DONG THAP UNIVERSITY JOURNAL OF SCIENCE
Tạp chí Khoa học Đại học Đồng Tháp
Natural Sciences Issue
ISSN 0866-7675 | e-ISSN 2815-567X

DOI: https://doi.org/10.52714/dthu.ns.3283.2031

SEARCH SPACE REPRESENTATION AND ANALYSIS OF THE BACKTRACKING ALGORITHM FOR THE N-QUEENS PROBLEM

Lam Thanh Toan, Nguyen Van Chi, and Nguyen Xuan Ha Giang*

Faculty of Information Technology, Cantho University of Technology, Vietnam

**Corresponding author, Email: nxhgiang@ctuet.edu.vn*

Article history

Received: 30/3/2026; Received in revised form: 28/8/2026; Accepted: 15/9/2026

Abstract

The N-Queens problem is a classical constraint satisfaction problem in artificial intelligence and combinatorial optimization. Although its formulation is simple, its search space grows rapidly as the board size increases, making exhaustive search computationally inefficient. This study analyzes the N-Queens problem within the search-space framework of the Backtracking algorithm, focusing on branching expansion, constraint checking, and early pruning of infeasible configurations. Three methods are implemented and compared: permutation-based exhaustive search, conventional Backtracking, and Bitmask Backtracking. Experimental results show that conventional Backtracking reduces the search workload from 40,320 permutations to 2,057 nodes for N=8, and from 3,628,800 permutations to 35,539 nodes for N=10. In addition, Bitmask Backtracking visits the same number of nodes as conventional Backtracking but achieves approximately 2.75× and 4.14× speed-up for N=8 and N=10, respectively. These results indicate that Backtracking improves efficiency through search-space pruning, while Bitmask representation further reduces the computational cost of constraint checking. The study provides a systematic and reproducible analysis of Backtracking for understanding constraint-based search and optimization problems.

Keywords: *Backtracking algorithm, Combinatorial optimization, Constraint satisfaction, Recursive search, Search space n-queens problem.*

Cite: Lam, T. T., Nguyễn, V. C., & Nguyen, X. H. G. (2026). Search space representation and analysis of the backtracking algorithm for the N-Queens problem. *Dong Thap University Journal of Science - Natural Sciences Issue*, *Online First*, 1-15. https://doi.org/10.52714/dthu.ns.3283.2031

BIỂU DIỄN KHÔNG GIAN TÌM KIẾM VÀ PHÂN TÍCH GIẢI THUẬT BACKTRACKING CHO BÀI TOÁN N-QUÂN HẬU

Lâm Thanh Toản, Nguyễn Văn Chí và Nguyễn Xuân Hà Giang*

Khoa Công nghệ Thông tin, Trường Đại học Kỹ thuật – Công nghệ Cần Thơ, Việt Nam

**Tác giả liên hệ, Email: nxhgiang@ctuet.edu.vn*

Lịch sử bài báo

Ngày nhận: 30/3/2026; Ngày nhận chỉnh sửa: 28/8/2026; Ngày duyệt đăng: 15/9/2026

Tóm tắt

Bài toán N-Queens là một bài toán thỏa mãn ràng buộc kinh điển trong trí tuệ nhân tạo và tối ưu hóa tổ hợp. Mặc dù có mô tả đơn giản, không gian tìm kiếm của bài toán tăng nhanh theo kích thước bàn cờ, khiến các phương pháp vét cạn trở nên kém hiệu quả về mặt tính toán. Nghiên cứu này phân tích bài toán N-Queens dưới góc độ không gian tìm kiếm của thuật toán Backtracking, tập trung vào cơ chế mở rộng nhánh, kiểm tra ràng buộc và cắt tỉa sớm các cấu hình không khả thi. Ba phương pháp được cài đặt và so sánh gồm vét cạn theo hoán vị (Permutation-based Exhaustive Search), Backtracking truyền thống và Bitmask Backtracking. Kết quả thực nghiệm cho thấy Backtracking truyền thống giảm khối lượng tìm kiếm từ 40.320 hoán vị xuống 2.057 nút đối với N=8, và từ 3.628.800 hoán vị xuống 35.539 nút đối với N=10. Bên cạnh đó, Bitmask Backtracking duyệt cùng số lượng nút với Backtracking truyền thống nhưng đạt tốc độ thực thi nhanh hơn khoảng 2,75 lần đối với N=8 và 4,14 lần đối với N=10. Những kết quả này cho thấy Backtracking nâng cao hiệu quả thông qua cơ chế cắt tỉa không gian tìm kiếm, trong khi biểu diễn bitmask tiếp tục giảm chi phí tính toán của quá trình kiểm tra ràng buộc. Nghiên cứu cung cấp một phân tích có hệ thống và có thể tái lập về thuật toán Backtracking, góp phần hỗ trợ việc nghiên cứu, giảng dạy và đánh giá các phương pháp giải bài toán thỏa mãn ràng buộc và tối ưu hóa tổ hợp.

Từ khóa: *Bài toán thỏa mãn ràng buộc, Bài toán N-Queens, Thuật toán quay lui, Tìm kiếm đệ quy, Tối ưu hóa tổ hợp.*

1. Introduction

The N-Queens problem is a classical benchmark in artificial intelligence and combinatorial search, which aims to place N queens on an N×N chessboard such that no two queens attack each other along rows, columns, or diagonals (Bell & Stevens, 2009; Thada & Dhaka, 2014). Despite its well-defined constraint structure, the problem exhibits an extremely large number of potential configurations, making it a widely used benchmark for studying search techniques, constraint satisfaction problems (CSPs), and the phenomenon of combinatorial explosion (Gent & Walsh, 1999). Although the problem formulation is relatively simple, the size of the search space grows exponentially with respect to N, leading to prohibitively high computational costs for exhaustive search approaches (Lijo & Jose, 2015; Haralick & Elliott, 1980). For example, when N=8, there exist 92 valid solutions, and this number increases to 724 when N=10 (Kumar, 1992). This combinatorial explosion is a common characteristic of real-world optimization and constraint-based problems (Norvig, 2021), rendering naive traversal of the entire search space computationally infeasible.

Among various solution strategies, the Backtracking algorithm is one of the most widely adopted approaches for solving the N-Queens problem due to its ability to systematically explore the search space while enforcing constraints at each step and backtracking upon encountering invalid configurations (Nudelman et al., 2004; Russell & Norvig, 2020). Backtracking also serves as a fundamental mechanism in many CSP solvers, as established in foundational studies (Schaefer, 1978; Simonis, 2005). Furthermore, advanced techniques such as bitmasking, heuristic pruning, and constraint programming are often built upon or extend the Backtracking framework to improve computational efficiency.

In addition to conventional Backtracking, numerous optimization techniques have been proposed to improve the efficiency of solving the N-Queens problem and related constraint satisfaction problems. Representative approaches include Bitmask-based implementations, heuristic search strategies, and constraint programming frameworks. While Bitmask-based methods mainly reduce the computational cost of constraint checking through compact bitwise representations, heuristic search techniques improve search efficiency by reducing the number of explored branches through informed variable ordering and pruning strategies, whereas constraint programming exploits global constraints and propagation mechanisms to further prune the search space (Haralick & Elliott, 1980; Tsang, 1993; Simonis, 2005; Russell & Norvig, 2020; Poole & Mackworth, 2023). These approaches have significantly improved the practical efficiency of solving the N-Queens problem and have also influenced the development of modern constraint satisfaction solvers. Nevertheless, while these approaches have significantly improved computational efficiency, comparatively fewer studies have investigated how the conventional Backtracking algorithm represents, explores, and prunes the search space. Such an analysis remains important because many advanced methods inherit the same depth-first search framework.

Motivated by this research gap, this paper does not aim to propose a novel variant of Backtracking. Instead, it focuses on formally modeling the N-Queens problem within the search space of Backtracking, providing an in-depth analysis of the branching process, constraint checking, and Backtracking behavior, as well as evaluating the effectiveness of search space pruning achieved by the algorithm. Based on this framework, an automated system is implemented to enumerate all valid solutions for multiple values of N, while generating empirical data to support the analysis of algorithmic behavior and computational complexity (Tsang, 1993; Wu & Tian, 2018). This approach not only clarifies the technical role and operational mechanisms of Backtracking in the context of a canonical problem, but also establishes a foundation for comparison and extension toward more advanced

techniques such as bitmasking, heuristic pruning, and general CSP solvers. Consequently, this study contributes to bridging the gap between theoretical analysis and practical applications of search algorithms in optimization, scheduling, and artificial intelligence problems characterized by similar constraint structures.

2. Theoretical Background

The N-Queens problem is a canonical combinatorial optimization task that can be naturally formulated within the framework of Constraint Satisfaction Problems (CSPs). The objective is to place N queens on an n×n chessboard such that no two queens attack each other under standard chess movement rules.

2.1. CSP Formulation

A CSP is defined by a triplet (X,D,C), where X denotes a set of variables, D represents their corresponding domains, and C is a set of constraints.

Variables:

$$X = \{x_1, x_2, \dots, x_n\} \tag{1}$$

where each variable x_i represents the column position of the queen placed in row i.

Domain:

$$D(x_i) = \{1,2, \dots, n\}, \forall i \in \{1, \dots, n\} \tag{2}$$

Constraints:

$$x_i \neq x_j, \forall i \neq j \ (column\ contraints) \tag{3}$$

$$|x_i - x_j| \neq |i - j|, \forall i \neq j (diagonal\ constraints) \tag{4}$$

2.2. Global Constraint Form

The problem can be compactly expressed using global constraints:

$$AllDifferent(x_1, x_2, \dots, x_n) \tag{5}$$

$$AllDifferent(x_i + i), AllDifferent(x_i - i) \tag{6}$$

2.3. Backtracking Solution

The solution is obtained via depth-first backtracking. At step i, a value $v \in D(x_i)$ is accepted if:

$$\forall k < i: v \neq x_k \wedge |v - x_k| \neq |i - k| \tag{7}$$

The algorithm incrementally builds a feasible assignment and prunes inconsistent branches early.

3. Data and Research Methodology

3.1. Backtracking Algorithm

The program implements the N-Queens problem using the Backtracking algorithm integrated with a graphical user interface (GUI) built on Tkinter (see Figure 1). Users input

the board size (N) through the GUI window, after which the program computes all valid configurations of placing (N) queens on an (NxN) chessboard such that no two queens attack each other. The core algorithm is implemented via the function *solve_n_queens* (see Algorithm 1).

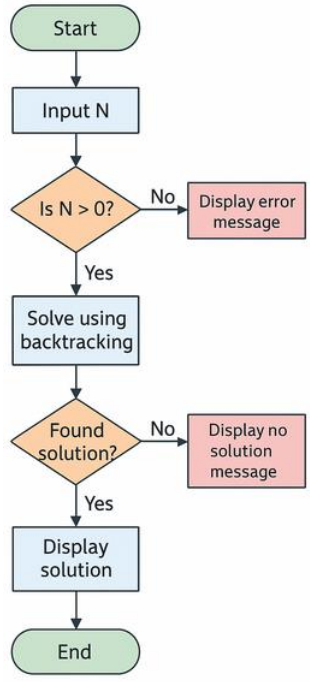

Figure 1. Backtracking algorithm to solve the N-Queens problem

Algorithm 1. Pseudocode of the Backtracking-Based N-Queens Solver

```
def solve_n_queens_util(board, row, size, sol):
    if row == size:
        sol.append(copy.deepcopy(board))
        return
    for col in range(size):
        if is_safe(board, row, col, size):
            board[row][col] = 1
            solve_n_queens_util(board, row + 1, size, sol)
            board[row][col] = 0
```

In this procedure, the algorithm traverses the chessboard row by row and evaluates the feasibility of placing a queen in each column using the **`is_safe`** function (see Algorithm 2). When a feasible position is identified, the algorithm recursively advances to the subsequent row in order to continue the solution construction process. If no admissible position can be found in the current search branch, the algorithm invokes the backtracking mechanism by removing the most recently placed queen and returning to the previous decision state. All valid configurations are recorded in the solutions list and subsequently presented through separate Tkinter windows, in which queens are visualized as red ♛ symbols on an alternating black-and-white chessboard. In addition, the program incorporates input validation and error-handling procedures to detect invalid user inputs or instances in which no feasible arrangement of queens exists. This design not only facilitates visual inspection of all admissible solutions but also provides an intuitive representation of how the backtracking

algorithm systematically explores the search space, enforces constraints, and reverses decisions during the search process.

Algorithm 2. Formal Specification of the Feasibility Checking Function (`is_safe`)

```
def is_safe(board, row, col, size):
    # Kiểm tra xem có queen nào trên cột này không
    for i in range(row):
        if board[i][col] == 1:
            return False
    # Kiểm tra đường chéo trái trên
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # Kiểm tra đường chéo phải trên
    for i, j in zip(range(row, -1, -1), range(col, size, 1)):
        if board[i][j] == 1:
            return False
    return True
```

3.2. Algorithm Procedure

Set N = 4

0	**0**	**0**	**0**
0	**0**	**0**	**0**
0	**0**	**0**	**0**
0	**0**	**0**	**0**

Solution 1

(1) At row 0, column 0 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (0,0).

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

(2) At row 1, column 0 is evaluated; a column conflict is detected due to an existing queen at position (0,0), thus the placement is infeasible (is_safe = False).

(3) At row 1, column 1 is evaluated; a conflict on the left diagonal is identified between positions (0,0) and (1,1), rendering the placement infeasible (is_safe = False).

(4) At row 1, column 2 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (1,2).

1	0	0	0
0	0	**1**	0
0	0	0	0
0	0	0	0

(5) At row 2, column 0 is evaluated; a column conflict is detected due to an existing queen at position (0,0), thus the placement is infeasible (is_safe = False).

(6) At row 2, column 1 is evaluated; a conflict on the right diagonal is identified between positions (1,2) and (2,1), rendering the placement infeasible (is_safe = False).

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

(7) At row 2, column 2 is evaluated; a column conflict is detected due to an existing queen at position (1,2), thus the placement is infeasible (is_safe = False).

(8) At row 2, column 3 is evaluated; a conflict on the left diagonal is identified between positions (1,2) and (2,3), rendering the placement infeasible (is_safe = False).

(9) As no feasible position is found at row 2, the algorithm performs backtracking by removing the queen previously placed at position (1,2).

(10) Returning to row 1, column 3 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (1,3).

1	0	0	0
0	0	0	**1**
0	0	0	0
0	0	0	0

(11) At row 2, column 0 is evaluated; a column conflict is detected due to an existing queen at position (0,0), thus the placement is infeasible (is_safe = False).

(12) At row 2, column 1 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (2,1).

1	0	0	0
0	0	0	1
0	**1**	0	0
0	0	0	0

(13) At row 3, column 0 is evaluated; a column conflict is detected due to an existing queen at position (0,0), thus the placement is infeasible (is_safe = False).

(14) At row 3, column 1 is evaluated; a column conflict is detected due to an existing queen at position (2,1), thus the placement is infeasible (is_safe = False).

(15) At row 3, column 2 is evaluated; a conflict on the left diagonal is identified with the queen at position (2,1), rendering the placement infeasible (is_safe = False).

(16) At row 3, column 3 is evaluated; a column conflict is detected due to an existing queen at position (1,3), thus the placement is infeasible (is_safe = False).

(17) As no feasible position is found at row 3, the algorithm performs backtracking by removing the queen previously placed at position (2,1).

1	0	0	0
0	0	0	1
0	**0**	0	0
0	0	0	0

(18) Returning to row 2, column 2 is evaluated; a conflict on the left diagonal is identified with the queen at position (0,0), rendering the placement infeasible (is_safe = False).

(19) Continuing at row 2, column 3 is evaluated; a column conflict is detected due to an existing queen at position (1,3), thus the placement is infeasible (is_safe = False).

(20) As no feasible position is found at row 2, the algorithm performs

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

backtracking by removing the queen previously placed at position (1,3).

(21) Since all columns at row 1 have been exhausted, the algorithm continues backtracking by removing the queen at position (0,0).

(22) At row 0, column 1 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (0,1).

0	**1**	0	0
0	0	0	0
0	0	0	0
0	0	0	0

(23) At row 1, column 0 is evaluated; a conflict on the right diagonal is detected due to an existing queen at position (0,1), thus the placement is infeasible (is_safe = False).

(24) At row 1, column 1 is evaluated; a column conflict is detected due to an existing queen at position (0,1), thus the placement is infeasible (is_safe = False).

(25) At row 1, column 2 is evaluated; a conflict on the upper-left diagonal is identified between positions (1,2) and (0,1), rendering the placement infeasible (is_safe = False).

(26) At row 1, column 3 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (1,3).

0	1	0	0
0	0	0	**1**
0	0	0	0
0	0	0	0

(27) At row 2, column 0 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (2,0).

0	1	0	0
0	0	0	1
1	0	0	0
0	0	0	0

(28) At row 3, column 0 is evaluated; a column conflict is detected due to an existing queen at position (2,0), thus the placement is infeasible (is_safe = False).

(29) At row 3, column 1 is evaluated; a conflict on the upper-left diagonal is identified with existing queens, rendering the placement infeasible (is_safe = False).

(30) At row 3, column 2 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (3,2).

0	1	0	0
0	0	0	1
1	0	0	0
0	0	**1**	0

(31) Hence, a valid solution is obtained.

Solution 2

(32) The algorithm performs backtracking by removing the queen at position (3,2).

(33) Continuing at row 3, column 3 is evaluated; a column conflict is detected due to an existing queen at position (1,3), thus the

0	1	0	0
0	0	0	1
1	0	0	0

placement is infeasible (is_safe = False).	0 0 0 **0**
(34) The algorithm performs backtracking by removing the queen at position (2,0).	
(35) Returning to row 2, column 1 is evaluated; a column conflict is detected due to an existing queen at position (0,1), thus the placement is infeasible (is_safe = False).	0 1 0 0 0 0 0 **1**
(36) Continuing at row 2, column 2 is evaluated; a conflict on the right diagonal is identified with the queen at position (1,3), rendering the placement infeasible (is_safe = False).	0 0 0 0 0 0 0 0
(37) Continuing at row 2, column 3 is evaluated; a column conflict is detected due to an existing queen at position (1,3), thus the placement is infeasible (is_safe = False).	
(38) The algorithm performs backtracking by removing the queen at position (1,3).	0 1 0 0 0 0 0 **0** 0 0 0 0 0 0 0 0
(39) As all columns at row 1 have been exhausted, the algorithm performs backtracking by removing the queen at position (0,1).	0 **0** 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(40) At row 0, column 2 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (0,2).	0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0
(41) At row 1, column 0 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (1,0).	0 0 1 0 **1** 0 0 0 0 0 0 0 0 0 0 0
(42) At row 2, column 0 is evaluated; a column conflict is detected due to an existing queen at position (1,0), thus the placement is infeasible (is_safe = False).	0 0 1 0
(43) At row 2, column 1 is evaluated; a conflict on the left diagonal is identified between positions (1,0) and (2,1), rendering the placement infeasible (is_safe = False).	1 0 0 0 0 0 0 **1** 0 0 0 0
(1) At row 2, column 2 is evaluated; a column conflict is detected due	

to an existing queen at position (0,2), thus the placement is infeasible (is_safe = False).

(2) At row 2, column 3 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (2,3).

(3) At row 3, column 0 is evaluated; a column conflict is detected due to an existing queen at position (1,0), thus the placement is infeasible (is_safe = False).

(4) At row 3, column 1 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (3,1).

(5) Hence, a second valid solution is obtained.

0	0	1	0
1	0	0	0
0	0	0	1
0	**1**	0	0

(6) The algorithm performs backtracking by removing the queen at position (3,1).

(7) Continuing at row 3, column 2 is evaluated; a conflict on the right diagonal is detected due to an existing queen at position (2,3), thus the placement is infeasible (is_safe = False).

(8) Continuing at row 3, column 3 is evaluated; a column conflict is detected due to an existing queen at position (2,3), thus the placement is infeasible (is_safe = False).

0	0	1	0
1	0	0	0
0	0	0	1
0	0	0	**0**

(9) The algorithm performs backtracking by removing the queen at position (2,3).

0	0	1	0
1	0	0	0
0	0	0	**0**
0	0	0	0

(10) the algorithm performs backtracking by removing the queen at position (1,0).

(11) Returning to row 1, column 1 is evaluated; a conflict on the right diagonal is detected due to an existing queen at position (0,2), thus the placement is infeasible (is_safe = False).

(12) Continuing at row 1, column 2 is evaluated; a column conflict is detected due to an existing queen at position (0,2), thus the placement is infeasible (is_safe = False).

(13) Continuing at row 1, column 3 is evaluated; a conflict on the left diagonal is identified with the queen at position (0,2), rendering the placement infeasible (is_safe = False).

0	0	1	0
0	0	0	**0**
0	0	0	0
0	0	0	0

(14) As all columns at row 1 have been exhausted, the algorithm performs backtracking by removing the queen at position (0,3).

0	0	0	**0**
0	0	0	0
0	0	0	0
0	0	0	0

(15) At row 0, column 3 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (0,3).

0	0	0	**1**
0	0	0	0
0	0	0	0
0	0	0	0

(16) At row 1, column 0 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (1,0).

0	0	0	1
1	0	0	0
0	0	0	0
0	0	0	0

(17) At row 2, column 0 is evaluated; a column conflict is detected due to an existing queen at position (1,0), thus the placement is infeasible (is_safe = False).

(18) At row 2, column 1 is evaluated; a conflict on the left diagonal is identified between positions (1,0) and (2,1), rendering the placement infeasible (is_safe = False).

(19) At row 2, column 2 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (2,2).

0	0	0	1
1	0	0	0
0	0	**1**	0
0	0	0	0

(20) At row 3, column 0 is evaluated; a column conflict is detected due to an existing queen at position (1,0), thus the placement is infeasible (is_safe = False).

(21) At row 3, column 1 is evaluated; a conflict on the right diagonal is detected due to an existing queen at position (2,2), rendering the placement infeasible (is_safe = False).

(22) At row 3, column 2 is evaluated; a column conflict is detected due to an existing queen at position (2,2), thus the placement is infeasible (is_safe = False).

(23) At row 3, column 3 is evaluated; a conflict on the left diagonal is identified with the queen at position (2,2), rendering the placement infeasible (is_safe = False).

(24) As no feasible position is found at row 3, the algorithm performs backtracking by removing the queen previously placed at position (2,2).

0	0	0	1
1	0	0	0
0	0	**0**	0
0	0	0	0

(25) Returning to row 2, column 3 is evaluated; a column conflict is detected due to an existing queen at position (0,3), thus the placement is infeasible (is_safe = False).

(26) As no feasible position is found at row 2, the algorithm performs backtracking.

0	0	0	1
1	0	0	0
0	0	0	**0**
0	0	0	0

(27) Returning to row 1, column 1 is evaluated and deemed feasible (is_safe = True); therefore, a queen is placed at position (1,1).

0	0	0	1
0	**1**	0	0
0	0	0	0
0	0	0	0

(28) At row 2, column 0 is evaluated; a conflict on the right diagonal is detected due to an existing queen at position (1,1), thus the placement is infeasible (is_safe = False).

(29) At row 2, column 1 is evaluated; a column conflict is detected due to an existing queen at position (1,1), thus the placement is infeasible (is_safe = False).

(30) At row 2, column 2 is evaluated; a conflict on the left diagonal is identified with the queen at position (1,1), rendering the placement infeasible (is_safe = False).

(31) At row 2, column 3 is evaluated; a column conflict is detected due to an existing queen at position (0,3), thus the placement is infeasible (is_safe = False).

0	0	0	1
0	0	0	0
0	0	0	**0**
0	0	0	0

(32) As no feasible position is found at row 2, the algorithm performs backtracking by removing the queen previously placed at position (1,1).

(33) Returning to row 1, column 2 is evaluated; a conflict on the right diagonal is detected due to an existing queen at position (0,3), thus the placement is infeasible (is_safe = False).

(34) Continuing at row 1, column 3 is evaluated; a column conflict is detected due to an existing queen at position (0,3), thus the placement is infeasible (is_safe = False).

(35) As all columns at row 1 have been exhausted, the algorithm performs backtracking by removing the queen at position (0,3).

0	0	0	**0**
0	0	0	0
0	0	0	0
0	0	0	0

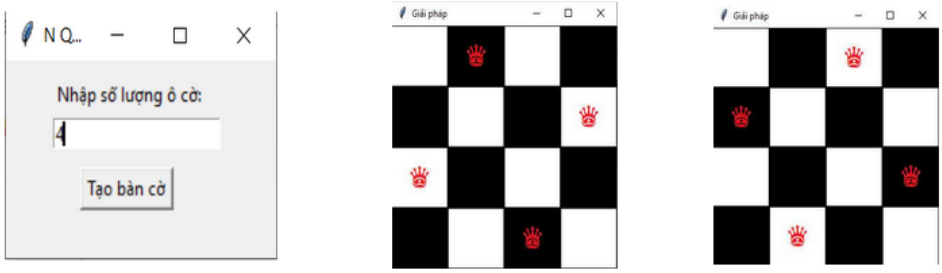

Figure 2. The user interface of the application (N=4)

4. Experimental Results and Evaluation

4.1. Experimental Design

The experimental design aims to evaluate both the effectiveness of search-space pruning and the computational efficiency of constraint representation in the Backtracking

algorithm. Two baseline methods are considered for comparison: Permutation-based Exhaustive Search and Bitmask Backtracking (BB). The permutation-based exhaustive search serves as a baseline for assessing the effectiveness of early pruning, whereas BB is used to isolate the effect of constraint representation while preserving the same search tree. To ensure a fair comparison, all methods are required to enumerate the complete set of valid solutions.

Experiments are conducted for board sizes $N \in \{8,10\}$. The recorded evaluation metrics include the number of valid solutions, the number of visited search nodes, the number of candidate position checks, execution time, pruning ratio, and runtime speed-up. All algorithms are implemented in Python and executed under identical experimental conditions using the same depth-first search strategy with left-to-right column ordering, without heuristics, symmetry-breaking techniques, or randomization. Execution time is measured using time.perf_counter(), while the overhead of the graphical user interface, board rendering, and input/output operations is excluded from the reported runtime. Conventional Backtracking and BB are each executed three times for every board size, and the median runtime is reported. The experiments are performed using CPython 3.8.3 on 64-bit Windows 10 (Build 19045) with an Intel64 Family 6 Model 142 processor, 8 GiB of physical memory, and single-threaded execution

4.2. Experimental Results

Table 2. Performance comparison of permutation-based exhaustive search, conventional Backtracking, and BB for the N-Queens problem

N	Method	Solutions	Search Workload	Candidate Checks	Runtime (s)
8	Exhaustive Search	92	40,320 perms.	–	0.17147
	Backtracking	92	2,057 nodes	15,720	0.01258
	BB	92	2,057 nodes	–	0.00458
10	Exhaustive Search	724	3,628,800 perms.	–	17.0396
	Backtracking	724	35,539 nodes	348,150	0.27044
	BB	724	35,539 nodes	–	0.06533

All three methods produced the expected numbers of valid solutions, namely 92 for **N=8** and 724 for **N=10**. This agreement confirms the correctness of the implementations and verifies that all methods performed the same complete enumeration task.

Table 2 shows that all three methods produced the expected numbers of valid solutions, namely 92 solutions for N=8 and 724 solutions for N=10, confirming the correctness of the implementations. Compared with permutation-based exhaustive search, Backtracking substantially reduced the search workload. For N=8, the number of examined states decreased from 40,320 permutations to 2,057 search nodes, while the execution time decreased from 0.171474 s to 0.012581 s. For N=10, exhaustive search examined 3,628,800 permutations, whereas Backtracking visited only 35,539 nodes, reducing the execution time from 17.039578 s to 0.270441 s. These results demonstrate that early constraint checking effectively eliminates infeasible branches, thereby significantly reducing the search effort. Table 2 also shows that BB and conventional Backtracking visited exactly the same number of search nodes. However, BB achieved substantially lower execution times, providing approximately 2.75× speed-up for N=8 and 4.14× speed-up for N=10. Since both methods explored the same search tree, the performance improvement can be attributed to the more efficient bitwise representation of constraints rather than to additional pruning.

5. Conclusion and Implications

This study investigated the Backtracking algorithm for the N-Queens problem from the perspective of search-space representation and constraint-based pruning. The experimental results demonstrate that all three implementations correctly reproduced the established solution counts of 92 for N=8 and 724 for N=10, confirming the correctness of the implementations. Compared with permutation-based exhaustive search, conventional Backtracking reduced the search workload from 40,320 to 2,057 examined states for N=8 and from 3,628,800 to 35,539 for N=10, while reducing the execution time from 0.171474 s to 0.012581 s and from 17.039578 s to 0.270441 s, respectively. Furthermore, although BB explored exactly the same search tree as conventional Backtracking, it achieved approximately 2.75× and 4.14× speed-up for N=8 and N=10, respectively, demonstrating that a more efficient constraint representation can significantly improve runtime without altering the search process.

The primary contribution of this work is not the introduction of a new algorithm but the systematic modeling and empirical analysis of Backtracking on a canonical constraint satisfaction problem. The results clearly distinguish two sources of performance improvement: search-space pruning substantially reduces the number of explored states, whereas Bitmask representation improves computational efficiency by lowering the cost of constraint checking. These findings provide a reproducible reference framework for understanding and evaluating Backtracking-based approaches in constraint satisfaction problems.This study has several limitations. The experimental evaluation is restricted to board sizes N=8 and N=10, and only complete solution enumeration is considered. Future work will extend the evaluation to larger problem instances and investigate advanced techniques, including symmetry breaking, heuristic variable ordering, forward checking, and constraint propagation, to further improve search efficiency and generalize the analytical framework to more complex constraint satisfaction and combinatorial optimization problems.

References

Bell, J., & Stevens, B. (2009). A survey of known results and research areas for the N-Queens problem. *Discrete Mathematics, 309*(1), 1–31. https://doi.org/10.1016/j.disc.2007.12.043

Gent, I. P., & Walsh, T. (1999). CSPLib: A benchmark library for constraints. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)* (pp. 480–481). Springer.

Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*(3), 263–313. https://doi.org/10.1016/0004-3702(80)90051-X

Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI Magazine, 13*(1), 32–44. https://doi.org/10.1609/aimag.v13i1.976

Lijo, V. P., & Jose, J. T. (2015). Solving N-Queen problem by prediction. *International Journal of Computer Science and Information Technologies, 6*(4), 3844–3848.

Poole, D. L., & Mackworth, A. K. (2023). Artificial Intelligence: *Foundations of Computational Agents (3rd ed.)*. Cambridge University Press.

Russell, S. J., & Norvig, P. (2020*). Artificial intelligence: A modern approach* (4th ed.). Pearson.

Schaefer, T. J. (1978). The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing* (pp. 216–226). ACM. https://doi.org/10.1145/800133.804350

Simonis, H. (2005). Sudoku as a constraint problem. In *Proceedings of the CP Workshop on Modelling and Reformulating Constraint Satisfaction Problems* (pp. 13–27).

Thada, V., & Dhaka, S. (2014). Performance analysis of N-Queen problem using backtracking and genetic algorithm techniques. *International Journal of Computer Applications, 102*(7), 26–29. https://doi.org/10.5120/17891-8867

Tsang, E. (1993). *Foundations of constraint satisfaction*. Academic Press.

Wu, Q., & Tian, Y. (2018). An improved backtracking algorithm for solving N-Queens problem. In *Advances in Intelligent Systems Research* (Vol. 157, pp. 101–105). Atlantis Press. https://doi.org/10.2991/emehss-18.2018.21