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) {
```

```cpp
// 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;
```

```cpp
    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
```

```cpp
    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

```
AIES  .\a.exe
Enter number of queens: 4
Best Value: 4
Best Value: 2
Best Value: 2
Best Value: 1
Best Value: 1
Best Value: 0

Final queen positions:
(2, 0)
(0, 1)
(3, 2)
(1, 3)
```

```
AIES  .\a.exe
Enter number of queens: 5
Best Value: 7
Best Value: 4
Best Value: 3
Best Value: 2
Best Value: 2
Best Value: 0

Final queen positions:
(4, 0)
(2, 1)
(0, 2)
(3, 3)
(1, 4)
```

```
AIES  .\a.exe
Enter number of queens: 6
Best Value: 11
Best Value: 7
Best Value: 5
Best Value: 3
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2
Best Value: 2

Local maximum found.
Final queen positions:
(5, 0)
(0, 1)
(0, 2)
(3, 3)
(4, 4)
(2, 5)
```

Assignment No. 5

Title

Implementing a local search algorithm or genetic algorithm e.g. n-queens

FAQs

1. Explain hill climbing algorithm in detail with example.

Hill climbing is a local search algorithm that continuously moves towards the direction of increasing value (in maximization problems) or decreasing value (in minimization problems) to find the optimal solution. It is an iterative algorithm that starts with an arbitrary solution and then makes incremental changes to it.

E.g. Consider finding the max value of a function f(x) on a curve.

i. Start at some initial point.
ii. Check the slope or neighbouring points around it.
iii. Move uphill to the point where the function value increases.
iv. Continues this until no further uphill movement is possible.
v. If the algorithm reaches the peak, it stops.

2. Explain limitations of hill climbing and solutions to it.

Limitations

- Local maxima - The algorithm may get stuck in a local maximum and miss the global maximum.

- Plateaus - The algorithm may reach a flat area where no neighbouring solution appears to improve the objective function.

- Ridges - The algorithm may have difficulty climbing along steep or narrow ridges.

Solutions

- Random restarts - Restart the algorithm from different initial solutions to explore different parts of the solution space.

Page No.
Date :

- Simulated annealing - Allows occasional moves to worse solutions to escape local maxima, mimicking the process of annealing in metalwork.

- Tabu search - Maintain a list of previously visited solutions to avoid cycling back to them.

- Stochastic hill climbing - Instead of choosing the best move, pick a random move to explore a wider space.

3. Solve n-queens problem using local search algorithm.

i. Generate a random state where N queens are placed in different columns, one per row.
ii. Evaluate the state by counting the number of pairs of queens that are attacking each other.
iii. Move a queen to a different row (same column) and check if the new state has a lower heuristic value.
iv. If a better state is found, move to that state.
v. Repeat (iii) and (iv) until no improvement can be made.
vi. If the algorithm gets stuck in a local maximum, restart with a new random configuration.

18-10-24