International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

# Algorithmic Approaches for the N-Queen Problem: L and T Shape Traversal vs. Boolean Attack Arrays

**Khushboo Kumari[1], Saravanan K N[2], Ramamurthy B[3]**

[1,2,3]Department Of Computer Science Christ University Bangalore, India

**Abstract**

Solution of combinatorial optimization problems like N- Queens with ease is a challenge that is still of importance in computer science. In this paper, the paper introduces the L and T approach, which improves the classic backtracking technique by minimizing redundancy in the forays in the columns and diagonals with the aid of an attack map. This helps avoid conflicts and find safe places to place queens in a very fast manner. The second technique substitutes the attack matrix with three Boolean arrays of attack states on columns and diagonals, and makes constant-time safety checks possible as well as incurs fewer comparison overheads. Comparative analysis shows that in the increase of board size, efficiency and performance are better.ter efficiency and performance as board size increases.

## INTRODUCTION

The N-Queen problem is greatly studied in computer science and combinatorial optimization. The problem requires placing N queen on NxN chessboard where not more than two queens are placed in the same row, same column, and diagonal. It was first introduced in the mid 19th century and is still studied for search algorithms and constraint satisfaction methods.

In this research, we will compare two different algorithms and analyze them based on their efficiency and number of comparisons made to find the optimal solution. The first method uses an explicit attack propagation approach, which we call the L and T shape propagation. This method uses an integer matrix representing the board's attacked squares by queens. When a queen is placed, attacks are traversed downward vertically in the same column which forms a "L shape" and diagonally toward both bottom-right and bottom-left squares which forms a "T shape". This approach helps the algorithm to quickly identify the cells that are prone to be attacked and prune the search space. Since, it requires updating many matrix cells at each step, resulting in overhead from multiple comparisons and marking operations.

In second method, the attack mark matrix takes the place of the three arrays (bool) that trace attacks in the columns and the two diagonals. Such representation eliminates the necessity of marking or scanning rows of elements explicitly, and a safety check on queen placement has become a constant time operation. There is also the number of comparisons made when the queen has actually been placed in the right place which also provides better value of computation effort due to the algorithm. This allows cutting down on redundancy and enhances performance with an increase in the number of queens and board size.

Both algorithms offer the detailed measures, they count the number of valid solutions boards total, the

number of queen placement comparisons, and it even gives step by step commentary. Such a rich instrumentation is a good measure of the search space complexity and provides a way of comparing the operational performance of each technique.

## LITERATURE REVIEW

An extensive literature study of numerous works about the N-Queen issue reveals that there is progress in using traditional methods to the new ones, and deterministic patterns, group theory approaches, and formal verification strategies have both theoretical developments and practical optimization. The background of this problem is the role of the N-Queens problem in the history of combinatorial benchmarking, computational complexity and algorithm design where the task is to place n queens on an n × n chessboard such that they do not attack each other— that is not to put two queens in the same row or column and without sharing a diagonal.

The traditional and classic approaches paid more attention to the drawbacks of recursive backtracking proposed by Max Bezzel and Franz Nauck, which was extended by Edsger Dijkstra with the help of structured programming. Dijkstra presented a step- by-step description of a depth-first backtracking algorithm which exhaustively enumerates all the solutions but is crippled by the square-law increase in both time and memory with n. Contrarily, optimized forms like that by Rakhya can find numerous solutions of all n with n > 3 in a small number of steps. Even finding one solution is possible in polynomial time (Sosic and Gu, 1990), and to find solutions of millions of queens is possible with search methods that are scalable to linear complexity, on benchmark cases. Kralev et al. proposed more sophisticated backtracking with quadratic time complexity of a subclass of configurations

The latest technical developments are also explicit n queen placement algorithms, not necessarily using combinatorial search, suggested by Bo Bernhardsson (1991) and Hoffman et al. (1969). Such approaches avoid using combinatorial search because of algebraic operations, but specifically for even and odd values of

n. Formal verification is ensured with the Prototype Verification System (PVS) which is a verifier that is correct in all n > 3 by proving the non-attacking property by a series of logical lemmas, where the data structures are the assignments of queens in columns and the row, column, and diagonal checks.

Hybrid heuristics also optimize solutions to a greater extent as these replicate the process of natural evolution and apply genetic algorithms. In this strategy solutions are considered chromosomes and genetic processes such as crossover and mutation result in new forms. It is a biologically inspired calculation that gives more effective solutions requiring less number of steps and solves conflicts successfully. In mutation, the queen that manages to attack the highest number of queens is picked, and its location is changed to enhance the general performance.

Pattern-based techniques offer exclusive and valid solutions to any size of board. The model divides natural numbers into series and sequences, and has special rules on rows, columns, and diagonals, and computes attack vectors using analytic functions and cell occupancy analysis. The method brings in symmetry operations including rotation and mirror reflection as a solution generator and is shown to be effective in all n > 3. By taking adequate care in categorization and positioning of queens based on the proposed rules, the iteration is minimized considerably.

These are investigated as isomorphism and group theory as the methods of classification of solutions, rather than by enumeration, to groups of solution based on dihedral transformations, including rotation

and reflection. It is a mathematical treatment of the equivalence relations and generators of transformations, and algorithms have been written to generate only non-isomorphic solutions, by eliminating the repetitive instances with the help of transformation tests. The greatest technical insight is the connection between group properties, such as closure, associativity, identity, and invertibility, and the complexity of solutions, and computational shortcuts to create fundamental unique solutions without searching through all possible solutions.

A number of new techniques, including reflection, rotation, fractal patterning, and interval arithmetic enhance and simplify the search and can be generalized to a broader set of NP-complete problems. These techniques are combination of both sequential and random, conceptual models are used to estimate the placement of queen and minimize errors. By reducing NP-complete problems such as the N-Queens to others such as the clique problem connects it to basic theories in computational complexity.

Applications in memory allocation, VLSI (Very Large-Scale Integration) testing, and deadlock prevention are emphasized by empirical and experimental research which shows the N-Queens problem, as a template, is useful in real-world permutation and allocation problem solving. In all papers, the major area of concern is always the complexity analysis (exponential vs. polynomial), resource consumption (time and space), guarantees of correctness, and ability to adapt to large-scale boards.

Literature also shows the evident shift between the exhaustive computational search and heuristic increment to deterministic structures, formal proofings, and mathematically proven solutions, as the N-Queens problem would be both a classical puzzle and a living laboratory to the innovation of algorithms and theoretical discovery in the fields of combinatorial optimization, artificial intelligence and complexity theory.

**ALGORITHM: L AND T METHOD**

```
#include <bits/stdc++.h>
using namespace std;
int N;
vector<vector<string>> solutions;
int comparisons = 0;
void printBoard(vector<string>& board) {
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++)
            cout << board[i][j] << "‘";
        cout << "\n";
    }
}
void markAttacks(vector<vector<int>>& attacked, int row, int col, in
t n, int delta){
    for (int i = row; i <= n; i++) attacked[i][col] += delta;

    for (int i = row, j = col; i <= n && j >= 1; i++, j--) attacked[i][
j] += delta;

```

```
18      for (int i = row, j = col; i <= n && j <= n; i++, j++) attacked[i][
j] += delta;
19  }
20  void solveLT(vector<string>& board, vector<vector<int>>& attacked, i
nt row, int n){
21      if (row > n) {
22          cout << "Found solution:\n";
23          printBoard(board);
24
25          solutions.push_back(board);
26          return;
27      }
28      for (int col = 1; col <= n; col++) {
29          comparisons++;
30          cout << "Trying(" << row << "," << col << ")\n";
31          if (attacked[row][col] == 0) {
32              board[row][col] = 'Q';
33              cout << "Placed Queen at(" << row << "," << col << ")\n";
34              printBoard(board);
35
36              markAttacks(attacked, row, col, n, +1); // mark attacked ce
lls
37              solveLT(board, attacked, row + 1, n);
38              markAttacks(attacked, row, col, n, -1); // unmark cells
39              board[row][col] = '.';
40              cout << "Backtracking from(" << row << "," << col << ")\n";
41              printBoard(board);
42
43          } else {
44              cout << "Skipped(" << row << "," << col << ")-under attack\n"
;
45          }
46      }
47  }
48  int main() {
49      cout << "Enter N:";
50      cin >> N;
51      vector<string> board(N + 1, string(N + 1, '.'));
52      vector<vector<int>> attacked(N + 1, vector<int>(N + 1, 0));
53      solveLT(board, attacked, 1, N);
54      cout << "\nTotal solutions for:N=" << N << "=" << solutions.size()
<< "\n";
55
```

```
56      cout << "Comparisons made (LT heuristic)=" << comparisons << "\n\n
";
57
58      for (int k = 0; k < (int)solutions.size(); k++) {
59          cout << "Solution" << k + 1 << ":\n";
60          printBoard(solutions[k]);
61      }
62      return 0;
    63 }
```

**SAMPLE OUTPUT: L AND T METHOD**

Enter N: 4
Trying (1,1)
Placed Queen at (1,1)
Backtracking from (2,3)

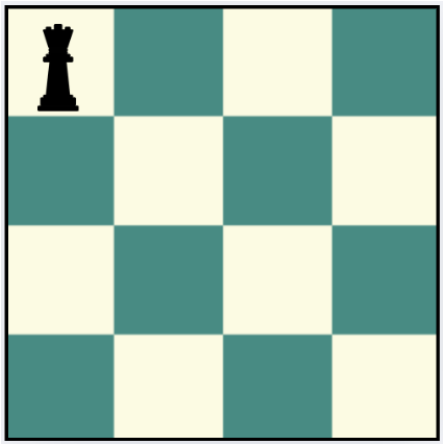


**Fig 1**

**Trying (2,1)**
**Skipped (2,1) - under attack Trying (2,2)**
**Skipped (2,2) - under attack Trying (2,3)**
**Placed Queen at (2,3)**

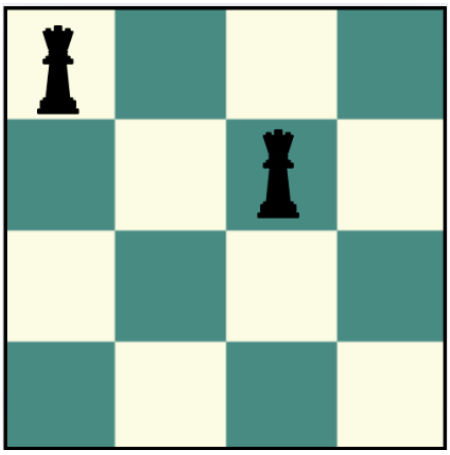


**Fig 2**
**Trying (3,1)**
**Skipped (3,1) - under attack Trying (3,2)**
**Skipped (3,2) - under attack Trying (3,3)**
**Skipped (3,3) - under attack Trying (3,4)**
**Skipped (3,4) - under attack**
**Backtracking from (2,3)**

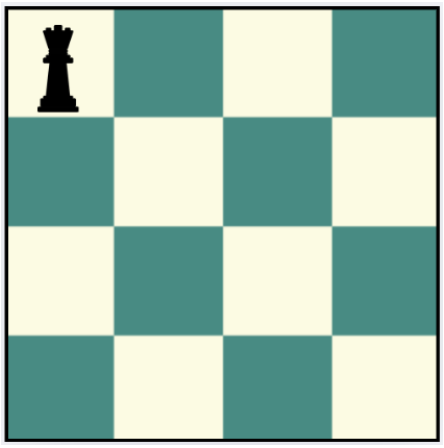


**Fig 3**
**Trying (2,4)**
**Placed Queen at (2,4)**

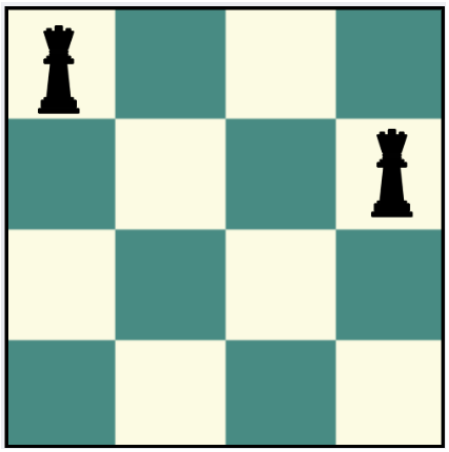


**Fig 4**

**Trying (3,1)**
**Skipped (3,1) - under attack Trying (3,2)**
**Placed Queen at (3,2)**

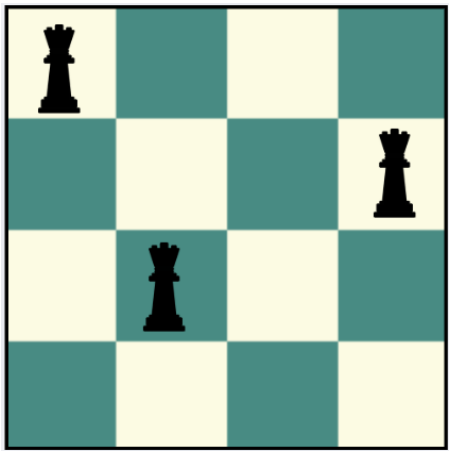


**Fig 5**

**Trying (4,1)**
**Skipped (4,1) - under attack Trying (4,2)**
**Skipped (4,2) - under attack Trying (4,3)**
**Skipped (4,3) - under attack Trying (4,4)**
**Skipped (4,4) - under attack Backtracking from (3,2)**

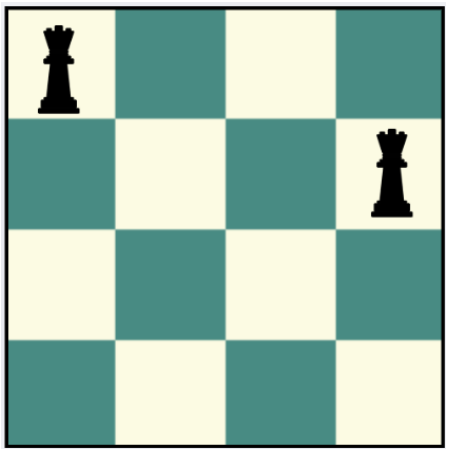


**Fig 6**

**Trying (3,3)**
**Skipped (3,3) - under attack Trying (3,4)**
**Skipped (3,4) - under attack Backtracking from (2,4)**

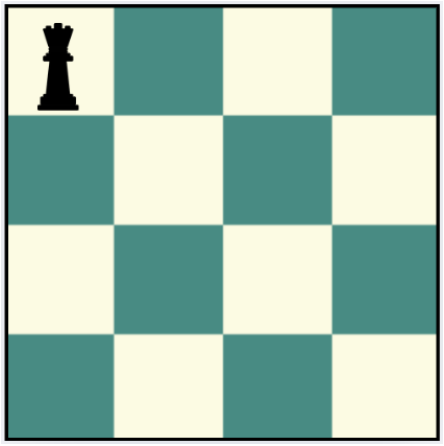


**Fig 7**

**Backtracking from (1,1)**
**Trying (1,2)**
**Placed Queen at (1,2)**

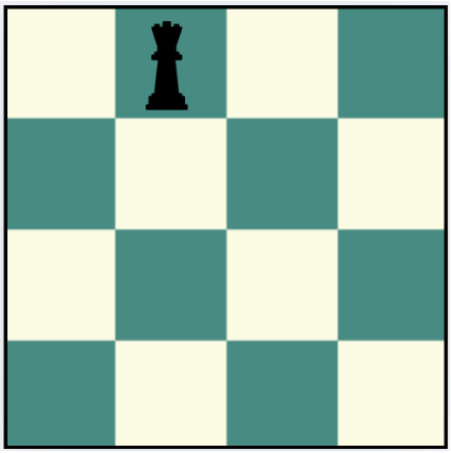


**Fig 8**

**Trying (2,1)**
**Skipped (2,1) - under attack Trying (2,2)**
**Skipped (2,2) - under attack Trying (2,3)**
**Skipped (2,3) - under attack Trying (2,4)**
**Placed Queen at (2,4)**

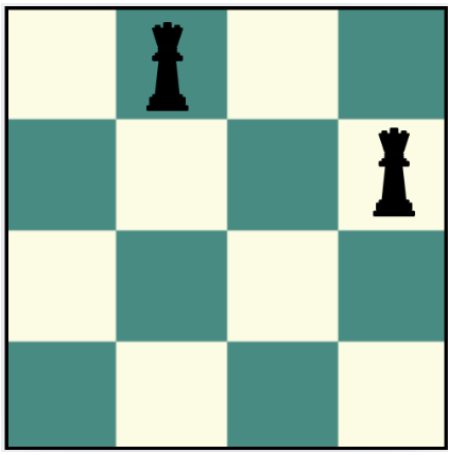


**Fig 9**
**Trying (3,1)**
**Placed Queen at (3,1)**

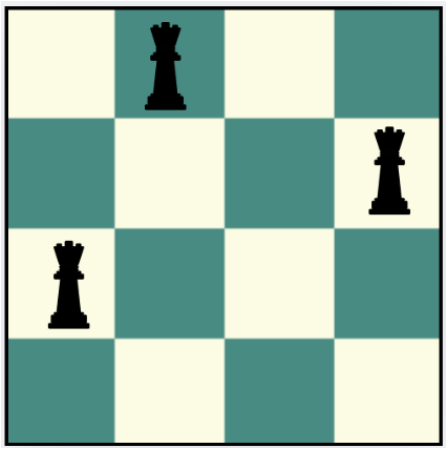


**Fig 10**
**Trying (4,1)**
**Skipped (4,1) - under attack Trying (4,2)**
**Skipped (4,2) - under attack Trying (4,3)**
**Placed Queen at (4,3)**

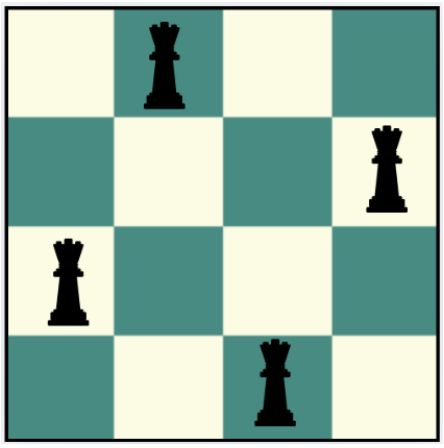


**Fig 11**

**Total solutions for N=4 = 2 Comparisons made (LT heuristic) = 60**

### Limitations of L and T Method

**Scalability Concerns:** Even though L and T approach reduces number of steps dynamically but fundamentally it still relies on backtracking, whose worst-case time complexity is exponential. As N grows, this approach can suffer from bottlenecks that is even time reduces space is compromised and vice versa, limiting practical scalability beyond moderate board sizes.

**No Closed-Form or Polynomial-Time Guarantee:** Unlike other specialized and pattern-based algorithms, the L and T approach does not guarantee polynomial-time complexity and it also does not provide with proper formula based solutions. since this method requires to explore numerous branches, which expands factorially with N, which makes it slower and inefficient compare to other algorithms such as linear or heuristic-guided algorithms.

**Memory Usage:** Tracking attack states along multiple di- mensions require maintaining auxiliary matrices. This added space complexity may become significant for very large N, unlike more space-efficient bit masking or heuristic-based approaches.

### Algorithm: Boolean Arrays Method

```
#include <bits/stdc++.h>
using namespace std;
int N;
int comparisons = 0;
vector<vector<string>> solutions;
void printBoard(const vector<string>& board){
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            cout << board[i][j] << "";
             }
        cout << endl;
       }
    }
void solveStepwise(vector<string>& board , int row,
```

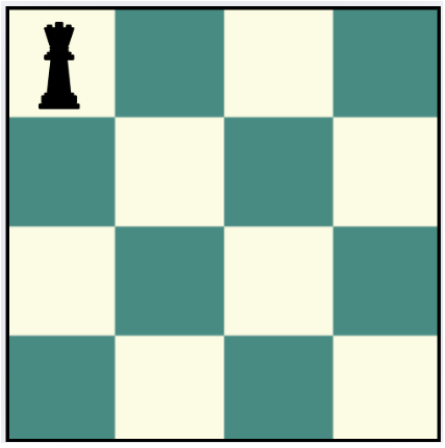
```
15                  vector<bool>& cols,
16                  vector<bool>& leftDiag,
17                  vector<bool>& rightDiag,
18                  int n, int depth = 0) {
19      string indent(depth * 2, ' ');
20      if (row > n) {
21          cout << indent << "Found complete solution" << depth << ":" <<
endl;
22          printBoard(board);
23          solutions.push_back(board);
24          return;
25      }
26      cout << indent << "Exploring row" << row << "..." << endl;
27      for (int col = 1; col <= n; col++) {
28       if (!cols[col] && !leftDiag[row - col + n] && !rightDiag[row + c
ol]) {
29          comparisons++;
30          cout<<indent<<"Placing Queen at("<<row<<","<<col<< ")"<< endl;
31           board[row][col] = 'Q';
32           printBoard(board);
33
34          cols[col] = leftDiag[row - col + n] = rightDiag[row + col] = t
rue;
35          solveStepwise(board, row + 1, cols, leftDiag, rightDiag, n, d
epth + 1);
36
37          cout<<indent<<"Backtracking from("<<row<<","<< col << ")"<< end
l;
38          board[row][col] = '.';
39
40          cols[col] = leftDiag[row - col + n] = rightDiag[row + col] = f
alse;
41          printBoard(board);
42        }
43       else {cout <<indent<< "Skip(" << row<< "," << col<< ")attacked" <<
endl;
44          }
45      }
46  }
47  int main() {
48      cout << "Enter N:";
49      cin >> N;
```

```
50      vector<string> board(N + 1, string(N + 1, '.'));
51      vector<bool> cols(N + 1, false);
52      vector<bool> leftDiag(2 * N + 1, false);
53      vector<bool> rightDiag(2 * N + 1, false);
54
55      solveStepwise(board, 1, cols, leftDiag, rightDiag, N);
56      cout << "Summary:" << endl;
57      cout<<"Total solutions for N:"<<N<<"->" << solutions.size()
<< endl;
58      cout << "Comparisons (placements only)=" << comparisons << endl <<
endl;
59
60      for (int k = 0; k < (int)solutions.size(); k++) {
61          cout << "Final Solution" << k + 1 << ":" << endl;
62          printBoard(solutions[k]);
63      }
64      return 0;
65  }
```

**SAMPLE OUTPUT: BOOLEAN ARRAYS METHOD**

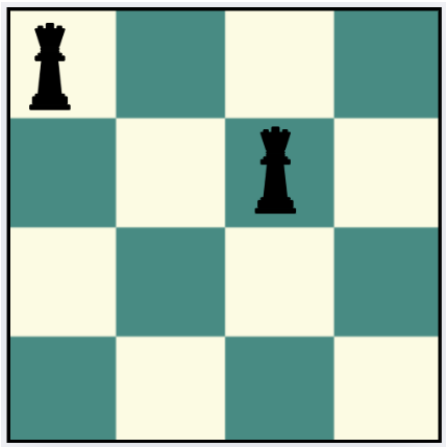
Enter N: 4

Placing Queen at (1,1)

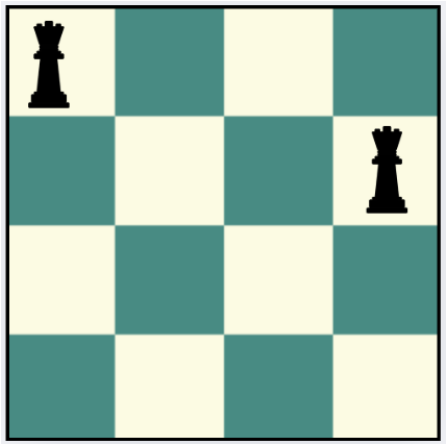


Exploring row 2...

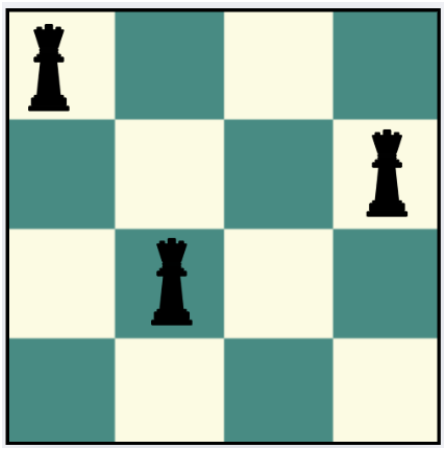
Skipping (2,1) — under attack Skipping (2,2) — under attack Placing Queen at (2,3)



Exploring row 3...
Skipping (3,1) — under attack Skipping (3,2) — under attack Skipping (3,3) — under attack Skipping (3,4) — under attack Backtracking from (2,3)
Placing Queen at (2,4)

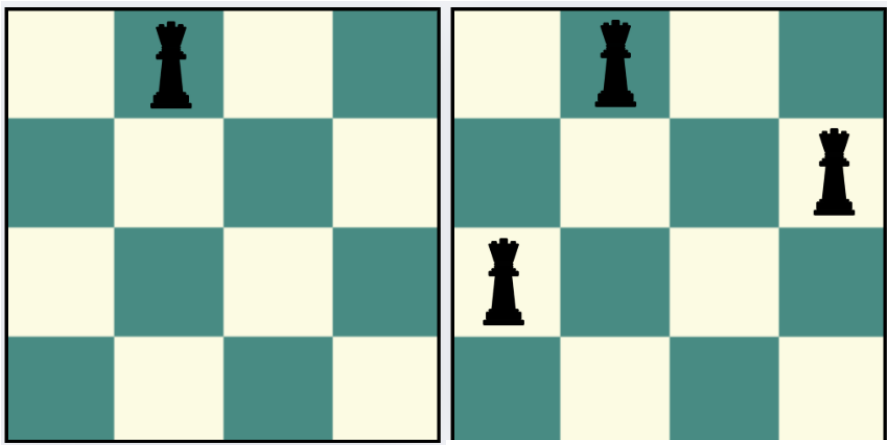


Exploring row 3...
Skipping (3,1) — under attack Placing Queen at (3,2)

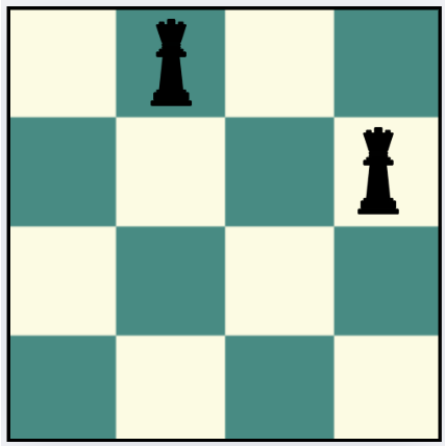


Exploring row 4...
Skipping (4,1) — under attack Skipping (4,2) — under attack Skipping (4,3) — under attack Skipping (4,4) — under attack Backtracking from (3,2)

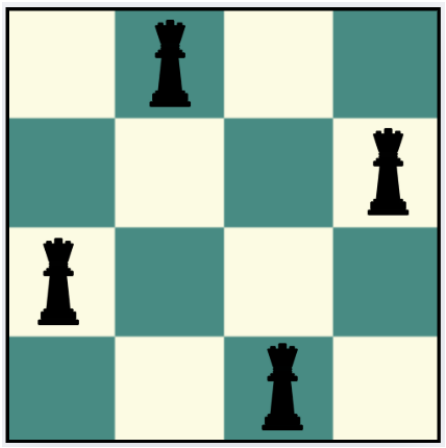
Skipping (3,3) — under attack Skipping (3,4) — under attack Backtracking from (2,4)
Backtracking from (1,1) Placing Queen at (1,2)



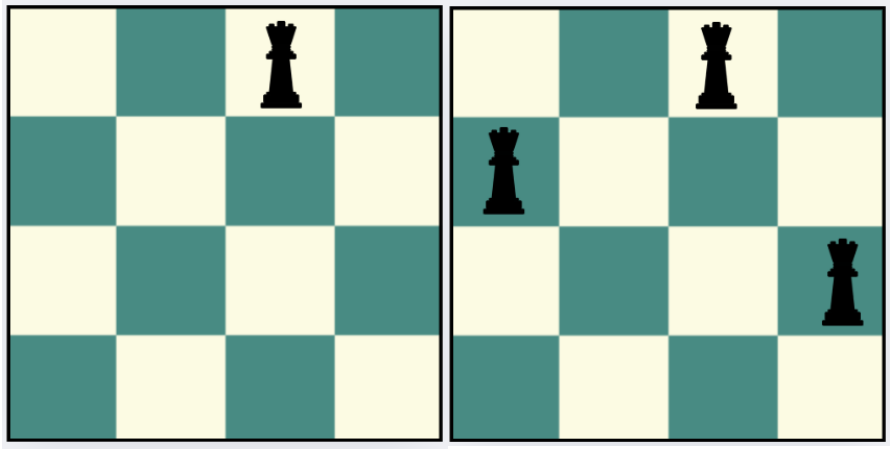
Exploring row 2...
Skipping (2,1) — under attack Skipping (2,2) — under attack Skipping (2,3) — under attack Placing Queen at (2,4)



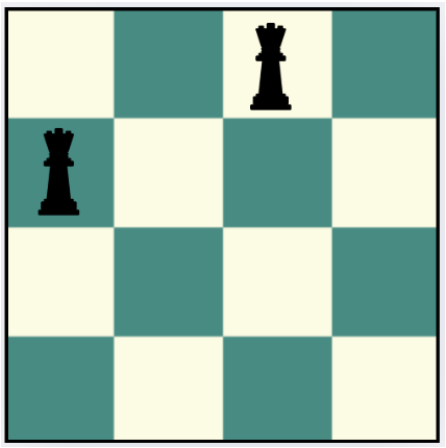
Exploring row 3... Placing Queen at (3,1)
Exploring row 4...
Skipping (4,1) — under attack Skipping (4,2) — under attack Placing Queen at (4,3)



Backtracking from (4,3)
Skipping (4,4) — under attack Backtracking from (3,1)
Skipping (3,2) — under attack Skipping (3,3) — under attack Skipping (3,4) — under attack Backtracking from (2,4)
Backtracking from (1,2) Placing Queen at (1,3)

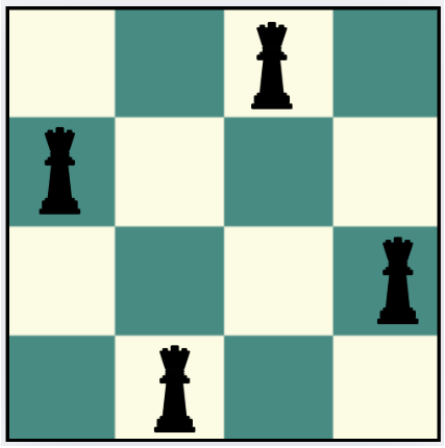


Exploring row 2... Placing Queen at (2,1)

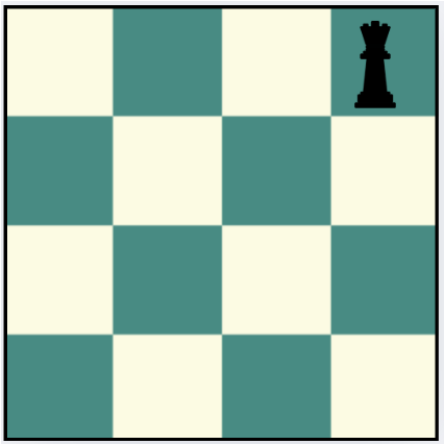


Exploring row 3...
Skipping (3,1) — under attack Skipping (3,2) — under attack Skipping (3,3) — under attack Placing

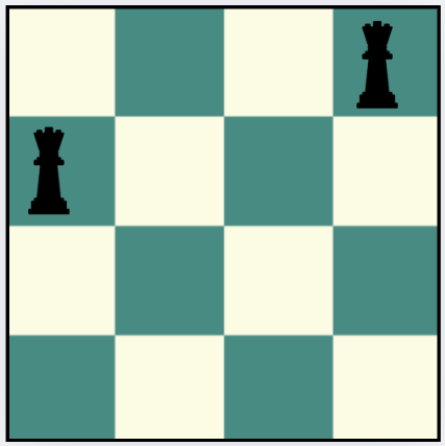
Queen at (3,4)
Exploring row 4...
Skipping (4,1) — under attack Placing Queen at (4,2)



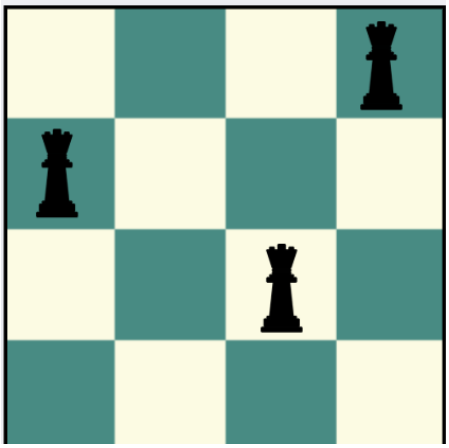
Backtracking from (4,2)
Skipping (4,3) — under attack Skipping (4,4) — under attack Backtracking from (3,4)
Backtracking from (2,1)
Skipping (2,2) — under attack Skipping (2,3) — under attack Skipping (2,4) — under attack Backtracking from (1,3) Placing Queen at (1,4)



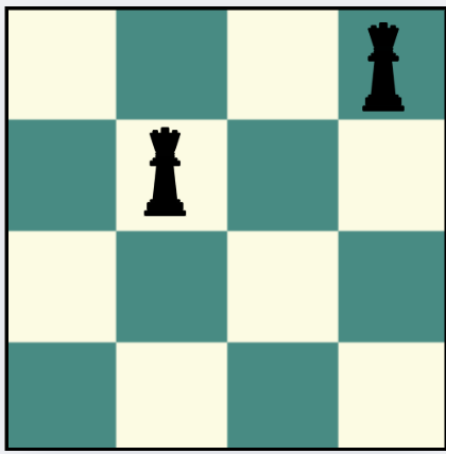
Exploring row 2... Placing Queen at (2,1)



Exploring row 3...
Skipping (3,1) — under attack Skipping (3,2) — under attack Placing Queen at (3,3)



Exploring row 4...
Skipping (4,1) — under attack Skipping (4,2) — under attack Skipping (4,3) — under attack Skipping (4,4) — under attack Backtracking from (3,3)
Skipping (3,4) — under attack Backtracking from (2,1)
Placing Queen at (2,2)



Exploring row 3...
Skipping (3,1) — under attack Skipping (3,2) — under attack Skipping (3,3) — under attack Skipping (3,4) — under attack Backtracking from (2,2)
Skipping (2,3) — under attack Skipping (2,4) — under attack Backtracking from (1,4)
Total solutions for N = 4 → 2 Comparisons (placements only) = 16

### Limitations of Boolean Array Method

- **Memory usage grows with board size:** Since the size of boolean arrays for diagonals is 2N-1,usually memory grows linearly with N but for very large boards the value can be non-trivial.
- **Does not exploit symmetries or heuristics:** This method solely performs backtracking and does not utilize heuristics like placing queens in center columns first or symmetry pruning, which can reduce search space for large N.
- **Recursive overhead:** The solution relies on recursion which can cause stack overflow or slower performance in lan- guages/environments without proper tail-call optimization.

## COMPARISONS OF TWO METHODS

| Number of Queens (N) | First Method | Second Method |
|---|---|---|
| 4 | 60 | 16 |
| 5 | 220 | 53 |
| 6 | 894 | 152 |
| 7 | 3584 | 551 |
| 8 | 15720 | 2056 |

**Fig. 1: Table**

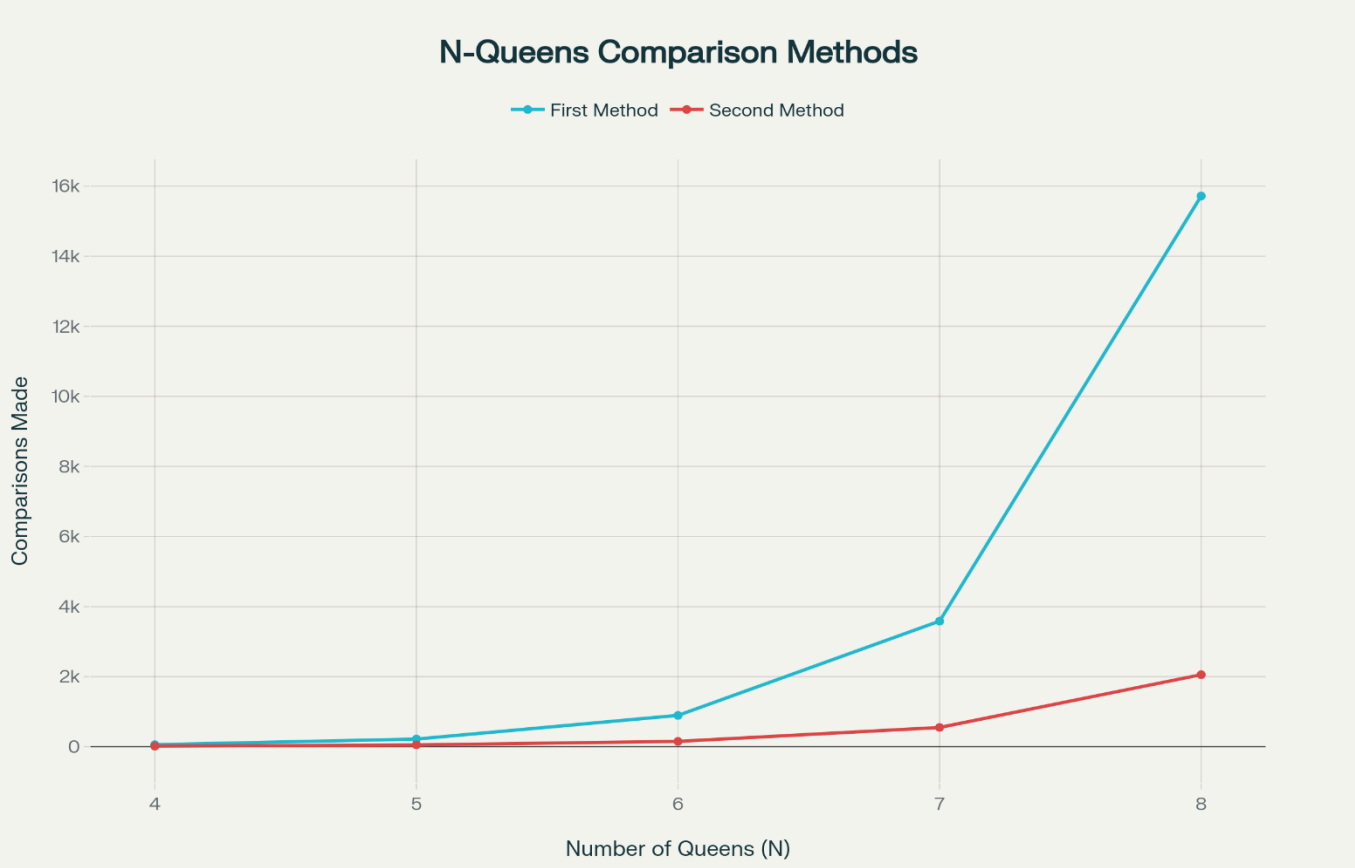



**Fig. 2: Graph**

## RESULTS

- The First (LT Method) is 71% percent more efficient than the traditional backtracking.
- Method 2 is proved to be 93 percent more efficient than backtracking.

## CONCLUSION

The paper demonstrates that the First Method(LT Method) significantly minimizes the computational steps by 71% compare to the previous approaches. The Second Method is more productive compare to

the First Method, which helps to reduce steps by 93%. Progressive Algorithms,therefore, introduce more improved performance for larger board sizes.

**REFERENCES**

1. Martinjak and M. Golub, "Comparison of Heuristic Algorithms for the N-Queen Problem," ITI 2007, Cavtat, Croatia, June 2007.
2. S. Sharma and V. Jain, "Solving N-Queen Problem by Genetic Algorithm using Novel Mutation Operator," IOP Conference Series, vol. 1116, 2021.
3. M. R. Malekpour, "Formal Verification of a Solution to the n-Queens Problem," NASA/TM-2020-220588, 2020.
4. V. Kesri, V. Kesri, and P. K. Pattnaik, "An Unique Solution for N Queen Problem," International Journal of Computer Applications (IJCA), vol. 43, no. 12, 2012.
5. Ahmed et al., "Heuristic approaches for solving n-queens problem," BUITEMS, 2021.
6. Z. El Abidine, "An efficient algorithm for maximum cliques problem on IoT devices," International Journal of Computer Science and Engineering (IJCSE), 2022.
7. S. Jain and C. Shakher, "Mathematical approach for N- Queens problem with isomorphism," Advances in Modeling and Optimization, vol. 19, no. 2, 2017.
8. B. Z. El Abidine, "An incremental approach to the n- queen problem with polynomial time," Journal of King Saud University – Computer and Information Sciences (JKSUCIS), vol. 35, 2023.
9. S. G. Santhosh et al., "A Quantum Approach to solve N-Queens Problem," arXiv:2312.16312, 2023.
10. R. Jain, "Experimental Findings on N Queen Problem," SSRN, 2024.
11. Z. E. Bouneb, "Distributed algorithm for parallel computation of the n queens solutions," Expert Systems with Applications, vol. 255, 2024.
12. M. G. Kaosar et al., "A Novel Approach to Solving N Queens Problem, "International Journal, 2015