200316059 İkram Celal KESKİN

2003160

2003160

**NQueens Search Algorithm Report**

**A – Development Environment**

We developed and tested the code in a Python 3.x environment. The operating system used was Windows 10. The code was written and executed in Visual Studio Code, a popular IDE for Python development. The CPU properties include an AMD Ryzen 3 3250U with Radeon Graphics processor.

**B – Problem Formulation**

The problem addressed in the code is the N-Queens problem, where N queens are placed on an N×N chessboard in such a way that no two queens threaten each other. The problem is formulated as a search problem using the SimpleAI library. Here are the key components:

* **State Specification**: The state is represented as a string of length N, where each character corresponds to the row position of the queen in the respective column.
* **Initial State**: The initial state is either manually entered by the user or generated randomly. The validity of the state is ensured through the \_is\_valid method.
* **Possible Actions**: The possible actions represent moving a queen to a different row within its column.
* **Transition Model**: The result method generates a new state by applying an action to the current state.
* **Goal Test**: The goal is reached when there are no attacking pairs of queens on the board.
* **Path Cost**: The cost of each step is constant, set to 1.

**C- Testing**

1. Experiment 1- N = 4, Initial State = "2323"

1.a) A\* Algorithm

Resulting State: “2413”

Resulting Path: [(None, '2323'), ((1, '4'), '2423'), ((2, '1'), '2413')]

Cost of Solution: 1

Viewer Statistics: {'max\_fringe\_size': 12, 'visited\_nodes': 2, 'iterations': 2}

1.b) Breadth-First Algorithm

Resulting State: “2413”

Resulting Path: [(None, '2323'), ((1, '4'), '2423'), ((2, '1'), '2413')]

Cost of Solution: 2

Viewer Statistics: {'max\_fringe\_size': 870, 'visited\_nodes': 80, 'iterations': 80}

1.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

1. Experiment 2- N = 4, Initial State = "4311"

2.a) A\* Algorithm

Resulting State: “2413”

Resulting Path: [(None, '4311'), ((3, '3'), '4313'), ((1, '4'), '4413'), ((0, '2'), '2413')]

Cost of Solution: 3

Viewer Statistics: {'max\_fringe\_size': 78, 'visited\_nodes': 8, 'iterations': 8}

2.b) Breadth-First Algorithm

Resulting State: “2413”

Resulting Path: [(None, '4311'), ((0, '2'), '2311'), ((1, '4'), '2411'), ((3, '3'), '2413')]

Cost of Solution: 3

Viewer Statistics: {'max\_fringe\_size': 4082, 'visited\_nodes': 372, 'iterations': 372}

2.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

1. Experiment 3- N = 4, Initial State = "3442"

3.a) A\* Algorithm

Resulting State: “3142”

Resulting Path: [(None, '3442'), ((1, '1'), '3142')]

Cost of Solution: 1

Viewer Statistics: {'max\_fringe\_size': 12, 'visited\_nodes': 2, 'iterations': 2}

3.b) Breadth-First Algorithm

Resulting State: “3142”

Resulting Path: [(None, '3442'), ((1, '1'), '3142')]

Cost of Solution: 1

Viewer Statistics: {'max\_fringe\_size': 45, 'visited\_nodes': 5, 'iterations': 5}

3.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

1. Experiment 4- N = 5, Initial State = "12345"

4.a) A\* Algorithm

Resulting State: “25314”

Resulting Path: [(None, '12345'), ((0, '2'), '22345'), ((1, '5'), '25345'), ((3, '1'), '25315'), ((4, '4'), '25314')]

Cost of Solution: 4

Viewer Statistics: {'max\_fringe\_size': 115, 'visited\_nodes': 7, 'iterations': 7}

4.b) Breadth-First Algorithm

Resulting State: “13524”

Resulting Path: [(None, '13154'), ((2, '5'), '13554'), ((3, '2'), '13524')]

Cost of Solution: 2

Viewer Statistics: {'max\_fringe\_size': 4827, 'visited\_nodes': 255, 'iterations': 255}

4.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

1. Experiment 5- N = 5, Initial State = "13154"

5.a) A\* Algorithm

Resulting State: “13524”

Resulting Path: [(None, '13154'), ((3, '2'), '13124'), ((2, '5'), '13524')]

Cost of Solution: 2

Viewer Statistics: {'max\_fringe\_size': 39, 'visited\_nodes': 3, 'iterations': 3}

5.b) Breadth-First Algorithm

Resulting State: “13524”

Resulting Path: [(None, '13154'), ((2, '5'), '13554'), ((3, '2'), '13524')]

Cost of Solution: 2

Viewer Statistics: {'max\_fringe\_size': 4827, 'visited\_nodes': 255, 'iterations': 255}

5.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

1. Experiment 1- N = 6, Initial State = "536142"

6.a) A\* Algorithm

Resulting State: “531642”

Resulting Path: [(None, '536142'), ((2, '1'), '531142'), ((3, '6'), '531642')]

Cost of Solution: 2

Viewer Statistics: {'max\_fringe\_size': 117, 'visited\_nodes': 5, 'iterations': 5}

6.b) Breadth-First Algorithm

Resulting State: “531642”

Resulting Path: [(None, '536142'), ((2, '1'), '531142'), ((3, '6'), '531642')]

Cost of Solution: 2

Viewer Statistics: {'max\_fringe\_size': 10151, 'visited\_nodes': 351, 'iterations': 351}

6.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

1. Experiment 7- N = 6, Initial State = "532512"

7.a) A\* Algorithm

Resulting State: “362514”

Resulting Path: [(None, '532512'), ((0, '6'), '632512'), ((5, '4'), '632514'), ((1, '6'), '662514'), ((0, '3'), '362514')]

Cost of Solution: 4

Viewer Statistics: {'max\_fringe\_size': 146, 'visited\_nodes': 6, 'iterations': 6}

7.b) Breadth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

7.c) Depth-First Algorithm

Resulting State: [state]

Resulting Path: [path]

Cost of Solution: [cost]

Viewer Statistics: [viewer\_stats]

**D – Discussion**

**Completeness and Optimality**

* **A**:\* A\* is complete and optimal, ensuring the optimal solution is found with an admissible heuristic. The heuristic used here is the count of attacking pairs.
* **Breadth-First**: Breadth-First is complete and optimal when all step costs are equal. It explores all nodes at the current depth before moving on to the next depth.
* **Depth-First**: Depth-First is not optimal because it might find a non-optimal solution if it reaches a goal state at a higher depth before finding the optimal solution at a shallower depth. It is also not complete in cases where there are infinite paths.

**Time and Space Complexity**

* **A\***:A\* has time and space complexity that depends on the heuristic's quality. In this case, the heuristic is admissible, ensuring optimality.
* **Breadth-First**: Breadth-First has higher time and space complexity compared to Depth-First. It explores all nodes at each depth level before moving on.
* **Depth-First**: Depth-First has lower memory requirements but may get stuck in deep paths. Its time complexity is heavily influenced by the depth of the solution.

**Conclusion**

In conclusion, the A\* algorithm tends to be the most reliable in finding optimal solutions, while Breadth-First is effective for relatively small problem sizes. Depth-First has its strengths in terms of memory efficiency but lacks optimality and completeness. The depth limit in DLS should be carefully chosen to balance completeness and memory usage.