Name: Shabbar Adamjee  
Roll No.: PB57  
PRN: 1032221508

*AIES ASSIGNMENT 5*

*Hill Climbing Algorithm – nQueens*

*Code*

#include <iostream>

#include <set>

#include <utility>

#include <vector>

struct Queen {

  int id;

  int x;

  int y;

  Queen(int id, int x, int y) : id(id), x(x), y(y) {}

  // Overload the == operator

  bool operator==(const Queen &other) const {

    return id == other.id && x == other.x && y == other.y;

  }

};

bool attacking(Queen q1, Queen q2, int dimension) {

  // Same row?

  if (q1.x == q2.x)

    return true;

  // Same column?

  if (q1.y == q2.y)

    return true;

  // Diagonal check

  if (abs(q1.x - q2.x) == abs(q1.y - q2.y))

    return true;

  return false;

}

bool calculateHeuristic(int \*\*board, std::vector<Queen> &qVec, int nQueens) {

  // Set to keep track of queen pairs attacking each other

  std::set<std::pair<int, int>> qCombo;

  std::vector<Queen> copyVec = qVec;

  // Lambda function to add queen attack combos

  // And to check that the same combo is not inserted

  // E.g. if (1, 3) is there, it won't add (3, 1)

  auto addPair = [&](const Queen &q1, const Queen &q2) {

    int a = std::min(q1.id, q2.id);

    int b = std::max(q1.id, q2.id);

    qCombo.insert({a, b}); // Store the pair {smaller\_id, larger\_id}

  };

  int h = 999;

  int sameValueCounter = 0;

  int prevValue = -1;

  while (true) {

    for (auto q : qVec) {

      Queen temp = q;

      // Move q to new sqaure in same column

      // The check how many combos of attacking

      for (int i = 0; i < nQueens; i++) {

        // Reset the copyVec

        copyVec = qVec;

        // Don't check og row

        if (i == q.x)

          continue;

        temp.x = i;

        // Send every other queen to be checked

        for (auto each : qVec) {

          if (q == each) // Skip same queen

            continue;

          if (attacking(temp, each, qVec.size())) {

            addPair(temp, each);

          }

        }

        // Remove queen

        copyVec.erase(copyVec.begin() + q.y);

        // Perform checks between other queens

        for (auto one : copyVec) {

          for (auto second : copyVec) {

            if (one == second)

              continue;

            if (attacking(one, second, qVec.size())) {

              addPair(one, second);

            }

          }

        }

        // Set the heuristic value for that square

        h = qCombo.size();

        board[temp.x][temp.y] = h;

        qCombo.clear();

        // Then move same queen to next sqaure

      }

      // Once that queen is finished, move to next queen

    }

    // After all queens finished

    // Scan board for bestMove

    int bestValue = board[0][0];

    std::pair<int, int> bestMove = std::make\_pair(0, 0);

    for (int i = 0; i < nQueens; i++) {

      for (int j = 0; j < nQueens; j++) {

        if (board[j][i] < bestValue) {

          if (j == qVec[i].x && i == qVec[i].y)

            continue;

          bestValue = board[j][i];

          bestMove = std::make\_pair(j, i);

        }

      }

    }

    // Move queen of the column to bestMove's row

    qVec[bestMove.second].x = bestMove.first;

    std::cout << "Best Value: " << bestValue << std::endl;

    if (prevValue == bestValue)

      sameValueCounter++;

    else

      prevValue = bestValue;

    if (sameValueCounter >= 10)

      return false;

    // Once no queens are attacking each other, return

    if (bestValue == 0)

      return true;

  }

}

int main() {

  int nQueens;

  std::cout << "Enter number of queens: ";

  std::cin >> nQueens;

  // int nQueens = 8;

  std::vector<Queen> qVec;

  // Make a Queen object for each queen

  for (int i = 0; i < nQueens; i++) {

    Queen q{i, i, i};

    qVec.push\_back(q);

  }

  // Make 4 rows

  int \*\*matrix = new int \*[nQueens];

  // For each row, make 4 columns

  for (int i = 0; i < nQueens; i++) {

    matrix[i] = new int[nQueens];

  }

  // Set all squares to -1

  for (int i = 0; i < nQueens; i++) {

    for (int j = 0; j < nQueens; j++) {

      matrix[i][j] = -1;

    }

  }

  // Calculate initial heuristic

  // Set to keep track of queen pairs attacking each other

  std::set<std::pair<int, int>> qCombo;

  auto addPair = [&](const Queen &q1, const Queen &q2) {

    // Always store the smaller id first

    int a = std::min(q1.id, q2.id);

    int b = std::max(q1.id, q2.id);

    // Insert the pair directly

    qCombo.insert({a, b}); // Store the pair {smaller\_id, larger\_id}

  };

  int h = 0;

  for (auto one : qVec) {

    for (auto second : qVec) {

      if (one == second)

        continue;

      if (attacking(one, second, nQueens))

        addPair(one, second);

    }

  }

  h = qCombo.size();

  // Place queens, different columns, different rows

  // Give those sqaures their initial heuristics

  for (int i = 0; i < nQueens; i++) {

    matrix[i][i] = h;

  }

  bool solved = calculateHeuristic(matrix, qVec, nQueens);

  if (solved) {

    std::cout << "\nFinal queen positions:\n";

    for (auto q : qVec) {

      std::cout << "(" << q.x << ", " << q.y << ")\n";

    }

  } else {

    std::cout << "\nLocal maximum found.\n";

    std::cout << "Final queen positions:\n";

    for (auto q : qVec) {

      std::cout << "(" << q.x << ", " << q.y << ")\n";

    }

  }

  std::cout << std::endl;

}

*Output*

