Aim

To implement the **8 Queen's Problem** using the **Breadth-First Search (BFS) approach** in C++.

Theory

The **8 Queen's Problem** is a classic problem in Artificial Intelligence and combinatorial optimization. It involves placing **8 queens on a standard 8×8 chessboard** in such a way that **no two queens attack each other**.

- A queen in chess can move **horizontally, vertically, and diagonally**.
- Thus, the solution must ensure that no two queens share the same row, column, or diagonal.

Search Strategies in AI

- **Breadth-First Search (BFS):**
    - A **blind search algorithm** that explores nodes level by level.
    - Starts from an initial state and generates all possible states (successor nodes) before moving to the next depth level.
    - Ensures completeness (finds solution if one exists).
    - BFS is implemented using a **queue** data structure (FIFO).

Relevance to 8-Queen's Problem

- Each state of the board is considered as a **node**.
- BFS generates states by placing queens column by column.
- Valid states (partial placements without conflicts) are expanded.
- The algorithm terminates when a valid placement of **8 queens** is found.

Complexity

- **Time Complexity:** Exponential in nature, worst-case ~O(N!) for N queens. BFS increases branching factor.
- **Space Complexity:** O(b^d) where *b* is the branching factor and *d* is the depth (here, d = 8).

Algorithm

1. Start with an empty chessboard.
2. Initialize a queue with the empty state.
3. While the queue is not empty:
   a. Dequeue a state (partial placement of queens).
   b. If the state has 8 queens, output it as a solution.
   c. Else, generate successor states by placing a queen in the next column in all valid rows.
   d. Check if the placement is valid (no conflicts with already placed queens).
   e. Enqueue valid successor states into the queue.
4. Repeat until a valid complete solution is found.
5. Stop.

Code (C++ Implementation)

```
// C++ Program to solve 8-Queen's Problem using BFS
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// Function to check if placing a queen is safe
bool isSafe(vector<int> &state, int row, int col) {
    for (int c = 0; c < col; c++) {
        int r = state[c];
        // Check same row
        if (r == row) return false;
        // Check diagonals
        if (abs(r - row) == abs(c - col)) return false;
    }
    return true;
}

// BFS function to solve 8-Queens
void solve8QueensBFS() {
    queue<vector<int>> q;
    q.push(vector<int>()); // Start with empty board

    while (!q.empty()) {
        vector<int> state = q.front();
        q.pop();

        int col = state.size();

        // If solution found
        if (col == 8) {
```

```
            cout << "Solution Found:\n";
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    if (state[j] == i)
                        cout << " Q ";
                    else
                        cout << " . ";
                }
                cout << endl;
            }
            return;
        }

        // Generate next states
        `for (int row = 0; row < 8; row++) {
            if (isSafe(state, row, col)) {
                vector<int> newState = state;
                newState.push_back(row);
                q.push(newState);
            }
        }
    }
}`

int main() {
    cout << "8 Queen's Problem using BFS\n";
    solve8QueensBFS();
    return 0;
}
```

Input/Output Example

Sample Output:

```
8 Queen's Problem using BFS
Solution Found:
 Q  .  .  .  .  .  .  .
 .  .  .  Q  .  .  .  .
 .  .  .  .  .  .  Q  .
 .  Q  .  .  .  .  .  .
 .  .  .  .  Q  .  .  .
 .  .  Q  .  .  .  .  .
 .  .  .  .  .  Q  .  .
 .  .  .  .  .  .  .  Q
```

(Note: Output may vary depending on BFS exploration order. Multiple solutions exist.)

Analysis of Code and Algorithm

- **Logic Flow:**
  - BFS explores board states column by column.
  - Only valid placements (no conflicts) are enqueued.
  - Stops upon finding the first complete valid solution.
- **Time Complexity:**
  - Worst case grows exponentially, ~O(N!) for N queens.
  - BFS ensures completeness but may explore more states than DFS.
- **Space Complexity:**
  - Requires memory to store multiple board states in the queue.
  - O(b^d) where b ≈ N and d = N.
- **Efficiency:**
  - BFS guarantees finding a solution if one exists.
  - Compared to DFS, BFS consumes more memory but avoids getting stuck in deep invalid branches.

Real-Life Applications

- **Constraint Satisfaction Problems (CSPs):**
  Scheduling tasks, exam timetables, or resource allocation problems.
- **AI Search Problems:**
  Pathfinding in robotics, automated reasoning, and puzzle-solving.
- **Optimization Problems:**
  Placement of processors on chips, circuit design without interference.

Conclusion

In this experiment, we successfully implemented the **8 Queen's Problem** using the **Breadth-First Search (BFS) approach**. The solution demonstrates how BFS can systematically explore states level by level to find a valid arrangement of queens. This practical reinforced key AI concepts such as **state-space search, BFS, constraint satisfaction, and computational complexity**.