Solving the 5-Queens Problem Using Uniform Cost Search (UCS)

Problem Description

The 5-Queens problem is a classical artificial intelligence problem. The goal is to place 5 queens on a 5×5 chessboard such that no two queens attack each other.

This means: - No two queens are in the same row - No two queens are in the same column - No two queens are on the same diagonal

State Representation

Each state is represented as a list of integers: - The index represents the column - The value represents the row of the queen in that column

Example:

```
[0, 2, 4]
```

- Queen at column 0, row 0
- Queen at column 1, row 2
- Queen at column 2, row 4

Only one queen is placed per column to automatically avoid column conflicts.

Initial State

The initial state is an empty list:

```
[]
```

No queens are placed yet.

Goal State

A state is considered a goal state when: - The length of the state is 5 - There are no conflicts between any two queens

Successor Function

From a given state, successor states are generated by: - Placing a queen in the next column - Trying all possible rows from 0 to 4 - Adding the queen only if the position is safe (no row or diagonal conflict)

Cost Function

Uniform Cost Search (UCS) assigns a cost to each move: - Each step has a cost of 1 - The total cost equals the number of queens placed so far

Since all steps have the same cost, UCS expands nodes in increasing order of depth.

Uniform Cost Search (UCS)

UCS always expands the node with the lowest cumulative cost first.

In the 5-Queens problem: - UCS guarantees finding a valid solution - Because all step costs are equal, UCS behaves similarly to Breadth-First Search (BFS)

Python Code

```python
import heapq

# Board size
N = 5

# Function to check if a queen can be safely placed in the given row
def is_safe(state, row):
    col = len(state)  # Next column to place
    for c, r in enumerate(state):
        # Check row and diagonal conflicts
        if r == row or abs(r - row) == abs(c - col):
            return False
    return True

# Uniform Cost Search function
def uniform_cost_search():
    # Priority Queue: (cost, state)
    pq = []
    heapq.heappush(pq, (0, []))  # Initial state with cost 0

    while pq:
        cost, state = heapq.heappop(pq)

        # Goal test: 5 queens placed safely
        if len(state) == N:
            return state

        # Generate successors
        for row in range(N):
            if is_safe(state, row):
                new_state = state + [row]
                new_cost = cost + 1
                heapq.heappush(pq, (new_cost, new_state))

    return None

# Run UCS
solution = uniform_cost_search()
print("Solution:", solution)
```

Example Output

```
Solution: [0, 2, 4, 1, 3]
```

Each number represents the row position of the queen in the corresponding column.

Conclusion

Uniform Cost Search successfully solves the 5-Queens problem by expanding states based on their cumulative cost. Although UCS guarantees an optimal solution, in this problem all actions have equal cost, making UCS functionally similar to BFS.

This approach demonstrates the application of classical search algorithms in constraint satisfaction problems.