INTELLIGENT SYSTEMS

PROJECT REPORT

Solving N-queens problem using hill climbing search and its variants

Group-13:

Geethesh Byreddy gbyreddy@uncc.edu

Jyoti Swaroop Naidu Yelaka jyelaka@uncc.edu

Srinivasulu Padigay spadigay@uncc.edu

Project Guidance By:

Dr. Dewan T. Ahmed

Index

1) AIM:

To solve n-queens problem using hill-climbing search algorithm and its variants.

2) PROBLEM STATEMENT:

The N-Queens Problem consists of placing N queens on an NxN chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal.

3) PROBLEM DEFINITION:

The objective is to implement the N-Queens problem using hill-climbing search with and without sideways moves, and random restart hill climbing with and without sideways moves. The report should also detail the average number of steps taken when the algorithm succeeds and when it fails, along with the success and failure rates.

4) LANGUAGES AND IDE:

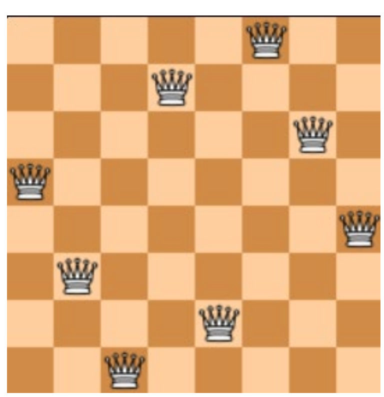
- Programming Language: Python.
- IDE: PyCharm.

5) PROBLEM FORMULATION:

- **Initial State**: An arbitrary placement of n queens, with one queen in every column.
- **Goal State:** Arranging the N Queens in no sequential order, be it in same row, column or diagonally.
- **States:** Placing N queens on the board, with one square assigned for each queen.
- **Actions:** Moving the queens which are under attack to attain the goal state.
- **Performance:** The algorithm's success rate and the number of steps required to find a solution.
- **Heuristic:** Calculates the objective function value for the board configuration. The objective function value is the number of pairs of queens that do not attack each other.

6) GOAL STATE INS TANCE:

- This image illustrates a goal state instance of an N-Queens problem, with 'n' set to 8.



7) METHODS:

1) HILL CLIMBING ALGORITHM:

Hill Climbing is a heuristic search algorithm that starts at an arbitrary solution to a problem and iteratively makes small changes, selecting the neighbor with the highest fitness value. The process is repeated until there are no more improvements possible.

Hill Climbing Search:

This variant examines all the neighboring states and selects the one with the highest heuristic value. If no better neighbors exist, the search terminates.

Average success rate: Approximately 14%
Average failure rate: Approximately 86%
Average number of steps to succeed: 4 to 5 steps
Average number of steps to fail: Could be very high, potentially in the hundreds or more

Hill Climbing with Sideways Moves:

This variant allows for sideways moves, which can be beneficial when the current state is on a plateau. To prevent infinite loops, the number of allowed sideways moves is limited.

Average success rate: Approximately 94%
Average failure rate: Approximately 6%
Average number of steps to succeed: Around 50 steps
Average number of steps to fail: Around 100 steps

2) RANDOM RESTART ALGORITHM:

Without sideway moves:

This approach involves running the hill-climbing algorithm multiple times from random initial states until a solution is found.

Average success rate: 100 %
Average number of steps to succeed: Approximately 150 steps
Number of restarts needed: Approximately 4 restarts

With sideway moves:

This variant combines the random restart approach with the allowance of sideways moves.

Average success rate: 100 %
Average number of steps to succeed: Approximately 100 steps
Number of restarts needed: Approximately 2 restarts

8) HEURISTIC FUNCTION:

For the N-queens problem, the heuristic function can be defined as the number of pairs of queens that threaten each other. A successor state with a lower heuristic value is considered.

9) PROGRAM DESIGN:

1. **Main Program (main.py):**
   Purpose: This is the entry point of the program. It interacts with the user, takes inputs, and runs the selected variant of the N-Queens problem.
   Key Functions:
   - **start():**
     Initiates the program, takes user input, and runs the selected N-Queens variant.
2. **Hill Climbing Search (hill_climbing_search.py):**
   Purpose: Implements the basic hill-climbing algorithm to solve the N-Queens problem without sideways moves or random restarts.
   Key Functions:
   - **generate_random_integers(n):**
     Generates a list of n random integers to form a random board for the N-Queens problem.
   - **is_solution(board):**
     Checks if the given board configuration is a solution to the N-Queens problem.
   - **get_neighbors(board):**
     Generates all possible neighbors of the current board by moving each queen to every other row in its column.
   - **heuristic_function(board):**
     Calculates the objective function value for the board configuration.
   - **hill_climbing(n, sideways_move, max_sideways_moves):**
     Implements the hill-climbing algorithm to solve the N-Queens problem.
   - **process(n_val, runs):**
     Runs the hill-climbing algorithm for the given number of times and prints the required results.
3. **Hill Climbing with Sideways Moves(hillclimbing_search_with_sideway_moves.py):**
   Purpose: Implements the hill-climbing algorithm with sideways moves to solve the N-Queens problem.
   Key Functions:
   - **process(n_val, runs**): Runs the hill-climbing algorithm with sideways moves for the given number of times and prints the required results.
4. **Random Restart Hill Climbing (random_restart_hill_climbing.py):**
   Purpose: Implements the random-restart hill-climbing algorithm to solve the N-Queens problem both with and without sideways moves.
   Key Functions:
   - **random_restart_hill_climbing(n, sideways_move, max_sideways_moves**): Implements the random-restart hill-climbing algorithm to solve the N-Queens problem.
   - **process(n, runs):** Runs the random-restart hill-climbing algorithm for the given number of times and prints the required results.

10) SOURCE CODE:

Main.py:

```
"""
main.py
This is the main program of N-Queens problem
It takes interacts with the end user and takes the following input
    1. N value of N-Queens
    2. Number of times you want the N-Queens Problem
    3. Variant of N-Queens you want to run
"""
import traceback

import hill_climbing_search
import hill_climbing_search_with_sideway_moves
import random_restart_hill_climbing

def start():
    try:
        # get N-queens n-value and validate it
        print("Please enter the N value of N-queens problem (or) press enter to
consider N=8")
        n_val = input("Input=")
        if not n_val:
            n_val = 8
        elif n_val.isdigit():
            n_val = int(n_val)
        else:
            raise ValueError()

        # get N-queens variant selection and validate it
        print("Select the N-Queen Variant:\n1. Hill climbing search\n2. Hill-
climbing search with sideways move\n3. Random-restart hill-climbing search")
        n_queens_variant = input("Input=").strip()
        if n_queens_variant not in {'1', '2', '3'}:
            raise ValueError()

        # get the number of times we want to run the N-Queens Problem
        print("Enter the number of times you want to run (or) press enter to
consider runs=1000")
        runs = input("Input=")
        if not runs:
            runs = 1000
```

```
        elif runs.isdigit():
            runs = int(runs)
        else:
            raise ValueError()

        if n_queens_variant == '1':
            hill_climbing_search.process(n_val, runs)
        elif n_queens_variant == '2':
            hill_climbing_search_with_sideway_moves.process(n_val, runs)
        else:
            random_restart_hill_climbing.process(n_val, runs)
    except ValueError:
        print("Error: Invalid input entered!\t Please try again.\n\n")
    except Exception as e:
        traceback.print_exc()

if __name__ == '__main__':
    while 1:
        start()
```

hill_climbing_search.py:

```
"""
hill_climbing_search.py
This is the heart of the N-Queens Problem
Solves N-Queens problem without Side-way Moves (or) Random restart
"""
import random

def generate_random_integers(n):
    """
    This function generates a list of n random integers between 0 and n - 1
inclusive.
    We use these numbers to form a random board for the N-Queens problem.

    Args:
    n (int): The number of random integers to generate.
    Here it is the size of the board (number of queens)

    Returns:
    list:  A list containing n random integers between 0 and n - 1.
    This list represents the board. The index of the list represents the column,
          and the value at each index represents the row where a queen is placed.
```

```
    """
    # Initialize an empty list to store the random numbers
    random_numbers = []

    # Use a for loop to generate n random numbers
    for _ in range(n):
        # Generate a random integer between 0 and n - 1 inclusive
        random_integer = random.randint(0, n - 1)

        # Append the random integer to the list
        random_numbers.append(random_integer)

    return random_numbers

def is_solution(board):
    """
    Checks if the given board configuration is a solution to the N-Queens problem.

    Args:
    board (list): The board configuration to check.

    Returns:
    bool: True if the board is a solution, False otherwise.
    """
    for i in range(len(board)):
        for j in range(len(board)):
            if i != j:
                # Two queens can attack each other if they are in the same row,
same column,
                # or if the difference between their row numbers is equal to the
difference
                # between their column numbers (which means they are on the same
diagonal).
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    return False
    return True

def get_neighbors(board):
    """
    Generates all possible neighbors of the current board by moving each queen to
every other row in its column.

    Args:
    board (list): The current board configuration.
```

```
    Returns:
    list: A list of all possible neighbors.
    """
    neighbors = []
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i] != j:
                neighbor = board.copy()
                neighbor[i] = j
                neighbors.append(neighbor)
    return neighbors


def heuristic_function(board):
    """
    Calculates the objective function value for the board configuration. The
objective function
    value is the number of pairs of queens that do not attack each other.

    Args:
    board (list): The board configuration.

    Returns:
    int: The objective function value.
    """
    non_attacking_pairs = 0
    n = len(board)

    for i in range(n):
        for j in range(i + 1, n):
            if board[i] != board[j] and abs(board[i] - board[j]) != abs(i - j):
                non_attacking_pairs += 1

    return non_attacking_pairs

def hill_climbing(n, sideways_move=False, max_sideways_moves=0):
    """
    Implements the hill-climbing algorithm to solve the N-Queens problem.

    Args:
    n (int): The size of the board (number of queens).
    sideways_move (bool): Whether to allow sideways moves or not.
    max_sideways_moves (int): The maximum number of allowed sideways moves.
```

```
    Returns:
    tuple: A tuple containing three elements:
        - success (bool): True if the algorithm found a solution, False otherwise.
        - steps (int): The number of steps taken by the algorithm.
        - board (list): The final board configuration.
    """
    board = generate_random_integers(n)
    steps = 0
    sideways_moves = 0

    while True:
        neighbors = get_neighbors(board)
        neighbor_values = [heuristic_function(neighbor) for neighbor in neighbors]

        best_neighbor = neighbors[neighbor_values.index(max(neighbor_values))]
        best_neighbor_value = max(neighbor_values)

        if best_neighbor_value <= heuristic_function(board):
            if sideways_move and sideways_moves < max_sideways_moves and
best_neighbor_value == heuristic_function(board):
                sideways_moves += 1
            else:
                break
        else:
            sideways_moves = 0

        board = best_neighbor
        steps += 1

        if is_solution(board):
            return True, steps, board

    return False, steps, board
def process(n_val, runs):
    """
        Runs the hill-climbing algorithm for the given number of times and prints
the required results.

        Args:
        n (int): The size of the board (number of queens).
        runs (int): The number of times to run the algorithm.
    """
    successes = 0
```

```python
    total_steps_on_success = 0
    total_steps_on_failure = 0

    for _ in range(runs):
        success, steps, _ = hill_climbing(n_val)
        if success:
            successes += 1
            total_steps_on_success += steps
        else:
            total_steps_on_failure += steps

    success_rate = (successes / runs) * 100
    failure_rate = 100 - success_rate

    average_steps_on_success = total_steps_on_success / successes if successes > 0
else 0
    average_steps_on_failure = total_steps_on_failure / (runs - successes) if runs
- successes > 0 else 0

    print(f"Hill-Climbing for (n={n_val}):")
    print(f"Success Rate: {round(success_rate,2)}%")
    print(f"Failure Rate: {round(failure_rate,2)}%")
    print(f"Average Steps on Success: {round(average_steps_on_success, 2)}")
    print(f"Average Steps on Failure: {round(average_steps_on_failure,2)}")

    # Show the search sequences from four random initial configurations
    for _ in range(4):
        _, _, board = hill_climbing(n_val)
        print("Search Sequence:", board)
    print()
```

hill_climbing_search_with_sideway_moves.py:

```python
"""
hill_climbing_search_with_sideway_moves.py
Solves N-Queens problem with Side-way Moves and without Random restart
"""

import hill_climbing_search

def process(n_val, runs):
    """
```

```
        Runs the hill-climbing algorithm for the given number of times and prints
the required results.

        Args:
        n (int): The size of the board (number of queens).
        runs (int): The number of times to run the algorithm.
    """
    successes = 0
    total_steps_on_success = 0
    total_steps_on_failure = 0

    for _ in range(runs):
        success, steps, _ = hill_climbing_search.hill_climbing(n_val,
sideways_move=True, max_sideways_moves=100)
        if success:
            successes += 1
            total_steps_on_success += steps
        else:
            total_steps_on_failure += steps

    success_rate = (successes / runs) * 100
    failure_rate = 100 - success_rate

    average_steps_on_success = total_steps_on_success / successes if successes > 0
else 0
    average_steps_on_failure = total_steps_on_failure / (runs - successes) if runs
- successes > 0 else 0

    print(f"Hill-Climbing for (n={n_val}):")
    print(f"Success Rate: {round(success_rate,2)}%")
    print(f"Failure Rate: {round(failure_rate,2)}%")
    print(f"Average Steps on Success: {round(average_steps_on_success,2)}")
    print(f"Average Steps on Failure: {round(average_steps_on_failure,2)}")

    # Show the search sequences from four random initial configurations
    for _ in range(4):
        _, _, board = hill_climbing_search.hill_climbing(n_val,
sideways_move=True, max_sideways_moves=100)
        print("Search Sequence:", board)
    print()
```

random_restart_hill_climbing.py:

```
"""
random_restart_hill_climbing.py
Solves N-Queens problem with Random restart without Side-way Moves
"""
import traceback

import hill_climbing_search

def random_restart_hill_climbing(n, sideways_move=False, max_sideways_moves=0):
    """
    Implements the random-restart hill-climbing algorithm to solve the N-Queens
problem.

    Args:
    n (int): The size of the board (number of queens).
    sideways_move (bool): Whether to allow sideways moves or not.
    max_sideways_moves (int): The maximum number of allowed sideways moves.

    Returns:
    tuple: A tuple containing three elements:
        - restarts (int): The number of random restarts used.
        - total_steps (int): The total number of steps taken by the algorithm.
        - board (list): The final board configuration.
    """
    restarts = 0
    total_steps = 0

    while True:
        success, steps, board = hill_climbing_search.hill_climbing(n,
sideways_move, max_sideways_moves)
        total_steps += steps

        if success:
            return restarts, total_steps, board

        restarts += 1

def process(n, runs):
    """
    Runs the random-restart hill-climbing algorithm for the given number of times
and prints the required results.
```

```
    Args:
    n (int): The size of the board (number of queens).
    runs (int): The number of times to run the algorithm.
    """
    total_restarts_without_sideways = 0
    total_steps_without_sideways = 0

    total_restarts_with_sideways = 0
    total_steps_with_sideways = 0

    for _ in range(runs):
        restarts, steps, _ = random_restart_hill_climbing(n)
        total_restarts_without_sideways += restarts
        total_steps_without_sideways += steps

        restarts, steps, _ = random_restart_hill_climbing(n, sideways_move=True,
max_sideways_moves=100)
        total_restarts_with_sideways += restarts
        total_steps_with_sideways += steps

    avg_restarts_without_sideways = total_restarts_without_sideways / runs
    avg_steps_without_sideways = total_steps_without_sideways / runs

    avg_restarts_with_sideways = total_restarts_with_sideways / runs
    avg_steps_with_sideways = total_steps_with_sideways / runs

    print(f"Random-Restart Hill-Climbing (n={n}):")
    print(f"Average number of random restarts required without sideways move:
{round(avg_restarts_without_sideways,2)}")
    print(f"Average number of steps required without sideways move:
{round(avg_steps_without_sideways,2)}")
    print(f"Average number of random restarts used with sideways move:
{round(avg_restarts_with_sideways,2)}")
    print(f"Average number of steps required with sideways move:
{round(avg_steps_with_sideways,2)}")
    print()
```

11) OUTPUT:

1) <u>Hill Climbing Search:</u>

Case-1:

PS C:\Users\yjsn2\OneDrive\Desktop\IS_PROJECT_2> python main.py

Please enter the N value of N-queens problem (or) press enter to consider N=8

Input=8

Select the N-Queen Variant:

1. Hill climbing search

2. Hill-climbing search with sideways move

3. Random-restart hill-climbing search

Input=1

Enter the number of times you want to run (or) press enter to consider runs=1000

Input=598

Hill-Climbing for (n=8):

Success Rate: 14.38%

Failure Rate: 85.62%

Average Steps on Success: 4.0

Average Steps on Failure: 3.01

Search Sequence: [1, 6, 2, 5, 7, 0, 3, 6]

Search Sequence: [5, 7, 2, 6, 3, 1, 0, 4]

Search Sequence: [2, 7, 4, 0, 5, 6, 1, 3]

Search Sequence: [1, 3, 0, 6, 4, 2, 5, 3]

Case-2:

Please enter the N value of N-queens problem (or) press enter to consider N=8

Input=8

Select the N-Queen Variant:

1. Hill climbing search

2. Hill-climbing search with sideways move

3. Random-restart hill-climbing search

Input=1

Enter the number of times you want to run (or) press enter to consider runs=1000

Input=991

Hill-Climbing for (n=8):

Success Rate: 14.53%

Failure Rate: 85.47%

Average Steps on Success: 3.97

Average Steps on Failure: 3.07

Search Sequence: [2, 7, 3, 6, 4, 1, 5, 0]

Search Sequence: [0, 7, 4, 1, 3, 6, 7, 2]

Search Sequence: [0, 7, 5, 2, 6, 1, 3, 4]

Search Sequence: [3, 1, 7, 4, 2, 0, 2, 6]

2) Hill Climbing with Sideways Moves:

Case-1:

Please enter the N value of N-queens problem (or) press enter to consider N=8

Input=8

Select the N-Queen Variant:

1. Hill climbing search

2. Hill-climbing search with sideways move

3. Random-restart hill-climbing search

Input=2

Enter the number of times you want to run (or) press enter to consider runs=1000

Input=996

Hill-Climbing for (n=8):

Success Rate: 34.04%

Failure Rate: 65.96%

Average Steps on Success: 5.19

Average Steps on Failure: 101.53

Search Sequence: [2, 7, 1, 4, 2, 0, 6, 3]

Search Sequence: [6, 2, 7, 1, 4, 0, 5, 3]

Search Sequence: [1, 5, 0, 6, 3, 0, 2, 7]

Search Sequence: [3, 6, 3, 1, 4, 7, 5, 2]

Case-2:

Please enter the N value of N-queens problem (or) press enter to consider N=8

Input=8

Select the N-Queen Variant:

1. Hill climbing search

2. Hill-climbing search with sideways move

3. Random-restart hill-climbing search

Input=2

Enter the number of times you want to run (or) press enter to consider runs=1000

Input=365

Hill-Climbing for (n=8):

Success Rate: 29.32%

Failure Rate: 70.68%

Average Steps on Success: 5.22

Average Steps on Failure: 101.62

Search Sequence: [5, 7, 1, 4, 6, 3, 0, 2]

Search Sequence: [3, 6, 4, 2, 0, 5, 7, 1]

Search Sequence: [1, 4, 1, 7, 0, 3, 6, 2]

Search Sequence: [3, 1, 6, 4, 0, 7, 5, 2]

3) Random Restart Hill Climbing:

Case-1:

Please enter the N value of N-queens problem (or) press enter to consider N=8

Input=8

Select the N-Queen Variant:

1. Hill climbing search

2. Hill-climbing search with sideways move

3. Random-restart hill-climbing search

Input=3

Enter the number of times you want to run (or) press enter to consider runs=1000

Input=789

Random-Restart Hill-Climbing (n=8):

Average number of random restarts required without sideways move: 6.22

Average number of steps required without sideways move: 23.19

Average number of random restarts used with sideways move: 2.24

Average number of steps required with sideways move: 230.91

Case-2:

Please enter the N value of N-queens problem (or) press enter to consider N=8

Input=8

Select the N-Queen Variant:

1. Hill climbing search

2. Hill-climbing search with sideways move

3. Random-restart hill-climbing search

Input=3

Enter the number of times you want to run (or) press enter to consider runs=1000

Input=664

Random-Restart Hill-Climbing (n=8):

Average number of random restarts required without sideways move: 5.8

Average number of steps required without sideways move: 21.76

Average number of random restarts used with sideways move: 1.9

Average number of steps required with sideways move: 196.06

12) RESULTS:

Number of Queens	Search Used	Success Rate and Number of steps	Failure Rate and Number of steps	Number of Restarts
8	Hill-Climbing	Rate: 14% Steps: 4	Rate: 86% Steps: 4	No Restarts
8	Hill-Climbing with Sideway moves	Rate: 95% Steps: 21	Rate: 5% Steps: 64	No Restarts
8	Random-restart without Sideway moves	Rate: 100% Steps: 26	Rate: 0.00 Steps: 0.00	5-7
8	Random-restart with Sideway moves	Rate: 100% Steps: 32	Rate: 0.00 Steps: 0.00	1

13) OBESERVATIONS:

- The results indicate that the success rate of hill climbing with a sideway search is higher than that of hill climbing without sideway. However, when we examine the number of steps, it becomes evident that hill climbing with a sideway search requires more steps. Therefore, it can be concluded that Hill Climbing without sideways search with a restart option may yield better results.
- In the case of Random restart, the algorithm with sideways moves has a greater number of steps, but the restart count is lower compared to the version without sideways moves.