CSE 3111 / CSE 3213

ARTIFICIAL INTELLIGENCE

FALL 2023

Programming Assignments Report

Furkan Bulut – 210316011

İsmet Eren Yurtcu – 210316028

Gonca Arabacı – 210316067

Submission Date: 5 January 2024

1 Development Environment

Windows 11 , Python 3.11.7 , PyCharm 2023.3.1 (Professional Edition) , jupyter==1.0.0 , simpleai==0.8.3

2 Problem Formulation

- *The state is represented as a tuple of length N, where each element corresponds to the row position of a queen in the respective column.*
- *The initial state is either provided by the user (manual input) or generated randomly. The state must be a valid configuration of N queens on the board.*
- *The possible actions at each step involve moving a queen to a different row within its column. This is represented as a tuple where the first element is the new state resulting from the move, and the second element is the cost of that action.*
- *The transition model is implemented in the actions and result methods. The actions method generates all possible moves for each queen, and the result method computes the new state when an action is performed.*
- *The goal test checks whether the current state represents a solution where no two queens threaten each other. This is done in the 'is_goal' method.*
- *The path cost is determined by the number of attacking pairs of queens in the current state. The lower the number of attacking pairs, the better the solution.*

The problem is given as a search issue, and different search techniques are used to find a reasonable queen location on the chessboard. The objective is to reach a state where there are no attacking pairs by moving queens to other rows to explore the state space. The path cost is determined by the number of attacking pairs, and the algorithms aim to minimize this cost to find an optimal solution.

3 Results

```
How do you want to set state?
1. Set state manually
2. Set state randomly
Enter state:

Testing hill_climbing:

Resulting state: (1, 1, 4, 2)
Resulting path: [(((1, 1, 4, 2), 1), (1, 1, 4, 2))]
Cost of the solution: 3
Time taken: 0:00:00.001117
```

```
How do you want to set state?
1. Set state manually
2. Set state randomly
Enter state:

Testing breadth_first:

Resulting state: (2, 4, 1, 3)
Resulting path: [(None, (1, 4, 3, 2)), (((2, 4, 3, 2), 1), (2, 4, 3, 2)), (((2, 4, 1, 2), 1), (2, 4, 1, 2)), (((2, 4, 1,
3), 1), (2, 4, 1, 3))]
Cost of the solution: 3
Time taken: 0:00:00.004785
```

```
How do you want to set state?
1. Set state manually
2. Set state randomly
Enter state:

Testing hill_climbing:

Resulting state: (1, 1, 4, 2)
Resulting path: [(((1, 1, 4, 2), 1), (1, 1, 4, 2))]
Cost of the solution: 3
Time taken: 0:00:00

Testing hill_climbing_restart_lambda:

Resulting state: (3, 1, 4, 2)
Resulting path: [(((3, 1, 4, 2), 1), (3, 1, 4, 2))]
Cost of the solution: 1
Time taken: 0:00:00.015905

Testing genetic:
Resulting state: [3, 1, 4, 2]
Resulting path: [('crossover', [3, 1, 4, 2])]
Cost of the solution: 0
Time taken: 0:00:05.000546
```

```
Testing breadth_first:

Resulting state: (2, 4, 1, 3)
Resulting path: [(None, (1, 4, 3, 2)), (((2, 4, 3, 2), 1), (2, 4, 3, 2)), (((2, 4, 1, 2), 1), (2, 4, 1, 2)), (((2, 4, 1,
 3), 1), (2, 4, 1, 3))]
Cost of the solution: 3
Time taken: 0:00:00.006192
```

```
Testing uniform_cost:

Resulting state: (2, 4, 1, 3)
Resulting path: [(None, (1, 4, 3, 2)), (((2, 4, 3, 2), 1), (2, 4, 3, 2)), (((2, 4, 1, 2), 1), (2, 4, 1, 2)), (((2, 4, 1,
 3), 1), (2, 4, 1, 3))]
Cost of the solution: 3
Time taken: 0:00:00.011984
```

```
Testing iterative_limited_depth_first:

Resulting state: (3, 1, 4, 2)
Resulting path: [(None, (1, 4, 3, 2)), (((1, 4, 4, 2), 1), (1, 4, 4, 2)), (((1, 1, 4, 2), 1), (1, 1, 4, 2)), (((3, 1, 4,
 2), 1), (3, 1, 4, 2))]
Cost of the solution: 3
Time taken: 0:00:00.001074
```

```
Testing greedy:

Resulting state: (2, 4, 1, 3)
Resulting path: [(None, (1, 4, 3, 2)), (((1, 4, 2, 2), 1), (1, 4, 2, 2)), (((1, 4, 2, 3), 1), (1, 4, 2, 3)), (((1, 4, 1,
 3), 1), (1, 4, 1, 3)), (((2, 4, 1, 3), 1), (2, 4, 1, 3))]
Cost of the solution: 4
Time taken: 0:00:00
```

```
Testing depth_first:

Resulting state: (3, 1, 4, 2)
Resulting path: [(None, (1, 4, 3, 2)), (((1, 4, 3, 4), 1), (1, 4, 3, 4)), (((1, 4, 4, 4), 1), (1, 4, 4, 4)), (((1, 4, 4,
 3), 1), (1, 4, 4, 3)), (((1, 4, 2, 3), 1), (1, 4, 2, 3)), (((1, 4, 2, 1), 1), (1, 4, 2, 1)), (((1, 4, 1, 1), 1), (1, 4,
 1, 1)), (((1, 3, 1, 1), 1), (1, 3, 1, 1)), (((1, 3, 1, 4), 1), (1, 3, 1, 4)), (((1, 3, 2, 4), 1), (1, 3, 2, 4)), (((1,
3, 2, 2), 1), (1, 3, 2, 2)), (((1, 3, 4, 2), 1), (1, 3, 4, 2)), (((1, 2, 4, 2), 1), (1, 2, 4, 2)), (((1, 2, 4, 1), 1),
(1, 2, 4, 1)), (((1, 2, 3, 1), 1), (1, 2, 3, 1)), (((1, 2, 3, 3), 1), (1, 2, 3, 3)), (((1, 2, 1, 3), 1), (1, 2, 1, 3)),
(((1, 1, 1, 3), 1), (1, 1, 1, 3)), (((1, 1, 1, 2), 1), (1, 1, 1, 2)), (((4, 1, 1, 2), 1), (4, 1, 1, 2)), (((4, 1, 1, 4),
 1), (4, 1, 1, 4)), (((4, 1, 4, 4), 1), (4, 1, 4, 4)), (((4, 1, 4, 3), 1), (4, 1, 4, 3)), (((4, 1, 3, 3), 1), (4, 1, 3,
3)), (((4, 1, 3, 1), 1), (4, 1, 3, 1)), (((4, 1, 2, 1), 1), (4, 1, 2, 1)), (((4, 3, 2, 1), 1), (4, 3, 2, 1)), (((4, 3,
2, 3), 1), (4, 3, 2, 3)), (((4, 3, 1, 3), 1), (4, 3, 1, 3)), (((4, 4, 1, 3), 1), (4, 4, 1, 3)), (((3, 4, 1, 3), 1), (3,
4, 1, 3)), (((3, 4, 1, 4), 1), (3, 4, 1, 4)), (((3, 4, 2, 4), 1), (3, 4, 2, 4)), (((3, 4, 2, 2), 1), (3, 4, 2, 2)),
(((3, 4, 4, 2), 1), (3, 4, 4, 2)), (((3, 1, 4, 2), 1), (3, 1, 4, 2))]
Cost of the solution: 35
Time taken: 0:00:00.009093
```

```
Testing limited_depth_first:

Resulting state: (3, 1, 4, 2)
Resulting path: [(None, (1, 4, 3, 2)), (((1, 4, 3, 4), 1), (1, 4, 3, 4)), (((1, 4, 4, 4), 1), (1, 4, 4, 4)), (((1, 4, 4,
 3), 1), (1, 4, 4, 3)), (((1, 4, 2, 3), 1), (1, 4, 2, 3)), (((1, 4, 2, 1), 1), (1, 4, 2, 1)), (((1, 4, 1, 1), 1), (1, 4,
 1, 1)), (((1, 3, 1, 1), 1), (1, 3, 1, 1)), (((1, 3, 1, 4), 1), (1, 3, 1, 4)), (((1, 3, 2, 4), 1), (1, 3, 2, 4)), (((1,
3, 2, 2), 1), (1, 3, 2, 2)), (((1, 3, 4, 2), 1), (1, 3, 4, 2)), (((1, 2, 4, 2), 1), (1, 2, 4, 2)), (((1, 2, 4, 1), 1),
(1, 2, 4, 1)), (((1, 2, 3, 1), 1), (1, 2, 3, 1)), (((1, 2, 3, 3), 1), (1, 2, 3, 3)), (((1, 2, 1, 3), 1), (1, 2, 1, 3)),
(((1, 1, 1, 3), 1), (1, 1, 1, 3)), (((1, 1, 1, 2), 1), (1, 1, 1, 2)), (((4, 1, 1, 2), 1), (4, 1, 1, 2)), (((4, 1, 1, 4),
 1), (4, 1, 1, 4)), (((4, 1, 4, 4), 1), (4, 1, 4, 4)), (((4, 1, 4, 3), 1), (4, 1, 4, 3)), (((4, 1, 3, 3), 1), (4, 1, 3,
3)), (((4, 1, 3, 1), 1), (4, 1, 3, 1)), (((4, 1, 2, 1), 1), (4, 1, 2, 1)), (((4, 3, 2, 1), 1), (4, 3, 2, 1)), (((4, 3,
2, 3), 1), (4, 3, 2, 3)), (((4, 3, 1, 3), 1), (4, 3, 1, 3)), (((4, 4, 1, 3), 1), (4, 4, 1, 3)), (((3, 4, 1, 3), 1), (3,
4, 1, 3)), (((3, 4, 1, 4), 1), (3, 4, 1, 4)), (((3, 4, 2, 4), 1), (3, 4, 2, 4)), (((3, 4, 2, 2), 1), (3, 4, 2, 2)),
(((3, 4, 4, 2), 1), (3, 4, 4, 2)), (((3, 1, 4, 2), 1), (3, 1, 4, 2))]
Cost of the solution: 35
Time taken: 0:00:00.008421
```

4 Discussion

Breadth-First Search (BFS):

Completeness: Breadth-First Search (BFS) is a complete algorithm, ensuring that it will find a solution if one exists by exploring the entire state space level by level.

Optimality: BFS guarantees optimality when the step costs are uniform. However, in cases where step costs vary, BFS may not always find the optimal solution.

Time Complexity: The time complexity of BFS is high due to exploring all nodes at a given depth before moving to the next level.

Space Complexity: BFS also has a high space complexity as it needs to store all nodes at a given level.

A*:

Completeness: A search is complete, as long as the heuristic used is admissible and consistent. It will find a solution if one exists.*

Optimality: A is optimal when the heuristic is admissible. It considers both the cost to reach the current node and the estimated cost to reach the goal.*

Time Complexity: A can be efficient, depending on the quality of the heuristic. The time complexity is influenced by the heuristic's accuracy.*

Space Complexity: The space complexity of A is generally higher than BFS due to the use of a priority queue for nodes based on their estimated costs.*

Depth-Limited Search (DLS):

Completeness: Depth-Limited Search (DLS) is not complete. If the depth limit is reached without finding a solution, DLS may terminate without realizing that a solution exists at a deeper level.

Optimality: DLS is not guaranteed to find the optimal solution. It may terminate at the depth limit, resulting in a suboptimal solution.

Time Complexity: DLS can have lower time complexity compared to BFS since it avoids exploring deeper levels.

Space Complexity: The space complexity of DLS is relatively low as it only needs to store nodes up to the specified depth limit.

Hill Climbing Search:

Completeness: Hill climbing is not a complete algorithm. It can get stuck in local optima, and there's no guarantee it will find the global optimum.

Optimality: Hill climbing is not guaranteed to find the optimal solution. Its effectiveness depends on the starting point and may converge to a local optimum.

Time Complexity: Hill climbing can be computationally efficient as it explores the immediate neighborhood of a state.

Space Complexity: The space complexity is low as hill climbing doesn't need to store a large number of nodes.

A uses a heuristic to effectively search the search space while finding a balance between completeness and optimality. On the other hand, the quality of the given heuristic determines how effective it is. A* is a strong candidate for a variety of search scenarios since it is especially appropriate when looking for an optimal solution in a wide range of problems. In addition, our experiences have shown that hill climbing search also demonstrates swift performance and may be well-suited for integration into various algorithms. The agility of hill climbing search makes it a viable option in algorithmic approaches, and its effectiveness can be observed in practical applications. Thus, alongside A*, hill climbing search proves to be a valuable tool, offering speed and applicability in certain contexts.*