Instructions on how to run the program:

The program was written and debugged on PyCharm, even though it could be run on any Python IDE. A function call to the main function is on the last line of the source code, so running 15_puzzle.py can get the program started right away. The program will print a message and ask for the name of the input file, which should be placed under the same directory as 15_puzzle.py. The next message to be printed asks the user to name the output file, which will also be placed under the same directory. The program terminates after writing the output file.

Output text files:

***Output1.txt* (output for *Input1.txt*):**
1 2 3 4
5 6 0 7
8 9 10 11
12 13 14 15

1 2 3 4
5 9 6 7
8 13 0 11
12 14 10 15

5
19
L D D R U
5 5 5 5 5 5

***Output2.txt* (output for *Input2.txt*):**
1 5 3 13
8 0 6 4
15 10 7 9
11 14 2 12

1 5 3 13
8 10 6 4
0 15 2 9
11 7 14 12

6
26
D R D L U L
6 6 6 6 6 6 6

***Output3.txt* (output for *Input3.txt*):**
9 13 7 4

12 3 0 1
2 15 5 6
14 10 11 8

13 3 7 4
9 1 0 6
12 2 5 8
14 15 10 11

12
38
R D D L L U L U U R D R
12 12 12 12 12 12 12 12 12 12 12 12 12

***Output4.txt* (output for *Input4.txt*):**
13 12 2 11
10 1 8 9
0 3 15 14
6 4 7 5

10 13 12 11
8 1 2 9
3 4 15 5
6 0 14 7

16
868
R U R D R D L U U U L L D R D D
12 12 14 14 16 16 16 16 16 16 16 16 16 16 16 16 16

Source code:

```
import io
import copy


class Node:
    def __init__(self, state, parent, path_cost, h_cost, path_history):
        self.state = state
        # Contains a representation of the current state
        self.parent = parent
        # Points to the parent node
        self.path_cost = path_cost
        # The path cost from the initial state to the current node
        self.h_cost = h_cost
```

```
        # The cost of the heuristic function
        self.path_history = path_history
        # The list of moves from the initial state that lead to the current state
        self.f = path_cost + h_cost
        # f(n) = g(n) + h(n)


def load_input(filename: str) -> list:
    """ Python 3 allows entering type hints in the parameter list as well as following an arrow
after the
    parentheses. Type hints only serve as annotations and do not require the arguments to be of
the
    specified type or the function to return a variable of the specified type. """

    text_stream = io.open(filename, 'r', encoding='utf-8', errors='ignore', newline='\n')
    """ Calls Python's io function to read the file with the specified name."""

    initial_state = []
    for i in range(0, 4):
        initial_state.append(list(map(int, text_stream.readline().rstrip().split(' '))))
        """ The rstrip method removes all trailing whitespace of the string. The split
        method uses the given character as the delimiter to break down the string and
        return a list of the substrings. The map function takes that list, converts
        the substrings into integers and returns a map object, which is eventually
        converted into a list by the exterior call to the list function. """

        """ A state is represented as a multi-layer list. The first layer contains
        the four rows, each of which is a second layer that consists of four tiles. """

    blank_line = text_stream.readline()
    """ In the input file, there is a blank line in between the two states."""

    goal_state = []
    for i in range(0, 4):
        goal_state.append(list(map(int, text_stream.readline().rstrip().split(' '))))
        """ The construct of this part is identical to the one above. """

    text_stream.close()

    ret = [initial_state, goal_state]
    """ Returns the two lists that represent the initial and goal states,
    respectively. """
    return ret


def state_to_locations(state: list) -> list:
```

```
    """ The function takes a state and return a list of sixteen tuples, each of which
    represents the location (row, column) of the number that corresponds to the current
    index. See below for further explanation."""

    locations = []
    for i in range(0, 16):
        locations.append((0, 0))
        # Each tuple represents a location on the board as (row, column)

    """ "locations" keeps track of all fifteen numbers in the given state and the goal
    state. The location of the blank in the state is stored as the tuple at locations[0],
    the location of the number 1 is stored as locations[1], so on and so forth."""

    """ Due to the nature of indices on a list, when a location is stored as a tuple
    (row, column), the four rows and four columns are represented as indices from 0
    to 3, even though the numbers 1 through 15 are represented as indices from 1 to
    15 on the list."""

    for i in range(0, 4):
        for j in range(0, 4):
            """ The loop scans the given state and reads the integer at [i][j]. The number
            is stored at its corresponding index in the list "locations". By the time the
            loop finishes, the locations of all fifteen numbers as well as the blank in
            the given state will have been stored in the list."""
            num = state[i][j]
            locations[num] = (i, j)

    return locations


def locations_to_state(locations: list) -> list:
    """ The function takes a list of locations and converts it to the format of a state,
    which can then become part of a node."""

    state = []
    for i in range(0, 4):
        state.append([])
        # The first layer of the list consists of the four rows of a state
        for j in range(0, 4):
            state[i].append(-1)
        """ The second layer consists of the four tiles of a row (one of them could be
        the blank)."""

    for i in range(0, 16):
        state[locations[i][0]][locations[i][1]] = i
        """ locations[i][0] stores the row number, locations[i][0] stores the column
```

```
        number, and i is the number on the tile."""

    return state


def heuristic_cal(current: list, goal: list) -> int:
    """ Parameters are two lists that represent the current state and the goal state,
    respectively. Returns the cost of the heuristic function for the current state,
    which is the sum of Manhattan distances of tiles from their goal positions."""

    current_locations = state_to_locations(current)
    goal_locations = state_to_locations(goal)

    h_val = 0  # Tracks the cost of the heuristic function
    for i in range(1, 16):
        h_val += (abs(current_locations[i][0] - goal_locations[i][0]) +
                  abs(current_locations[i][1] - goal_locations[i][1]))
        """ Loops through both lists of locations and adds the Manhattan distance
        of each number to the sum h_val. The range is from 1 to 16 because the
        blank in either state is not taken into account."""

    return h_val


def create_child_nodes(current_node: Node, goal: list, generated: set) -> list:
    """ The function takes a node, the goal state and the set of generated
    nodes and returns a list of its child nodes. """

    children = []
    locations = state_to_locations(current_node.state)
    blank = locations[0]

    # Moving blank to the left
    if blank[1] != 0:
        new_locations = copy.deepcopy(locations)
        new_locations[0] = (new_locations[0][0], new_locations[0][1] - 1)
        # Modifies the location of the blank in the new list
        """ Note that the index 0 represents the first column. So long as
        the blank is not in the first column, it can be moved to the left."""
        neighbor = current_node.state[blank[0]][blank[1] - 1]
        # Finds the number on the tile to the left of the blank
        new_locations[neighbor] = (new_locations[neighbor][0], new_locations[neighbor][1] + 1)
        # Modifies the location of the neighbor in the new list
        new_path_history = copy.deepcopy(current_node.path_history)
        new_path_history.append('L')
        new_state = locations_to_state(new_locations)
```

```
    # Constructs the new state by calling locations_to_state
    new_node = Node(new_state, current_node, current_node.path_cost + 1,
                    heuristic_cal(new_state, goal), new_path_history)
    if new_node not in generated:
        children.append(new_node)
        """ Append the child node to the list only if it's not a
        repeated state."""

# Moving blank to the right
if blank[1] != 3:
    new_locations = copy.deepcopy(locations)
    new_locations[0] = (new_locations[0][0], new_locations[0][1] + 1)
    """ Similar to the case above: so long as the blank is not in the fourth
    column, it can be moved to the right."""
    neighbor = current_node.state[blank[0]][blank[1] + 1]
    # Finds the number on the tile to the right of the blank
    new_locations[neighbor] = (new_locations[neighbor][0], new_locations[neighbor][1] - 1)
    new_path_history = copy.deepcopy(current_node.path_history)
    new_path_history.append('R')
    new_state = locations_to_state(new_locations)
    new_node = Node(new_state, current_node, current_node.path_cost + 1,
                    heuristic_cal(new_state, goal), new_path_history)
    if new_node not in generated:
        children.append(new_node)

# Moving blank up
if blank[0] != 0:
    new_locations = copy.deepcopy(locations)
    new_locations[0] = (new_locations[0][0] - 1, new_locations[0][1])
    """ So long as the blank is not in the first row, it can be moved up."""
    neighbor = current_node.state[blank[0] - 1][blank[1]]
    # Finds the number on the tile above the blank
    new_locations[neighbor] = (new_locations[neighbor][0] + 1, new_locations[neighbor][1])
    new_path_history = copy.deepcopy(current_node.path_history)
    new_path_history.append('U')
    new_state = locations_to_state(new_locations)
    new_node = Node(new_state, current_node, current_node.path_cost + 1,
                    heuristic_cal(new_state, goal), new_path_history)
    if new_node not in generated:
        children.append(new_node)

# Moving the blank down
if blank[0] != 3:
    new_locations = copy.deepcopy(locations)
    new_locations[0] = (new_locations[0][0] + 1, new_locations[0][1])
    """ So long as the blank is not in the fourth row, it can be moved down."""
```

```
        neighbor = current_node.state[blank[0] + 1][blank[1]]
        # Finds the number on the tile below the blank
        new_locations[neighbor] = (new_locations[neighbor][0] - 1, new_locations[neighbor][1])
        new_path_history = copy.deepcopy(current_node.path_history)
        new_path_history.append('D')
        new_state = locations_to_state(new_locations)
        new_node = Node(new_state, current_node, current_node.path_cost + 1,
                    heuristic_cal(new_state, goal), new_path_history)
        if new_node not in generated:
            children.append(new_node)

    return children


def next_expansion(successors: set, goal: list, generated: set) -> list:
    """ The function takes a set of successors, the goal state and the set of all nodes
    that have been generated. It selects the best successor and expands it by returning
    a call to the create_child_nodes function, which returns a list of child nodes."""

    t_successors = tuple(successors)
    # Type-casts the set of successors to a tuple for the purpose of looping
    best = t_successors[0]
    for a_successor in t_successors:
        if a_successor.f < best.f:
            best = a_successor
        """ Goes through every successor in the set to find the one with the
        lowest f(n)."""

    child_nodes = create_child_nodes(best, goal, generated)
    for child in child_nodes:
        generated.add(child)
        # Add each generated child node to the set
        successors.add(child)
        # Add each generated child to the set of successors

    successors.remove(best)
    # Remove the selected node, which has been expanded, from the set of successors

    return child_nodes


def generate_output(input_filename: str, output_filename: str, goal_node: Node,
                  generated: set) -> None:
    """ The function takes the filename of the input, a filename for the output
    to be generated, the goal node, and the set of generated nodes. It creates
    an output file with the filename provided and returns nothing."""
```

```
    input_stream = io.open(input_filename, 'r', encoding='utf-8', errors='ignore',
                           newline='\n')
    with open(output_filename, 'w') as out_file:
        for i in range(0, 10):
            out_file.write(input_stream.readline().rstrip())
            out_file.write('\n')
        """ The first ten lines of the output file are identical to those in the
        input file. The tenth line should be skipped because it's blank."""
        out_file.write(str(goal_node.path_cost) + '\n')
        # Line 11 of the output, the depth level d
        out_file.write(str(len(generated)) + '\n')
        # Line 12 of the output, the total number of nodes generated

        # Writing Line 13 of the output, the sequence of moves
        length = len(goal_node.path_history)
        for i in range(length - 1):
            out_file.write(goal_node.path_history[i] + ' ')
        out_file.write(goal_node.path_history[length - 1] + '\n')

        # Writing Line 14 of the output, the f(n) values
        f_line = str(goal_node.f) + ' '
        parent = goal_node.parent
        while parent:  # Loop stops when parent == None
            f_line += (str(parent.f) + ' ')
            parent = parent.parent
        f_list = f_line.split(' ')
        # Breaks down the string to the integers it contains
        reverse = ''
        for i in range(len(f_list) - 2, -1, -1):
            # f_line[len(f_line)-1] is an extra whitespace character and
            # thus shouldn't be copied
            reverse += str(f_list[i])
            if i != 0:
                reverse += ' '
        """ The order of the f(n) values in f_line is from goal node
        to root node. The four lines above reverse the order, which
        is what the output format expects."""
        out_file.write(reverse)

    out_file.close()


def main() -> int:
    in_filename = input("""Please enter below the input filename, e.g. "Input1.txt".
    The filename is case-sensitive.\n""")
```

```
    states = load_input(in_filename)
    initial_state = states[0]
    goal_state = states[1]
    """ Asks for the input filename and passes it to load_input. The return variable
    is a list whose first element is the initial state and second element is the
    goal state."""

    root_node = Node(initial_state, None, 0, heuristic_cal(initial_state, goal_state), [])
    generated = {root_node}  # Stores all generated nodes
    successors = {root_node}  # A set of successors that could be expanded
    goal_node = None
    while len(successors) != 0:
        # Keep calling next_expansion as long as there are successors remaining
        child_nodes = next_expansion(successors, goal_state, generated)
        goal_found = False
        for child in child_nodes:
            # Goal-state check
            if child.state == goal_state:
                goal_found = True
                goal_node = child
                break
        if goal_found:
            break

    out_filename = input("""Now please enter the output filename, in the same manner as
earlier:\n""")
    generate_output(in_filename, out_filename, goal_node, generated)

    return 0


main()
```