Term Project: n-Queens CSP Design Document

Programming Language: Python

CP468- Artificial Intelligence

Dr. Ilias Kotsireas

December 6th, 2024

GroupID: 4

Group Members:

Salam Al-Rifaie (210580330) - alri0330@mylaurier.ca

Aseel Adeinat (169040674) - adei0674@mylaurier.ca

Bilal Asad (210847340) - asad7340@mylaurier.ca

Noah Haddad (210428500) - noahhaddad56@gmail.com

Sevde Korkut (169033770) - kork3770@mylaurier.ca

Zach Reid(210334450) - reid4450@mylaurier.ca

Elza Jung (000006974) - jung1603@mylaurier.ca

Gabriel Labayan (210769570) - laba9570@mylaurier.ca

Aidan Gibson (210233290) - gibs3290@mylaurier.ca

Huda Cheema (169027250) - chee7250@mylaurier.ca

Table of Contents

1. Project Description: N-Queens as a CSP

The n-Queens problem can be represented as a Constraint Satisfaction Problem (CSP) where the objective is to place $N \times N$ queens on a $N \times N$ chessboard such that no two queens can attack each other.

1.1 Problem Definition

Variables- Each row on the $N \times N$ chessboard represents a variable, and the column position of a queen on that row is the value that the variable can be assigned. For a $N \times N$ chessboard, there are N variables X_1 - X_N where each variable X_i corresponds to the queen's column in the row i.

Domains- The domain of each variable X_i is the set of possible positions for a queen on her row, that is, the set of column indices $\{1, 2, \ldots, N\}$.

Constraints- The constraints of the CSP n-queens problem ensure that no two queens are attacking each other, meaning, no two queens exist in the same row, column, or diagonally.

- Row constraints: satisfied inherently since every queen is placed in her own row (each variable represents a different row).
- Column constraints- for any two variables X_i and X_j, $X_i \neq X_j$
- Diagonal constraints- For any two variables X_i and X_j, the following must hold:

  $$|X_i - X_j| \neq |i - j|$$

  There are two types of diagonals:
  Major Diagonals: Diagonal from top-left to bottom-right
  Minor Diagonals: Diagonal from top-right to bottom-left

Goal State- A solution to the n-queens CSP is an assignment of column indices to all queens on the board (N variables X_1 - X_N) such that all constraints are satisfied—no pair of queens are threatening each other.

Worst case scenario of conflicts-
In the worst case, each queen would conflict with other queens diagonally. In other words, every queen would conflict with *n - 1* queens (all other queens except itself).

The initial total conflicts can be calculated as *n*(n-1)* maximum pairwise conflicts where there are n queens and each queen conflicts with *n - 1* others. This considers each conflict twice; to remove the overcount, we can divide our calculation by half: *(n*(n-1))/2*

The worst case of conflicts provides the maximum amount of conflicts, which is useful as a metric to test the Min-Conflicts algorithm.

1.2 Project Decisions

We have made several design choices to make our program more efficient, scalable, and clear. We implemented the MIN-CONFLICTS algorithm in Python, a high-level language

that is suited for problem-solving because of its simplicity and access to powerful libraries. We used the data structure lists to represent the positions of queens on the board and sets to track conflicts in rows and diagonals, making it more efficient when checking for conflicts. We also used the `random.shuffle` to initialize the queens' positions randomly, which is essential for the MIN-CONFLICTS algorithm.

To verify our solution, we implemented the `is_valid` function, checking if there are zero conflicts in the rows or diagonals. To visualize the problem, we chose to create scatter plots that represent the positions of queens on an N×N grid. Initially, we tried different formats for visualizations such as heatmaps and the chess board itself. We selected the scatter plot format for our visualization because it is easy to understand and visually appealing, even for larger N.

2. Implementation

The n-Queens CSP has been implemented using a *list* data structure. The queen positions are randomized initially.

Code section 1: Puzzle Initialization

```
def initialize_puzzle(n):
    """
    Initializes a random position for the queens with no initial row
conflicts.
    """
    queens = list(range(n))  # Place one queen per row
    random.shuffle(queens)  # Shuffle to randomize initial positions
    return queens
```

2.1 The MIN-CONFLICTS Heuristic

The Min-Conflicts heuristic selects the next value for a variable in a local search algorithm based on the number of violated constraints. Min-conflicts heuristic selects the value that results in the fewest constraints violated. In the context of the n-queens problem, the Min-Conflicts algorithm chooses the queen position on the board such that the fewest queens attack each other. Similar to hill-climbing, for min-conflict h(n) = total number of violated constraints.

The min-conflict function uses a for loop to check if the current configuration of queens is a valid solution.

```
for step in range(max_steps):
        if sum(conflicts) == 0:  # Check if solved
            # Confirmation: print the number of conflicts at the last
step
            print(f"Final number of conflicts: {sum(conflicts)}")
            return queens, step
```

Code section 1: MIN-CONFLICTS Function

```
def min_conflict(N, max_steps):
    """
    Implementation of the min_conflict algorithm to solve the n-queens
problem.
    """
    queens = initialize_puzzle(N)
    conflicts, rows, maj_diag, min_diag = build_conflicts(queens, N)

    # Print the number of conflicts at the first step
    print(f"Initial number of conflicts: {sum(conflicts)}")

    for step in range(max_steps):
        if sum(conflicts) == 0:  # Check if solved
            # Print the number of conflicts at the last step
            print(f"Final number of conflicts: {sum(conflicts)}")
            return queens, step

        # Pick a random conflicting column
        col = pick_position(conflicts, lambda x: x > 0)

        # Calculate row conflicts for the selected column
        min_conf = float('inf')
        best_rows = []
        for row in range(N):
            conflict_count = (
                rows[row] +
                maj_diag[row - col + N - 1] +
                min_diag[row + col] -
                3  # Subtract current queen's own conflicts
            )
            if conflict_count < min_conf:
                min_conf = conflict_count
                best_rows = [row]
            elif conflict_count == min_conf:
                best_rows.append(row)

        # Choose one of the best rows randomly
```

```
        new_row = random.choice(best_rows)

        # Update conflicts and move the queen
        update_conflicts(queens, conflicts, rows, maj_diag, min_diag,
col, new_row, N)

    return queens, max_steps  # Return final state if max_steps reached
```

Min-conflicts calls the pick_position function which returns a random position in the array based on a condition.

Code section 2: pick_position Function

```
def pick_position(arr, condition):
    """
    Returns a random position in the array based on a given condition.
    """
    candidates = [i for i, val in enumerate(arr) if condition(val)]
    return random.choice(candidates)
```

The min_conflicts function calls the build_conflicts function, and it initializes the conflicts.

Code section 3: build_conflicts Function

```
def build_conflicts(queens, N):
    """
    Initializes the conflict counters for rows and diagonals.
    """
    rows = [0] * N
    maj_diag = [0] * (2 * N - 1)
    min_diag = [0] * (2 * N - 1)
    conflicts = [0] * N

    for col, row in enumerate(queens):
        rows[row] += 1
        maj_diag[row - col + N - 1] += 1
        min_diag[row + col] += 1

    for col, row in enumerate(queens):
        conflicts[col] = (
            rows[row] - 1 +
            maj_diag[row - col + N - 1] - 1 +
            min_diag[row + col] - 1
        )
```

```
    return conflicts, rows, maj_diag, min_diag
```

The update_conflicts function is used by the MIN-CONFLICTS algorithm as it adjusts conflict counts whenever a queen is moved to a new position. First, it checks if the queen's new position is different from its current one; if not, it skips any updates. When a move occurs, the function reduces conflict counts for the queen's old row and diagonals, removing its influence from the previous position. It then increases the counts for the new row and diagonals, reflecting the updated placement. The column where the queen was moved is recalculated to ensure its conflicts are accurate, and adjustments are made for other columns if their conflicts are affected by the move. Finally, the queen's position is updated in the list. This method focuses on updating only the impacted areas of the board, making it efficient for large chess boards while ensuring the conflict data remains consistent and accurate throughout the algorithm.

Code section 4: update_conflicts Function

```
def update_conflicts(queens, conflicts, rows, maj_diag, min_diag, col,
new_row, N):
    """
    Incrementally updates conflicts when a queen is moved.
    """
    old_row = queens[col]
    if old_row == new_row:
        return  # No need to update if the position is unchanged

    # Remove the old position's conflicts
    rows[old_row] -= 1
    maj_diag[old_row - col + N - 1] -= 1
    min_diag[old_row + col] -= 1

    # Add the new position's conflicts
    rows[new_row] += 1
    maj_diag[new_row - col + N - 1] += 1
    min_diag[new_row + col] += 1

    # Update the specific column's conflict count
    conflicts[col] = (
        rows[new_row] - 1 +
        maj_diag[new_row - col + N - 1] - 1 +
        min_diag[new_row + col] - 1
    )

    # Update conflicts for other columns affected by the move
    for col2, row2 in enumerate(queens):
        if col2 == col:
```

```
            continue
        if row2 == old_row or abs(row2 - old_row) == abs(col2 - col):
            conflicts[col2] -= 1
        if row2 == new_row or abs(row2 - new_row) == abs(col2 - col):
            conflicts[col2] += 1

    # Update the queen's position
    queens[col] = new_row
```

2.2 Solution Verification

The is_valid function takes a configuration of n queens as input and decides whether or not this is a solution to the n-queens problem.

Code section 5: is_valid Function

```
def is_valid(assignment):
    """
    Validates whether the assignment has no conflicts.
    """
    rows = set()
    major_diag = set()
    minor_diag = set()
    for col, row in enumerate(assignment):
        if row in rows or (row - col) in major_diag or (row + col) in
minor_diag:
            return False
        rows.add(row)
        major_diag.add(row - col)
        minor_diag.add(row + col)
    return True
```

2.3 Error Handling

The error handling of the code is done in the Main function via a for-loop statement. It ensures that invalid inputs or unsolvable cases for the N-queens problem are managed easily and provides feedback to the user. It covers several scenarios to prevent runtime errors and any invalid input:

a) **Non-integer Input:** ValueError is raised if the user provides a value for N that isn't an integer. The program catches the error and prints a statement to the user to enter a valid value, preventing the program from crashing.

b) **Non-positive Integer Values**: The program identifies any integer that is zero or negative, displaying an error message while prompting the user to input a positive number since it exits early. This guarantees that only acceptable chess board sizes are handled, as a chessboard with zero or negative dimensions is not significant.

c) **Unsolvable board size:** Certain chess board sizes, specifically N=2 and N=3, have no solution to the n-Queens problem due to the small size of the board that won't satisfy the problem constraints. For these cases, the program outputs a message informing the user that this problem doesn't have any solution.

d) **Failure to Solve:** Even with valid inputs, there is a chance that the algorithm may not solve the problem due to an unanticipated issue. To handle this, the program uses the is_valid function to verify if the solution found by the min_conflict algorithm satisfies all constraints. If it does not, the program outputs: "N-Queens not solved." and exits, ensuring the user is informed about the failure without proceeding with invalid results.

For general cases with valid inputs and solvable board sizes: The program executes the MIN-CONFLICTS algorithm and provides detailed feedback to the user. This includes a start notification indicating that the algorithm has begun, details of the solution process such as the number of steps taken, and the time required to find a solution for the given N value. It also visualizes the solution using a scatter plot and confirms that the solution has been saved to a text file, as discussed earlier in the document.

Code section 6: Main Function

```
def main(N):
    """
    Main function to solve the N-Queens problem and visualize the final
solution.
    """
    try:
        N = int(N)
        if N <= 0:
            print("Error: N must be a positive integer greater than
0.")
            return
        if N in [2, 3]:
            print(f"Error: N = {N} has no solutions. Please try a
different value.")
            return

        max_steps = min(10 * N, 10**6)
        print("\nStarting MIN-CONFLICTS algorithm...")
        start_time = time.perf_counter()
        queens, steps = min_conflict(N, max_steps)
        end_time = time.perf_counter()
        elapsed_time = end_time - start_time

        if not is_valid(queens):
            print("N-Queens not solved.")
            return
```

```
        print(f"N-Queens solved for N = {N} in {steps} steps.")
        print(f"Time taken = {elapsed_time:.4f}")
        visualize_solution(queens, N)

        # output to text file
        save_solution(queens, N)

    except ValueError:
        print("Error: Please enter a valid integer value for N.")
```

2.4 Visualization

The visualize_solution function creates a scatter plot to display the solution of the N-Queens problem. It places queens (red dots) on a grid where the x-axis represents columns, and the y-axis represents rows.

Code section 7: visualize_solution Function

```
def visualize_solution(queens, N):
    """
    Visualizes the final N-Queens solution using a scatter plot.
    """
    cols = range(N)  # Columns
    rows = queens    # Corresponding rows

    plt.figure(figsize=(10, 10))
    plt.scatter(cols, rows, c='red', s=20 if N <= 50 else 2)  # Adjust
size for readability
    plt.title(f"N-Queens Solution for N = {N}", fontsize=14)
    plt.xlabel("Columns")
    plt.ylabel("Rows")
    plt.xlim(-1, N)
    plt.ylim(-1, N)
    plt.gca().invert_yaxis()  # Match chessboard orientation
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.show()
```

As a second solution format, the save_solution function prints the solved queen row and column positions to a text file.

Code section 8: save_solution Function

```
def save_solution(queens, N):
    """
    Saves the N-Queens solution to a text file.
    parameters-
```

```
    queens: List of queen positions (row for each column).
    N: Size of the chessboard (NxN).
    """
    with open(f"nqueens_solution_{N}.txt", "w") as f:
        for col, row in enumerate(queens):
            f.write(f"Column {col + 1}, Row {row + 1}\n")
    print(f"Solution saved to nqueens_solution_{N}.txt")
```

3. Installation and Execution

To install and run this code, you simply need the term_project.py file downloaded in your files. Then, open a command terminal and run the following command:
python <filepath_to_termproject.py> <N as int>

N should be an integer representing the size of the puzzle. For best results, fill in the file path by right-clicking term_project.py and clicking “Copy as Path”.

4. Test results: Graphical Representation & Text File

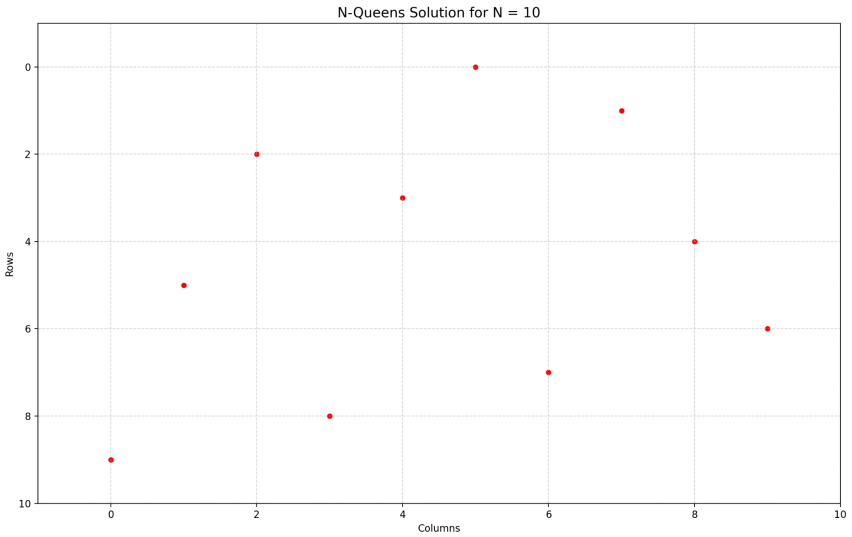
The scatter plot visualizes the N-queens problem for the given N. Each red dot represents a queen's position on the chessboard, where the x-axis corresponds to the column and the y-axis to the row. You can hover over a red dot which will reveal the exact row and column of the queen, showing their exact positions and ensuring no two queens threaten each other.

Figure 1: Example of Text File Output for n=10

```
Column 1, Row 4
Column 2, Row 2
Column 3, Row 8
Column 4, Row 5
Column 5, Row 9
Column 6, Row 1
Column 7, Row 6
Column 8, Row 10
Column 9, Row 3
Column 10, Row 7
```

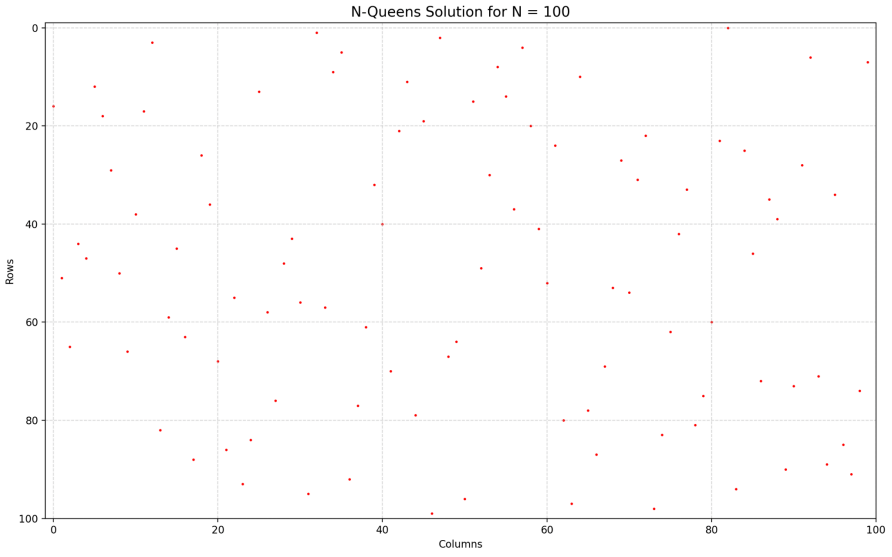
For N = 10
Output:
Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 34
Final number of conflicts: 0
N-Queens solved for N = 10 in 58 steps.
Time taken = 0.0005 => 0 minutes and 0.001 seconds
Solution saved to nqueens_solution_10.txt

Figure 2: Visual Output for n=10



For N = 100
Output:
Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 138
Final number of conflicts: 0
N-Queens solved for N = 100 in 144 steps.
Time taken = 0.0055 => 0 minutes and 0.01 seconds
Solution saved to nqueens_solution_100.txt

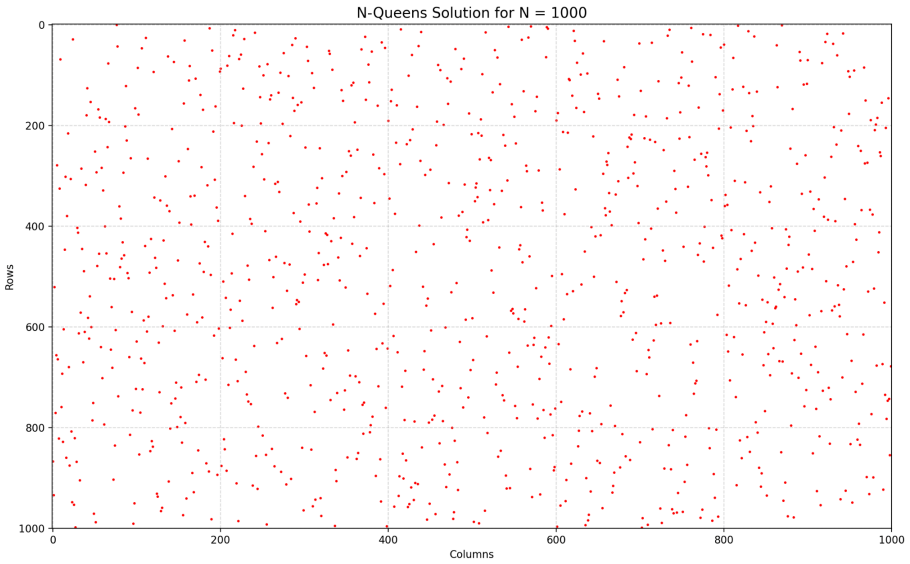
Figure 3: Visual Output for n=100



For N = 10000
Output:
Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 1374

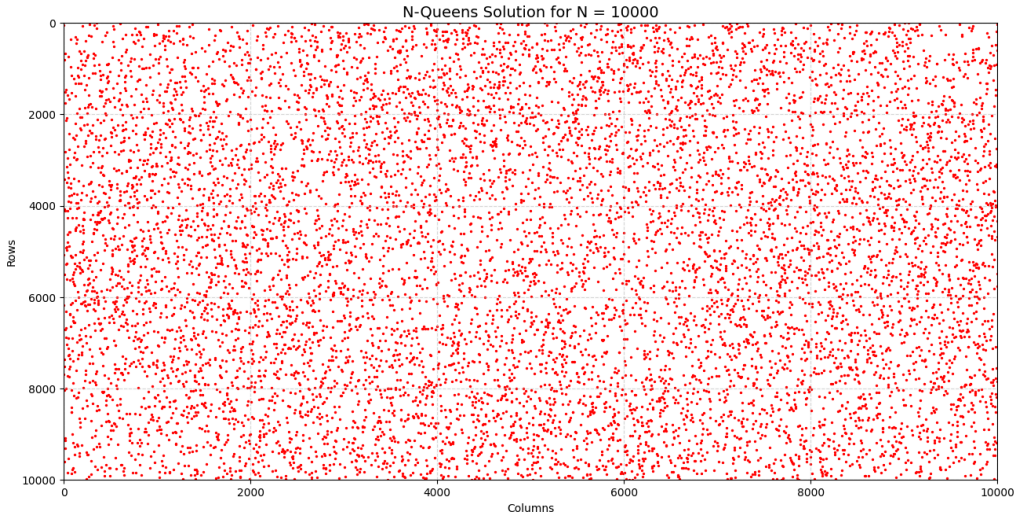
Final number of conflicts: 0
N-Queens solved for N = 1000 in 547 steps.
Time taken = 0.2222 => 0 minutes and 0.22 seconds
Solution saved to nqueens_solution_1000.txt

Figure 2: Visual Output for n=1000



For N = 10,000
Output:
Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 13460
Final number of conflicts: 0
N-Queens solved for N = 10000 in 4838 steps.
Time taken = 20.8412
Solution saved to nqueens_solution_10000.txt

Figure 4: Visual Output for n=10,000



For N = 100,000
Output:
Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 133184
Final number of conflicts: 0
N-Queens solved for N = 100000 in 44729 steps.
Time taken = 1770.7252 => 29 minutes and 30.73 seconds
Solution saved to nqueens_solution_100000.txt

Figure 5: Visual Output for n=100,000

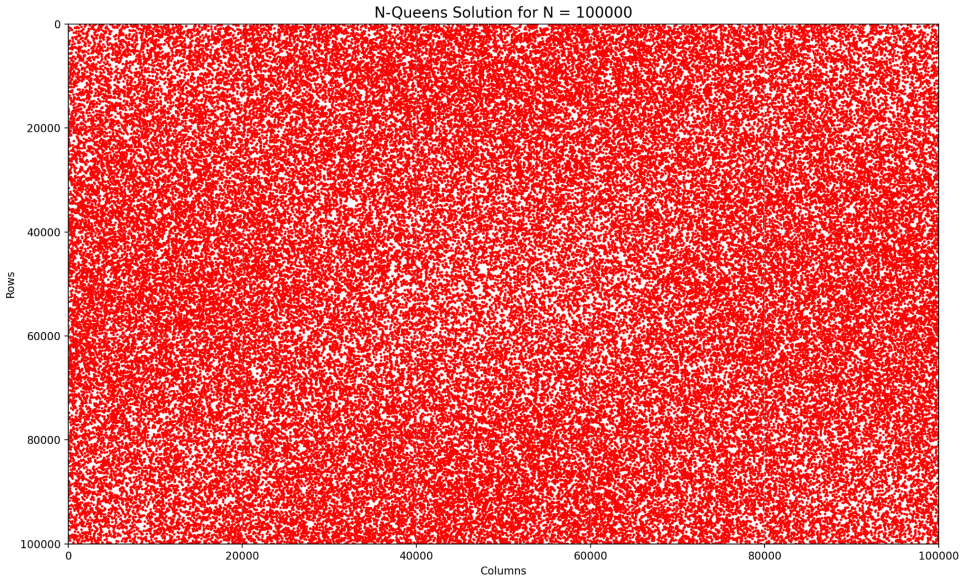
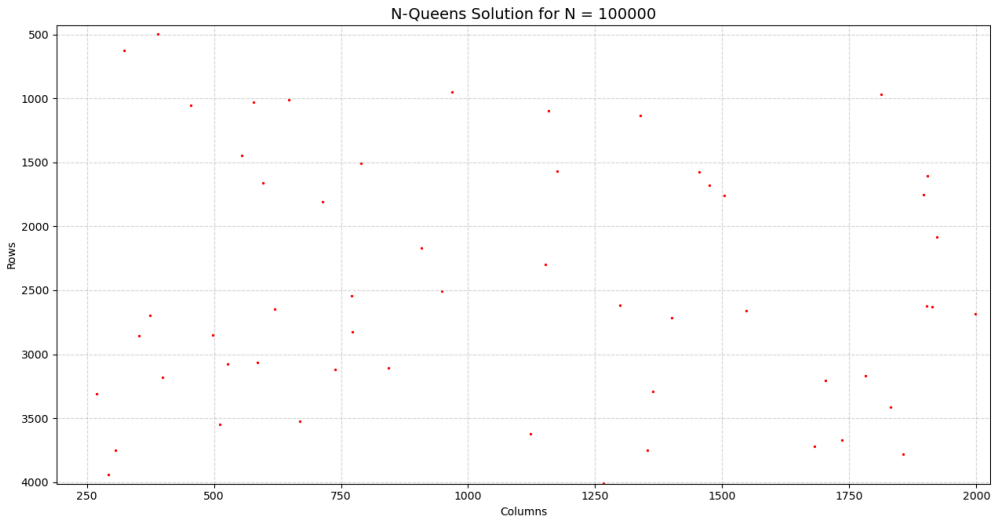


Figure 6: Zoomed-in Visual Output for n=100,000



For N = 1,000,000

Output:
Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 1336336

Our implementation of the MIN-CONFLICTS algorithm is built to handle large n, and for n=1,000,000, the algorithm runs as intended. However, given the size of the board, it naturally takes significantly more time to compute. From our earlier results, we estimate solving n=1,000,000 will take roughly 4 days of continuous computation.

This isn't a flaw in the code but rather a reflection of the complexity of the problem. The algorithm works through conflicts step by step, and with a million queens, there's a massive number of moves to calculate and adjust. While running, the code provides intermediate outputs like the remaining conflicts(see figure below) so we can see that it is progressing. This showcases the scalability of our solution, even if the runtime grows with the problem size.

(Progress of algorithm with n=1,000,000, with obvious progress within 10 hrs)

```
C:\Users\aseel\Downloads>python projectaiwith1ksteps.py 1000000

Starting MIN-CONFLICTS algorithm...
Initial number of conflicts: 1336336
Step 0: Current conflicts = 1336336
Step 1000: Current conflicts = 1332632
Step 2000: Current conflicts = 1328964
Step 3000: Current conflicts = 1325136
Step 4000: Current conflicts = 1321586
Step 5000: Current conflicts = 1317850
Step 6000: Current conflicts = 1314238
Step 7000: Current conflicts = 1310598
Step 8000: Current conflicts = 1307040
Step 9000: Current conflicts = 1303444
Step 10000: Current conflicts = 1299786
Step 11000: Current conflicts = 1296148
Step 12000: Current conflicts = 1292590
Step 13000: Current conflicts = 1288770
Step 14000: Current conflicts = 1285108
Step 15000: Current conflicts = 1281398
Step 16000: Current conflicts = 1277776
Step 17000: Current conflicts = 1274152
Step 18000: Current conflicts = 1270626
Step 19000: Current conflicts = 1266988
Step 20000: Current conflicts = 1263498
Step 21000: Current conflicts = 1259882
Step 22000: Current conflicts = 1256296
Step 23000: Current conflicts = 1252774
Step 24000: Current conflicts = 1249100
Step 25000: Current conflicts = 1245518
Step 26000: Current conflicts = 1242024
Step 27000: Current conflicts = 1238378
Step 28000: Current conflicts = 1234730
Step 29000: Current conflicts = 1231114
Step 30000: Current conflicts = 1227618
Step 31000: Current conflicts = 1224098
Step 32000: Current conflicts = 1220608
Step 33000: Current conflicts = 1217014
Step 34000: Current conflicts = 1213450
Step 35000: Current conflicts = 1209912
Step 36000: Current conflicts = 1206456
Step 37000: Current conflicts = 1202914
Step 38000: Current conflicts = 1199318
Step 39000: Current conflicts = 1195720
```

4.1 Solving Time Graph

Full Code here for solving time: time1.txt. This code is for the sake of understanding the solving times and getting the graphical representation of solving time below. The main function measures the time of the min_conflict algorithm to solve the N-Queens problem for various N sizes. It looks at the solving time for each N, checks whether the solution is valid, then saves it to a file, and then puts the results to visualize the between N size and time it took to solve. This helps understand the algorithm's efficiency and scalability.

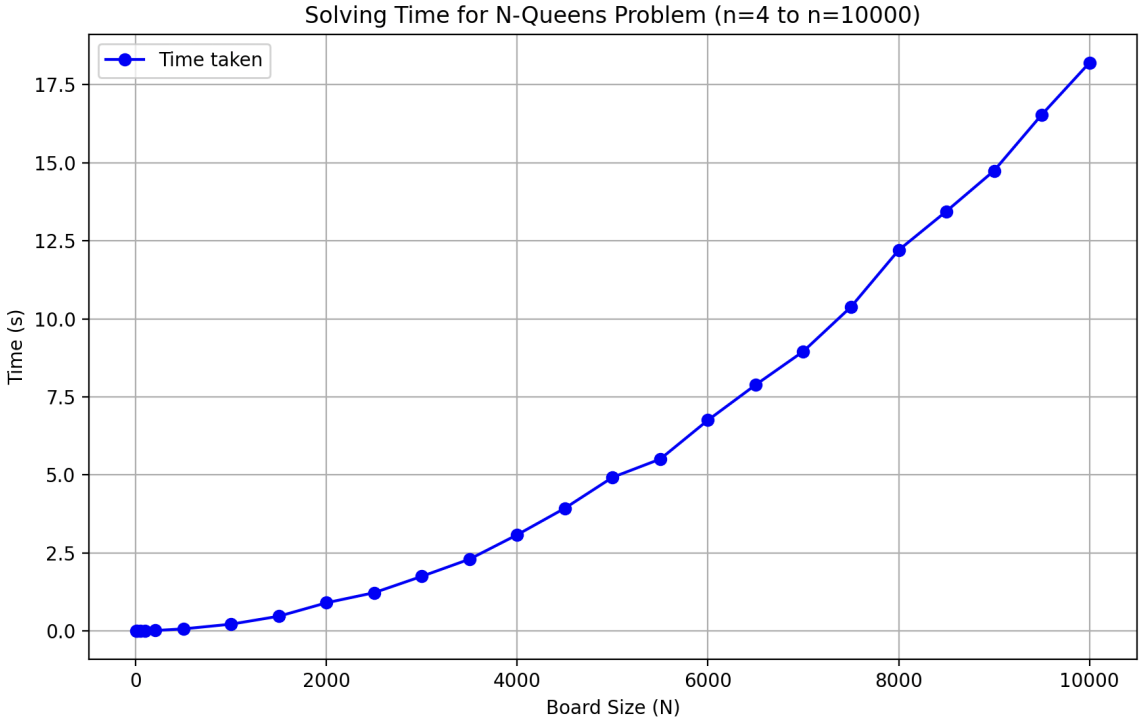
```
def main(n_values):
    """
    Measures time taken for solving the N-Queens problem for different sizes and plots the results.
    parameters-
    n_values: Size of N values to test.
    """
    max_steps = 10**6
    times = []
    n_values = int(n_values)
    #N_values = list(range(4, n_values+4, 400))
    N_values = [8, 10, 50, 100, 200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000, 9500, 10000]
    for N in N_values:
        start_time = time.perf_counter()
        queens, steps = min_conflict(N, max_steps)
        end_time = time.perf_counter()

        if(is_valid(queens)):
            # Calculate time taken and store it
            elapsed_time = end_time - start_time
        else: elapsed_time = max_steps
        times.append(elapsed_time)


        # Output to show progress
        print(f"N = {N}, Time taken = {elapsed_time:.4f} seconds, Steps = {steps}")
        save_solution(queens, N)
```

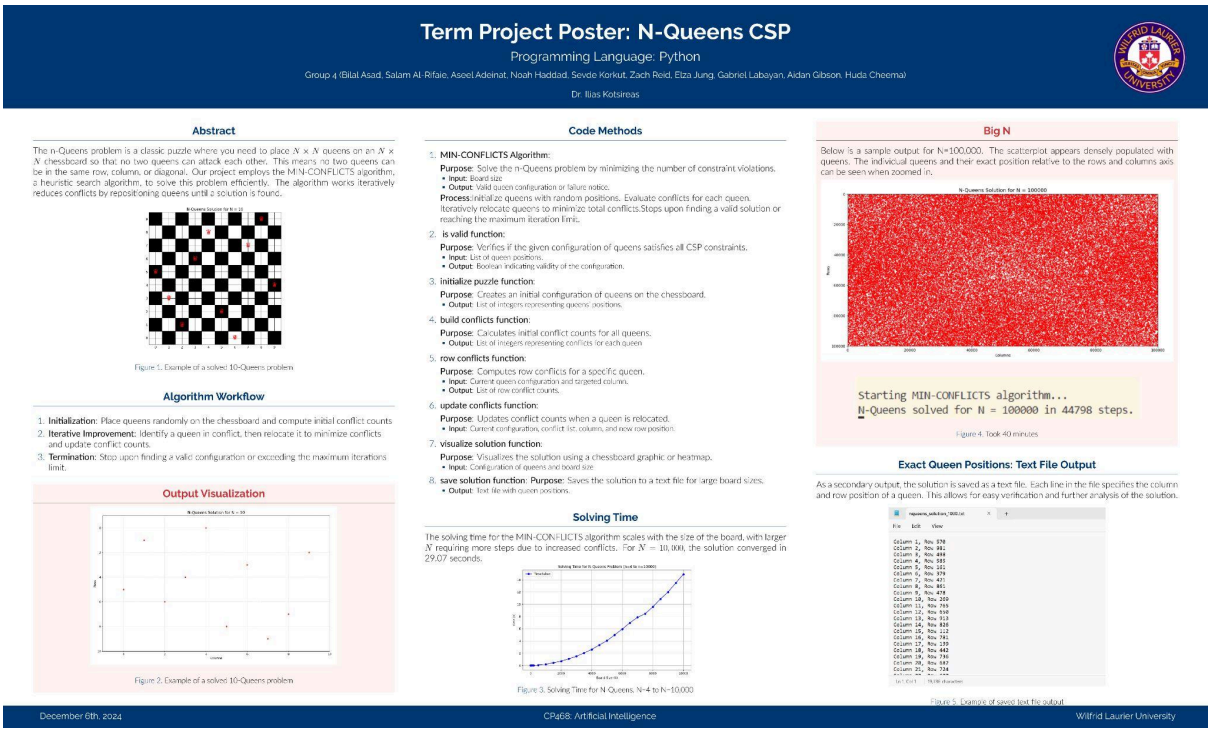
```
    # Plot the results
    plt.figure(figsize=(10, 6))
    plt.plot(N_values, times, marker='o', linestyle='-', color='b',
label='Time taken')
    plt.title("Solving Time for N-Queens Problem (n=4 to n=10000)")
    plt.xlabel("Board Size (N)")
    plt.ylabel("Time (s)")
    plt.grid(True)
    plt.legend()
    plt.show()
```

Figure 7: Graphical Representation of Solving Time



4.2 Graphical Representation: Poster

Figure 8: Graphical Representation of Big N



Link to pdf file:
https://drive.google.com/file/d/1MI2DU2t69tsTsHO-huapDoXj-tmS2OnM/view?usp=sharing

Resources

Full Term Project Code:
https://drive.google.com/file/d/1oV9Er8-TxE8pYRZOU7Y5shABBJQYaNMB/view?usp=sharing

Term Project Graphical Poster PDF file:
https://drive.google.com/file/d/1MI2DU2t69tsTsHO-huapDoXj-tmS2OnM/view?usp=sharing