King Fahd University of Petroleum & Minerals

College of Computing and Mathematics

Information & Computer Science

ICS 381: Principles of Artificial Intelligence

Assignment #1

Student Name: Abdullah Alzeid

1. Comment on the *branching factor*

The average number of subsequent possibilities in a state space is referred to as the branching factor. When you're anywhere other than the starting point, the step that takes you back isn't generally considered.

1. In the case of an 8-puzzle on a 3x3 grid:

Where the blank tile can be:

- Central spot (1 location): Initially 4 options, but with the backward step removed, it's 3.
- Along the sides (4 locations): 3 options reduce to 2 after removing the previous move.
- At the corners (4 locations): Begins with 2, but then narrows down to 1 after deduction.

Computed Average:

1(3) + 4(2) + 4(1) = 1.67

2. When dealing with a 15-puzzle on a 4x4 grid:

Positions for the blank tile:

- Inner spots (4 locations): Starts at 4 moves but becomes 3 without the return move.
- Along the sides (8 locations): From 3, we get 2 without the move backward.

- At the corners (4 locations): Reducing from 2 to 1 after deduction.

Computed Average:

4(3) + 8(2) + 4(1) = 2.00

3. For the 24-puzzle on a 5x5 grid:

Possible blank tile placements:
- Inner section (9 locations): 4 moves originally, but only 3 once the previous move is excluded.
- Along the perimeter (12 locations): Goes from 3 to 2 without the backward move.
- Corners (4 locations): Initially 2, but 1 remains after deduction.

Computed Average:

9(3) + 12(2) + 4(1) = 2.20

4. Regarding the 35-puzzle on a 6x6 grid:

Blank tile placements:
- Center section (16 locations): 4 moves cut down to 3 after omitting the backward step.
- Perimeter (16 locations): Starts at 3 but reduces to 2 without the return move.

- Corners (4 locations): Initially 2, but the count drops to 1 after removing the backward move.

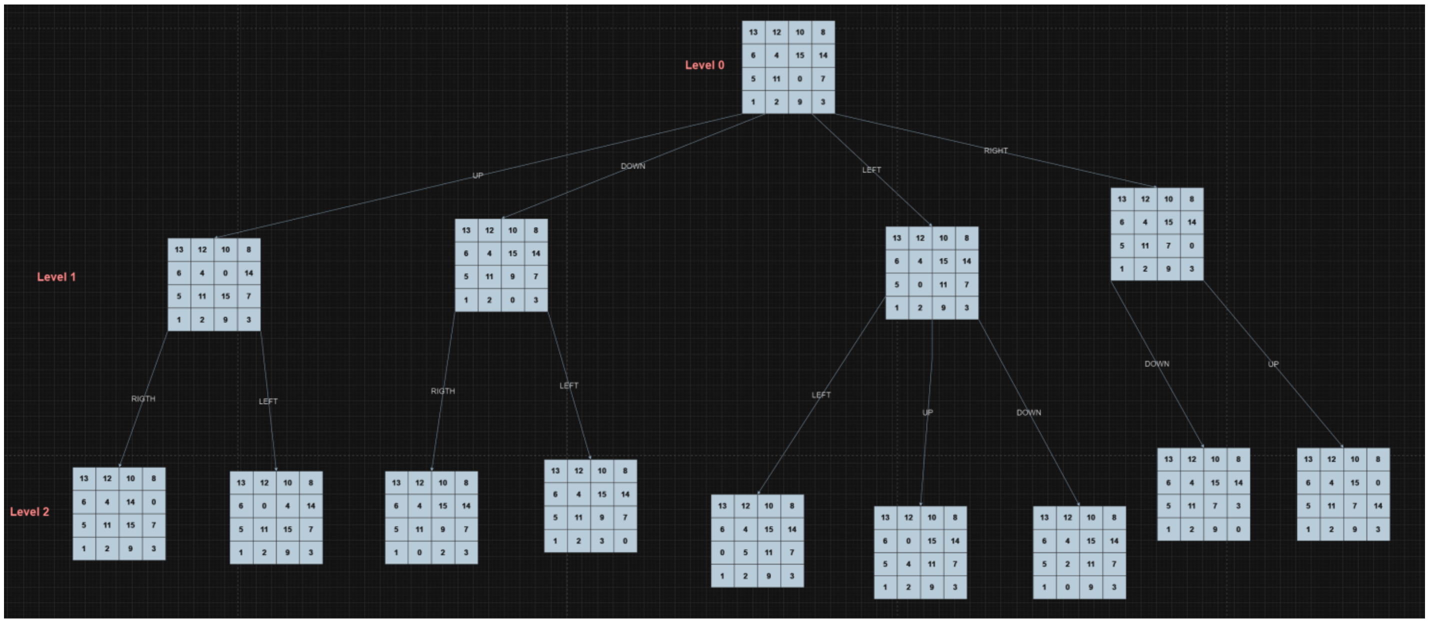
Computed Average:

16(3) + 16(2) + 4(1) = 2.33

In a more straightforward summary:

- For a matrix size of n = 3, the average number of moves is around 1.67.
- With n = 4, it comes out to be 2.00.
- For n = 5, the average is 2.20.
- Lastly, with n = 6, it's close to 2.33.

2. Draw a diagram of the corresponding state space search tree (depth 0, 1, and 2) for n=4.



3. Implement *BFS*, *DFS*, and *DFS with revisit check* search solutions to the problem.

Representation: The code provides a representation and solution to the N x N sliding puzzle game. This game usually has (N^2 - 1) tiles labeled with numbers and one blank space, represented as 0. The objective is to move tiles around using the blank space until the puzzle reaches its goal state.

Function Breakdown:

1. **get_puzzle_size()**:

   • Obtains the desired puzzle size from the user.

   • Accepts only sizes between 3x3 and 6x6.

2. **solvable(state)**:

   • Checks if a given puzzle configuration is solvable.

   • Flattens the 2D state for easier calculation.

   • Computes the inversion count by checking pairs of tiles in the wrong order.

   • Uses specific logic depending on grid width and blank tile position to determine solvability.

3. **generate_initial_state(n)**:

   • Generates a random, solvable initial state for the puzzle of size NxN.

   • It ensures the generated state is solvable before returning it.

4. **generate_goal_state(n)**:

   • Creates the goal state for an N x N puzzle, which is an ascending order of numbers with the last spot being blank.

5. **GenerateChildren(state, last_move=None)**:

   • Generates all possible child states by sliding the blank spot.

   • The last_move parameter helps to avoid making a move that reverts the puzzle to its previous state.

   • Uses a mapping of opposite moves to filter out unnecessary moves.

6. **Bfs(initial_state, goal_state)**:

   • Implements the Breadth-First Search algorithm to solve the puzzle.

   • Uses a queue to explore nodes level-wise, ensuring the shortest path is found.

   • Tracks visited states to avoid repetitive exploration.

   • Uses the GenerateChildren function to get possible moves.

7. **Original_Dfs(initial_state, goal_state)**:

   • Implements the Depth-First Search algorithm to solve the puzzle.

   • Uses a stack to explore paths deeply before backtracking.

   • Doesn't track visited states, so it can revisit states.

   • Also uses the GenerateChildren function, with the optimization to avoid reverting moves.

8. **Dfs_with_revisit_check(initial_state, goal_state)**:

   • An enhanced version of DFS, which keeps a record of visited states.

   • This ensures that the algorithm doesn't get stuck in loops.

9. **main()**:

   • The primary execution function.

   • Retrieves the puzzle size and prints the initial and goal states.

   • Allows users to choose an algorithm (BFS, DFS, or DFS with revisit checks) to solve the puzzle.

   • Outputs the solution path, its depth, and memory usage statistics (maximum states stored at once).

   - Allows the user to run generate report function with desired algorithms

DFS(initial_state, goal_state):

• Implements the Iterative Deepening Depth-First Search (IDDFS) algorithm to solve the puzzle.

• Utilizes a stack to store states and continuously explores deeper depths until a solution is found.

• Ensures complete exploration by gradually increasing the depth limit and restarting the search.

• Leverages the **GenerateChildren** function to determine possible next moves and avoid backtracking to the immediate prior state.

• Dynamically adjusts the depth in an outer loop until the solution is found, providing a breadth-like exploration in a depth-first manner.

generate_report(algorithm_name, algorithm, n):

• Generates a report summarizing the results of the provided algorithm's runs.

• Performs 10 runs of the specified algorithm, storing and logging the solution sequence, solution depth, and maximum states stored for each run.

• Writes each run's data into a text file for detailed analysis, including initial and final puzzle states.

• After the 10 runs, calculates descriptive statistics such as minimum, maximum, and average solution depths and states stored.

• Organizes the data in a structured manner, separating each run's details with defined splitters for readability.

4. **Run your program for $3 \leq n \leq 6$.**

   a. **For each *n*, repeat the run *10 times* with a different random *Initial State*. *Final State* should be the standard sorted one.**
   Answers in the next pages

   b. **Report the Initial State, Final State, and the Solution Sequence of Actions.**
   Answers in the next pages

   c. **Across the 10 repetitions, report the descriptive statistics (minimum, maximum, and average) of the solution depth for each n for each solution.**
   Answers in the next pages

   d. **Across the 10 repetitions, report the descriptive statistics (*minimum*, *maximum*, and *average*) of the *maximum number of states concurrently stored* for each *n* for each solution.**
   Answers in the next pages

   e. **Comment on the results.**
   Answers in the next pages

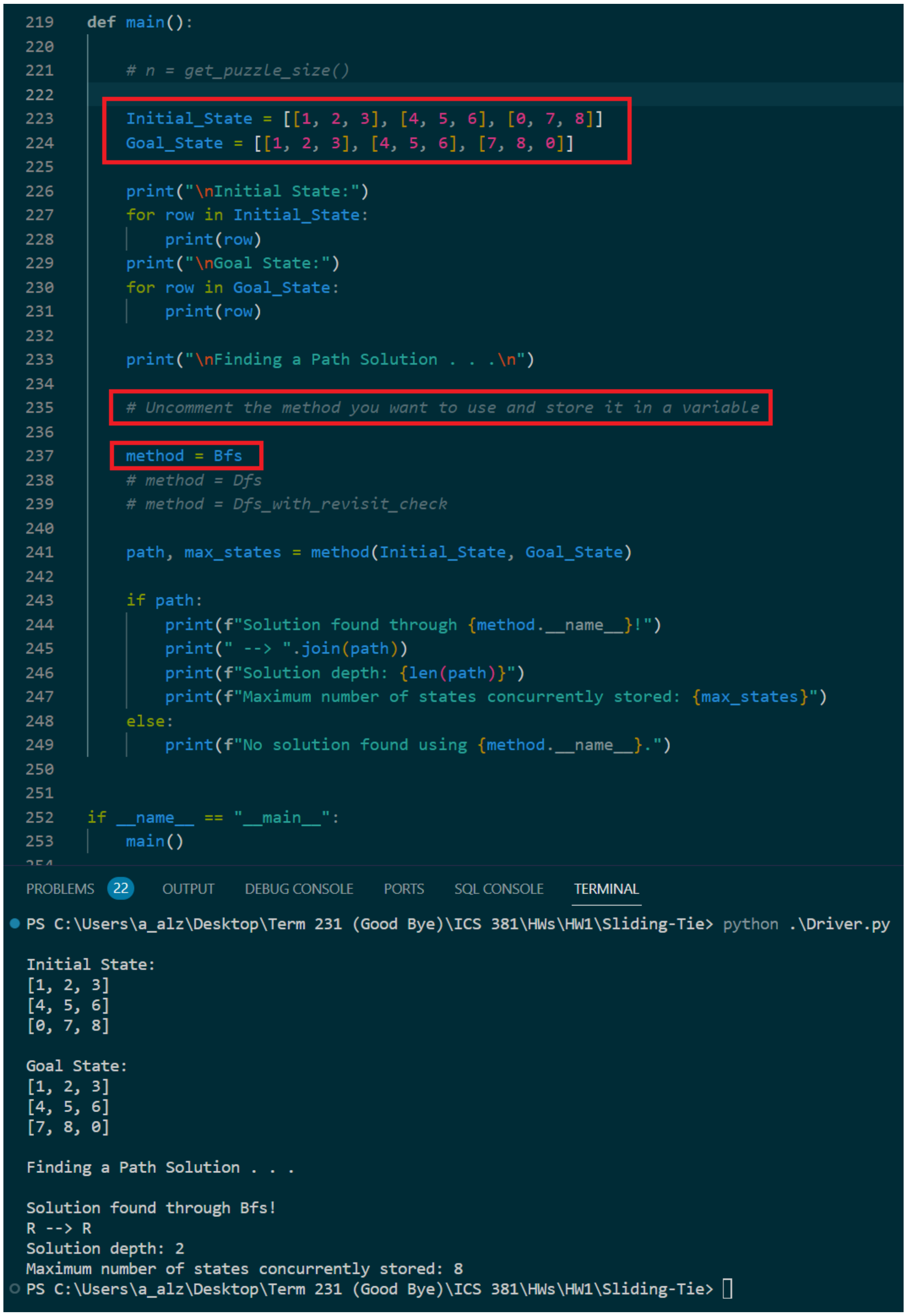
Please note that I designed the code to be dynamic for any NxN puzzle but due to resource limitations I will be showing analysis only for 3x3 randomly generated puzzles so that's what I included in the analysis. Anything larger than 3x3 puzzles would crash and cause memory errors. But I will show simple enforced initial states for larger N sizes to prove that the program logic is dynamic and working correctly and optimized and the only limitation is the machine resources.

<u>With each algorithm I'll first start by enforcing a simple initial state to show that the logic is correct and working.</u>

Analysis for N=3:-

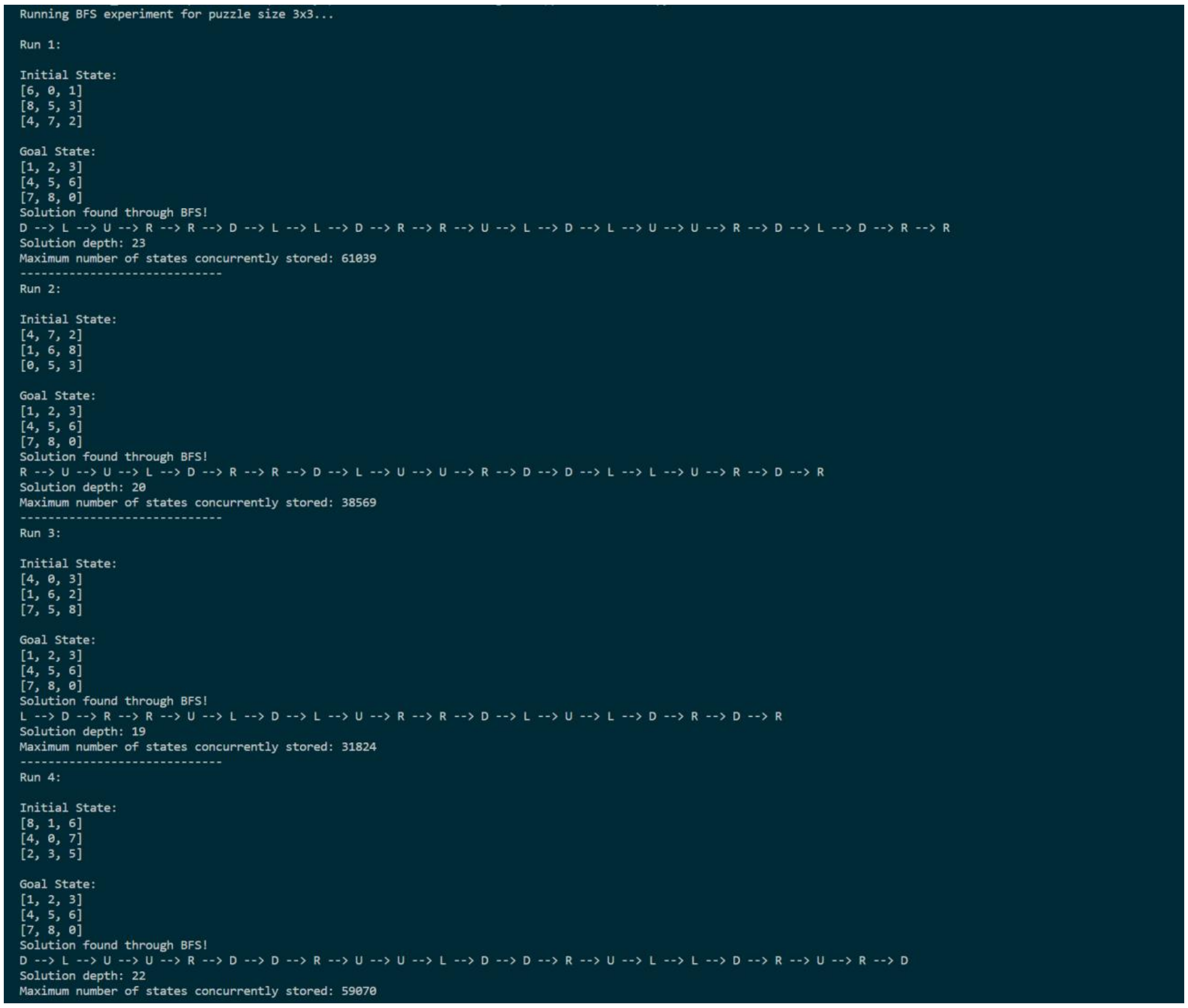
Next Page

BFS:

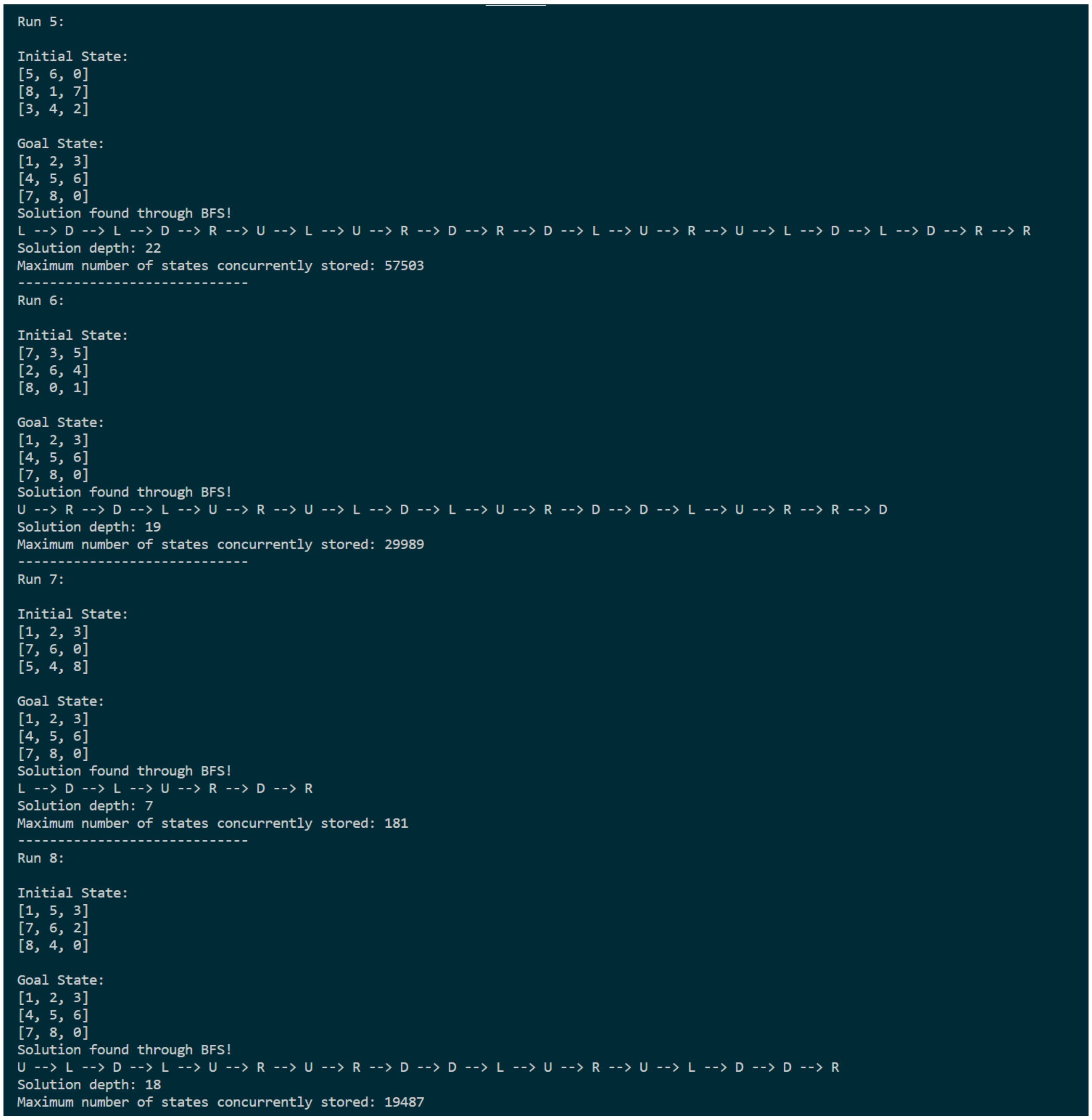


```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
    Goal_State = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    method = Bfs
    # method = Dfs
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```

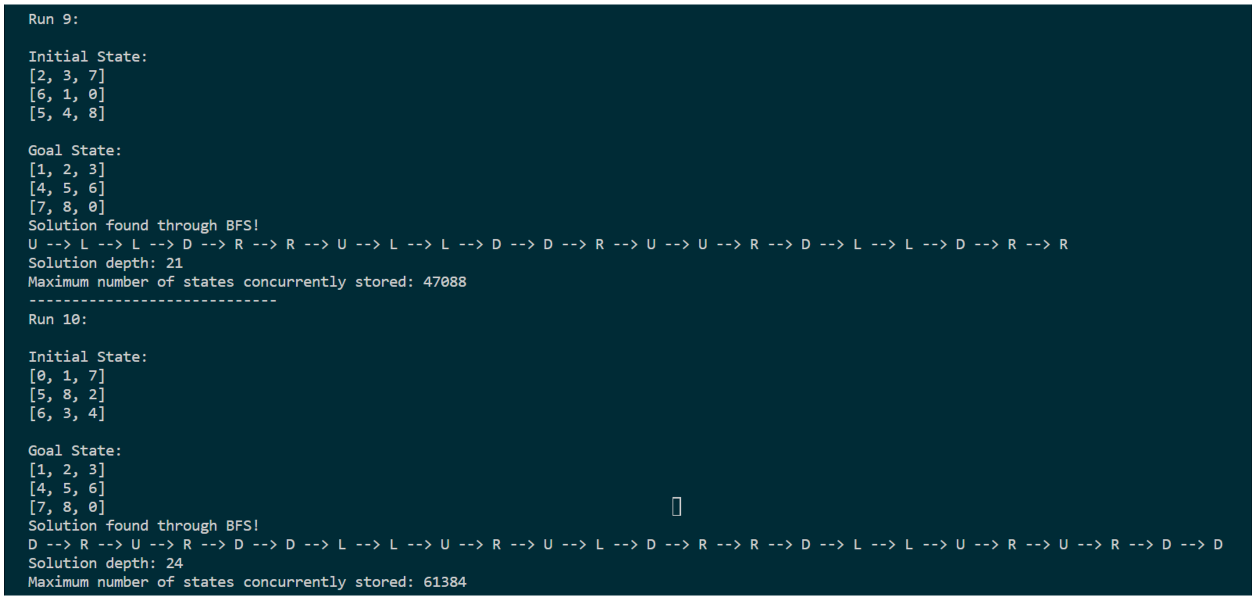
```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Finding a Path Solution . . .

Solution found through Bfs!
R --> R
Solution depth: 2
Maximum number of states concurrently stored: 8
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```



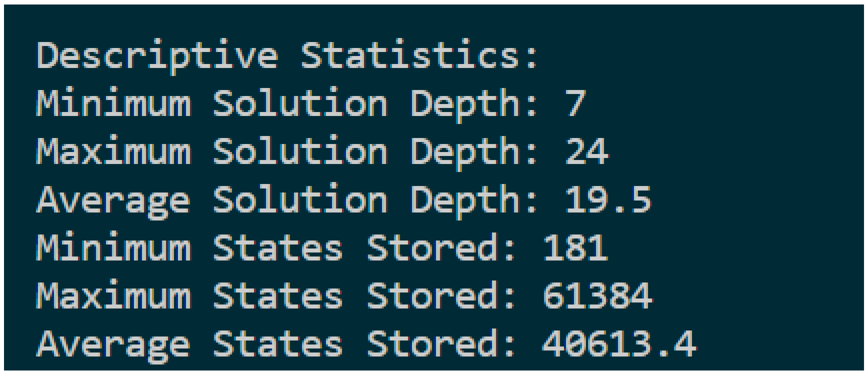
```
Running BFS experiment for puzzle size 3x3...

Run 1:

Initial State:
[6, 0, 1]
[8, 5, 3]
[4, 7, 2]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
D --> L --> U --> R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> D --> L --> U --> U --> R --> D --> L --> D --> R --> R
Solution depth: 23
Maximum number of states concurrently stored: 61039
-----------------------------
Run 2:

Initial State:
[4, 7, 2]
[1, 6, 8]
[0, 5, 3]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
R --> U --> U --> L --> D --> R --> R --> D --> L --> U --> U --> R --> D --> D --> L --> L --> U --> R --> D --> R
Solution depth: 20
Maximum number of states concurrently stored: 38569
-----------------------------
Run 3:

Initial State:
[4, 0, 3]
[1, 6, 2]
[7, 5, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
L --> D --> R --> R --> U --> L --> D --> L --> U --> R --> R --> D --> L --> U --> L --> D --> R --> D --> R
Solution depth: 19
Maximum number of states concurrently stored: 31824
-----------------------------
Run 4:

Initial State:
[8, 1, 6]
[4, 0, 7]
[2, 3, 5]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
D --> L --> U --> U --> R --> D --> D --> R --> U --> U --> L --> D --> D --> R --> U --> L --> L --> D --> R --> U --> R --> D
Solution depth: 22
Maximum number of states concurrently stored: 59070
```



```
Run 5:

Initial State:
[5, 6, 0]
[8, 1, 7]
[3, 4, 2]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
L --> D --> L --> D --> R --> U --> L --> U --> R --> D --> R --> D --> L --> U --> R --> U --> L --> D --> L --> D --> R --> R
Solution depth: 22
Maximum number of states concurrently stored: 57503
-----------------------------
Run 6:

Initial State:
[7, 3, 5]
[2, 6, 4]
[8, 0, 1]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
U --> R --> D --> L --> U --> R --> U --> L --> D --> L --> U --> R --> D --> D --> L --> U --> R --> R --> D
Solution depth: 19
Maximum number of states concurrently stored: 29989
-----------------------------
Run 7:

Initial State:
[1, 2, 3]
[7, 6, 0]
[5, 4, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
L --> D --> L --> U --> R --> D --> R
Solution depth: 7
Maximum number of states concurrently stored: 181
-----------------------------
Run 8:

Initial State:
[1, 5, 3]
[7, 6, 2]
[8, 4, 0]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
U --> L --> D --> L --> U --> R --> U --> R --> D --> D --> L --> U --> R --> U --> L --> D --> D --> R
Solution depth: 18
Maximum number of states concurrently stored: 19487
```



```
Run 9:

Initial State:
[2, 3, 7]
[6, 1, 0]
[5, 4, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> D --> R --> U --> U --> R --> D --> L --> L --> D --> R --> R
Solution depth: 21
Maximum number of states concurrently stored: 47088
-----------------------------
Run 10:

Initial State:
[0, 1, 7]
[5, 8, 2]
[6, 3, 4]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Solution found through BFS!
D --> R --> U --> R --> D --> D --> L --> L --> U --> R --> U --> L --> D --> R --> R --> D --> L --> L --> U --> R --> U --> R --> D --> D
Solution depth: 24
Maximum number of states concurrently stored: 61384
```



```
Descriptive Statistics:
Minimum Solution Depth: 7
Maximum Solution Depth: 24
Average Solution Depth: 19.5
Minimum States Stored: 181
Maximum States Stored: 61384
Average States Stored: 40613.4
```

DFS:

Note: due to the nature of DFS algorithm the solution paths are very long because there is no revisit check and it can easily get stuck in loops and its dependent on the complexity of the random states generated. **A considerable amount of execution time and trials and optimization went into trying to make the original DFS work but constant crashes and memory issues were faced**. I succeeded in getting only one run with a solution depth of 1810 (The details of the run are on the next pages). **There for I decided to modify the original DFS and make it work in an iterative deepening fashion**. Even though both the original DFS and the iterative deepening version revisit already visited nodes, the iterative deepening version outperforms the original one in the following aspects:

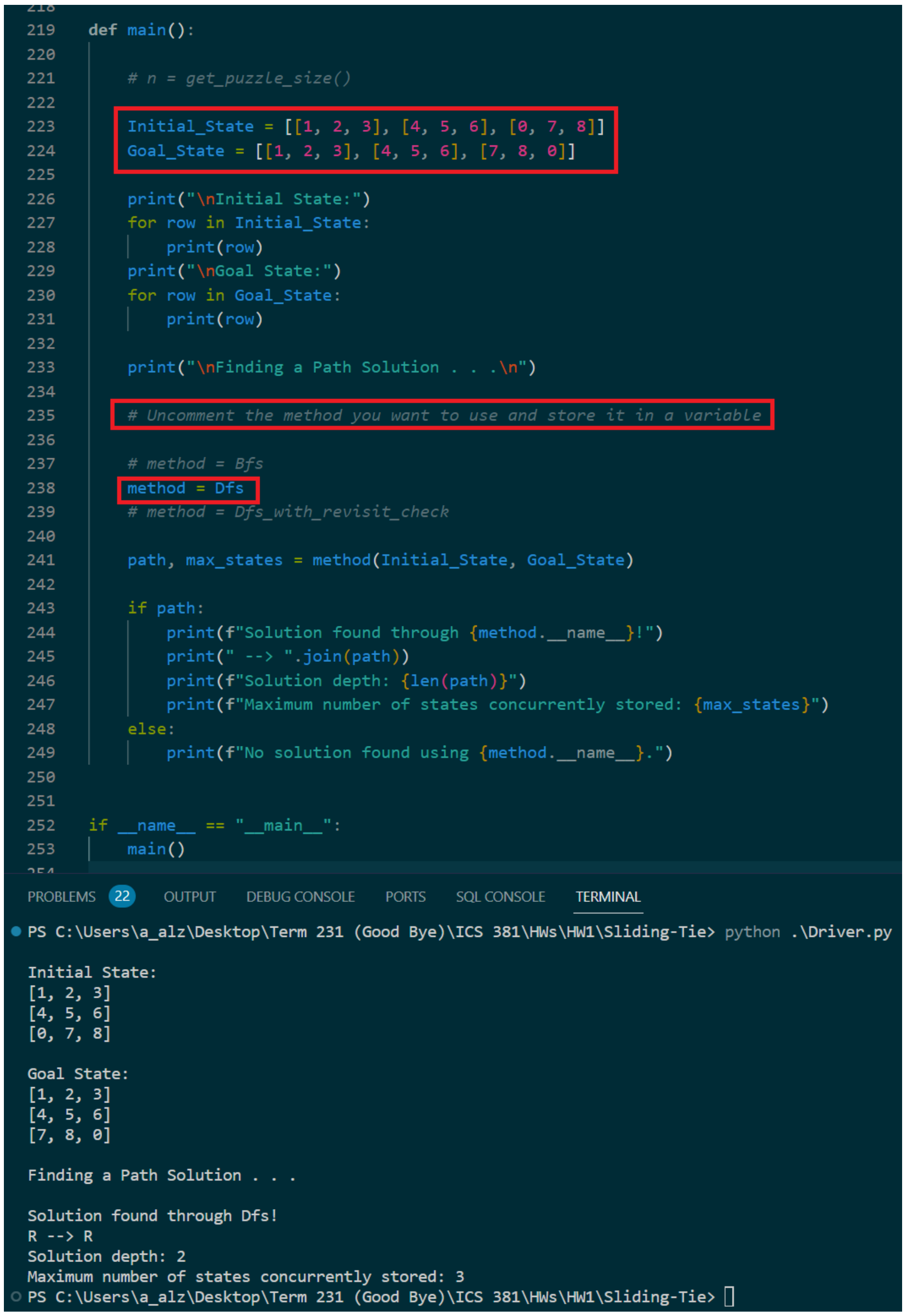
1. **Completeness**:
    - **IDDFS**: It's complete. Given enough iterations, IDDFS will eventually search every level of the state space, ensuring that if a solution exists, it will be found.
    - **DFS**: It might not be complete in infinite state spaces. DFS can get trapped in a branch that goes infinitely deep and never find a solution, even if one exists.

2. **Space Complexity**:
    - **IDDFS**: Has linear space complexity, since it only needs to store a single branch of the tree at any given depth level. This makes it memory-efficient.
    - **DFS**: Also has linear space complexity. However, the branch it explores might be extremely long, especially if it goes down a path without a solution.

3. **Time Complexity**:
   - **IDDFS**: It might seem inefficient because it revisits nodes multiple times. But for many problems, the overhead of revisiting states is small compared to the exponential growth of the state space. The bulk of the work is done at the deepest level, making it comparable to BFS in terms of time complexity for many problems.
   - **DFS**: If it goes down the wrong path, especially one without a solution, it might explore a large or infinite number of states without any benefit, wasting time. Also, if solutions exist at shallower depths, DFS might miss a quick solution by delving deeper than necessary.

4. **Optimality (in terms of path length)**:
   - **IDDFS**: Finds the shallowest (shortest) solution first, since it starts searching from depth 0 and increases the depth iteratively.
   - **DFS**: It does not guarantee finding the shortest solution. It finds the first solution it encounters, which might be down a very deep path even if shorter paths exist.

Please Be aware that from this point onward whenever we refer to DFS we are referring to the Iterative deepening version of it unless specified otherwise.



```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
    Goal_State = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    method = Dfs
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Finding a Path Solution . . .

Solution found through Dfs!
R --> R
Solution depth: 2
Maximum number of states concurrently stored: 3
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

(**Test Run for Original DFS**)

```
Initial State:
6 8 1
4 5 2
0 7 3

Goal State:
1 2 3
4 5 6
7 8 0

Solution found through DFS !
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> U --> R --> R --> D --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> U --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> U --> R --> R --> D --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> U --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> U --> R --> R -->
D --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
```

U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> U --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> U -->
R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> U --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> U --> R --> R --> D --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> U --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> U --> R --> R --> D --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> U -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> U --> R --> R --> D --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> U -->
R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> U --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> U --> R --> R --> D --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> U --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->

L --> L --> U --> U --> R --> R --> D --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> U -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> U --> R --> R --> D --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> U --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> U --> R --> R --> D --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> U --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> U --> R --> R -->
D --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> U --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> U -->
R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> U --> R --> R --> D --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> U --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> U --> R --> R --> D --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> U --> R --> D --> L --> L --> U --> R --> R --> D -->

L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> U --> R --> R -->
D --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> U --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> U -->
R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> U --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> U --> R --> R --> D --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> U --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> U --> R --> R --> D --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> U -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> U --> R --> D --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> U --> R --> R --> D --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> U --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> U --> R --> D --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->

R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> U -->
R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> U --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> U --> R --> R --> D --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> U --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> U --> R --> R --> D --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> U -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> U --> R --> R --> D --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> U --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> U --> R --> R --> D --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R --> U --> L -->
L --> D --> R --> U --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> R --> R --> D -->
L --> L --> U --> R --> R --> D --> L --> L --> U --> U --> R --> R -->
D --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R -->
U --> L --> L --> D --> R --> U --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> R -->
R --> D --> L --> L --> U --> R --> R --> D --> L --> L --> U --> U -->

```
R --> R --> D --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> U --> R --> R --> D --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> R --> U --> L --> L --> D --> R --> R --> U --> L --> L --> D -->
R --> U --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> R --> R --> D --> L --> L -->
U --> R --> R --> D --> L --> L --> U --> U --> R --> R --> D --> L -->
L --> D --> R --> R --> U --> L --> L --> D --> R --> R

Solution depth: 1810

Maximum number of states concurrently stored: 23
```

DFS after making iterative deepening modifications

The 10 Runs are on the next page

Run 1:

Initial State:

0 6 2

4 7 3

8 1 5

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

D --> R --> U --> R --> D --> L --> D --> R --> U --> U --> L --> D -->
D --> L --> U --> U --> R --> R --> D --> D

Solution Depth: 20

Max States Stored: 17

Run 2:

Initial State:

5 1 6

0 8 4

3 2 7

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

R --> R --> D --> L --> L --> U --> R --> D --> L --> U --> U --> R --> D --> R --> D --> L --> U --> R --> U --> L --> D --> L --> D --> R --> R

Solution Depth: 25

Max States Stored: 23

Run 3:

Initial State:

8 7 5

2 0 1

4 6 3

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

U --> R --> D --> D --> L --> U --> U --> R --> D --> L --> L --> U --> R --> D --> L --> D --> R --> U --> R --> D

Solution Depth: 20

Max States Stored: 19

Run 4:

Initial State:

3 7 6

4 0 1

8 5 2

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

U --> L --> D --> R --> R --> U --> L --> D --> D --> R --> U --> L --> D --> L --> U --> U --> R --> D --> R --> D

Solution Depth: 20

Max States Stored: 19

Run 5:

Initial State:

6 8 0

2 7 1

5 3 4

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

D --> L --> D --> R --> U --> L --> U --> R --> D --> L --> L --> U -->
R --> D --> L --> D --> R --> R --> U --> L --> L --> D --> R --> R

Solution Depth: 24

Max States Stored: 21

Run 6:

Initial State:

8 0 5

1 6 7

2 4 3

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

L --> D --> D --> R --> R --> U --> U --> L --> D --> D --> R --> U --> U --> L --> D --> L --> D --> R --> R --> U --> U --> L --> D --> D --> R

Solution Depth: 25

Max States Stored: 22

Run 7:

Initial State:

0 5 2

1 8 6

7 4 3

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

R --> R --> D --> L --> D --> R --> U --> U --> L --> L --> D --> R --> D --> R --> U --> L --> U --> R --> D --> D

Solution Depth: 20

Max States Stored: 17

Run 8:

Initial State:

8 1 7

0 4 5

2 3 6

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

R --> D --> L --> U --> U --> R --> D --> D --> R --> U --> U --> L --> D --> R --> D --> L --> U --> L --> D --> R --> U --> R --> D

Solution Depth: 23

Max States Stored: 21

Run 9:

Initial State:

3 6 5

2 0 7

4 8 1

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

R --> D --> L --> U --> R --> U --> L --> L --> D --> R --> R --> U -->
L --> L --> D --> D --> R --> R

Solution Depth: 18

Max States Stored: 17

Run 10:

Initial State:

5 2 8

3 6 1

7 0 4

Final State:

1 2 3

4 5 6

7 8 0

Algorithm Used: DFS

Solution Sequence:

U --> R --> U --> L --> L --> D --> R --> R --> D --> L --> U --> R -->
U --> L --> L --> D --> R --> R --> U --> L --> D --> R --> D

Solution Depth: 23

Max States Stored: 21

Descriptive Statistics:

Minimum Solution Depth: 18

Maximum Solution Depth: 25

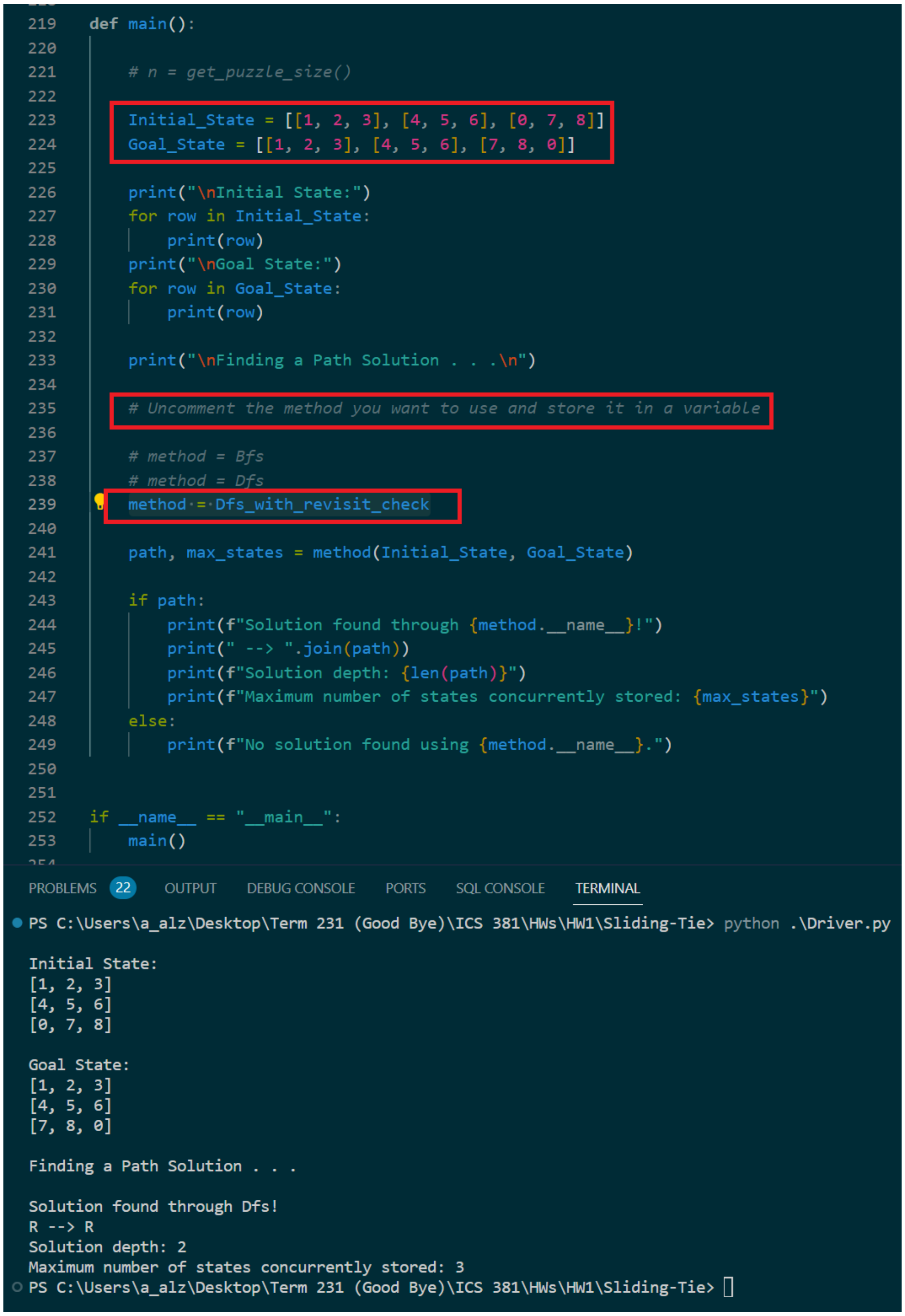
Average Solution Depth: 21.80

Minimum States Stored: 17

Maximum States Stored: 23

Average States Stored: 19.70

DFS (With Revisit Check):



```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
    Goal_State = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    # method = Dfs
    method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Finding a Path Solution . . .

Solution found through Dfs!
R --> R
Solution depth: 2
Maximum number of states concurrently stored: 3
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

Note: Due to the long solution sequences generated by DFS (With Revisit Check) and for the brevity of this document I will be providing the complete output of the 10 runs in a separate file called **Output.txt** that will be submitted alongside this report.

1- Solution Depth: 49523

Max States Stored: 39008

2- Solution Depth: 89508

Max States Stored: 70582

3-Solution Depth: 105411

Max States Stored: 83069

4-Solution Depth: 61807

Max States Stored: 48716

5-Solution Depth: 41594

Max States Stored: 32759

6-Solution Depth: 41465

Max States Stored: 32658

7-Solution Depth: 91214

Max States Stored: 71931

8-Solution Depth: 93816

Max States Stored: 73936

9-Solution Depth: 79204

Max States Stored: 62433

10-Solution Depth: 25449

Max States Stored: 20053

Descriptive Statistics:

Minimum Solution Depth: 25449

Maximum Solution Depth: 105411

Average Solution Depth: 70549.2

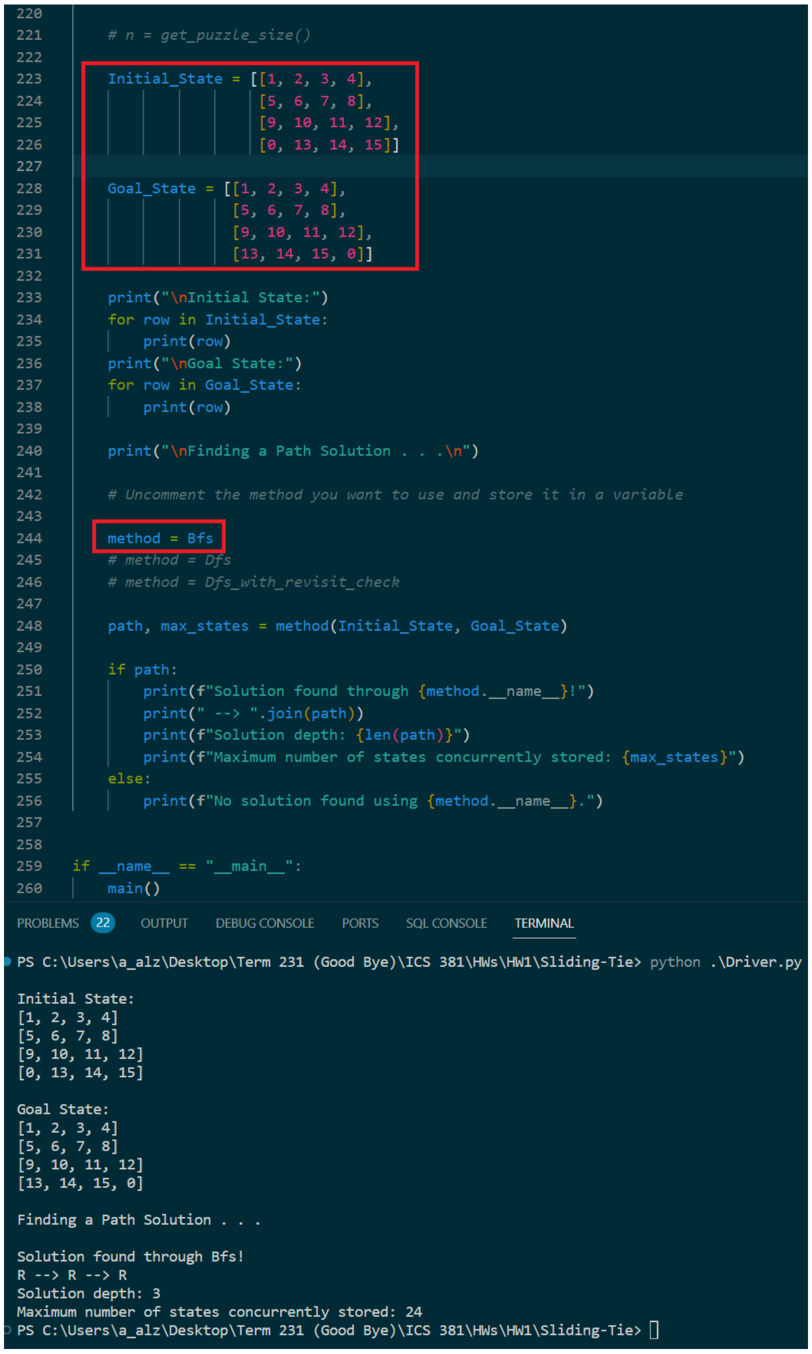
Minimum States Stored: 20053

Maximum States Stored: 83069

Average States Stored: 55419.5

Demo Runs For N=4:-

BFS:

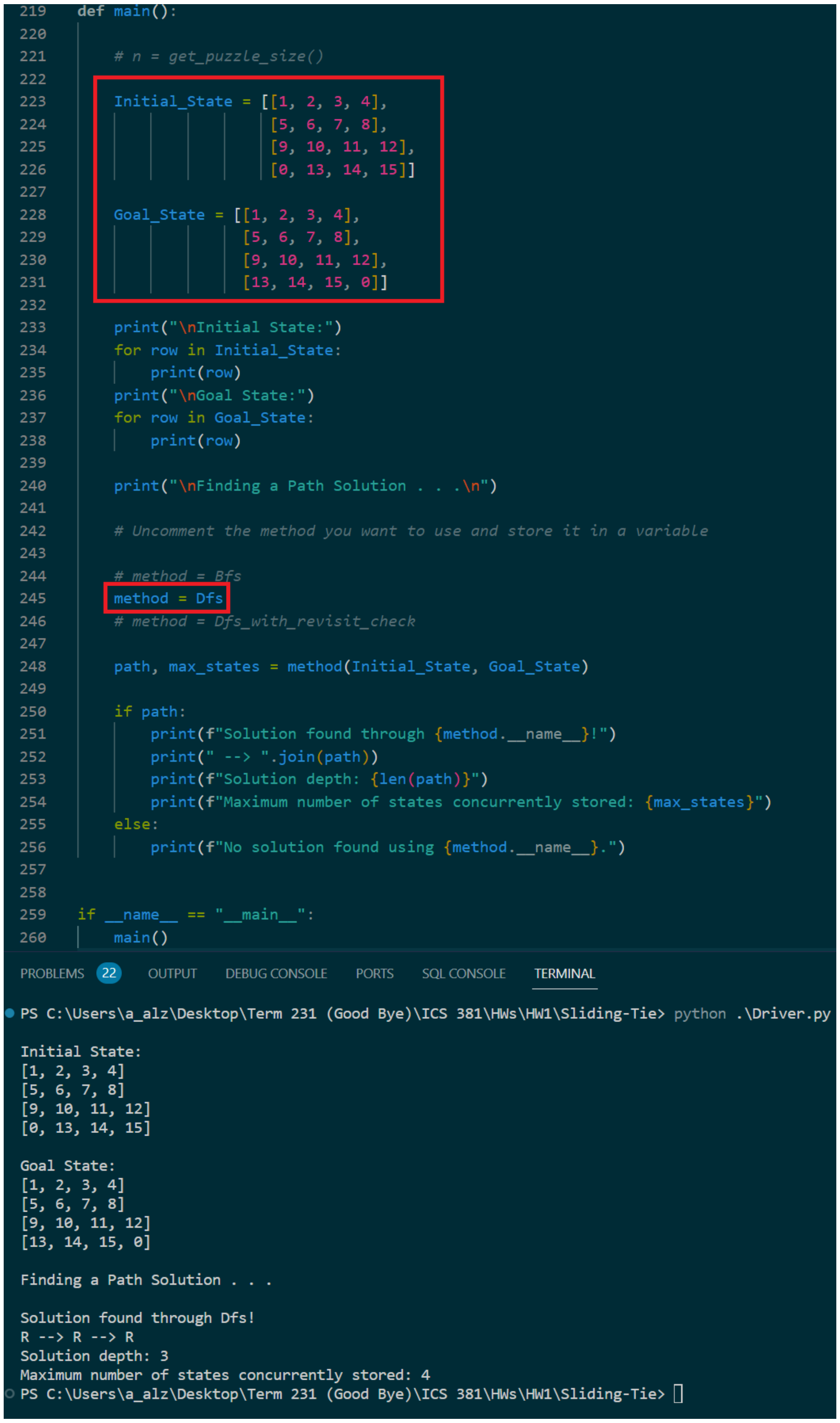


```python

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12],
                     [0, 13, 14, 15]]

    Goal_State = [[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    method = Bfs
    # method = Dfs
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```

PROBLEMS 22   OUTPUT   DEBUG CONSOLE   PORTS   SQL CONSOLE   TERMINAL

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[0, 13, 14, 15]

Goal State:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]

Finding a Path Solution . . .

Solution found through Bfs!
R --> R --> R
Solution depth: 3
Maximum number of states concurrently stored: 24
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> 
```

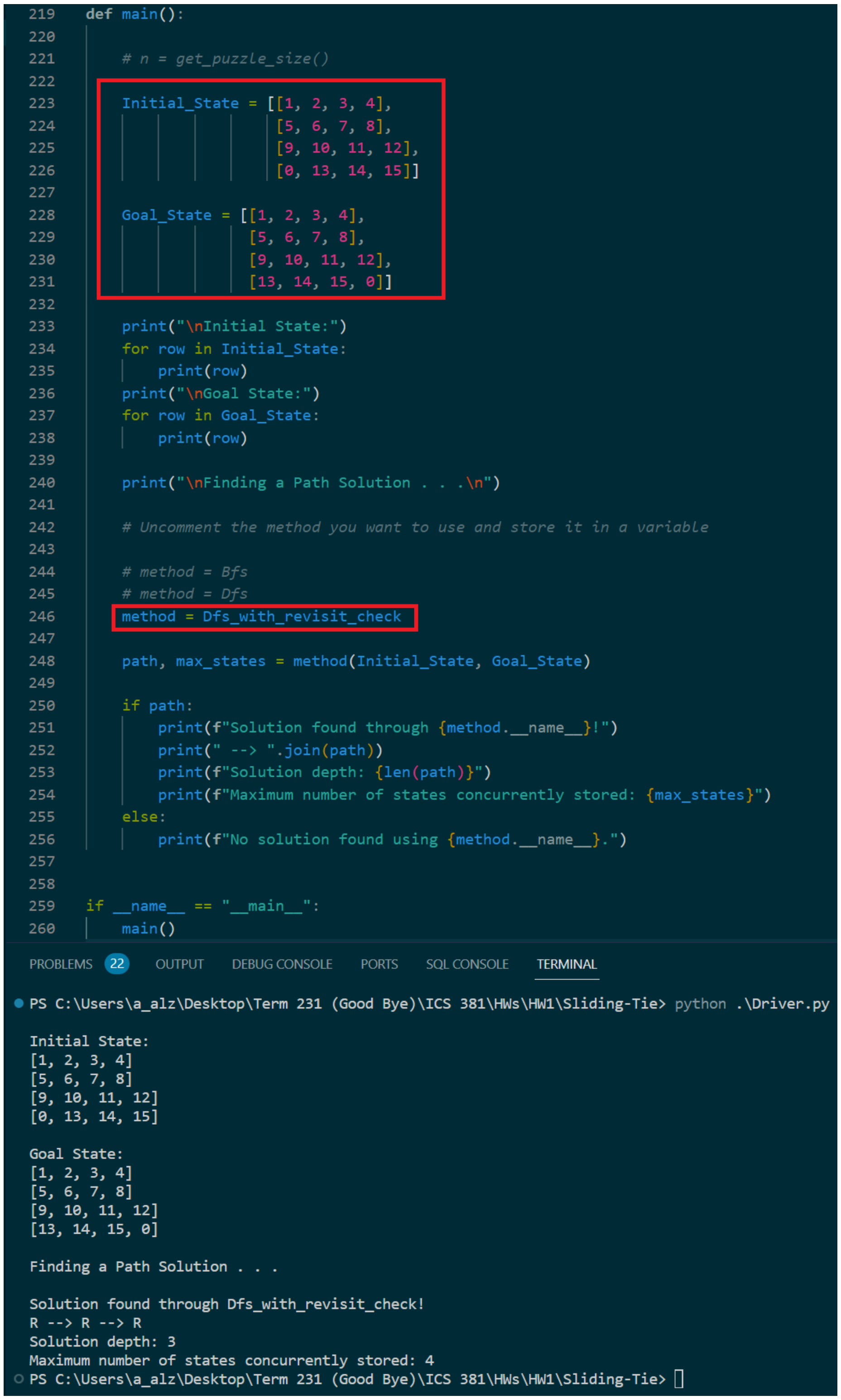
DFS:



```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12],
                     [0, 13, 14, 15]]

    Goal_State = [[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    method = Dfs
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[0, 13, 14, 15]

Goal State:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]

Finding a Path Solution . . .

Solution found through Dfs!
R --> R --> R
Solution depth: 3
Maximum number of states concurrently stored: 4
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

DFS (Revisit):

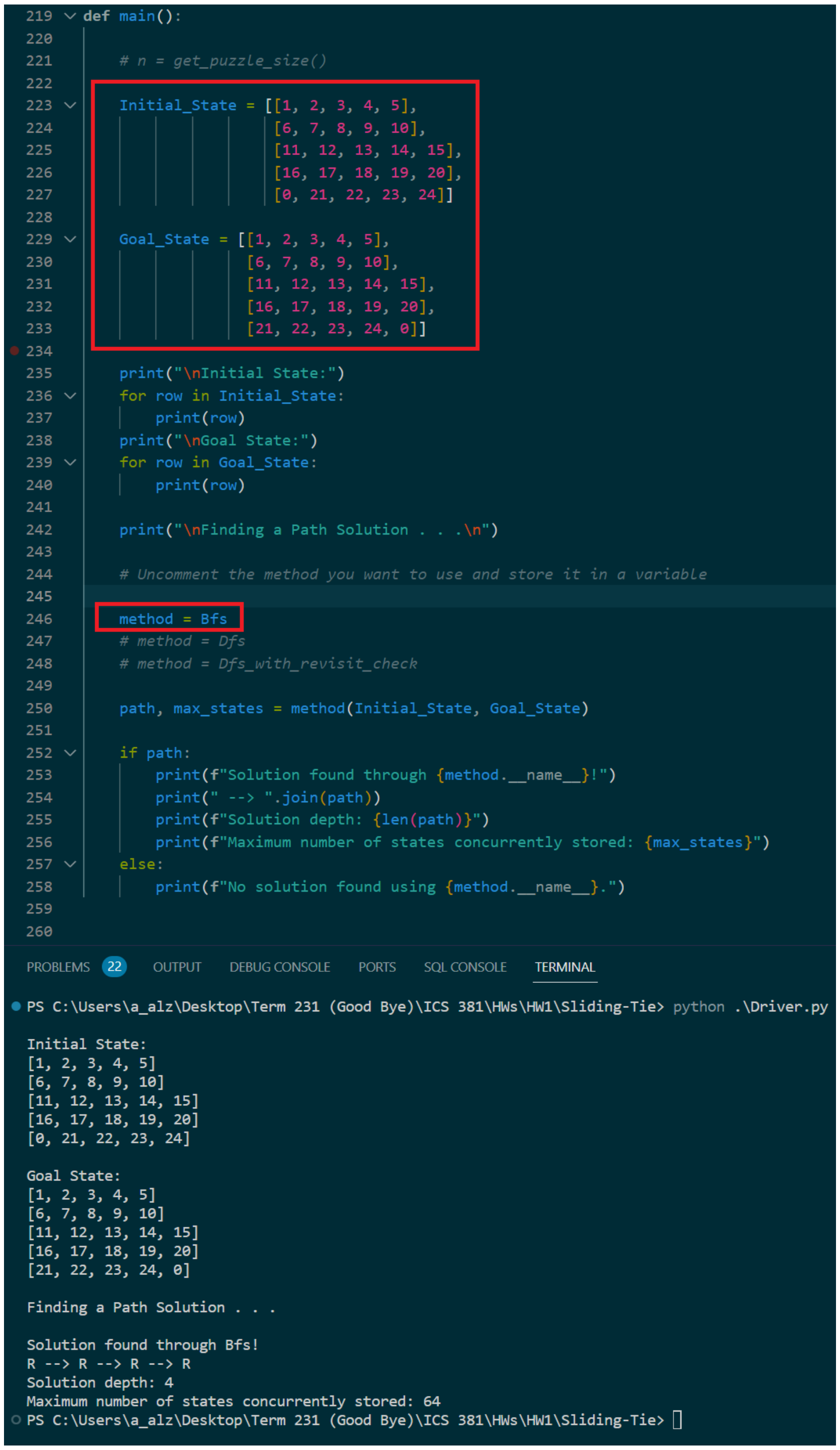


```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12],
                     [0, 13, 14, 15]]

    Goal_State = [[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    # method = Dfs
    method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[0, 13, 14, 15]

Goal State:
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]

Finding a Path Solution . . .

Solution found through Dfs_with_revisit_check!
R --> R --> R
Solution depth: 3
Maximum number of states concurrently stored: 4
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

Demo Runs For N=5:-

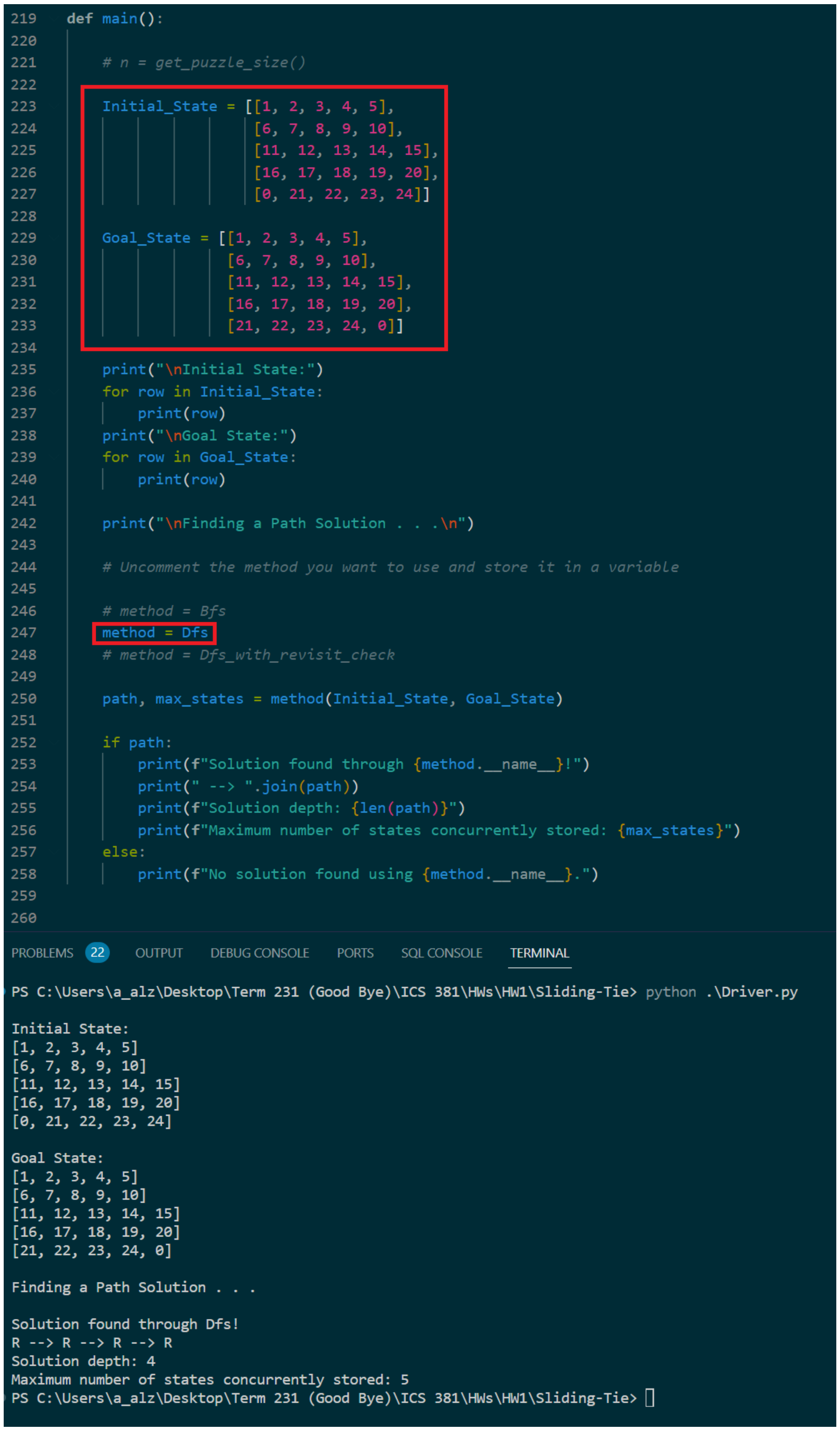
BFS:



```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3, 4, 5],
                     [6, 7, 8, 9, 10],
                     [11, 12, 13, 14, 15],
                     [16, 17, 18, 19, 20],
                     [0, 21, 22, 23, 24]]

    Goal_State = [[1, 2, 3, 4, 5],
                  [6, 7, 8, 9, 10],
                  [11, 12, 13, 14, 15],
                  [16, 17, 18, 19, 20],
                  [21, 22, 23, 24, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    method = Bfs
    # method = Dfs
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[0, 21, 22, 23, 24]

Goal State:
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[21, 22, 23, 24, 0]

Finding a Path Solution . . .

Solution found through Bfs!
R --> R --> R --> R
Solution depth: 4
Maximum number of states concurrently stored: 64
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

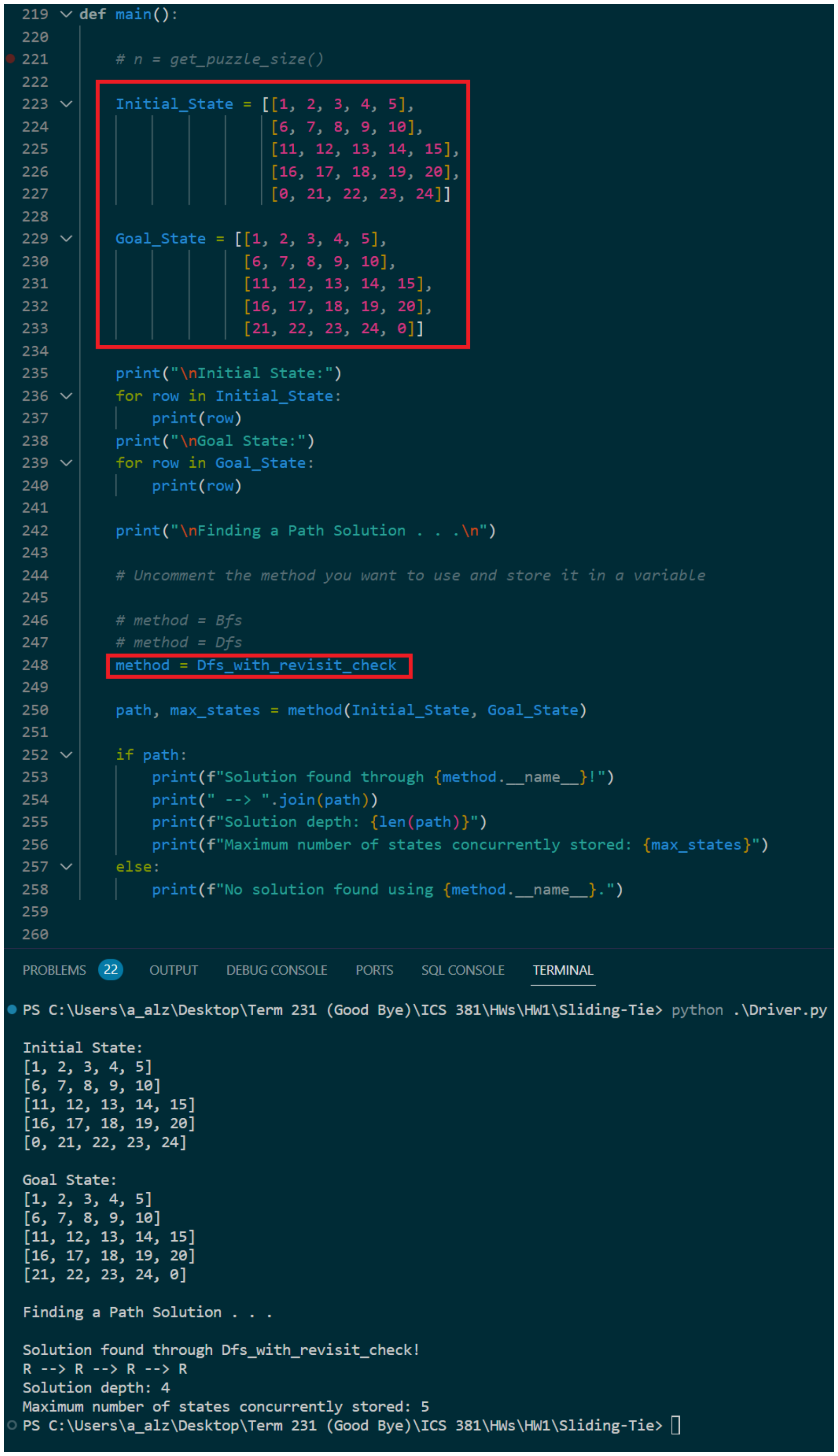
DFS:



```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3, 4, 5],
                     [6, 7, 8, 9, 10],
                     [11, 12, 13, 14, 15],
                     [16, 17, 18, 19, 20],
                     [0, 21, 22, 23, 24]]

    Goal_State = [[1, 2, 3, 4, 5],
                  [6, 7, 8, 9, 10],
                  [11, 12, 13, 14, 15],
                  [16, 17, 18, 19, 20],
                  [21, 22, 23, 24, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    method = Dfs
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")

```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[0, 21, 22, 23, 24]

Goal State:
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[21, 22, 23, 24, 0]

Finding a Path Solution . . .

Solution found through Dfs!
R --> R --> R --> R
Solution depth: 4
Maximum number of states concurrently stored: 5
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

DFS (Revisit):

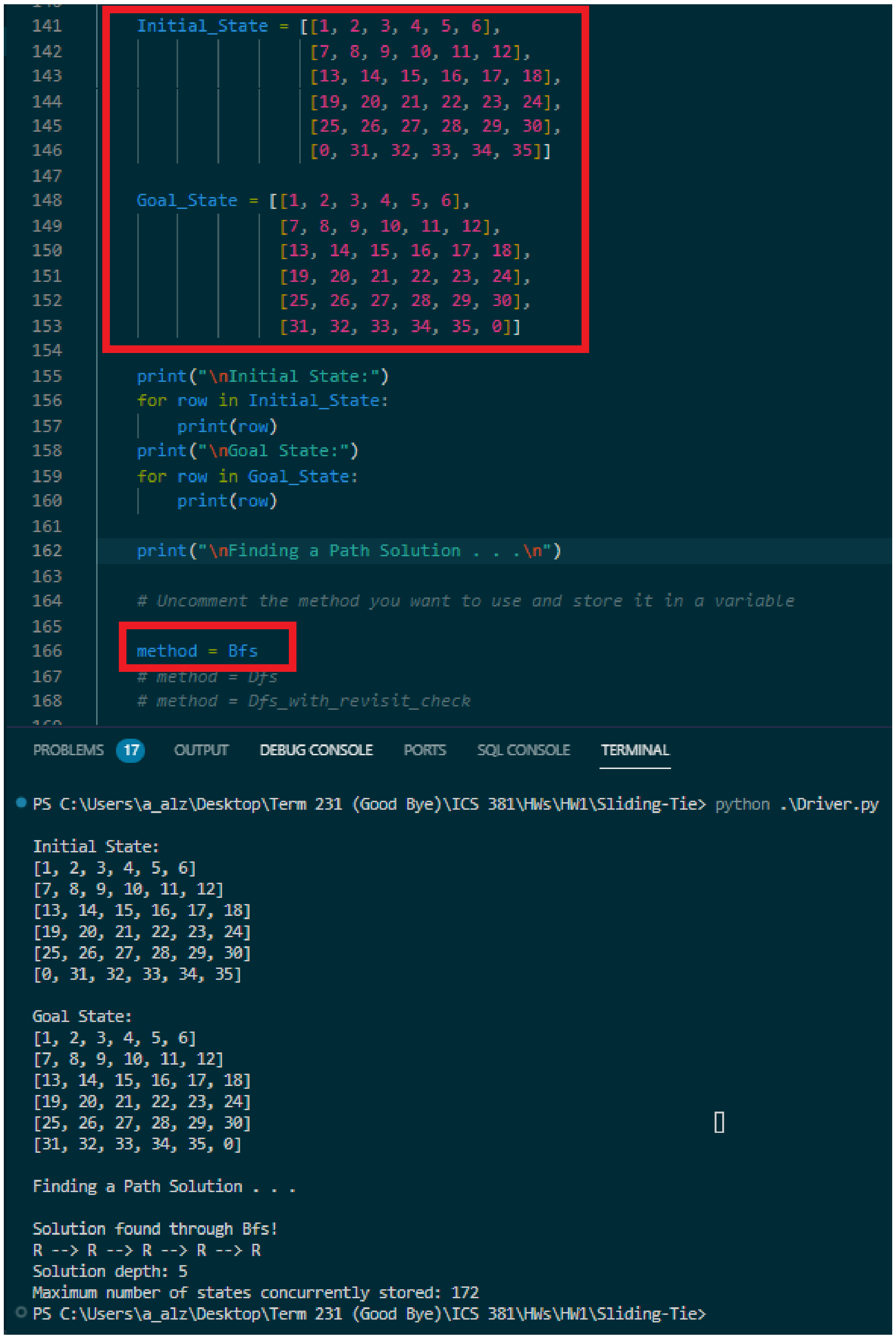


```python
def main():

    # n = get_puzzle_size()

    Initial_State = [[1, 2, 3, 4, 5],
                     [6, 7, 8, 9, 10],
                     [11, 12, 13, 14, 15],
                     [16, 17, 18, 19, 20],
                     [0, 21, 22, 23, 24]]

    Goal_State = [[1, 2, 3, 4, 5],
                  [6, 7, 8, 9, 10],
                  [11, 12, 13, 14, 15],
                  [16, 17, 18, 19, 20],
                  [21, 22, 23, 24, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    # method = Dfs
    method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[0, 21, 22, 23, 24]

Goal State:
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
[11, 12, 13, 14, 15]
[16, 17, 18, 19, 20]
[21, 22, 23, 24, 0]

Finding a Path Solution . . .

Solution found through Dfs_with_revisit_check!
R --> R --> R --> R
Solution depth: 4
Maximum number of states concurrently stored: 5
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

Demo Runs For N=6:-

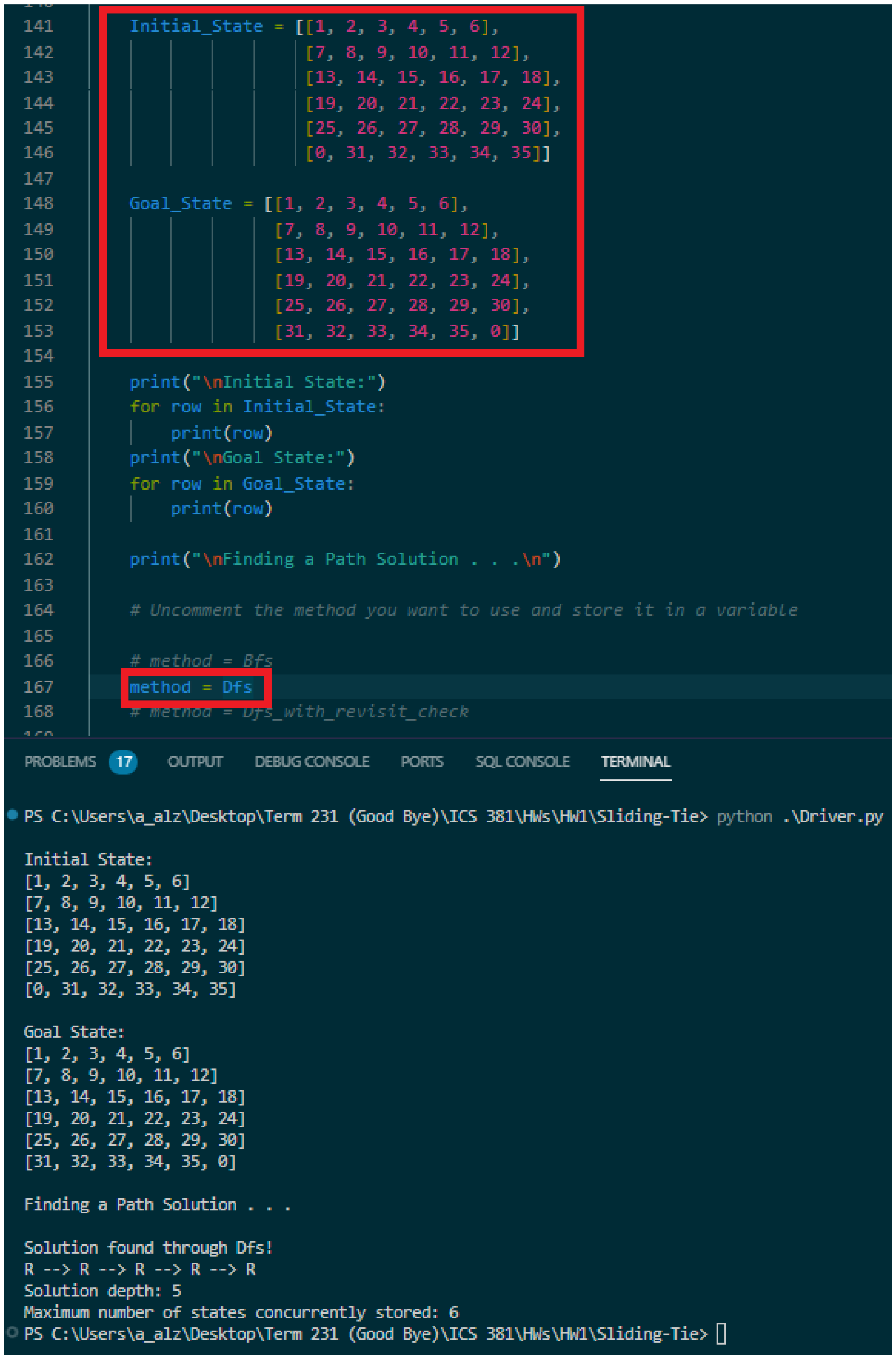
BFS:



```
    Initial_State = [[1, 2, 3, 4, 5, 6],
                     [7, 8, 9, 10, 11, 12],
                     [13, 14, 15, 16, 17, 18],
                     [19, 20, 21, 22, 23, 24],
                     [25, 26, 27, 28, 29, 30],
                     [0, 31, 32, 33, 34, 35]]

    Goal_State = [[1, 2, 3, 4, 5, 6],
                  [7, 8, 9, 10, 11, 12],
                  [13, 14, 15, 16, 17, 18],
                  [19, 20, 21, 22, 23, 24],
                  [25, 26, 27, 28, 29, 30],
                  [31, 32, 33, 34, 35, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    method = Bfs
    # method = Dfs
    # method = Dfs_with_revisit_check
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18]
[19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30]
[0, 31, 32, 33, 34, 35]

Goal State:
[1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18]
[19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30]
[31, 32, 33, 34, 35, 0]

Finding a Path Solution . . .

Solution found through Bfs!
R --> R --> R --> R --> R
Solution depth: 5
Maximum number of states concurrently stored: 172
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

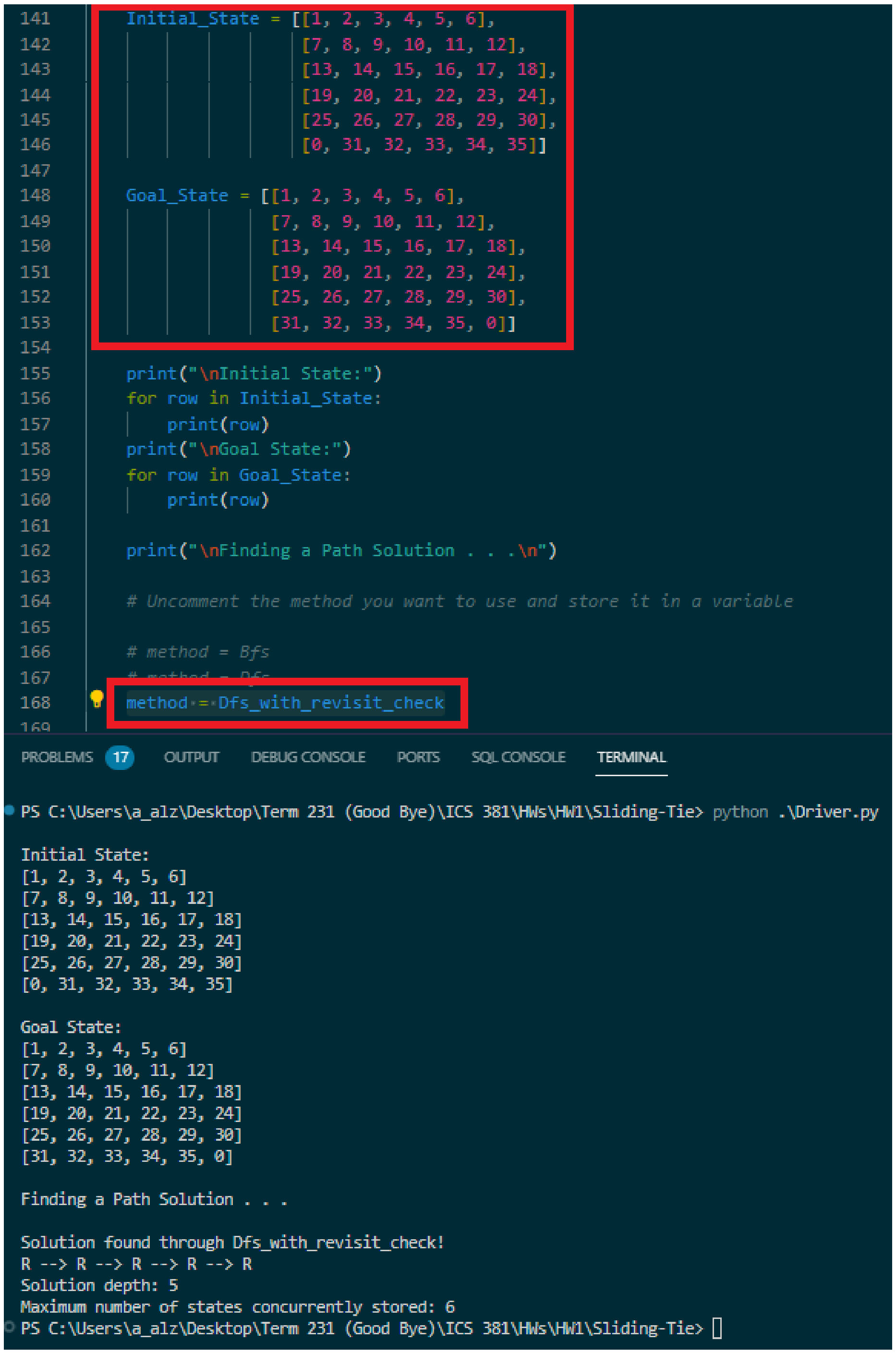
DFS:



```python
    Initial_State = [[1, 2, 3, 4, 5, 6],
                     [7, 8, 9, 10, 11, 12],
                     [13, 14, 15, 16, 17, 18],
                     [19, 20, 21, 22, 23, 24],
                     [25, 26, 27, 28, 29, 30],
                     [0, 31, 32, 33, 34, 35]]

    Goal_State = [[1, 2, 3, 4, 5, 6],
                  [7, 8, 9, 10, 11, 12],
                  [13, 14, 15, 16, 17, 18],
                  [19, 20, 21, 22, 23, 24],
                  [25, 26, 27, 28, 29, 30],
                  [31, 32, 33, 34, 35, 0]]

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    method = Dfs
    # method = Dfs_with_revisit_check
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18]
[19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30]
[0, 31, 32, 33, 34, 35]

Goal State:
[1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18]
[19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30]
[31, 32, 33, 34, 35, 0]

Finding a Path Solution . . .

Solution found through Dfs!
R --> R --> R --> R --> R
Solution depth: 5
Maximum number of states concurrently stored: 6
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

DFS (Revisit):



```python
Initial_State = [[1, 2, 3, 4, 5, 6],
                 [7, 8, 9, 10, 11, 12],
                 [13, 14, 15, 16, 17, 18],
                 [19, 20, 21, 22, 23, 24],
                 [25, 26, 27, 28, 29, 30],
                 [0, 31, 32, 33, 34, 35]]

Goal_State = [[1, 2, 3, 4, 5, 6],
              [7, 8, 9, 10, 11, 12],
              [13, 14, 15, 16, 17, 18],
              [19, 20, 21, 22, 23, 24],
              [25, 26, 27, 28, 29, 30],
              [31, 32, 33, 34, 35, 0]]

print("\nInitial State:")
for row in Initial_State:
    print(row)
print("\nGoal State:")
for row in Goal_State:
    print(row)

print("\nFinding a Path Solution . . .\n")

# Uncomment the method you want to use and store it in a variable

# method = Bfs
# method = Dfs
method = Dfs_with_revisit_check
```

```
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie> python .\Driver.py

Initial State:
[1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18]
[19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30]
[0, 31, 32, 33, 34, 35]

Goal State:
[1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12]
[13, 14, 15, 16, 17, 18]
[19, 20, 21, 22, 23, 24]
[25, 26, 27, 28, 29, 30]
[31, 32, 33, 34, 35, 0]

Finding a Path Solution . . .

Solution found through Dfs_with_revisit_check!
R --> R --> R --> R --> R
Solution depth: 5
Maximum number of states concurrently stored: 6
PS C:\Users\a_alz\Desktop\Term 231 (Good Bye)\ICS 381\HWs\HW1\Sliding-Tie>
```

Comment:

Based on the descriptive analysis provided for each algorithm BFS gives the optimal solution but suffers in terms of states stored. While on the other hand the modified DFS (iterative deepening) although comparable in terms of numbers to BFS it does not give the optimal solution but it actually outperforms BFS in terms of concurrently stored states. DFS (With Revisit Check) is the worst of the 3 algorithms owing to the nature of the algorithm itself producing very long solution sequences. While the revisit check ensures that the algorithm doesn't get caught in infinite loops or cycles by revisiting already explored states, it doesn't inherently lead to shorter paths. A state could be revisited from a different, longer path, and the algorithm might explore from there, leading to a longer solution sequence.

In conclusion after many tests and trials of the 3 algorithms here are my observations for each:

1. **DFS with Revisit Check**:
    - **Non-optimal Solutions**: Just like the standard DFS, DFS with revisit check does not guarantee the shortest solution.
    - **Memory Overhead**: Although the revisit check prevents expanding already seen states, the memory overhead from the stack can still be significant for deep or long paths.
    - **Slower Due to Check**: The revisit check, which involves searching in the **visited** set, adds an overhead, which can slow down the search, especially when the set becomes large.

2. **Depth-First Search (Iterative Deepening Version aka IDDFS):**

   - **Overhead of Revisits**: The most significant limitation of IDDFS is that it repeatedly visits and expands the same states multiple times due to its iterative nature. For example, nodes at depth 1 are expanded once for depth limit 1, again for depth limit 2, and so on.
   - **Memory Overhead**: Although it generally requires less memory than BFS because it doesn't store all frontier nodes at once, IDDFS can still have significant memory usage, especially when the depth limit increases.
   - **Slower for Shallow Solutions**: If the solution is located at a shallow depth, IDDFS might be slower than straightforward DFS because of the repeated expansion of states at lower depths.
   - **No Benefit for Deep Solutions**: If the solution is located at a significant depth, IDDFS provides no advantage over standard DFS, as both will search deeply before finding a solution.
   - **State Explosion**: Similar to other blind search methods, the number of possible states in an NxN puzzle grows significantly with depth. As the depth increases, the number of states can grow exponentially.

3. **Breadth-First Search (BFS)**:
    - **Optimal Solution**: BFS does guarantee an optimal solution in terms of the number of moves, but this comes at a cost.
    - **Memory Overhead**: BFS can have a significant memory overhead, especially in the later stages of the search, because it stores all frontier nodes in the queue. For larger NxN puzzles, the number of frontier nodes can grow exponentially.
    - **Slower for Deep Solutions**: If the solution is located at a significant depth, BFS might be slower than other methods that can luck into a solution faster (like DFS). However, the solution BFS finds will be optimal.
    - **Time Complexity**: The time complexity can be high for large puzzles because it examines a large number of nodes.

4. **Original Depth-First Search (DFS)**:

- **Exponential Time Complexity**: The nature of DFS can lead it to explore a vast number of states in the N-puzzle problem. The time complexity can be exponential in the worst case, making it impractical for larger N values.

- **Exponential Space Complexity**: DFS, in its basic form, can require storing an exponential number of states on the stack (or equivalent data structure), leading to potential memory issues.

- **Suboptimal Solutions**: Unlike certain other algorithms, DFS does not guarantee finding the shortest path to a solution for the N-puzzle. Instead, it can often produce a solution that is much longer than the optimal one.

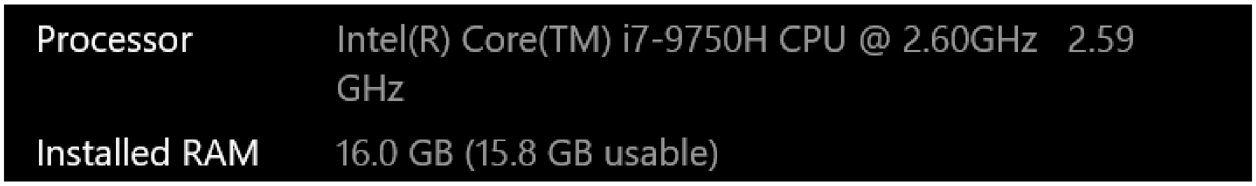
- **Possibility of Infinite Loops**: Without mechanisms to detect repeated states, DFS can get trapped in infinite loops, especially in problems where states can be revisited.

- **Depth Limitation**: Without any predefined depth limit, DFS could dive very deep into a particular branch of the search tree without finding a solution, while the solution could have been much shallower in another branch.

- **No Guarantees of Solution**: There's no assurance that DFS will find a solution in a reasonable amount of time. For larger instances of the N-puzzle, DFS could run for an impractically long time without reaching a solution.

- **Heavy Reliance on Initial Configuration**: The efficiency of DFS can vary significantly based on the initial configuration of the puzzle. For some configurations, it might find a solution relatively quickly, while for others, it might take an impractical amount of time.

Specifications:



Processor	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
Installed RAM	16.0 GB (15.8 GB usable)

Appendix:

```python
from copy import deepcopy
import random
from collections import deque


def get_puzzle_size():
    while True:
        try:
            n = int(input('Enter the N value for puzzle size (3<=N<=6): '))
            if 3 <= n <= 6:
                return n
            else:
                print('Please Enter a Correct N Value!')
        except ValueError:
            print('Enter a valid integer!')


def solvable(state):
    n = len(state)
    flat_state = [tile for row in state for tile in row]
    inv_count = 0
    for i in range(n*n):
        for j in range(i+1, n*n):
            if flat_state[j] != 0 and flat_state[i] != 0 and flat_state[i] >
flat_state[j]:
                inv_count += 1

    if n % 2 == 0:
        blank_row = [i for i, row in enumerate(state) if 0 in row][0]
        if blank_row % 2 == 0:
            return inv_count % 2 == 1
    return inv_count % 2 == 0


def generate_initial_state(n):
    while True:
        elements = list(range(1, n * n)) + [0]
        random.shuffle(elements)
        state = [elements[i * n:(i + 1) * n] for i in range(n)]
        if solvable(state):
            return state
```

```python
def generate_goal_state(n):
    elements = list(range(1, n * n)) + [0]
    return [elements[i * n:(i + 1) * n] for i in range(n)]


def GenerateChildren(state, last_move=None):
    for i in range(len(state)):
        for j in range(len(state)):
            if state[i][j] == 0:
                oldi, oldj = (i, j)

    directions = [(-1, 0, "U"), (1, 0, "D"), (0, -1, "L"), (0, 1, "R")]
    successors = []

    opposite_moves = {"U": "D", "D": "U", "L": "R", "R": "L"}
    if last_move and last_move in opposite_moves:
        directions = [d for d in directions if d[2]
                            != opposite_moves[last_move]]

    for d in directions:
        newi, newj = oldi + d[0], oldj + d[1]

        if 0 <= newi < len(state) and 0 <= newj < len(state):
            child = deepcopy(state)
            child[oldi][oldj], child[newi][newj] = child[newi][newj],
child[oldi][oldj]
            successors.append((child, d[2]))

    return successors
```

```python
def Bfs(initial_state, goal_state):
    visited = set()
    queue = deque([(initial_state, [])])
    max_states = 0

    while queue:
        max_states = max(max_states, len(queue))
        current_state, path = queue.popleft()

        if current_state == goal_state:
            return path, max_states

        current_state_str = str(current_state)
        if current_state_str not in visited:
            visited.add(current_state_str)

            last_move = path[-1] if path else None
            for neighbor, move in GenerateChildren(current_state, last_move):
                queue.append((neighbor, path + [move]))

    return [], max_states

# This the Original DFS algorithm (Testing was done on the Iterative Deepening
version as this version can get stuck in infinite loops and is heavily dependent
on the initial configuration of the puzzle)
# def Original_Dfs(initial_state, goal_state):
#     stack = [(initial_state, [])]
#     max_states = 0

#     while stack:
#         max_states = max(max_states, len(stack))
#         current_state, path = stack.pop()

#         if current_state == goal_state:
#             return path, max_states

#         last_move = path[-1] if path else None
#         for neighbor, move in GenerateChildren(current_state, last_move):
#             stack.append((neighbor, path + [move]))

#     return [], max_states
# This is the Iterative Deepening version of DFS
def DFS(initial_state, goal_state):
    depth = 0
    while True:
        max_states = 0
        stack = [(initial_state, [], 0)]
        solution = None

        while stack:
            current_state, path, current_depth = stack.pop()

            if current_state == goal_state:
                return path, max_states

            max_states = max(max_states, len(stack))

            if current_depth < depth:
                last_move = path[-1] if path else None
                for neighbor, move in GenerateChildren(current_state, last_move):
                    stack.append((neighbor, path + [move], current_depth + 1))

        depth += 1
```

```python
def Dfs_with_revisit_check(initial_state, goal_state):
    visited = set()
    stack = [(initial_state, [])]
    max_states = 0

    while stack:
        max_states = max(max_states, len(stack))
        current_state, path = stack.pop()

        if current_state == goal_state:
            del visited
            del stack
            return path, max_states

        current_state_str = str(current_state)
        if current_state_str not in visited:
            visited.add(current_state_str)

            last_move = path[-1] if path else None
            for neighbor, move in GenerateChildren(current_state, last_move):
                stack.append((neighbor, path + [move]))

    del visited
    del stack
    return [], max_states


def generate_report(algorithm_name, algorithm, n):

    splitter = "-" * 40 + "\n"

    with open("algorithm_report_DFS(Revisit check)4.txt", "w") as f:
        solution_depths = []
        states_stored = []

        for i in range(10):
            initial = generate_initial_state(n)
            final = generate_goal_state(n)

            solution_sequence, max_states_stored = algorithm(initial, final)
            solution_depth = len(solution_sequence)

            f.write(f"Run {i + 1}:\n")
            f.write("Initial State:\n")
            for row in initial:
                f.write(" ".join(map(str, row)) + "\n")
            f.write("\nFinal State:\n")
            for row in final:
                f.write(" ".join(map(str, row)) + "\n")
            f.write(f"\nAlgorithm Used: {algorithm_name}\n")
            f.write("Solution Sequence:\n")
            f.write(" --> ".join(solution_sequence) + "\n")
            f.write(f"\nSolution Depth: {solution_depth}\n")
            f.write(f"Max States Stored: {max_states_stored}\n\n")

            solution_depths.append(solution_depth)
            states_stored.append(max_states_stored)

            f.write(splitter)

        f.write("Descriptive Statistics:\n")
        f.write(f"Minimum Solution Depth: {min(solution_depths)}\n")
        f.write(f"Maximum Solution Depth: {max(solution_depths)}\n")
        f.write(
            f"Average Solution Depth: {sum(solution_depths) /
len(solution_depths):.2f}\n")
        f.write(f"Minimum States Stored: {min(states_stored)}\n")
        f.write(f"Maximum States Stored: {max(states_stored)}\n")
        f.write(
            f"Average States Stored: {sum(states_stored) /
len(states_stored):.2f}\n")


def main():

    n = get_puzzle_size()

    # # generate report used to get summary of the algorithm of choice

    # generate_report("DFS (With Revisit Check)", Dfs_with_revisit_check, n)

    Initial_State = generate_initial_state(n)

    Goal_State = generate_goal_state(n)

    print("\nInitial State:")
    for row in Initial_State:
        print(row)
    print("\nGoal State:")
    for row in Goal_State:
```

```python
        print(row)

    print("\nFinding a Path Solution . . .\n")

    # Uncomment the method you want to use and store it in a variable

    # method = Bfs
    method = DFS
    # method = Dfs_with_revisit_check

    path, max_states = method(Initial_State, Goal_State)

    if path:
        print(f"Solution found through {method.__name__}!")
        print(" --> ".join(path))
        print(f"Solution depth: {len(path)}")
        print(f"Maximum number of states concurrently stored: {max_states}")
    else:
        print(f"No solution found using {method.__name__}.")


if __name__ == "__main__":
    main()
```