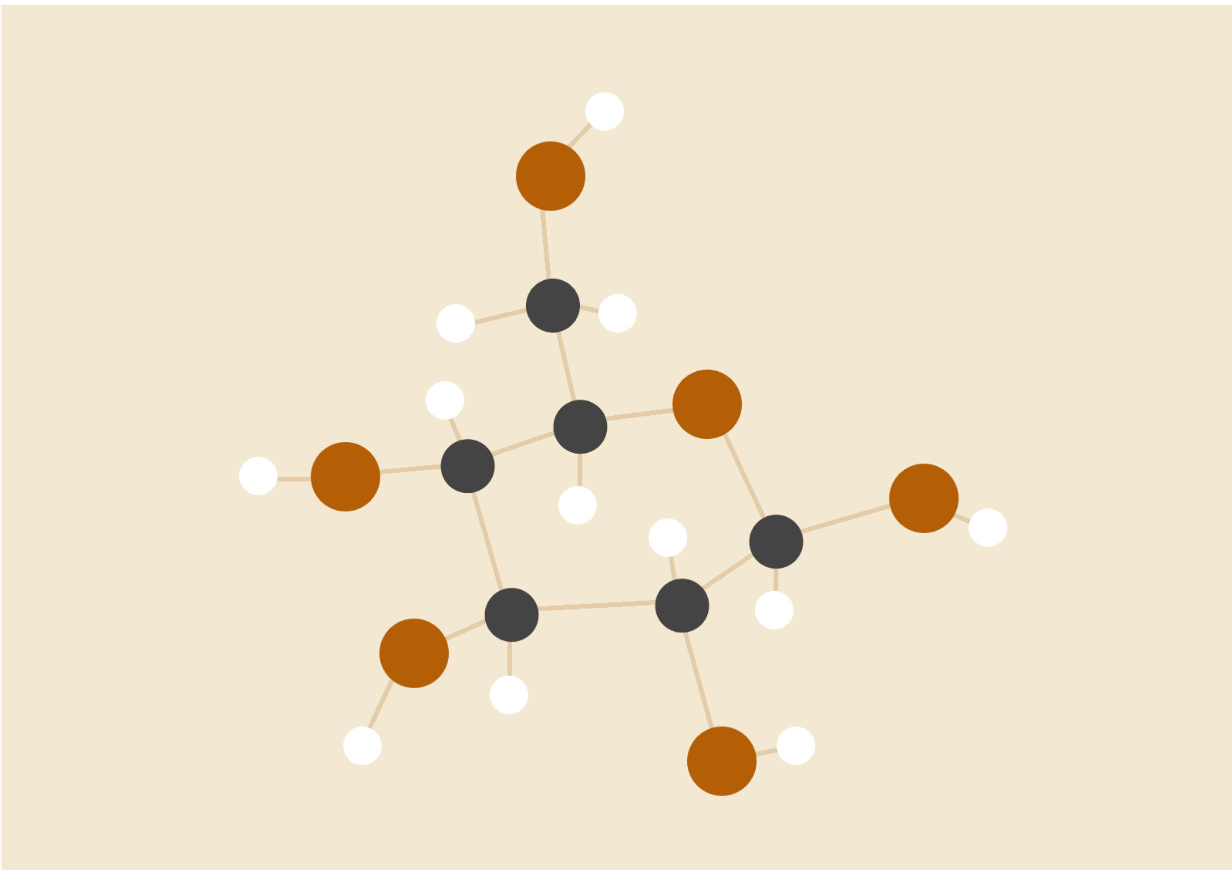
N-Queens Problem Report



Rupsa Mukhopadhyay

21052094

AI_CS:31

10.11.2023

INTRODUCTION

The N-Queen problem is a classic problem in computer science that involves placing N chess queens on an N x N chessboard so that no two queens attack each other. The problem can be solved using various algorithms, including backtracking, hill-climbing, and branch and bound. In this report, we will focus on the implementation of the hill-climbing algorithm in Python.

IMPLEMENTATION:

Implementation of the N-Queen problem using the hill-climbing algorithm can be divided into the following points:

1. Initialize a chessboard of size N and place N queens randomly on the board.
2. Define an evaluation function that calculates the number of attacks on the board by counting the number of diagonal, row, and column attacks.
3. Define functions to calculate the number of diagonal, row, and column attacks.
4. Define a function to generate all possible successors of the current board by moving one queen at a time to a different column in the same row.
5. Implement the hill-climbing algorithm by iteratively improving the current solution until it reaches a local maximum.
6. Define a modified version of the hill-climbing algorithm that allows for sideways moves to escape local maxima.
7. Initialize a chessboard of size N, run the hill-climbing algorithm to find a solution, and print the final chessboard configuration after the algorithm has been completed.

ASSUMPTIONS

Assumptions related to the N-Queen problem and the hill-climbing algorithm include:

1. The chessboard is a square grid of size N x N.
2. Each chessboard row must have exactly one queen.
3. The goal is to find an arrangement of N queens on the chessboard such that no queen can attack any other queen on the board.
4. The hill-climbing algorithm is a local search algorithm that starts with an initial solution and then iteratively improves the solution by making small changes.
5. The algorithm terminates when it reaches a local maximum, i.e., a solution that cannot be improved further.
6. The algorithm makes minimal assumptions about problem structure, which enables broader applicability than problem-specific algorithms.
7. The landscape of the problem is statistically isotropic, which means that it is symmetric in all directions.

CODELT

```python
import random

def initial_state(N):
    chessboard = [[0 for _ in range(N)] for _ in range(N)]
    queens = random.sample(range(N), N)
    for i in range(N):
        chessboard[i][queens[i]] = 1
    return chessboard

def evaluation_function(chessboard):
    count = 0
    N = len(chessboard)
    for i in range(N):
        for j in range(N):
            if chessboard[i][j] == 1:
                count += diagonal_attacks(chessboard, i, j)
                count += row_attacks(chessboard, i, j)
                count += column_attacks(chessboard, i, j)
    return count

def diagonal_attacks(chessboard, i, j):
    N = len(chessboard)
    count = 0
    for d in range(1, N):
        if i - d >= 0 and j - d >= 0 and chessboard[i-d][j-d] == 1:
            count += 1
        if i - d >= 0 and j + d < N and chessboard[i-d][j+d] == 1:
            count += 1
        if i + d < N and j - d >= 0 and chessboard[i+d][j-d] == 1:
            count += 1
        if i + d < N and j + d < N and chessboard[i+d][j+d] == 1:
            count += 1
    return count
```

```python

def row_attacks(chessboard, i, j):
    count = 0
    N = len(chessboard)
    for d in range(1, N):
        if i - d >= 0 and chessboard[i-d][j] == 1:
            count += 1
        if i + d < N and chessboard[i+d][j] == 1:
            count += 1
    return count

def column_attacks(chessboard, i, j):
    count = 0
    N = len(chessboard)
    for d in range(1, N):
        if j - d >= 0 and chessboard[i][j-d] == 1:
            count += 1
        if j + d < N and chessboard[i][j+d] == 1:
            count += 1
    return count

def generate_successors(chessboard):
    successors = []
    N = len(chessboard)
    for i in range(N):
        for j in range(N):
            if chessboard[i][j] == 1:
                new_board = [row.copy() for row in chessboard]
                new_board[i][j] = 0
                for k in range(N):
                    if k != j:
                        new_board[i][k] = 1
                successors.append(new_board)
    return successors

```

```python
def hill_climbing(chessboard):
    steps = 0
    current_board = chessboard
    current_eval = evaluation_function(current_board)
    while True:
        steps += 1
        successors = generate_successors(current_board)
        best_successor = None
        best_eval = float('inf')
        for successor in successors:
            eval = evaluation_function(successor)
            if eval < best_eval:
                best_eval = eval
                best_successor = successor
        if best_eval >= current_eval:
            return current_board, steps
        current_eval = best_eval
        current_board = best_successor

def hill_climbing_sideways_moves(chessboard):
    steps = 0
    current_board = chessboard
    current_eval = evaluation_function(current_board)
    sideways_moves = 0
    while True:
        steps += 1
        successors = generate_successors(current_board)
        best_successor = None
        best_eval = float('inf')
        for successor in successors:
            eval = evaluation_function(successor)
            if eval < best_eval:
                best_eval = eval
                best_successor = successor
        if best_eval >= current_eval:
            if sideways_moves > 10:
                return current_board, steps
            sideways_moves += 1
        else:
            sideways_moves = 0
        current_eval = best_eval
        current_board = best_successor

N = 8
chessboard = initial_state(N)
resulting_board, steps = hill_climbing_sideways_moves(chessboard)
print(f"The final chessboard after {steps} steps is:")
for row in resulting_board:
    print(row)
```

(TYPED)

```
import random


def initial_state(N):
    chessboard = [[0 for _ in range(N)] for _ in range(N)]
    queens = random.sample(range(N), N)
    for i in range(N):
        chessboard[i][queens[i]] = 1
    return chessboard


def evaluation_function(chessboard):
    count = 0
    N = len(chessboard)
    for i in range(N):
        for j in range(N):
            if chessboard[i][j] == 1:
                count += diagonal_attacks(chessboard, i, j)
                count += row_attacks(chessboard, i, j)
                count += column_attacks(chessboard, i, j)
    return count


def diagonal_attacks(chessboard, i, j):
    N = len(chessboard)
    count = 0
```

```
    for d in range(1, N):
        if i - d >= 0 and j - d >= 0 and chessboard[i-d][j-d] == 1:
            count += 1
        if i - d >= 0 and j + d < N and chessboard[i-d][j+d] == 1:
            count += 1
        if i + d < N and j - d >= 0 and chessboard[i+d][j-d] == 1:
            count += 1
        if i + d < N and j + d < N and chessboard[i+d][j+d] == 1:
            count += 1
    return count

def row_attacks(chessboard, i, j):
    count = 0
    N = len(chessboard)
    for d in range(1, N):
        if i - d >= 0 and chessboard[i-d][j] == 1:
            count += 1
        if i + d < N and chessboard[i+d][j] == 1:
            count += 1
    return count

def column_attacks(chessboard, i, j):
    count = 0
    N = len(chessboard)
```

```
    for d in range(1, N):
        if j - d >= 0 and chessboard[i][j-d] == 1:
            count += 1
        if j + d < N and chessboard[i][j+d] == 1:
            count += 1
    return count

def generate_successors(chessboard):
    successors = []
    N = len(chessboard)
    for i in range(N):
        for j in range(N):
            if chessboard[i][j] == 1:
                new_board = [row.copy() for row in chessboard]
                new_board[i][j] = 0
                for k in range(N):
                    if k != j:
                        new_board[i][k] = 1
                successors.append(new_board)
    return successors

def hill_climbing(chessboard):
    steps = 0
    current_board = chessboard
```

```
    current_eval = evaluation_function(current_board)
    while True:
        steps += 1
        successors = generate_successors(current_board)
        best_successor = None
        best_eval = float('inf')
        for successor in successors:
            eval = evaluation_function(successor)
            if eval < best_eval:
                best_eval = eval
                best_successor = successor
        if best_eval >= current_eval:
            return current_board, steps
        current_eval = best_eval
        current_board = best_successor

def hill_climbing_sideways_moves(chessboard):
    steps = 0
    current_board = chessboard
    current_eval = evaluation_function(current_board)
    sideways_moves = 0
    while True:
        steps += 1
        successors = generate_successors(current_board)
```

```
        best_successor = None
        best_eval = float('inf')
        for successor in successors:
            eval = evaluation_function(successor)
            if eval < best_eval:
                best_eval = eval
                best_successor = successor
        if best_eval >= current_eval:
            if sideways_moves > 10:
                return current_board, steps
            sideways_moves += 1
        else:
            sideways_moves = 0
        current_eval = best_eval
        current_board = best_successor


N = 8
chessboard = initial_state(N)
resulting_board, steps = hill_climbing_sideways_moves(chessboard)
print(f"The final chessboard after {steps} steps is:")
for row in resulting_board:
    print(row)
```

OUTPUT

The final chessboard after 14 steps is:

[1, 1, 0, 1, 1, 1, 1, 1]

[1, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 1, 0]

[0, 0, 0, 0, 0, 1, 0, 0]

[0, 0, 0, 1, 0, 0, 0, 0]

[0, 1, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 1, 0, 0, 0]

[0, 0, 1, 0, 0, 0, 0, 0]