14 Puzzle- Instructions

1) Have a text file in the following format:

Input1.txt

```
1 2 3 4
5 0 6 7
8 9 0 10
11 12 13 14

1 2 4 0
8 5 3 7
11 9 6 10
0 12 13 14
```

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:

```
323                                    fron
324
325
326     main('input3.txt')
327
328
```

-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

<u>How to run using VSCode:</u>

```
PS C:\Users\jiach\Documents\College Fall 2019\AI> python project_1.py
```

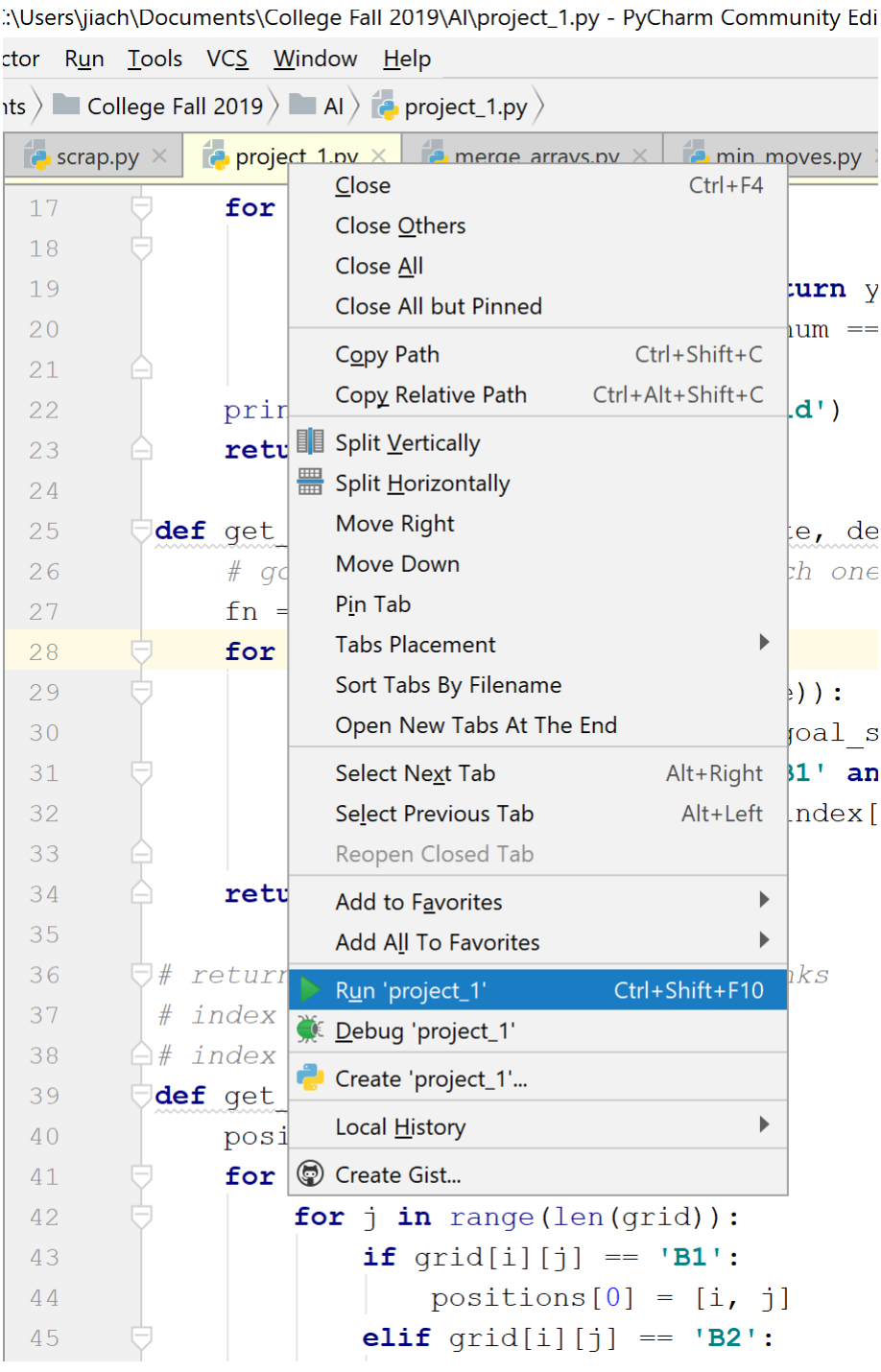
-Go to the directory containing the source code file; in my case it is …..\AI

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

-In my case the name is project_1.py

<u>How to run using PyCharm:</u>

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


:\Users\jiach\Documents\College Fall 2019\AI\project_1.py - PyCharm Community Edi
ctor Run Tools VCS Window Help
College Fall 2019
AI
project_1.py
scrap.py
project_1.py
merge_arrays.py
min_moves.py
Close Ctrl+F4
Close Others
Close All
Close All but Pinned
Copy Path Ctrl+Shift+C
Copy Relative Path Ctrl+Alt+Shift+C
Split Vertically
Split Horizontally
Move Right
Move Down
Pin Tab
Tabs Placement
Sort Tabs By Filename
Open New Tabs At The End
Select Next Tab Alt+Right
Select Previous Tab Alt+Left
Reopen Closed Tab
Add to Favorites
Add All To Favorites
Run 'project_1' Ctrl+Shift+F10
Debug 'project_1'
Create 'project_1'...
Local History
Create Gist...
for j in range(len(grid)):
if grid[i][j] == 'B1':
positions[0] = [i, j]
elif grid[i][j] == 'B2':

Output 1:

output1.txt

```
1 2 4 0
8 5 3 7
11 9 6 10
0 12 13 14

1 2 4 0
8 5 3 7
11 9 6 10
0 12 13 14

6
16
L1 D1 U2 U2 R2 D1
6 6 6 6 6 6 6
```

Output 2:

output2.txt

```
1 3 4 13
8 5 7 9
10 0 6 12
11 14 0 2

1 3 4 13
8 5 7 9
10 0 6 12
11 14 0 2

12
117
R2 R1 U2 U2 R2 R1 D2 D2 D1 D1 L1 L2
10 10 12 12 12 12 12 12 12 12 12 12 12
```

Output 3:

output3.txt

```
9 3 13 4
2 7 1 0
10 12 0 5
14 11 8 6

9 3 13 4
2 7 1 0
10 12 0 5
14 11 8 6

14
80
D2 R2 R2 U2 U1 L1 D1 L1 D1 R1 U1 R1 R1 L2
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
```

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 = s
um 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')
```