_state, i+'1'])

    # get actions for moving second blank

    for i in actions:

        # create deep copy of node state

        res\_state = get\_copy\_state(node\_state)

        res\_state = apply\_actions(i, node, (b2\_y, b2\_x), res\_state, 'B2')

        if res\_state != None:

            state\_lst.append([res\_state, i+'2'])

    return state\_lst

# check if the given state is == goal\_state

def is\_goal(node\_state, goal\_state):

    # create deep copy of node state

    curr\_state = []

    for row in node\_state:

        curr\_state.append(row[:])

    # switch B1 and B2 into 0

    switch\_count = 0

    for i in range(len(node\_state)):

        for x in range(len(node\_state[i])):

            if (node\_state[i][x] == 'B1') or (node\_state[i][x] == 'B2'):

                curr\_state[i][x] = 0

                switch\_count += 1

            if switch\_count == 2: break

    # check

    # print('checking---------')

    # print('curr:')

    # for i in curr\_state: print(i)

    # print('goal:')

    # for i in goal\_state: print(i)

    return curr\_state == goal\_state

# prints the final result once the state is == goal\_state

def print\_res(node, goal\_state, counter):

    leaf = node

    action\_lst = []

    man\_dist\_lst = []

    # print('depth:', leaf.depth)

    while leaf.parent != None:

        action\_lst.append(leaf.action)

        man\_dist\_lst.append(leaf.fn)

        leaf = leaf.parent

    man\_dist\_lst.append(leaf.fn)

    # print final state

    for i in node.state:

        row = ''

        for j in i:

            if j == 'B1' or j == 'B2':

                j = 0

            row += str(j) + ' '

        print(row)

    # print goal state

    print('')

    for i in node.state:

        row = ''

        for j in i:

            if j == 'B1' or j == 'B2':

                j = 0

            row += str(j) + ' '

        print(row)

    print('')

    # print depth of tree

    print(node.depth)

    # print total number of nodes in tree

    print(counter)

    res = ''

    # print actions

    for i in action\_lst[::-1]:

        res += str(i) + ' '

    print(res)

    res = ''

    # print the f(n) values for each node

    for i in man\_dist\_lst[::-1]:

        res += str(i) + ' '

    print(res)

def main(file\_name):

    '''

    test grid

    '''

    i\_grid = [0]\*4

    for i in range(len(i\_grid)):

        i\_grid[i] = [0] \* 4

    for i in i\_grid:

        print(i)

    '''

    Extract inital and goal state from file

    '''

    with open(file\_name) as f:

        lines = f.readlines()

    # strip \n from the rows

    for i in range(len(lines)):

        lines[i] = lines[i].strip() # gets rid of any \n

    print(lines)

    set\_blank\_2 = 0

    initial\_state = []

    goal\_state = []

    change\_flag = 0

    for line in lines:

        # empty line seperates input from goal state

        if line == '':

            change\_flag = 1

            continue

        if change\_flag:

            row = line.split(' ')

            for i in range(len(row)):

                row[i] = int(row[i])

            goal\_state.append(row)

        else:

            row = line.split(' ')

            for i in range(len(row)):

                row[i] = int(row[i])

            initial\_state.append(row)

    # set names for blanks

    for i in range(len(initial\_state)):

        for x in range(len(initial\_state[i])):

            if set\_blank\_2 and initial\_state[i][x] == 0:

                initial\_state[i][x] = 'B2'

                break

            elif initial\_state[i][x] == 0:

                initial\_state[i][x] = 'B1'

                set\_blank\_2 = 1

    # see grid formation

    print('initial:')

    for i in initial\_state:

        print(i)

    print('final:')

    for i in goal\_state:

        print(i)

    frontier = [] # priority queue of all unexplored nodes

    explored = []

    man\_dist = get\_fn\_dist(initial\_state, goal\_state, 0)

    root = Node(initial\_state, None, None, 0, man\_dist)

    frontier.append(root)

    depth = 0

    counter= 0

    while len(frontier) != 0:

        counter += 1

        print('counter:', counter, '----------------------------')

        # search for node with lowest f(n)

        node = frontier[0]

        count = 0

        index = 0

        for i in frontier:

            if i.fn < node.fn:

                index = count

                node = i

            count += 1

        # check if current is == goal\_state

        done = is\_goal(node.state, goal\_state)

        if done:

            print('done')

            print\_res(node, goal\_state, counter)

            break

        # add state to explored

        explored.append(node.state)

        # pop node

        # swap node with lowest f(n) with last index and pop

        frontier[index], frontier[-1] = frontier[-1], frontier[index]

        node = frontier.pop()

        # get index of blank states

        blank\_pos = get\_blank\_positions(node.state)

        # print(blank\_pos)

        b1 = blank\_pos[0]

        b2 = blank\_pos[1]

        children = get\_states(blank\_pos, node)

        # make each map into a node

        # res[0] gives the grid

        # res[1] gives the action

        # add state to frontier if it is not already in frontier and explored

        for res in children:

            if res[0] not in explored:

                in\_frontier = 0

                for i in frontier:

                    if i.state == res[0]:

                        in\_frontier = 1

                        break

                if not in\_frontier:

                    fn = get\_fn\_dist(res[0], goal\_state, node.depth+1)

                    frontier.append(Node(res[0], node, res[1], node.depth+1, fn))

main('input1.txt')