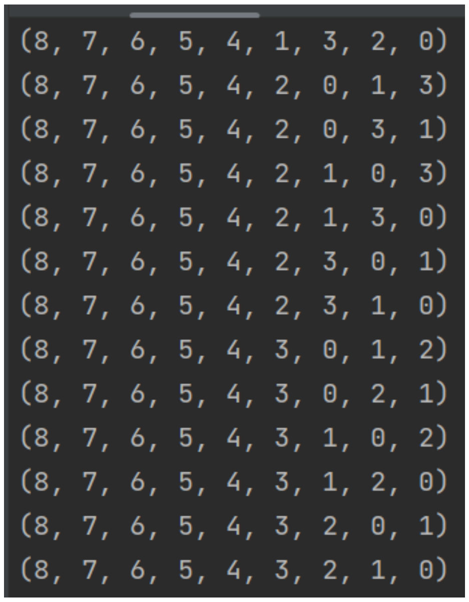
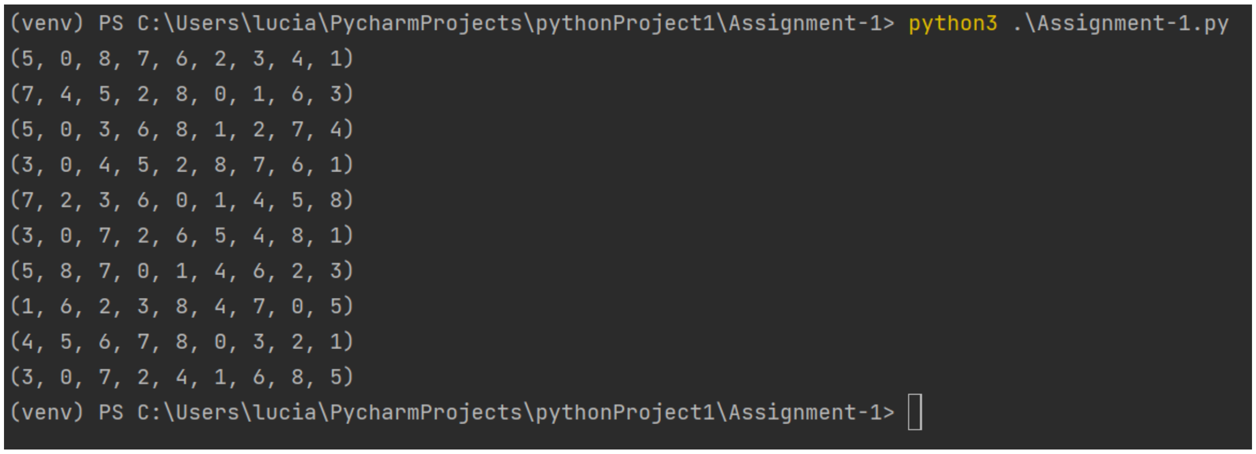
MAIN FUNCTION DRIVER CODE IS COMMENTED OUT, TO RUN PROGRAM, UNCOMMENT EACH PART AND RUN AGAIN

1. To find all states, I called itertools and used the permutation function to find all permutations of a 3 x 3 board. My code (included in zip submission) generated all 9! states and here is a short snippet of the results:



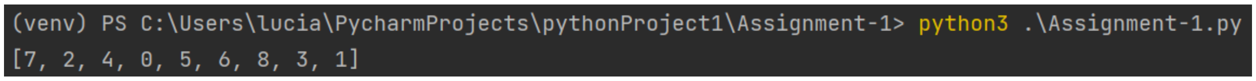
```
(8, 7, 6, 5, 4, 1, 3, 2, 0)
(8, 7, 6, 5, 4, 2, 0, 1, 3)
(8, 7, 6, 5, 4, 2, 0, 3, 1)
(8, 7, 6, 5, 4, 2, 1, 0, 3)
(8, 7, 6, 5, 4, 2, 1, 3, 0)
(8, 7, 6, 5, 4, 2, 3, 0, 1)
(8, 7, 6, 5, 4, 2, 3, 1, 0)
(8, 7, 6, 5, 4, 3, 0, 1, 2)
(8, 7, 6, 5, 4, 3, 0, 2, 1)
(8, 7, 6, 5, 4, 3, 1, 0, 2)
(8, 7, 6, 5, 4, 3, 1, 2, 0)
(8, 7, 6, 5, 4, 3, 2, 0, 1)
(8, 7, 6, 5, 4, 3, 2, 1, 0)
```

2. For part B, I randomly picked a state generated from part A, checked it if all odd numbers were not neighbors, added it to a set if it was, and repeated until 10 states were found:



```
(venv) PS C:\Users\lucia\PycharmProjects\pythonProject1\Assignment-1> python3 .\Assignment-1.py
(5, 0, 8, 7, 6, 2, 3, 4, 1)
(7, 4, 5, 2, 8, 0, 1, 6, 3)
(5, 0, 3, 6, 8, 1, 2, 7, 4)
(3, 0, 4, 5, 2, 8, 7, 6, 1)
(7, 2, 3, 6, 0, 1, 4, 5, 8)
(3, 0, 7, 2, 6, 5, 4, 8, 1)
(5, 8, 7, 0, 1, 4, 6, 2, 3)
(1, 6, 2, 3, 8, 4, 7, 0, 5)
(4, 5, 6, 7, 8, 0, 3, 2, 1)
(3, 0, 7, 2, 4, 1, 6, 8, 5)
(venv) PS C:\Users\lucia\PycharmProjects\pythonProject1\Assignment-1>
```

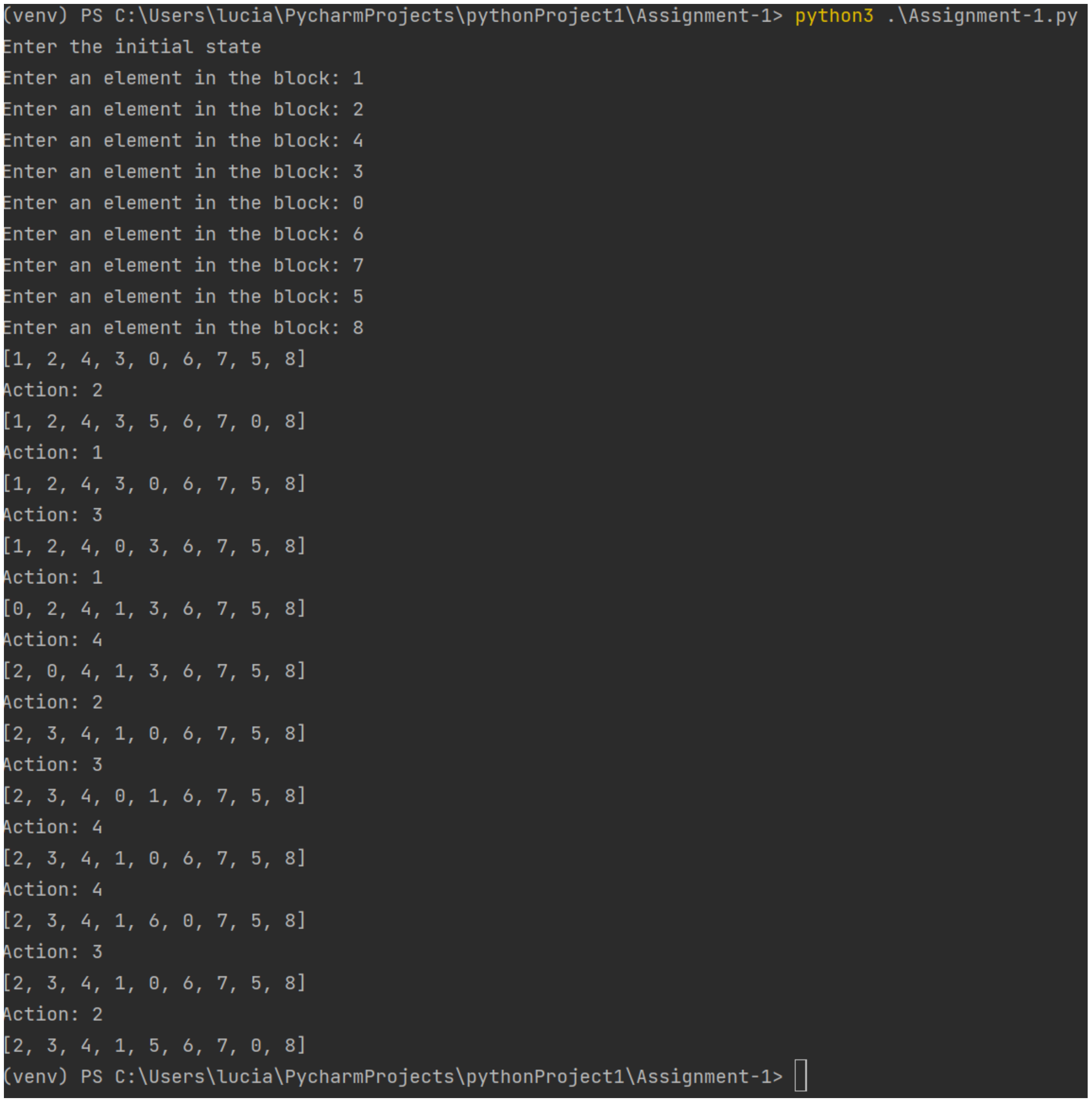
3. For part C, I passed in the example state and action from the handout, checked if it was possible to do the action, and then swapped 0 depending on the action number passed in:



```
(venv) PS C:\Users\lucia\PycharmProjects\pythonProject1\Assignment-1> python3 .\Assignment-1.py
[7, 2, 4, 0, 5, 6, 8, 3, 1]
```

4. For part D, I prompted the user for an input with error checking to guarantee a valid state. I then called my function random_moves() from the state passed in that randomly made moves and

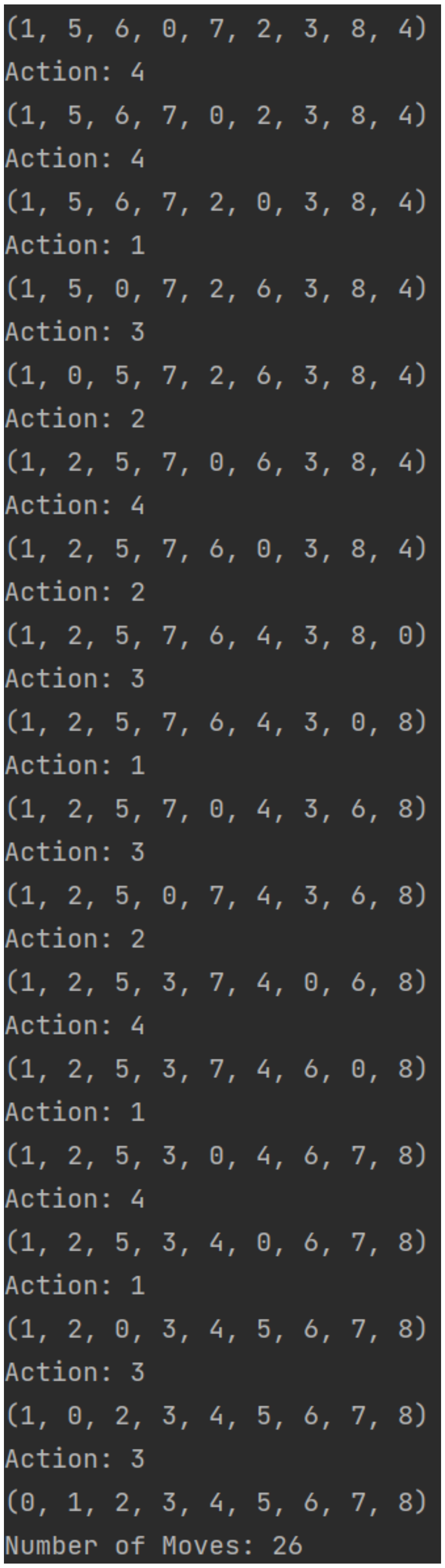
ran until the rows could be divided by 3:



```
(venv) PS C:\Users\lucia\PycharmProjects\pythonProject1\Assignment-1> python3 .\Assignment-1.py
Enter the initial state
Enter an element in the block: 1
Enter an element in the block: 2
Enter an element in the block: 4
Enter an element in the block: 3
Enter an element in the block: 0
Enter an element in the block: 6
Enter an element in the block: 7
Enter an element in the block: 5
Enter an element in the block: 8
[1, 2, 4, 3, 0, 6, 7, 5, 8]
Action: 2
[1, 2, 4, 3, 5, 6, 7, 0, 8]
Action: 1
[1, 2, 4, 3, 0, 6, 7, 5, 8]
Action: 3
[1, 2, 4, 0, 3, 6, 7, 5, 8]
Action: 1
[0, 2, 4, 1, 3, 6, 7, 5, 8]
Action: 4
[2, 0, 4, 1, 3, 6, 7, 5, 8]
Action: 2
[2, 3, 4, 1, 0, 6, 7, 5, 8]
Action: 3
[2, 3, 4, 0, 1, 6, 7, 5, 8]
Action: 4
[2, 3, 4, 1, 0, 6, 7, 5, 8]
Action: 4
[2, 3, 4, 1, 6, 0, 7, 5, 8]
Action: 3
[2, 3, 4, 1, 0, 6, 7, 5, 8]
Action: 2
[2, 3, 4, 1, 5, 6, 7, 0, 8]
(venv) PS C:\Users\lucia\PycharmProjects\pythonProject1\Assignment-1>
```

5. For part E, I prompted the user to enter both a start_state and a goal_state and passed the two into my breadth first search function. My BFS function utilized a queue and a set containing visited states. For the start_state and the goal_state provided in the handout, there was no solution, so another random state was chosen and solved (for the following output snippet, the

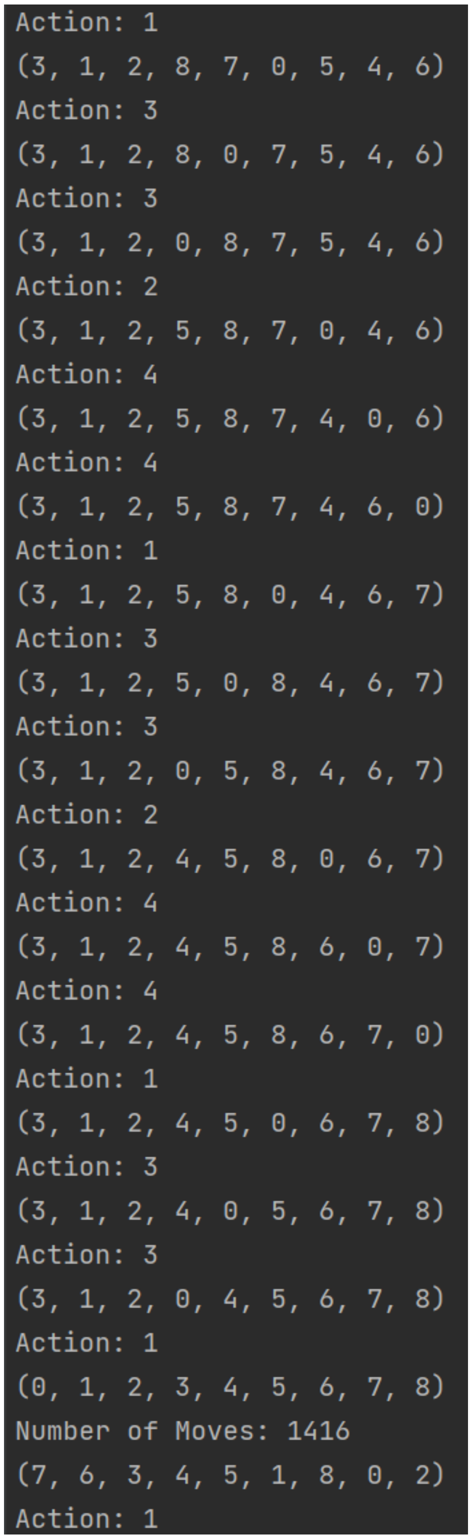
random state was [5, 7, 0, 8, 2, 6, 1, 3, 4] and it took 26 moves):



```
(1, 5, 6, 0, 7, 2, 3, 8, 4)
Action: 4
(1, 5, 6, 7, 0, 2, 3, 8, 4)
Action: 4
(1, 5, 6, 7, 2, 0, 3, 8, 4)
Action: 1
(1, 5, 0, 7, 2, 6, 3, 8, 4)
Action: 3
(1, 0, 5, 7, 2, 6, 3, 8, 4)
Action: 2
(1, 2, 5, 7, 0, 6, 3, 8, 4)
Action: 4
(1, 2, 5, 7, 6, 0, 3, 8, 4)
Action: 2
(1, 2, 5, 7, 6, 4, 3, 8, 0)
Action: 3
(1, 2, 5, 7, 6, 4, 3, 0, 8)
Action: 1
(1, 2, 5, 7, 0, 4, 3, 6, 8)
Action: 3
(1, 2, 5, 0, 7, 4, 3, 6, 8)
Action: 2
(1, 2, 5, 3, 7, 4, 0, 6, 8)
Action: 4
(1, 2, 5, 3, 7, 4, 6, 0, 8)
Action: 1
(1, 2, 5, 3, 0, 4, 6, 7, 8)
Action: 4
(1, 2, 5, 3, 4, 0, 6, 7, 8)
Action: 1
(1, 2, 0, 3, 4, 5, 6, 7, 8)
Action: 3
(1, 0, 2, 3, 4, 5, 6, 7, 8)
Action: 3
(0, 1, 2, 3, 4, 5, 6, 7, 8)
Number of Moves: 26
```

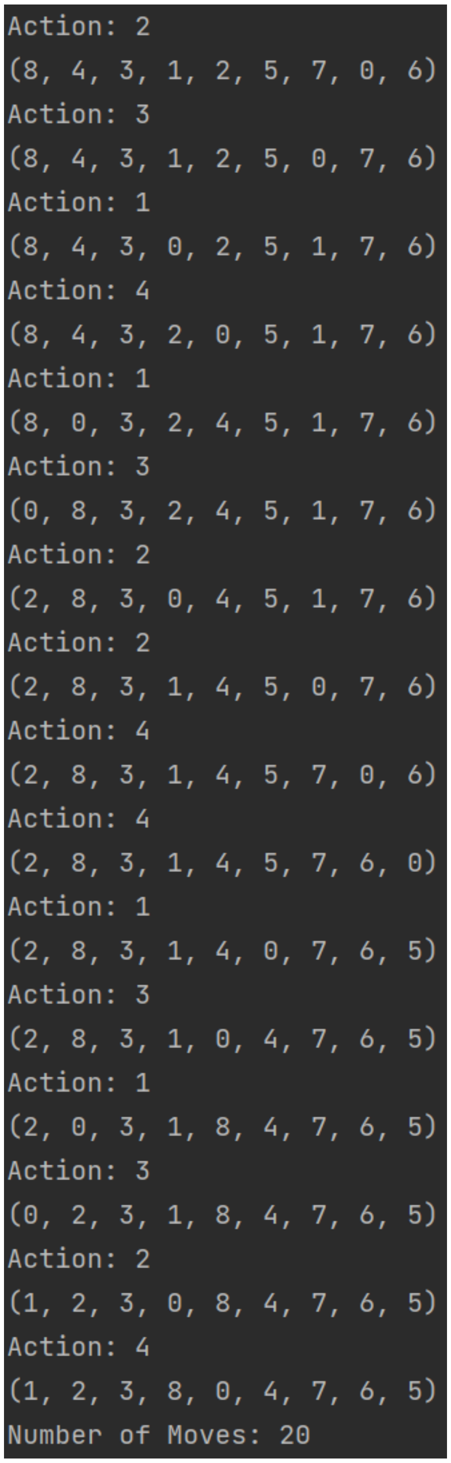
6. For part F, I ran both DFS and BFS back to back and saved the valid initial_state from bfs so that both functions performed on the same state. The valid state randomly chosen was [7, 6, 3, 4, 5,

1, 8, 0, 2]. My DFS utilized a stack and a maximum_iter variable for when a solution was unsolvable. DFS took 1416 moves compared to 19 moves from BFS to find the goal_state, so DFS took much more moves than BFS took (the following snippet shows a snippet of the end of the DFS search):



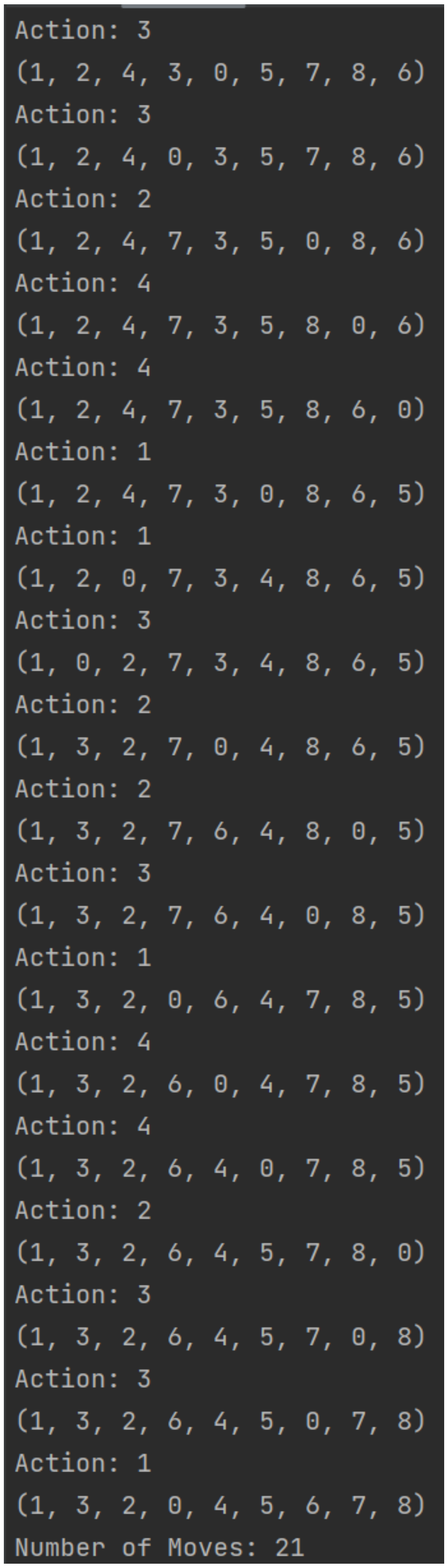
```
Action: 1
(3, 1, 2, 8, 7, 0, 5, 4, 6)
Action: 3
(3, 1, 2, 8, 0, 7, 5, 4, 6)
Action: 3
(3, 1, 2, 0, 8, 7, 5, 4, 6)
Action: 2
(3, 1, 2, 5, 8, 7, 0, 4, 6)
Action: 4
(3, 1, 2, 5, 8, 7, 4, 0, 6)
Action: 4
(3, 1, 2, 5, 8, 7, 4, 6, 0)
Action: 1
(3, 1, 2, 5, 8, 0, 4, 6, 7)
Action: 3
(3, 1, 2, 5, 0, 8, 4, 6, 7)
Action: 3
(3, 1, 2, 0, 5, 8, 4, 6, 7)
Action: 2
(3, 1, 2, 4, 5, 8, 0, 6, 7)
Action: 4
(3, 1, 2, 4, 5, 8, 6, 0, 7)
Action: 4
(3, 1, 2, 4, 5, 8, 6, 7, 0)
Action: 1
(3, 1, 2, 4, 5, 0, 6, 7, 8)
Action: 3
(3, 1, 2, 4, 0, 5, 6, 7, 8)
Action: 3
(3, 1, 2, 0, 4, 5, 6, 7, 8)
Action: 1
(0, 1, 2, 3, 4, 5, 6, 7, 8)
Number of Moves: 1416
(7, 6, 3, 4, 5, 1, 8, 0, 2)
Action: 1
```

7. For part G, I repeated the BFS search from part E and passed in the arguments from the handout. The initial_state to goal_state was unsolvable, so a random_state was chosen instead:

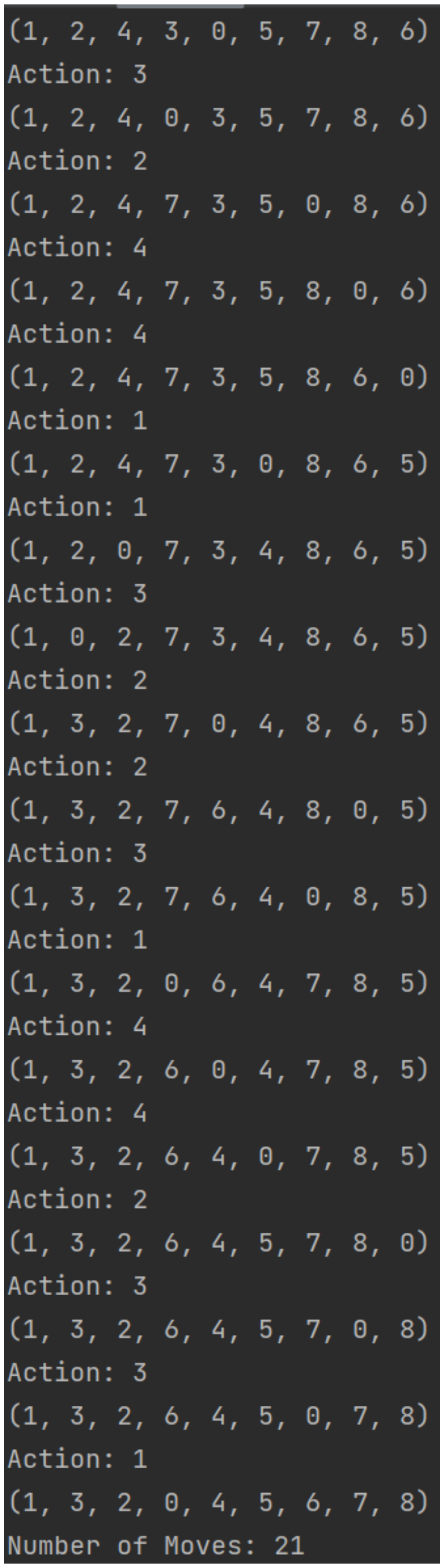


```
Action: 2
(8, 4, 3, 1, 2, 5, 7, 0, 6)
Action: 3
(8, 4, 3, 1, 2, 5, 0, 7, 6)
Action: 1
(8, 4, 3, 0, 2, 5, 1, 7, 6)
Action: 4
(8, 4, 3, 2, 0, 5, 1, 7, 6)
Action: 1
(8, 0, 3, 2, 4, 5, 1, 7, 6)
Action: 3
(0, 8, 3, 2, 4, 5, 1, 7, 6)
Action: 2
(2, 8, 3, 0, 4, 5, 1, 7, 6)
Action: 2
(2, 8, 3, 1, 4, 5, 0, 7, 6)
Action: 4
(2, 8, 3, 1, 4, 5, 7, 0, 6)
Action: 4
(2, 8, 3, 1, 4, 5, 7, 6, 0)
Action: 1
(2, 8, 3, 1, 4, 0, 7, 6, 5)
Action: 3
(2, 8, 3, 1, 0, 4, 7, 6, 5)
Action: 1
(2, 0, 3, 1, 8, 4, 7, 6, 5)
Action: 3
(0, 2, 3, 1, 8, 4, 7, 6, 5)
Action: 2
(1, 2, 3, 0, 8, 4, 7, 6, 5)
Action: 4
(1, 2, 3, 8, 0, 4, 7, 6, 5)
Number of Moves: 20
```

8. For part H, I wrote 2 versions of a UCS algorithm using a priority queue and a cost counter to mark actions – one where every cost was 1 and one where the cost depended on what was given in the handout. Both gave the same number of moves back and also in the same order (21 moves to solve [1, 2, 4, 3, 0, 6, 7, 5, 8] to [1, 3, 2, 0, 4, 5, 6, 7, 8]):



```
Action: 3
(1, 2, 4, 3, 0, 5, 7, 8, 6)
Action: 3
(1, 2, 4, 0, 3, 5, 7, 8, 6)
Action: 2
(1, 2, 4, 7, 3, 5, 0, 8, 6)
Action: 4
(1, 2, 4, 7, 3, 5, 8, 0, 6)
Action: 4
(1, 2, 4, 7, 3, 5, 8, 6, 0)
Action: 1
(1, 2, 4, 7, 3, 0, 8, 6, 5)
Action: 1
(1, 2, 0, 7, 3, 4, 8, 6, 5)
Action: 3
(1, 0, 2, 7, 3, 4, 8, 6, 5)
Action: 2
(1, 3, 2, 7, 0, 4, 8, 6, 5)
Action: 2
(1, 3, 2, 7, 6, 4, 8, 0, 5)
Action: 3
(1, 3, 2, 7, 6, 4, 0, 8, 5)
Action: 1
(1, 3, 2, 0, 6, 4, 7, 8, 5)
Action: 4
(1, 3, 2, 6, 0, 4, 7, 8, 5)
Action: 4
(1, 3, 2, 6, 4, 0, 7, 8, 5)
Action: 2
(1, 3, 2, 6, 4, 5, 7, 8, 0)
Action: 3
(1, 3, 2, 6, 4, 5, 7, 0, 8)
Action: 3
(1, 3, 2, 6, 4, 5, 0, 7, 8)
Action: 1
(1, 3, 2, 0, 4, 5, 6, 7, 8)
Number of Moves: 21
```



```
(1, 2, 4, 3, 0, 5, 7, 8, 6)
Action: 3
(1, 2, 4, 0, 3, 5, 7, 8, 6)
Action: 2
(1, 2, 4, 7, 3, 5, 0, 8, 6)
Action: 4
(1, 2, 4, 7, 3, 5, 8, 0, 6)
Action: 4
(1, 2, 4, 7, 3, 5, 8, 6, 0)
Action: 1
(1, 2, 4, 7, 3, 0, 8, 6, 5)
Action: 1
(1, 2, 0, 7, 3, 4, 8, 6, 5)
Action: 3
(1, 0, 2, 7, 3, 4, 8, 6, 5)
Action: 2
(1, 3, 2, 7, 0, 4, 8, 6, 5)
Action: 2
(1, 3, 2, 7, 6, 4, 8, 0, 5)
Action: 3
(1, 3, 2, 7, 6, 4, 0, 8, 5)
Action: 1
(1, 3, 2, 0, 6, 4, 7, 8, 5)
Action: 4
(1, 3, 2, 6, 0, 4, 7, 8, 5)
Action: 4
(1, 3, 2, 6, 4, 0, 7, 8, 5)
Action: 2
(1, 3, 2, 6, 4, 5, 7, 8, 0)
Action: 3
(1, 3, 2, 6, 4, 5, 7, 0, 8)
Action: 3
(1, 3, 2, 6, 4, 5, 0, 7, 8)
Action: 1
(1, 3, 2, 0, 4, 5, 6, 7, 8)
Number of Moves: 21
```