# Exploring CSP Algorithms: The N-Queen's Problem

*Authors:*
Mehyar MLAWEH
mehyar.mlaweh@dauphine.tn

Nadia BENYOUSSEF
nadia.benyoussef@dauphine.tn

# Contents


**1 Introduction to Constraint Satisfaction Problems (CSPs) 2**
1.1 Motivation . . . 2
1.2 Definition and Components . . . 2
1.2.1 Definition . . . 2
1.2.2 Components . . . 2
1.3 Conclusion . . . 3

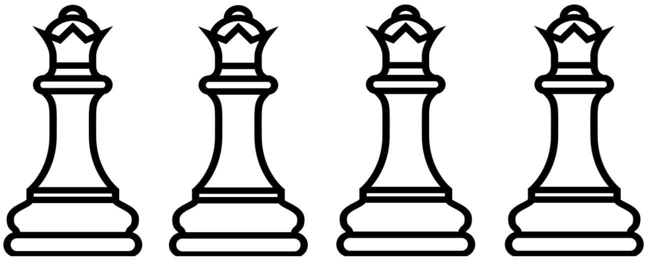
**2 Algorithms for Solving CSPs 4**
2.1 AC3 . . . 4
2.1.1 Functioning . . . 4
2.1.2 Advantages . . . 4
2.1.3 Complexity . . . 5
2.1.4 Pseudo Code . . . 5
2.2 Depth-First Search with Backtracking . . . 6
2.2.1 Functioning . . . 6
2.2.2 Advantages . . . 7
2.2.3 Complexity . . . 7
2.2.4 Pseudo Code . . . 8
2.3 Forward Checking . . . 8
2.3.1 Functioning . . . 8
2.3.2 Advantages . . . 9
2.3.3 Complexity . . . 9
2.3.4 Pseudo Code . . . 10
2.4 Conclusion . . . 10

**3 Solving the N-Queens Problem 11**
3.1 CSP Problem Formalization . . . 11
3.2 Implementation of Algorithms . . . 13
3.2.1 AC3 Algorithm . . . 13
3.2.1.1 Execution . . . 13
3.2.1.2 Implementation . . . 14
3.2.2 Forward Checking . . . 15
3.2.2.1 Execution . . . 15
3.2.2.2 Implementation . . . 16
3.2.3 Depth-First Search with Backtracking . . . 20
3.2.3.1 Execution . . . 20
3.2.3.2 Implementation . . . 21
3.3 Conclusion . . . 23

**4 Evaluation and Comparison of Methods 24**
4.1 Experimental Setup . . . 24
4.2 Results and Observations . . . 24
4.3 Discussion . . . 28
4.4 Conclusion . . . 28

**Annex A: Jupyter Notebook 32**

# Problem Statement



The N-Queens puzzle, a classic chessboard problem, tasks us with placing N queens on an NxN grid so that no two queens threaten each other. Despite its apparent simplicity, the puzzle poses significant challenges due to its vast number of potential configurations. Traditional trial-and-error methods quickly become unfeasible as the board size increases.

To overcome these challenges, we use algorithmic techniques and Constraint Satisfaction Programming as two main strategies. In order to effectively reduce the search space, CSP divides the problem into manageable parts by specifying variables, domains, and constraints. Then, algorithms such as Depth-First Search and Backtracking work their way through these options, methodically examining various combinations in an effort to identify an answer.

We hope to address the N-Queens problem's complexity by merging various methods and investigating the trade-off between computer capacity and human intuition. Our objective is to identify approaches that not only solve the puzzle but also provide insight into more general issues related to optimization and problem-solving in the context of combinatorial puzzles.

# Chapter 1

# Introduction to Constraint Satisfaction Problems (CSPs)

## 1.1 Motivation

In many different disciplines, Constraint Satisfaction Problems (CSPs) are crucial for handling complex decision-making circumstances. They offer a methodical framework for expressing constraints and identifying effective ways to satisfy them.

**Importance in Problem-Solving**
CSPs are used in scheduling, planning, resource allocation, and design. By effectively allocating resources while respecting limitations, they improve decision-making processes and optimize operations.

**Real-World Applications**
Industries employ CSP-based methodologies to optimize operations and streamline processes, addressing intricate problems while adhering to regulatory requirements.

## 1.2 Definition and Components

### 1.2.1 Definition

A CSP is a computational challenge where the goal is to assign values to variables in a way that satisfies all the constraints. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. [Feder, 2024]

This problem class is extensively researched in artificial intelligence and operations research.[Minton et al., 1992]

CSPs in general are **NP-complet** and **P-Complete**. [Gent et al., 2018]

### 1.2.2 Components

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$ where:

- $X = X1, \ldots, Xn$ is a set of **variables** : Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints.
- $D = D1, \ldots, Dn$ is a set of their respective **domains** of values : The range of potential values that a variable can have is represented by domains.
- $C = C1, \ldots, Cm$ is a set of **constraints** : The guidelines that control how variables relate to one another are known as constraints.

## 1.3 Conclusion

In summary, CSPs provide a powerful framework for modeling and solving complex problems in computer science, artificial intelligence, and many other fields, making them a valuable tool for problem-solving in diverse contexts.

# Chapter 2

# Algorithms for Solving CSPs

In the literature, numerous algorithms have been proposed to resolve CSPs, showcasing the varied methodologies for resolving these complex computational problems. In this report, we focus on three fundamental algorithms: **Arc Consistency Algorithm 3 (AC3)**, **Depth-First Search with Backtracking**, and **Forward Checking**. While many other algorithms exist, our attention is directed towards these three, chosen for their widespread applicability and effectiveness in addressing a wide range of CSP instances.

## 2.1 AC3

### 2.1.1 Functioning

The AC-3 algorithm [Mackworth, 1977]tackles CSPs by operating on constraints, variables, and their domains (scopes). A constraint is a relationship that dictates which values a variable can hold, potentially considering values of other related variables. During execution, the CSP can be visualized as a directed graph where edges, or arcs, represent symmetric constraints linking variables.

AC-3 works by systematically examining paths between each pair of variables $(x, y)$. It then removes values from the domain of $x$ that violate the constraint with y. This effectively prunes the search space by eliminating values that would lead to inconsistencies. The algorithm maintains a list of arcs that require checking, which becomes finite at each step due to the limited size of variable domains. This list ensures all relevant constraint checks are performed throughout the process.[Van Hentenryck et al., 1992]

### 2.1.2 Advantages

- **Faster Solutions:** By pruning inconsistent values, AC3 shrinks the search space, leading to quicker problem-solving.
- **Simpler Constraints:** Representing constraints as arcs simplifies their handling and helps identify inconsistencies.

- **Compatible with Solvers:** AC3 preprocesses problems for existing solvers, leveraging the simplified constraint landscape.
- **Scales Well:** AC3 tackles problems with large search spaces and complex constraints, making it versatile.[Toolify AI, 2022]

#### 2.1.3 Complexity

The complexity of the AC3 algorithm primarily depends on the structure and constraints of the CSP instance being solved. However, the worst-case time complexity of the AC3 algorithm can be analyzed as follows:

Let $n$ be the number of variables in the CSP, and $d$ be the maximum domain size among all variables.

**Initialization:** The initialization step typically requires $O(nd^2)$ time, as it involves setting up the initial queue of arcs.

**Main Loop:** In the main loop of the algorithm, each arc is processed at most once. For each arc, the algorithm may need to revise the domains of the connected variables. In the worst case, each domain revision operation takes $O(d^2)$ time, resulting in a total worst-case complexity of $O(nd^3)$ for the main loop.

Overall, the worst-case time complexity of the AC3 algorithm is $O(nd^3)$.

#### 2.1.4 Pseudo Code

**Algorithm 1:** Revise function - AC3

**Input:** CSP, $X_i, X_j$
**Output:** Domain of $X_i$ and boolean indicating if any change was made
anyChangeToDomain$_i$ ← false ; `// Initialize change flag`
**for** *each $x$ in Domain($X_i$)* **do**
  **if** *no value $y$ in Domain($X_j$) allows $(x, y)$ to satisfy constraint between $(X_i, X_j)$* **then**
    delete $x$ from Domain($X_i$) ; `// Delete` $x$ `if no value in` $X_j$ `allows` $(x, y)$
    anyChangeToDomain$_i$ ← true ; `// Set change flag`
  **end**
**end**
**return** Domain of $X_i$ and anyChangeToDomain$_i$ ; `// Return updated domain and change flag`

**Algorithm 2:** AC3 Algorithm

**Input:** CSP
**Output:** CSP, possibly with reduced domains for variables, or inconsistent
**Local variables:** queue, initially queue of all arcs (binary constraints in CSP);

```
while queue is not empty do
    (X_i, X_j) ← Remove-First(queue) ; // Select and remove first arc from the queue
    (domain_i, anyChangeToDomain_i) ← Revise(CSP, X_i, X_j) ; // Revise domains of variables involved in the arc
    if anyChangeToDomain_i then
        if size(domain_i) = 0 then
            return inconsistent ; // Return inconsistent if domain becomes empty
        end
        else
            for each X_k in Neighbors(X_i) except X_j do
                add (X_k, X_i) to queue ; // Add arcs to queue for neighbors of X_i
            end
        end
    end
end
return CSP ; // Return CSP with reduced domains or inconsistent
```

## 2.2 Depth-First Search with Backtracking

### 2.2.1 Functioning

Backtracking is a powerful depth-first search algorithm(DFS)[Gouda and Zaki, 2001] used in CSPs. It systematically explores the solution space by assigning values to variables one at a time. However, unlike simple generate-and-test, backtracking incorporates a limited form of arc consistency to improve efficiency.

Here's how it works:

After selecting a variable, backtracking assigns a value from its domain. It then checks for constraint violations by considering only those constraints involving the newly assigned variable and previously instantiated variables. Since instantiated variables have only one possible value remaining in their domain, this check becomes simpler and focused. If any constraint is violated (meaning no value in the remaining domains satisfies the constraint), backtracking acknowledges the inconsistency and returns to the previous variable. It then tries a different value for that variable, effectively pruning the search space and avoiding dead-end paths. This iterative process continues until a complete solution is found that satisfies all constraints, or all possibilities for variable assignments have been exhausted.

In essence, backtracking leverages a basic form of arc consistency to avoid pursuing infeasible assignments and focuses its search on promising branches of the solution space. This combination of systematic exploration and limited consistency checking makes backtracking a powerful technique for finding solutions to CSPs.

#### 2.2.2 Advantages

- **Systematic Exploration:** Backtracking ensures all possible assignments are explored, guaranteeing a solution if one exists.
- **Constraint Enforcement:** It checks for constraint violations, preventing solutions that wouldn't satisfy all relationships.
- **Dead-End Detection:** Backtracking identifies infeasible paths early on, focusing on promising solutions.

#### 2.2.3 Complexity

The time complexity of DFS with Backtracking is often expressed in terms of the number of **edges** ($E$) and **vertices** ($V$) in the search space, which are typically denoted as $O(E + V)$, when implemented using an adjacency list.

### 2.2.4 Pseudo Code

**Algorithm 3:** Backtracking

```
Input: CSP net, Assignment a
Output: Assignment
Function Backtracking(CSP net, Assignment a):
    if is_complete(a) then
        return a;
    end
    var ← select_unassigned_variable(net) ;        // Select next
     unassigned variable
    foreach val in domain of var do
        a.assign(var, val) ;                       // Assign value to variable
        if consistent(net, a) then
            result ← Backtracking(net, a);
            if result is None then
                a.unassign(var) ;        // Backtrack if no solution found
            end
            else
                return result ;                    // Return solution if found
            end
        end
    end
    return None ;         // No solution found for this variable
```

## 2.3 Forward Checking

### 2.3.1 Functioning

Forward checking[Haralick and Elliott, 1979] is a powerful technique used in constraint satisfaction problems (CSPs) to enhance backtracking and prevent future conflicts. It achieves this by applying a type of local consistency check, focusing on the constraints between the currently assigned variable and unassigned variables (future variables).

Here's how it works:
When a value is assigned to the current variable, forward checking examines the domains of future variables. Any value in a future variable's domain that conflicts with the current assignment is temporarily removed. This proactive approach significantly prunes the search space. A key advantage is the immediate detection of inconsistencies: if a future variable's domain becomes empty after forward checking, it signifies that the current partial solution is invalid. This allows the algorithm to backtrack much earlier than with basic backtracking, leading to a

more efficient search.

It's important to note that due to the nature of forward checking, once a new variable is considered, its remaining values are guaranteed to be consistent with previously assigned variables. This eliminates the need for redundant checks against past assignments, further streamlining the process.

#### 2.3.2 Advantages

Here are 3 concise advantages of forward checking:

- **Faster failure detection:** Catches conflicts early, avoiding exploration of dead-end paths.
- **Less backtracking:** Prunes the search space, reducing unnecessary backtracking steps.
- **Efficient focus:** Only checks relevant constraints, avoiding redundant checks.

#### 2.3.3 Complexity

The time complexity of Forward Checking (FC) in Constraint Satisfaction Problems (CSPs) is typically expressed as $O(d^2nm)$, where:

- $d$ is the maximum domain size among all variables,
- $n$ is the number of variables in the CSP, and
- $m$ is the maximum number of constraints involving any single variable.

### 2.3.4 Pseudo Code

**Algorithm 4:** Forward Checking

```
Input: CSP net, Assignment a
Output: Assignment
Function ForwardChecking(CSP net, Assignment a):
    if the assignment a is complete then
        return a;
    end
    var ← next non-assigned variable ;     // Select next non-assigned variable
    foreach val in the domain of var do
        if {var = val} does not violate any constraint in net then
            Add {var = val} to a ;                       // Add assignment to a
            foreach variable v connected to var do
                Apply arc-consistency from v to var ;    // Apply arc-consistency
            end
            result ← ForwardChecking(net, a) ;           // Recursive call
            if result = None then
                remove {var = val} from a ;              // Backtrack if failure
            end
            else
                return result ;                          // Return result if found
            end
        end
    end
    return None ;                          // No solution found for this variable
```

## 2.4 Conclusion

In the next chapters, we will apply these concepts to our project problem, testing the algorithms discussed to resolve real-world constraints. By doing so, we aim to not only deepen our understanding of CSPs but also to provide practical solutions to complex problems.

# Chapter 3

# Solving the N-Queens Problem

## 3.1 CSP Problem Formalization

The N-Queens problem can be formalized as a Constraint Satisfaction Problem (CSP). In this formalization, we define variables, domains, and constraints that represent the requirements of the problem.

**Variables:**
Let $X$ be the set of variables representing the placement of queens on the chessboard. Each variable $X_i$ corresponds to a column on the chessboard, where $i$ ranges from 0 to $N-1$, with $N$ being the size of the chessboard.

**Domains:**
The domain of each variable $X_i$ consists of the integers from 0 to $N-1$, representing the rows where the queen in column $i$ can be placed.

**Constraints:**
The constraints ensure that no two queens threaten each other, meaning no two queens share the same row, column, or diagonal. Thus, we impose the following constraints:

1. **Row Constraint:** Each queen must be placed in a different row. This constraint can be represented as $X_i \neq X_j$ for all $i$ and $j$ where $i \neq j$.
2. **Column Constraint:** Each queen must be placed in a different column. This is implicit in the variable definition, as each variable represents a different column.
3. **Diagonal Constraint:** No two queens should be placed on the same diagonal. This can be expressed as $|X_i - X_j| \neq |i - j|$ for all $i$ and $j$ where $i \neq j$.

To visually demonstrate the problem formulation, let's present an illustration of the variables, their corresponding domains, and the constraints they must respect.

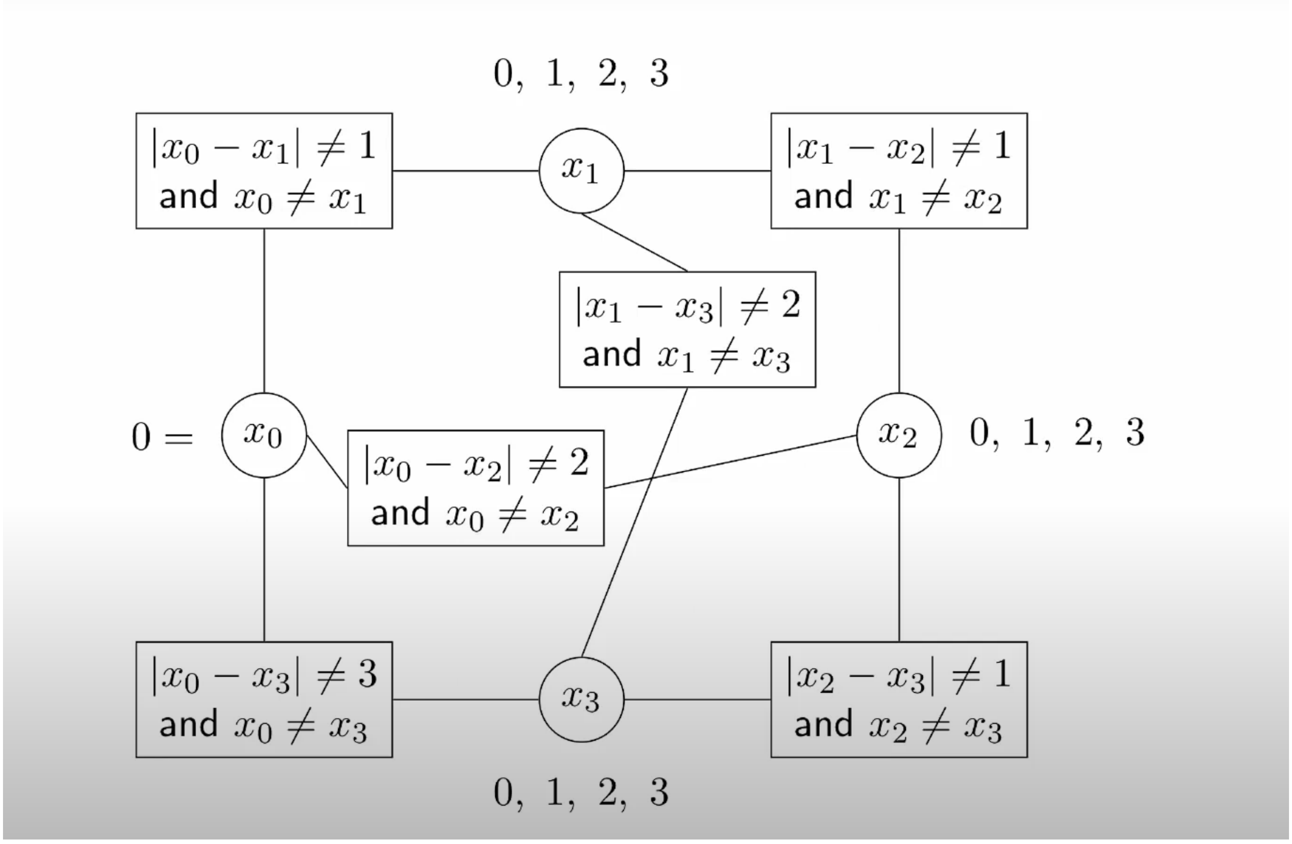



Figure 3.1: Illustration of the N-Queens Problem with $N = 4$

### Implementation of Problem Formalization in Python

```
class NQueensCSP:
    def __init__(self, N):
        # The given number of queens N
        self.N = N
        # Define the variables as the columns of the
          chessboard
        self.variables = list(range(N))
        # Set the domain of each variable to be the
          rows of the chessboard
        self.domains = {i: list(range(N)) for i in
          range(N)}
        #List of constarints
        self.constraints = []

        # Row constraint
        for i in range(N):
            for j in range(i + 1, N):
                self.constraints.append((i, j, lambda x
```

```
                      , y: x != y))

        # Diagonal constraint
        for i in range(N):
            for j in range(i + 1, N):
                self.constraints.append((i, j, lambda x
                      , y, i=i, j=j: abs(x - y) != abs(i -
                       j)))

    def checkConstraint(self, var1, val1, var2, val2):
        for c in self.constraints:
            if (var1 == c[0] and var2 == c[1]) or (var1
                 == c[1] and var2 == c[0]):
                if not c[2](val1, val2):
                    return False
        # If no constraints are violated, return True
        return True
```

### 3.2 Implementation of Algorithms

Now that we have formalized the N-Queens problem as a CSP, we can proceed to implement various algorithms to solve it. In this section, we will discuss the implementation details of the AC3 algorithm, Forward Checking, and DFS with backtracking.

#### 3.2.1 AC3 Algorithm

##### 3.2.1.1 Execution

In this section, we present the execution table detailing the steps of the AC-3 algorithm for the N queens problem with $N = 3$. Table 3.1 illustrates the reduction of variable domains and the discovery of solutions throughout the algorithm's execution.

- Queue for $N = 3$:
  - $(X0, X1)$
  - $(X1, X0)$
  - $(X0, X2)$
  - $(X2, X0)$
  - $(X1, X2)$

– $(X2, X1)$

Table 3.1: Execution Table for $N = 3$

| Queue | Domain of $X_0$ | Domain of $X_1$ | Domain of $X_2$ |
|---|---|---|---|
| $(X0, X1)$ | $\{0, 1, 2\}$ | $\{0, 1, 2\}$ | $\{0, 1, 2\}$ |
| $(X1, X0)$ | $\{0, 2\}$ | $\{0, 1, 2\}$ | $\{0, 1, 2\}$ |
| $(X0, X2)$ | $\{0, 2\}$ | $\{0, 2\}$ | $\{0, 1, 2\}$ |
| $(X2, X0)$ | $\{0, 2\}$ | $\{0, 2\}$ | $\{0, 1, 2\}$ |
| $(X1, X2)$ | $\{0, 2\}$ | $\{0, 2\}$ | $\{1\}$ |
| $(X2, X1)$ | $\{0, 2\}$ | $\{\}$ | $\{1\}$ |

Inconsistency found: **Domain of $X_1$ is empty**.

#### 3.2.1.2 Implementation

```python
def ac3(csp, queue=None):
    #  Initialize if not given with all arcs (Xi, Xj) where Xi is a variable and Xj is its neighbor
    if queue is None:
        queue = [(Xi, Xj) for Xi in csp.variables for Xj in csp.neighbors(Xi)]

    # Until the queue is empty
    while queue:
        # Remove the first arc
        (Xi, Xj) = queue.pop(0)
        # Attempt to revise the domains based on the constraint between Xi and Xj
        if revise(csp, Xi, Xj):
            # If the domain of Xi becomes empty, then inconsistent
            if len(csp.domains[Xi]) == 0:
                return False
            # If the domain of Xi is revised add arcs (Xk, Xi) to the queue for all neighbors Xk of Xi except Xj
            for Xk in csp.neighbors(Xi):
                if Xk != Xj:
                    queue.append((Xk, Xi))

```

```
    # If no domain becomes empty, return consistent
      TRUE
    return True


def revise(csp, Xi, Xj):
    revised = False
    # Iterate over each value in the domain of variable
       Xi
    for x in list(csp.domains[Xi]):
        # Check if there is no value y in the domain of
           Xj that satisfies the constraint between Xi
           and Xj
        if not any(csp.checkConstraint(Xi, x, Xj, y)
          for y in csp.domains[Xj]):
            # If no such value is found, remove x from
              the domain of Xi
            csp.domains[Xi].remove(x)
            # Set the flag to indicate that a revision
              is made
            revised = True
    return revised
```

### 3.2.2 Forward Checking

#### 3.2.2.1 Execution

With a view to demonstrating how the implemented approach works, again the visualization part of FC algorithm has been given, as illustrated in Fig.4. (Illustration taken from [Ayub et al., 2017])

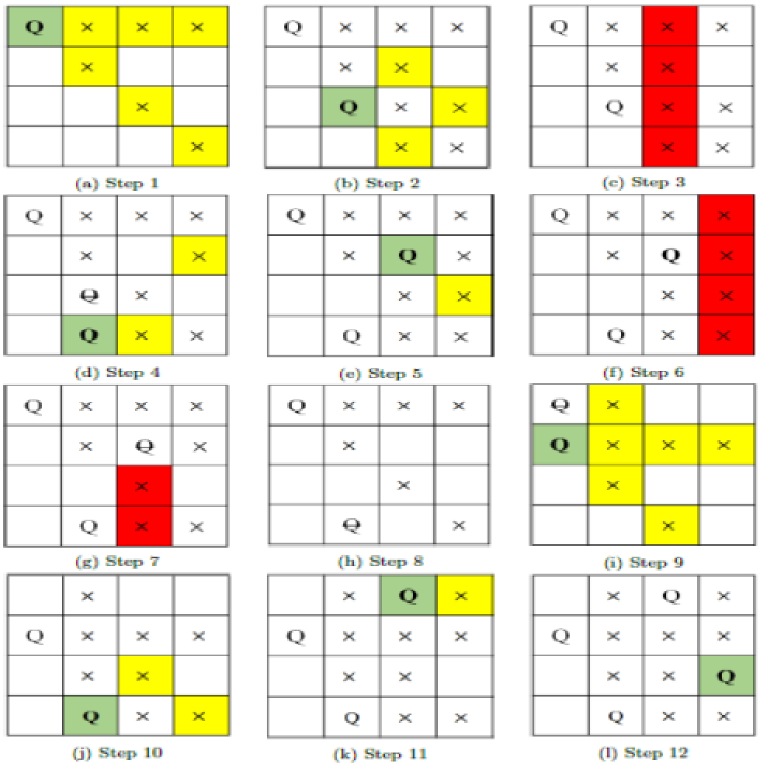



Figure 3.2: Forward checking execution for N = 4

#### 3.2.2.2 Implementation

```
class ForwardChecking:
    def __init__(self, csp):
        """
        Constructor for the ForwardChecking class.

        Args:
            csp: An instance of the CSP class representing the Constraint Satisfaction Problem.
        """
        self.csp = csp
        self.solutions = []

    def forward_checking(self):
        """
        Main method to perform forward checking and find all solutions.

        Returns:
            list: A list of all solutions found.
```

```
        """
        assignment = {}
        # Call the recursive forward checking method with the initial assignment
        self.solutions = self._recursive_forward_checking(assignment)
        return self.solutions

    def _recursive_forward_checking(self, assignment):
        """
        Recursive function to perform forward checking.

        Args:
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            list: A list of all solutions found from the current assignment.
        """
        # Base case: if all variables are assigned
        if len(assignment) == self.csp.N:
            return [assignment.copy()]  # Return the current assignment as a solution

        var = self.select_unassigned_variable(assignment)  # Select an unassigned variable
        solutions = []  # Initialize an empty list to store solutions

        if var is not None:  # Check if there is an unassigned variable
            for val in range(self.csp.N):  # Iterate over possible values for the variable
                if not self.violates_constraints(var, val, assignment):  # Check if value violates constraints
                    assignment[var] = val  # Assign the value to the variable
```

```python
                    neighbors = self.get_neighbors(var, assignment)  # Get neighboring variables
                    forward_check = self._recursive_forward_checking(assignment)  # Recursively explore further assignments
                    solutions.extend(forward_check)  # Add solutions from further exploration
                    del assignment[var]  # Backtrack: remove the assigned value

        return solutions  # Return the list of solutions found

    def select_unassigned_variable(self, assignment):
        """
        Method to select an unassigned variable.

        Args:
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            int or None: The selected unassigned variable, or None if all variables are assigned.
        """
        unassigned_vars = [var for var in self.csp.variables if var not in assignment]
        if unassigned_vars:
            return min(unassigned_vars, key=lambda var: len(self.csp.domains[var]))
        else:
            return None

    def violates_constraints(self, var, value, assignment):
```

```python
        """
        Method to check if assigning a value to a variable violates constraints.

        Args:
            var (int): The variable to assign a value to.
            value (int): The value to assign to the variable.
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            bool: True if the assignment violates constraints, False otherwise.
        """
        for constraint_var in assignment:
            if assignment[constraint_var] == value or abs(var - constraint_var) == abs(value - assignment[constraint_var]):
                return True
        return False

    def get_neighbors(self, var, assignment):
        """
        Method to get neighboring variables that are not yet assigned.

        Args:
            var (int): The variable to find neighbors for.
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            list: A list of neighboring variables.
        """
```

```python
        return [neighbor for neighbor in self.csp.
          variables if neighbor != var and neighbor
          not in assignment]

    def restore_domains(self, neighbors, assignment):
        """
        Method to restore the domains of neighboring
          variables.

        Args:
            neighbors (list): A list of neighboring
              variables.
            assignment (dict): A dictionary
              representing the current assignment of
              variables.
        """
        for neighbor in neighbors:
            self.csp.domains[neighbor].extend([val for
              val in range(self.csp.N) if val not in
              self.csp.domains[neighbor]])
```

### 3.2.3 Depth-First Search with Backtracking

#### 3.2.3.1 Execution

This section demonstrates the application of DFS with Backtracking.

Figure 3.3 showcases the exploration of solutions using DFS with backtracking for$N = 3$. Despite exhaustive search, no solution was found within the constraints.

However, for N = 4, Figure 3.4displays the successful application DFS with backtracking, leading to the discovery of a valid solution, as shown in Figure 3.5.

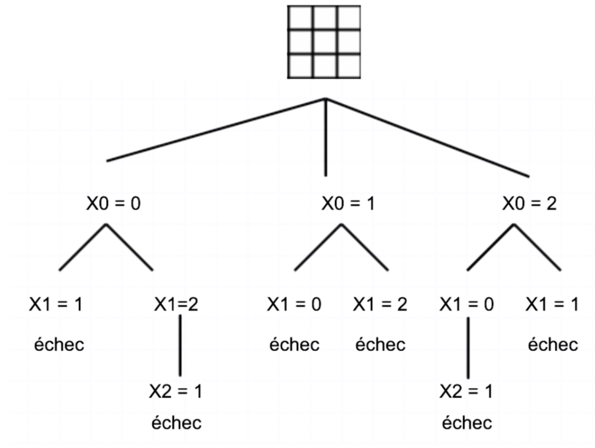



Figure 3.3: DFS execution for $N = 3$. No solution found.

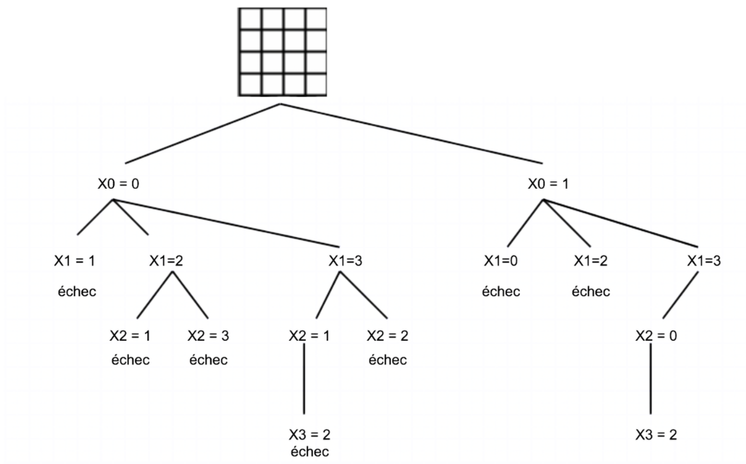



Figure 3.4: Successful DFS for $N = 4$.

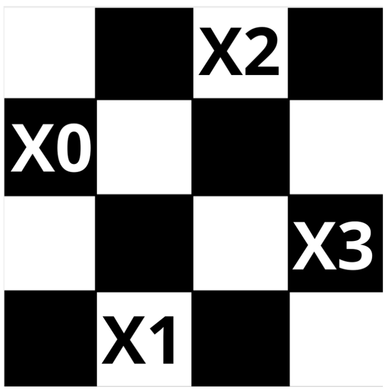



Figure 3.5: Solution configuration for $N = 4$.

#### 3.2.3.2 Implementation

```
class DFS:
def __init__(self, csp):
    self.csp = csp
```

```
        self.solutions = []

    def solve_all(self):
        self.solutions = []  # Reset solutions list
        self.backtrack(0, [])  # Start the backtracking process from column 0
        return self.solutions

    def backtrack(self, col, solution):
        # If all columns are explored (base case)
        if col == self.csp.N:
            # Add the solution to the list of solutions
            self.solutions.append(solution[:])
            return

        # Iterate over each possible row in the current column
        for row in self.csp.domains[col]:
            # Check if placing a queen in the current position is safe
            if self.is_safe(col, row, solution):
                # Place the queen in the current position
                solution.append(row)
                # Recursively move to the next column
                self.backtrack(col + 1, solution)
                # Backtrack by removing the last queen placed
                solution.pop()

    def is_safe(self, col, row, solution):
        # Iterate over previously placed queens
        for prev_col, prev_row in enumerate(solution):
            # Check if the new queen conflicts with any previous queen
            if not self.csp.checkConstraint(col, row, prev_col, prev_row):
                return False
            # Check if the new queen is on the same diagonal as any previous queen
```

```python
            if abs(col - prev_col) == abs(row -
              prev_row):
                return False
        return True
```

### 3.3 Conclusion

In summary, we've formulated the N-Queens problem as a CSP, defining variables, domains, and constraints. We've implemented efficient solving algorithms including AC3, Forward Checking, and DFS with Backtracking. Now, we aim to compare their performances through executions for different board sizes, and that what we going to see in the next chapter.

# Chapter 4

# Evaluation and Comparison of Methods

## 4.1 Experimental Setup

In this section, we outline the experimental setup used to evaluate the performance of the algorithms for solving the N-Queens problem. The experiments were conducted on a standard laptop computer with the following specifications:

- Processor: AMD Ryzen, 4.0 GHz, 8 cores
- RAM: 16 GB
- Operating System: Windows 11
- Programming Language: Python 3.11.4

We implemented three different algorithms, DFS with backtracking, Forward Checking , and AC3, to solve the N-Queens problem. Each algorithm was tested for various values of N (number of queens), ranging from **2** to **13**.(Check Annex A)

The experiments were designed to measure the **memory consumption** and **execution time** of each algorithm under different problem sizes.

## 4.2 Results and Observations

In this section, we present the results obtained from the experiments.

We observed the following results as shown in the Figures 4.1 & 4.2.

**Observations**: The DFS algorithm, despite being a fundamental approach, demonstrates limitations in solving larger instances of the N-Queens problem efficiently. As the problem size increases, the execution time and memory consumption also increase significantly, as evidenced by the results.

The FC algorithm shows relatively better performance compared to DFS, particularly in terms of execution time and memory consumption. However, it still exhibits notable inefficiencies for larger problem sizes.

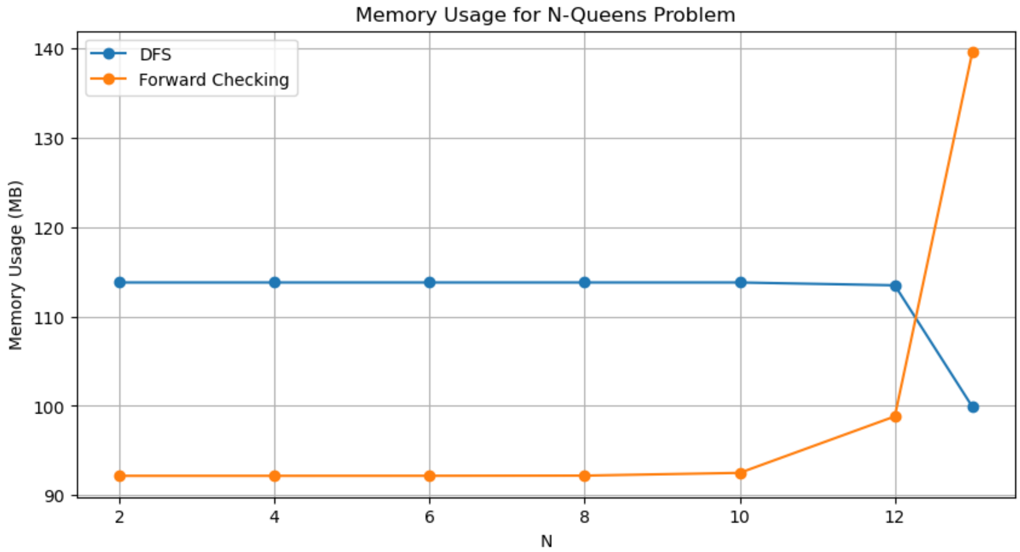



Figure 4.1: Memory Usage FC vs. Simple DFS

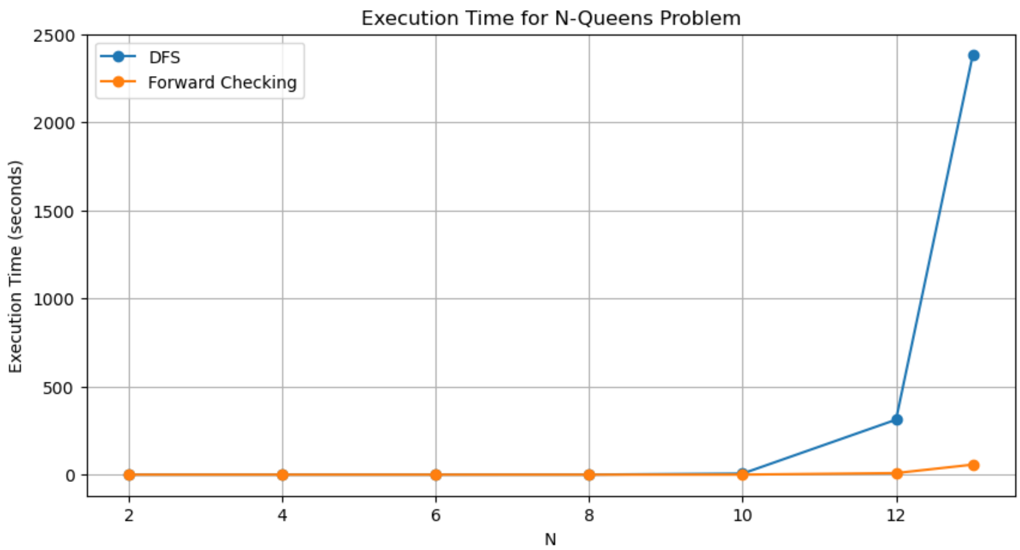



Figure 4.2: Execution Time FC vs. Simple DFS

Given the observed limitations of DFS and FC algorithms, we aim to enhance the DFS algorithm's efficiency by incorporating the **AC3 algorithm**. This approach intends to reduce the search space by enforcing constraints, potentially mitigating the scalability issues observed in the previous experiments.

We observed the new results as shown in the Figures 4.3 & 4.4.

**Observations**: The DFSAC3 algorithm demonstrates improvements in memory consumption compared to the basic DFS algorithm. However, the execution time remains largely unchanged, indicating that while AC3 helps reduce memory usage, it does not significantly impact the computational efficiency of DFS.

The FC algorithm consistently performs with slight memory usage variations compared to DFSAC3. However it still encountering challenges in efficiently solving larger N-Queens instances.

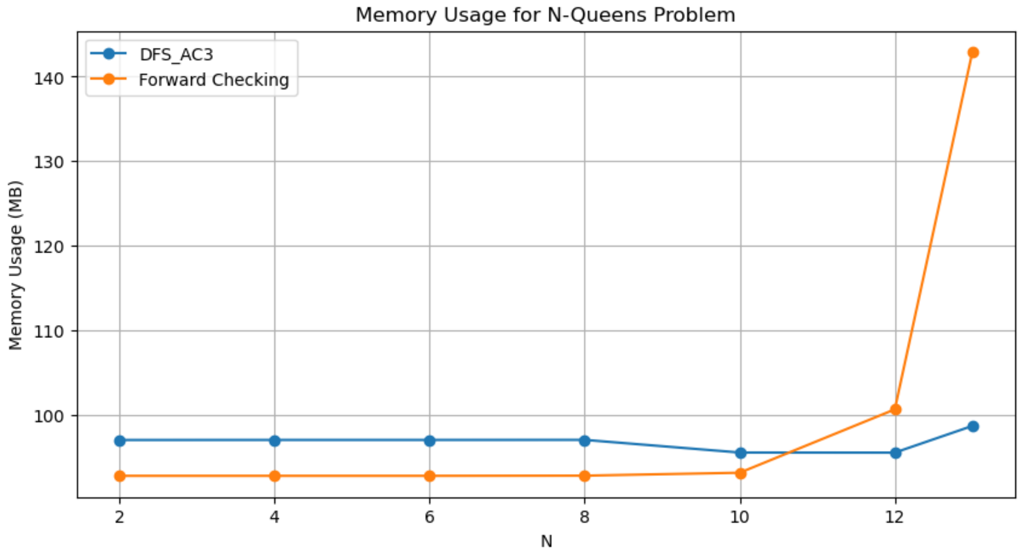



Figure 4.3: Memory Usage DFSAC3 vs. FC

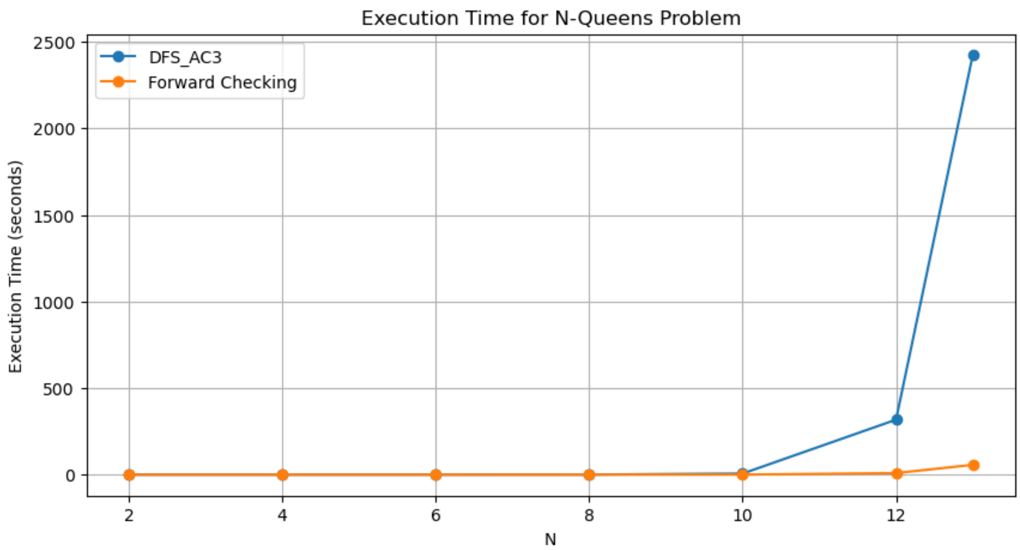



Figure 4.4: Execution Time DFSAC3 vs. FC

Table 4.1 presents a summary of the results obtained for $N = 2$ to 13, including the number of solutions found, execution time, and memory usage for the simple DFS, DFSAC3 and FC .

Table 4.1: Summary of Algorithm Results

| **N** | **Algorithm** | **Solutions** | **Time (s)** | **Memory (MB)** |
|---|---|---|---|---|
| 4 | DFSAC3 | 2 | 0.0 | 96.988 |
| | FC | 2 | 0.0 | 92.738 |
| | DFS | 2 | 0.0 | 113.804 |
| 6 | DFSAC3 | 4 | 0.006 | 96.988 |
| | FC | 4 | 0.0 | 92.738 |
| | DFS | 4 | 0.004 | 113.804 |
| 8 | DFSAC3 | 92 | 0.142 | 96.996 |
| | FC | 92 | 0.015 | 92.758 |
| | DFS | 92 | 0.142 | 113.804 |
| 10 | DFSAC3 | 724 | 6.205 | 95.496 |
| | FC | 724 | 0.282 | 93.105 |
| | DFS | 724 | 6.121 | 113.804 |
| 12 | DFSAC3 | 14200 | 317.351 | 95.484 |
| | FC | 14200 | 9.088 | 100.605 |
| | DFS | 92 | 311.949 | 113.488 |
| 13 | DFSAC3 | 73712 | 2422.607 | 98.645 |
| | FC | 73712 | 56.602 | 142.902 |
| | DFS | 92 | 2381.305 | 99.906 |

## 4.3 Discussion

The experimental results provide insights into the performance of DFS algorithm, including DFS enhanced with AC3 (DFSAC3), and FC algorithms in solving the N-Queens problem.

DFS remains a fundamental approach for solving combinatorial optimization problems like the N-Queens problem. However, the results indicate its limitations in handling larger problem instances efficiently. As the problem size increases, DFS shows significant increases in execution time and memory consumption. This trend is consistent across all tested problem sizes, with execution times reaching several minutes for $N = 12$ and $N = 13$.

In an attempt to address these limitations, DFSAC3 was introduced by using AC3 to reduce the search space. While DFSAC3 shows improvements in memory consumption compared to basic DFS, its impact on execution time is negligible. The algorithm's execution time remains largely unchanged, indicating that while AC3 helps reduce memory usage, it does not significantly enhance computational efficiency.

Comparatively, FC algorithm demonstrates consistent performance with slight variations in memory usage when compared to DFSAC3. Although FC generally consumes slightly less memory, it also faces challenges in efficiently solving larger instances of the N-Queens problem. Notably, for $N = 12$ and $N = 13$ or larger N, both FC and DFSAC3 algorithms exhibit significant increases in execution time and memory usage, highlighting the scalability issues faced by these algorithms.

Overall, the experimental results underscore the need for further research and optimization to develop algorithms capable of efficiently solving larger instances of the N-Queens problem. [Ayub et al., 2017]

## 4.4 Conclusion

In conclusion, the experimental evaluation of Depth-First Search (DFS) algorithms, including DFS enhanced with Arc-Consistency 3 (DFSAC3), and FC algorithms provides valuable insights into their performance in solving the N-Queens problem. Despite efforts to mitigate inefficiencies, such as incorporating AC3 into DFS, scalability challenges persist, particularly for larger problem sizes. Further research and optimization are necessary to develop more efficient algorithms capable of handling larger instances of the N-Queens problem.

# Conclusion and Future Perspectives

In conclusion, our initiative aimed to test the performance of various algorithms on the existing N-Queens problem, a well-known problem in computational complexity theory known for its NP-Complete and P-Complete nature[Gent et al., 2018]. Through our experimental evaluation, we compared the effectiveness of AC3, FC, and DFS across different problem sizes (N values).

Our findings underscored the importance of considering factors such as problem size, available computational resources, and desired solution quality when choosing an algorithm.

As a perspective, exploring alternative algorithms such as Genetic Algorithms (GAs) could provide novel approaches to solving the N-Queens problem.

# Bibliography

[Ayub et al., 2017] Ayub, M. A., Kalpoma, K. A., Proma, H. T., Kabir, S. M., and Chowdhury, R. I. H. (2017). Exhaustive study of essential constraint satisfaction problem techniques based on n-queens problem. In *2017 20th International Conference of Computer and Information Technology (ICCIT)*, Dhaka, Bangladesh. IEEE.

[Feder, 2024] Feder, T. (2024). Constraint satisfaction: A personal perspective. *Journal Name*, Volume Number:Page Numbers.

[Gent et al., 2018] Gent, I. P., Jefferson, C., and Nightingale, P. (2018). Complexity of n-queens completion (extended abstract). In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, Fife KY16 9SX, UK. School of Computer Science, University of St Andrews.

[Gouda and Zaki, 2001] Gouda, K. and Zaki, M. (2001). Efficiently mining maximal frequent itemsets. In *Proceedings 2001 IEEE International Conference on Data Mining*, ICDM-01. IEEE Comput. Soc.

[Haralick and Elliott, 1979] Haralick, R. and Elliott, G. (1979). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

[Mackworth, 1977] Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118.

[Minton et al., 1992] Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1):161–205.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299.

[Toolify AI, 2022] Toolify AI (2022). Simplifying constraints: Discover the ac3 algorithm. `https://www.toolify.ai/ai-news/simplifying-constraints-discover-the-ac3-algorithm-2042532`. Accessed on March 30, 2024.

[Van Hentenryck et al., 1992] Van Hentenryck, P., Deville, Y., and Teng, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321.

# Annex A: Jupyter Notebook

# Exploring CSP Algorithms: The N-Queen's Problem


### Mehyar MLAWEH : mehyar.mlaweh@dauphine.tn

### Nadia BENYOUSSEF : nadia.benyoussef@dauphine.tn


===========================================================

In [1]:
```
import time
import random

import matplotlib.pyplot as plt

from memory_profiler import memory_usage

import statistics as st

import copy

import os, psutil
```

## CSP Problem Formalization

In [2]:
```
class NQueensCSP:
    def __init__(self, N):
        self.N = N
        # Define the variables as the columns of the chessboard
        self.variables = list(range(N))
        # Set the domain of each variable to be the rows of the chessboard
        self.domains = {i: list(range(N)) for i in range(N)}
        # List of constraints
        self.constraints = []

        # Row constraint
        for i in range(N):
            for j in range(i + 1, N):
                self.constraints.append((i, j, lambda x, y: x != y))

        # Diagonal constraint
        for i in range(N):
            for j in range(i + 1, N):
                self.constraints.append((i, j, lambda x, y, i=i, j=j: abs(x - y) != abs(i - j)))

    def checkConstraint(self, var1, val1, var2, val2):
        for c in self.constraints:
            if (var1 == c[0] and var2 == c[1]) or (var1 == c[1] and var2 == c[0]):
                if not c[2](val1, val2):
                    return False
        # If no constraints are violated, return True
        return True

    def neighbors(self, var):
        return [v for v in self.variables if v != var]
```

In [3]:
```
#Testing The Formalization
N = 2
csp = NQueensCSP(N)

# Test constraints
print("Testing :")
for i in range(N):
    for j in range(N):
        if i != j:
            for val1 in csp.domains[i]:
                for val2 in csp.domains[j]:
                    if not csp.checkConstraint(i, val1, j, val2):
                        print(f"Constraint NO: ({i}, {val1}) and ({j}, {val2})")
                    if csp.checkConstraint(i, val1, j, val2):
                        print(f"Constraint YES: ({i}, {val1}) and ({j}, {val2})")

print("Testing complete.")
```

```
Testing :
Constraint NO: (0, 0) and (1, 0)
Constraint NO: (0, 0) and (1, 1)
Constraint NO: (0, 1) and (1, 0)
Constraint NO: (0, 1) and (1, 1)
Constraint NO: (1, 0) and (0, 0)
Constraint NO: (1, 0) and (0, 1)
Constraint NO: (1, 1) and (0, 0)
Constraint NO: (1, 1) and (0, 1)
Testing complete.
```

## AC3 Algorithm

In [4]:
```python
def revise(csp, Xi, Xj):
    revised = False
    for x in list(csp.domains[Xi]):
        if not any(csp.checkConstraint(Xi, x, Xj, y) for y in csp.domains[Xj]):
            csp.domains[Xi].remove(x)
            print("Removed value:", x, "from domain of", Xi)
            revised = True
    return revised

def ac3(csp, queue=None):
    if queue is None:
        queue = [(Xi, Xj) for Xi in csp.variables for Xj in csp.neighbors(Xi)]
    while queue:
        (Xi, Xj) = queue.pop(0)
        if revise(csp, Xi, Xj):
            print("Revise done")
            if len(csp.domains[Xi]) == 0:
                return False
            for Xk in csp.neighbors(Xi):
                if Xk != Xj:
                    queue.append((Xk, Xi))
    return True
```

## Testing AC3

In [5]:
```python
N = 3
csp = NQueensCSP(N)
consistent = ac3(csp)

if consistent:
    print("The CSP is consistent after applying AC3.")
else:
    print("The CSP is inconsistent after applying AC3.")
```

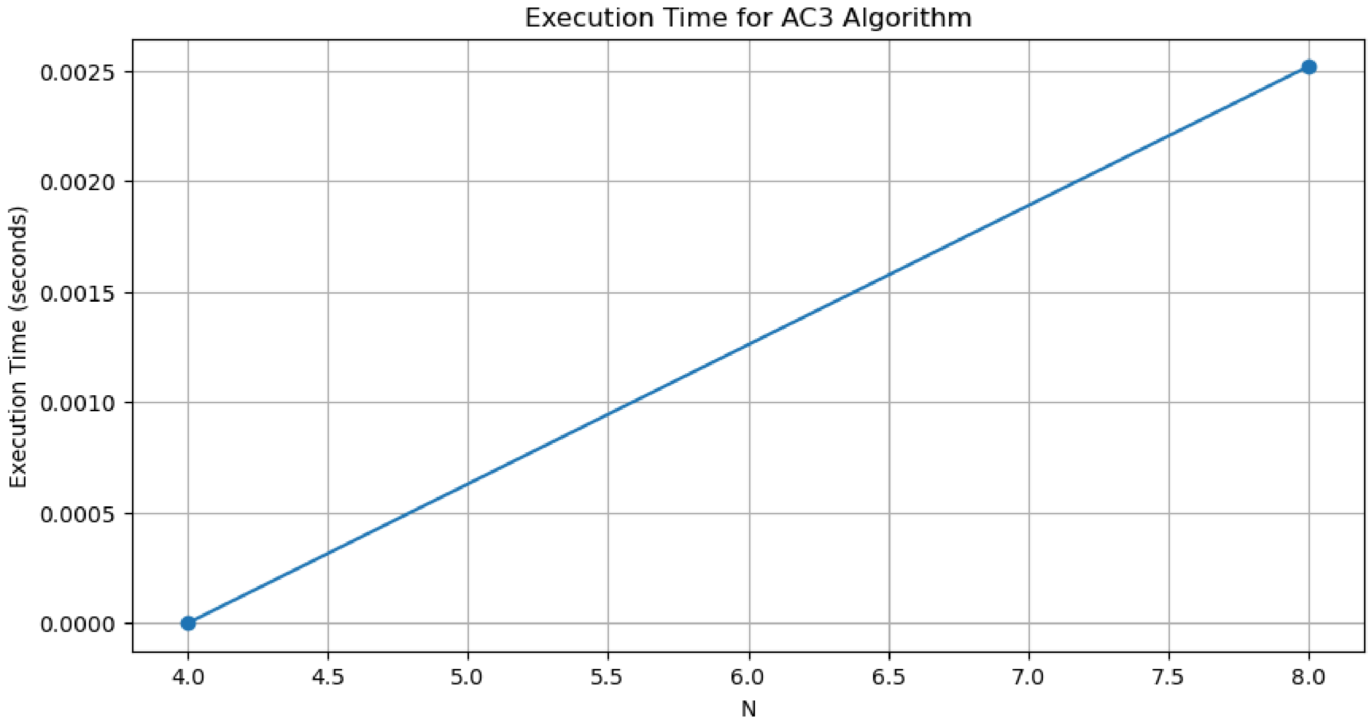
```
Removed value: 1 from domain of 0
Revise done
Removed value: 1 from domain of 1
Revise done
Removed value: 0 from domain of 2
Removed value: 2 from domain of 2
Revise done
Removed value: 1 from domain of 2
Revise done
The CSP is inconsistent after applying AC3.
```

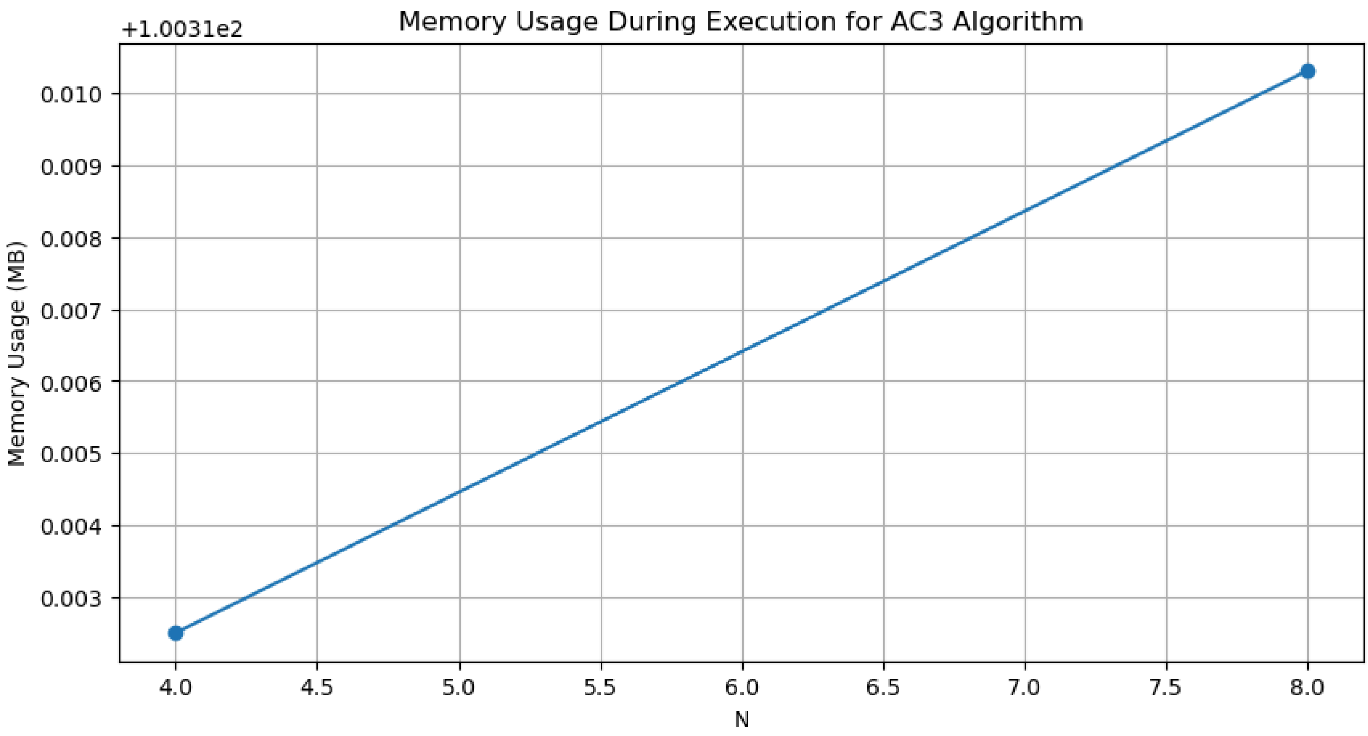
## Evaluation

In [6]:
```python
def test_ac3_for_n(N):
    start_time = time.time()

    csp = NQueensCSP(N)

    ac3_result = ac3(csp)

    end_time = time.time()
    execution_time = end_time - start_time

    end_memory = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2

    print(f"For N = {N}:")
    print("Consistent? :", ac3_result)
    print("Execution time:", execution_time, "seconds")
    print("Memory used during execution:", end_memory , "MB")

    return execution_time, end_memory

Ns = [4, 8]
```

```python
execution_times = []
memory_usages = []
for n in Ns:
    execution_time, memory_usage_during_execution = test_ac3_for_n(n)
    execution_times.append(execution_time)
    memory_usages.append(memory_usage_during_execution)

plt.figure(figsize=(10, 5))
plt.plot(Ns, execution_times, marker='o')
plt.title('Execution Time for AC3 Algorithm')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

# Plot memory usage
plt.figure(figsize=(10, 5))
plt.plot(Ns, memory_usages, marker='o')
plt.title('Memory Usage During Execution for AC3 Algorithm')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.grid(True)
plt.show()
```

```
For N = 4:
Consistent? : True
Execution time: 0.0 seconds
Memory used during execution: 100.3125 MB
For N = 8:
Consistent? : True
Execution time: 0.0025179386138916016 seconds
Memory used during execution: 100.3203125 MB
```






## Depth-First Search with Backtracking

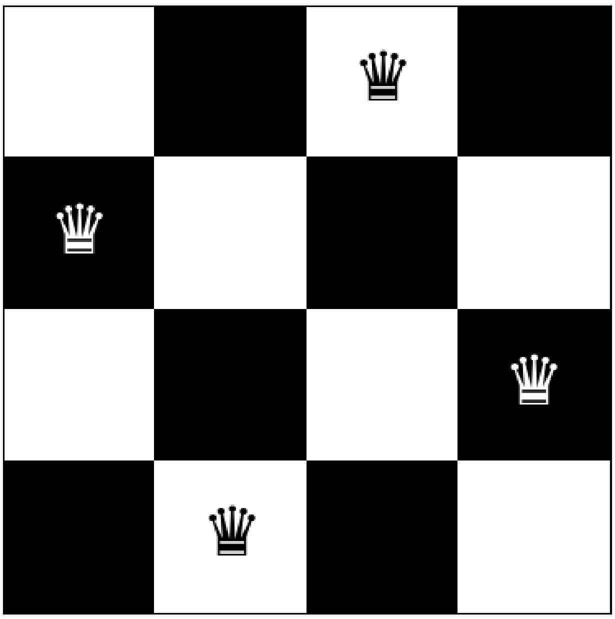
In [7]:
```
class DFS:
    def __init__(self, csp):
        self.csp = csp
        self.solutions = []

    def solve_all(self):
        """
        Method to find all solutions using depth-first search with backtracking.

        Returns:
            list: A list of all solutions found.
        """

        self.solutions = []  # Reset solutions list
        self.backtrack(0, [])  # Start the backtracking process from column 0
        return self.solutions

    def backtrack(self, col, solution):
        """
        Recursive function to perform depth-first search with backtracking.

        Args:
            col (int): The current column being explored.
            solution (list): The current partial solution.

        Returns:
            None
        """

        # If all columns are explored (base case)
        if col == self.csp.N:
            # Add the solution to the list of solutions
            self.solutions.append(solution[:])
            return

        # Iterate over each possible row in the current column
        for row in self.csp.domains[col]:
            # Check if placing a queen in the current position is safe
            if self.is_safe(col, row, solution):
                # Place the queen in the current position
                solution.append(row)
                # Recursively move to the next column
                self.backtrack(col + 1, solution)
                # Backtrack by removing the last queen placed
                solution.pop()

    def is_safe(self, col, row, solution):
        """
        Method to check if placing a queen in a given position is safe.

        Args:
```

```
            col (int): The column of the queen being placed.
            row (int): The row of the queen being placed.
            solution (list): The current partial solution.

        Returns:
            bool: True if placing the queen is safe, False otherwise.
        """

        # Iterate over previously placed queens
        for prev_col, prev_row in enumerate(solution):
            # Check if the new queen conflicts with any previous queen
            if not self.csp.checkConstraint(col, row, prev_col, prev_row):
                return False
            # Check if the new queen is on the same diagonal as any previous queen
            if abs(col - prev_col) == abs(row - prev_row):
                return False
        return True

    def plot_chessboard(self, board):
        N = len(board)
        chessboard = [[(i + j) % 2 for i in range(N)] for j in range(N)]  # Generate a chessboard pattern

        fig, ax = plt.subplots()  # Create subplots
        ax.imshow(chessboard, cmap='binary')  # Plot the chessboard pattern

        # Plot queens on the chessboard
        for i in range(N):
            for j in range(N):
                if board[j] == i:  # If there is a queen in the position
                    # Plot the queen symbol
                    ax.text(j, i, u'\u265B', fontsize=30, ha='center', va='center', color='black' if (i + j) % 2 == 0

        ax.set_xticks([])  # Hide x-axis ticks
        ax.set_yticks([])  # Hide y-axis ticks
        plt.show()  # Show the plot

    def plot_solutions(self):
        for i, solution in enumerate(self.solutions):
            print(f"Solution {i + 1}:")
            self.plot_chessboard(solution)
```

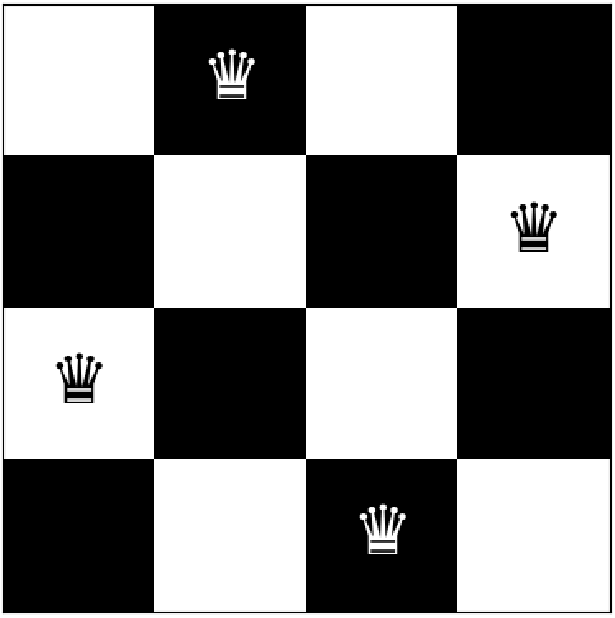
## Testing DFS With Backtracking

In [8]:
```
N = 4
csp = NQueensCSP(N)
solver = DFS(csp)
solutions = solver.solve_all()
if(len(solutions)==0):
    print("No solution")

solver.plot_solutions()
```

Solution 1:



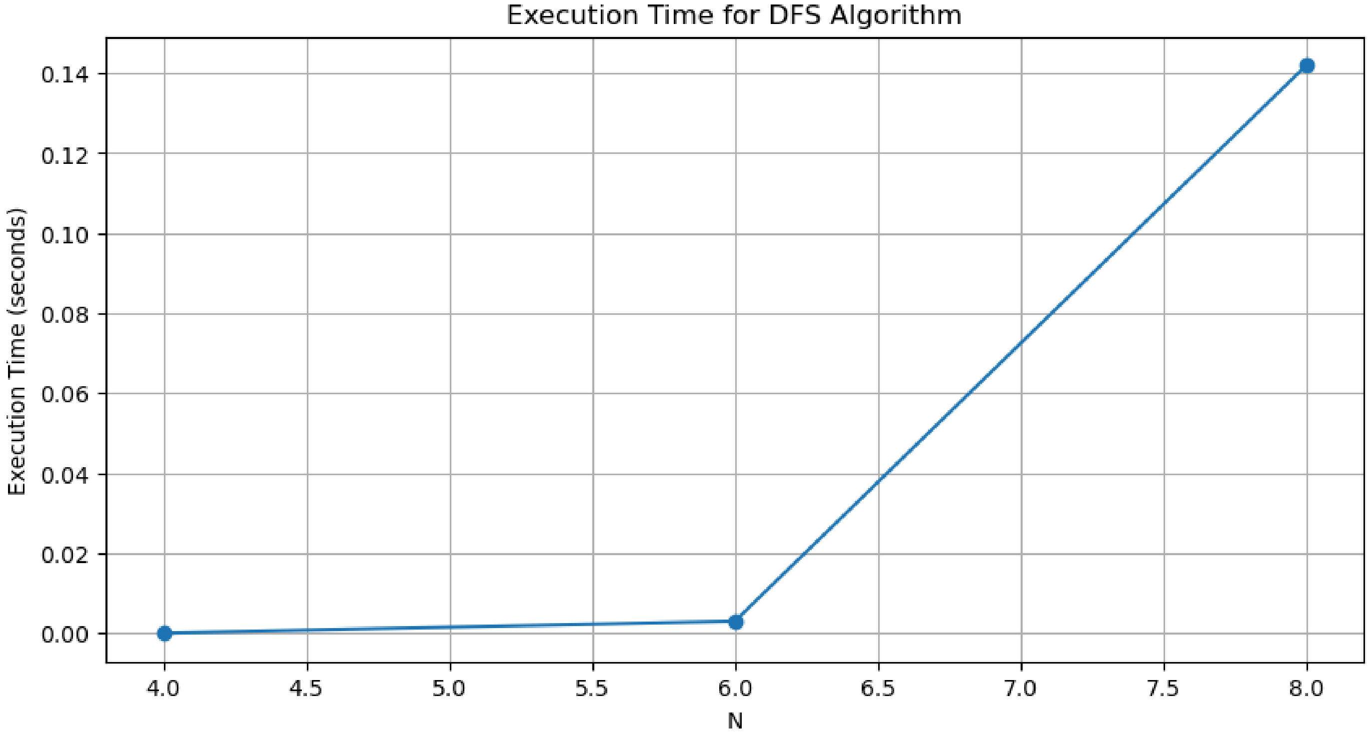
Solution 2:

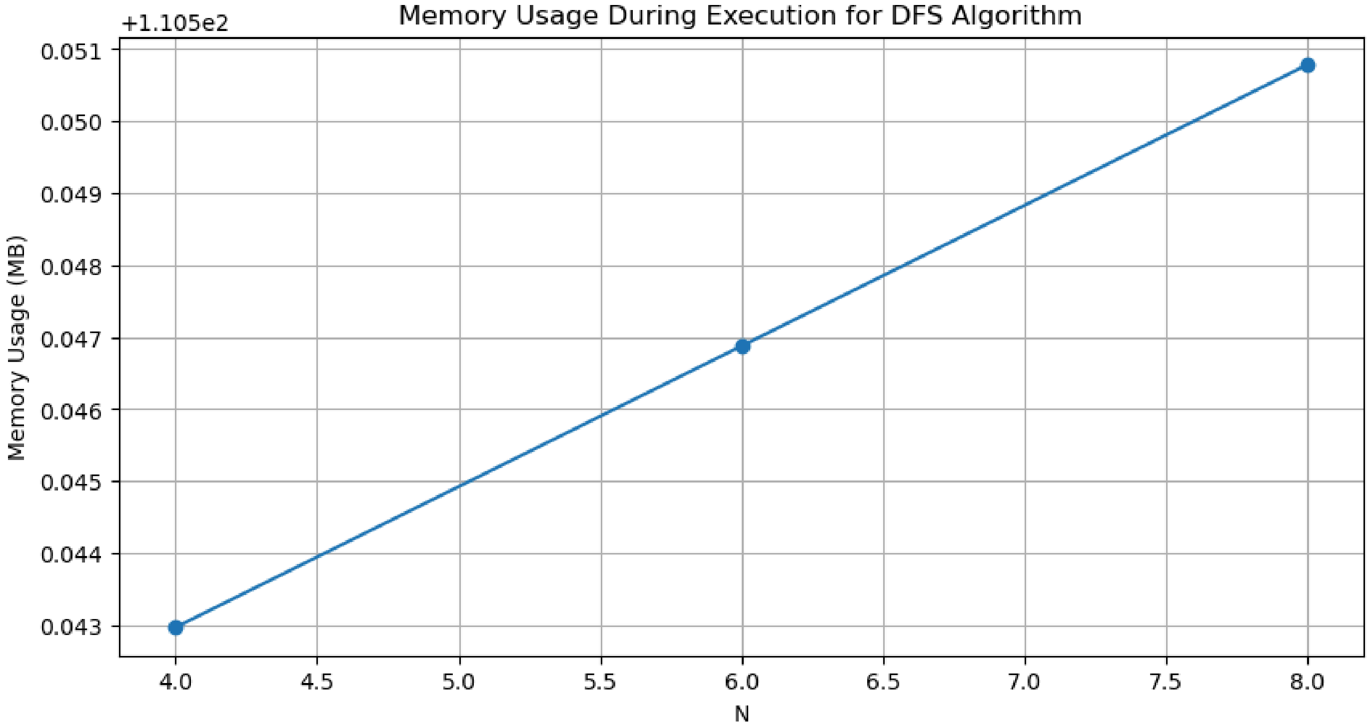


## Evaluation

In [9]:
```
def test_dfs_for_n(N):
    start_time = time.time()

    # Create NQueensCSP instance
    csp = NQueensCSP(N)

    # Create DFS instance
    dfs_solver = DFS(csp)

    # Run DFS algorithm
    solutions = dfs_solver.solve_all()

    end_time = time.time()
    execution_time = end_time - start_time

    end_memory = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2

    print(f"For N = {N}:")
    print("Number of solutions found:", len(solutions))
    print("Execution time:", execution_time, "seconds")
    print("Memory used during execution:", end_memory, "MB")

    return execution_time, end_memory

# Test for various values of N
Ns = [4,6, 8]
execution_times = []
memory_usages = []
for n in Ns:
    execution_time, memory_usage_during_execution = test_dfs_for_n(n)
    execution_times.append(execution_time)
    memory_usages.append(memory_usage_during_execution)

# Plot execution time
plt.figure(figsize=(10, 5))
plt.plot(Ns, execution_times, marker='o')
plt.title('Execution Time for DFS Algorithm')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

# Plot memory usage
plt.figure(figsize=(10, 5))
plt.plot(Ns, memory_usages, marker='o')
plt.title('Memory Usage During Execution for DFS Algorithm')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.grid(True)
plt.show()
```

```
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 110.54296875 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.0029633045196533203 seconds
Memory used during execution: 110.546875 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.14200997352600098 seconds
Memory used during execution: 110.55078125 MB
```




## Forward Checking

In [10]:
```
class ForwardChecking:
    def __init__(self, csp):
        """
        Constructor for the ForwardChecking class.

        Args:
            csp: An instance of the CSP class representing the Constraint Satisfaction Problem.
        """
        self.csp = csp
        self.solutions = []
```

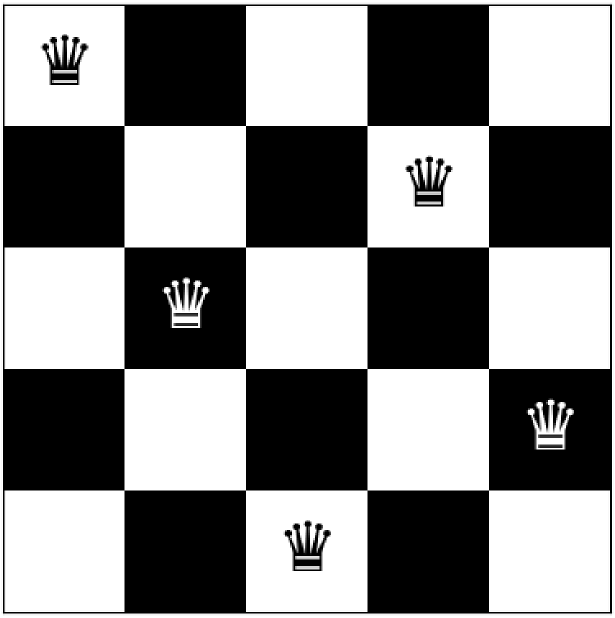
```python
    def forward_checking(self):
        """
        Main method to perform forward checking and find all solutions.

        Returns:
            list: A list of all solutions found.
        """
        assignment = {}
        # Call the recursive forward checking method with the initial assignment
        self.solutions = self._recursive_forward_checking(assignment)
        return self.solutions

    def _recursive_forward_checking(self, assignment):
        """
        Recursive function to perform forward checking.

        Args:
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            list: A list of all solutions found from the current assignment.
        """
        # Base case: if all variables are assigned
        if len(assignment) == self.csp.N:
            return [assignment.copy()]  # Return the current assignment as a solution

        var = self.select_unassigned_variable(assignment)  # Select an unassigned variable
        solutions = []  # Initialize an empty list to store solutions

        if var is not None:  # Check if there is an unassigned variable
            for val in range(self.csp.N):  # Iterate over possible values for the variable
                if not self.violates_constraints(var, val, assignment):  # Check if value violates constraints
                    assignment[var] = val  # Assign the value to the variable
                    neighbors = self.get_neighbors(var, assignment)  # Get neighboring variables
                    forward_check = self._recursive_forward_checking(assignment)  # Recursively explore further assign
                    solutions.extend(forward_check)  # Add solutions from further exploration
                    del assignment[var]  # Backtrack: remove the assigned value

        return solutions  # Return the list of solutions found

    def select_unassigned_variable(self, assignment):
        """
        Method to select an unassigned variable.

        Args:
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            int or None: The selected unassigned variable, or None if all variables are assigned.
        """
        unassigned_vars = [var for var in self.csp.variables if var not in assignment]
        if unassigned_vars:
            return min(unassigned_vars, key=lambda var: len(self.csp.domains[var]))
        else:
            return None

    def violates_constraints(self, var, value, assignment):
        """
        Method to check if assigning a value to a variable violates constraints.

        Args:
            var (int): The variable to assign a value to.
            value (int): The value to assign to the variable.
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            bool: True if the assignment violates constraints, False otherwise.
        """
        for constraint_var in assignment:
            if assignment[constraint_var] == value or abs(var - constraint_var) == abs(value - assignment[constraint_v
                return True
        return False

    def get_neighbors(self, var, assignment):
        """
        Method to get neighboring variables that are not yet assigned.

        Args:
            var (int): The variable to find neighbors for.
            assignment (dict): A dictionary representing the current assignment of variables.

        Returns:
            list: A list of neighboring variables.
```

```
        """
        return [neighbor for neighbor in self.csp.variables if neighbor != var and neighbor not in assignment]

    def restore_domains(self, neighbors, assignment):
        """
        Method to restore the domains of neighboring variables.

        Args:
            neighbors (list): A list of neighboring variables.
            assignment (dict): A dictionary representing the current assignment of variables.
        """
        for neighbor in neighbors:
            self.csp.domains[neighbor].extend([val for val in range(self.csp.N) if val not in self.csp.domains[neighbo

    def plot_chessboard(self, board):
        """
        Method to plot the chessboard with queens placed according to the solution.

        Args:
            board (list): A list representing the positions of queens on the chessboard.
        """
        N = len(board)
        chessboard = [[(i + j) % 2 for i in range(N)] for j in range(N)]

        fig, ax = plt.subplots()
        ax.imshow(chessboard, cmap='binary')

        for i in range(N):
            for j in range(N):
                if board[j] == i:
                    ax.text(j, i, u'\u265B', fontsize=30, ha='center', va='center', color='black' if (i + j) % 2 == 0

        ax.set_xticks([])
        ax.set_yticks([])
        plt.show()
```

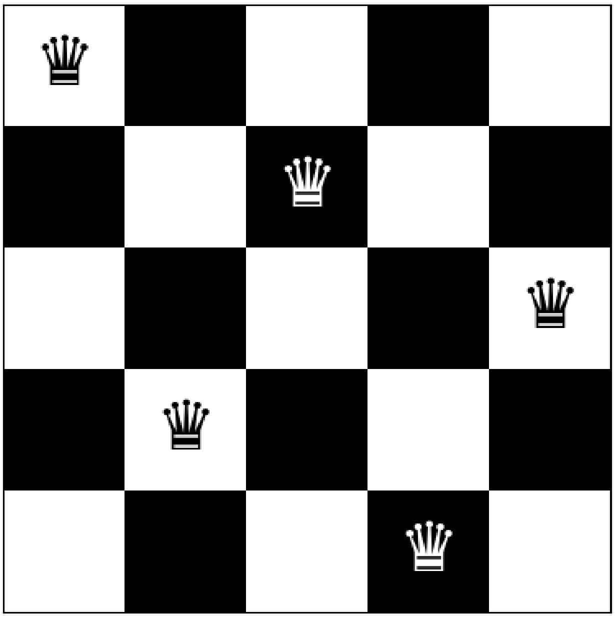
## Testing FC

In [11]:
```
N = 5
csp = NQueensCSP(N)
forward_checker = ForwardChecking(csp)

solutions = forward_checker.forward_checking()
if solutions:
    for i, solution in enumerate(solutions):
        print(f"Solution {i + 1}:")
        forward_checker.plot_chessboard(solution)
else:
    print("No solution found")
```

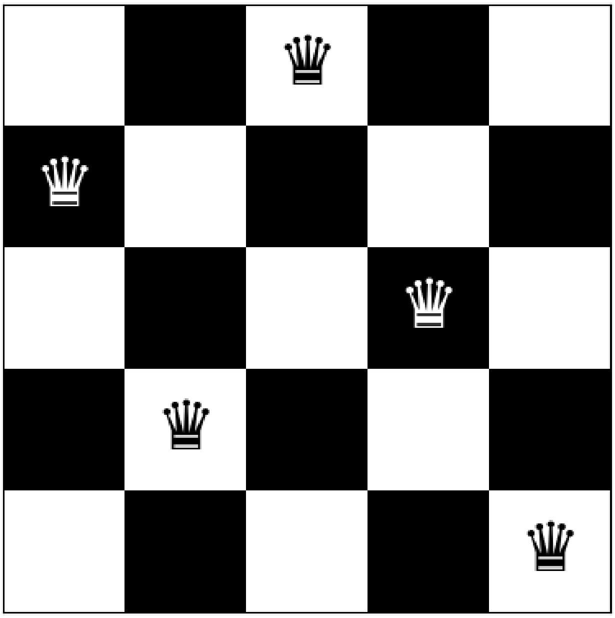
```
Solution 1:
```



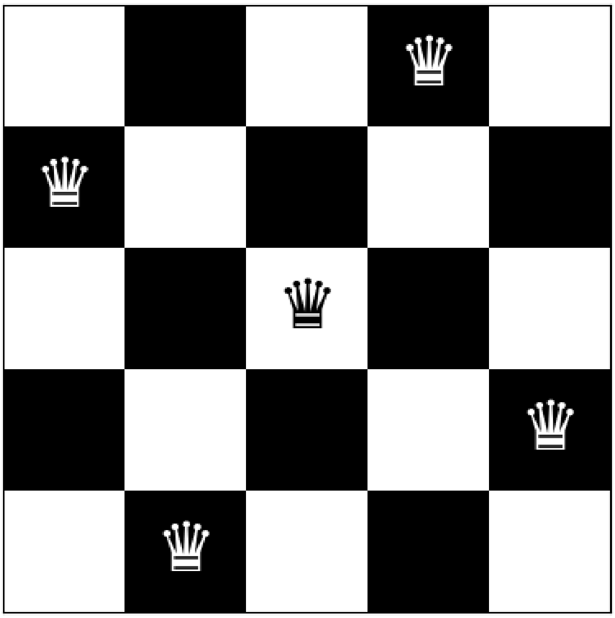
```
Solution 2:
```



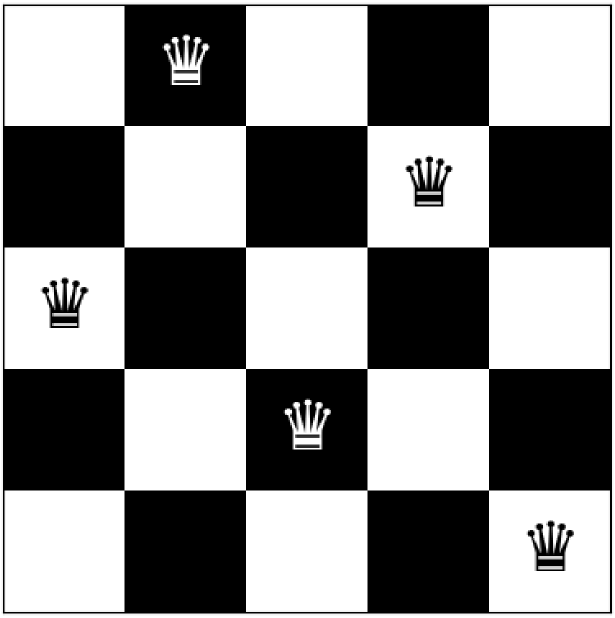
Solution 3:



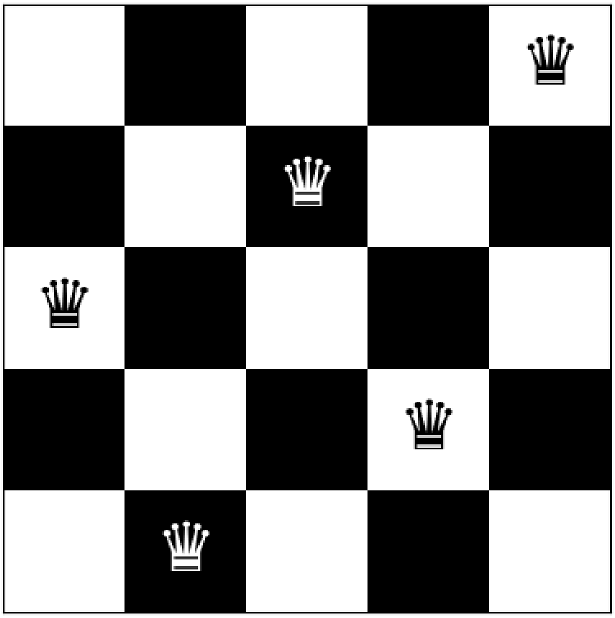
Solution 4:



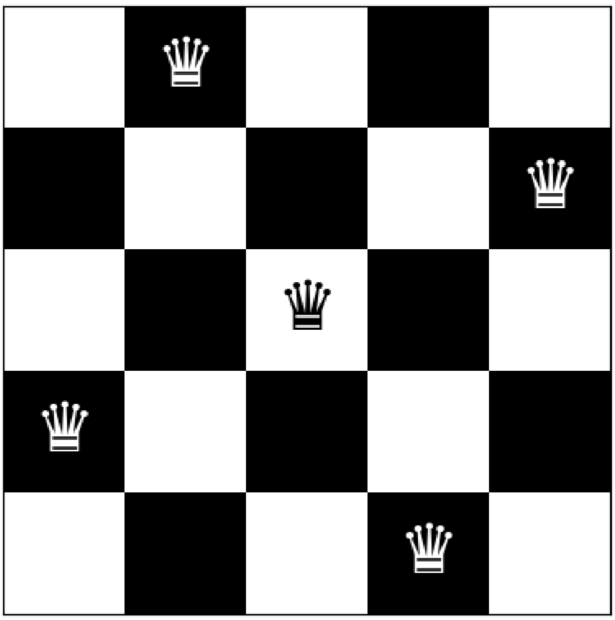
Solution 5:



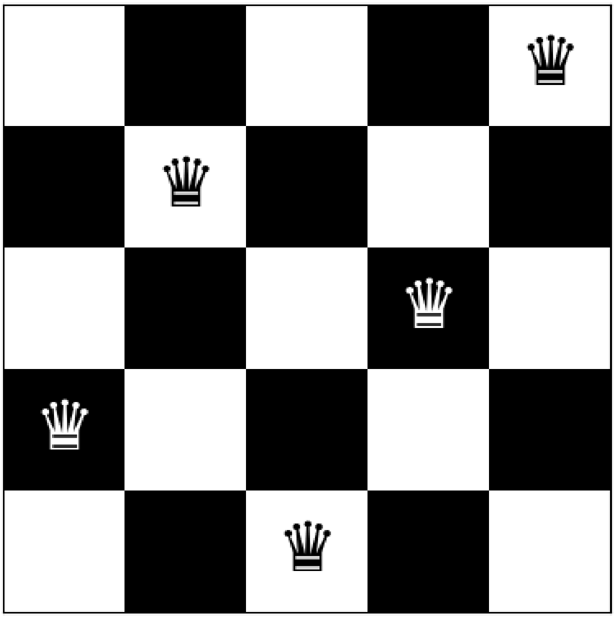
Solution 6:



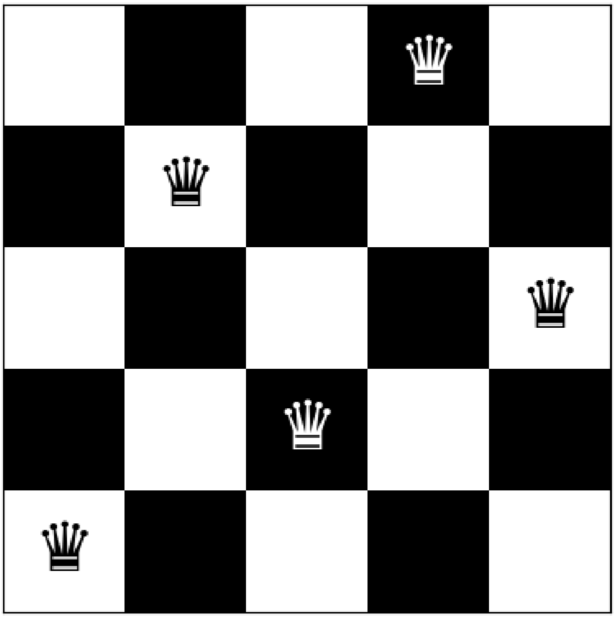
Solution 7:



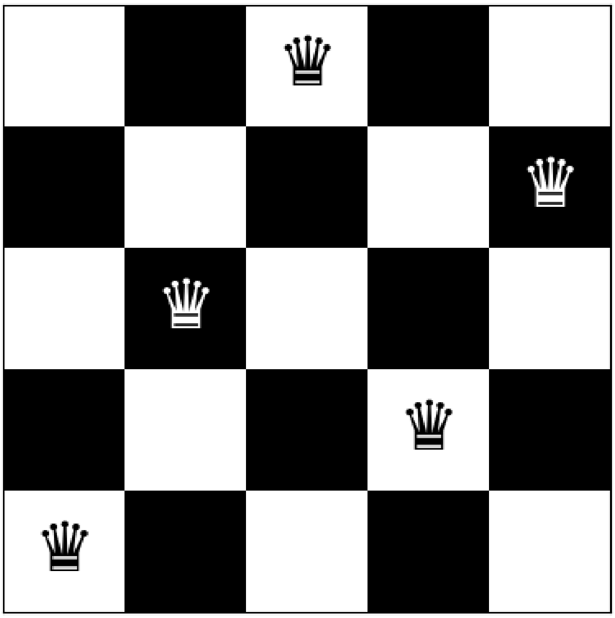
Solution 8:



Solution 9:



Solution 10:

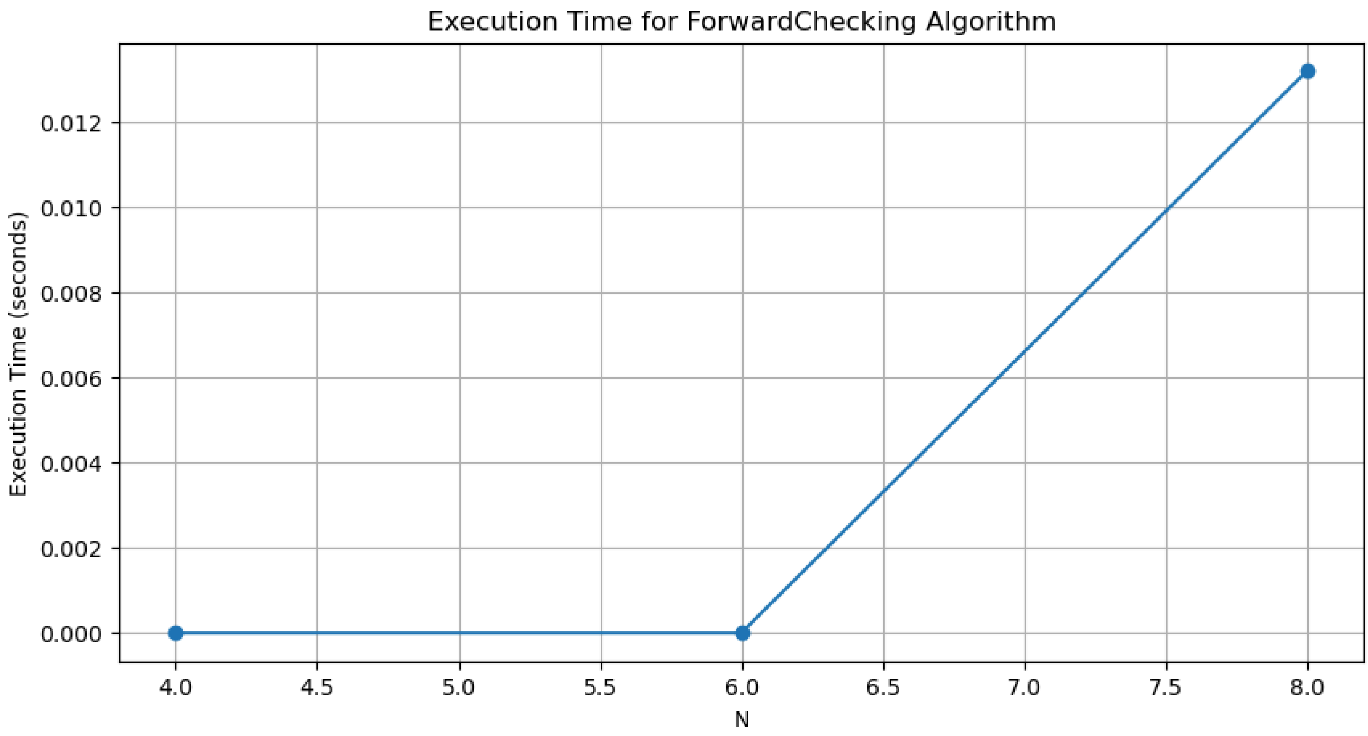


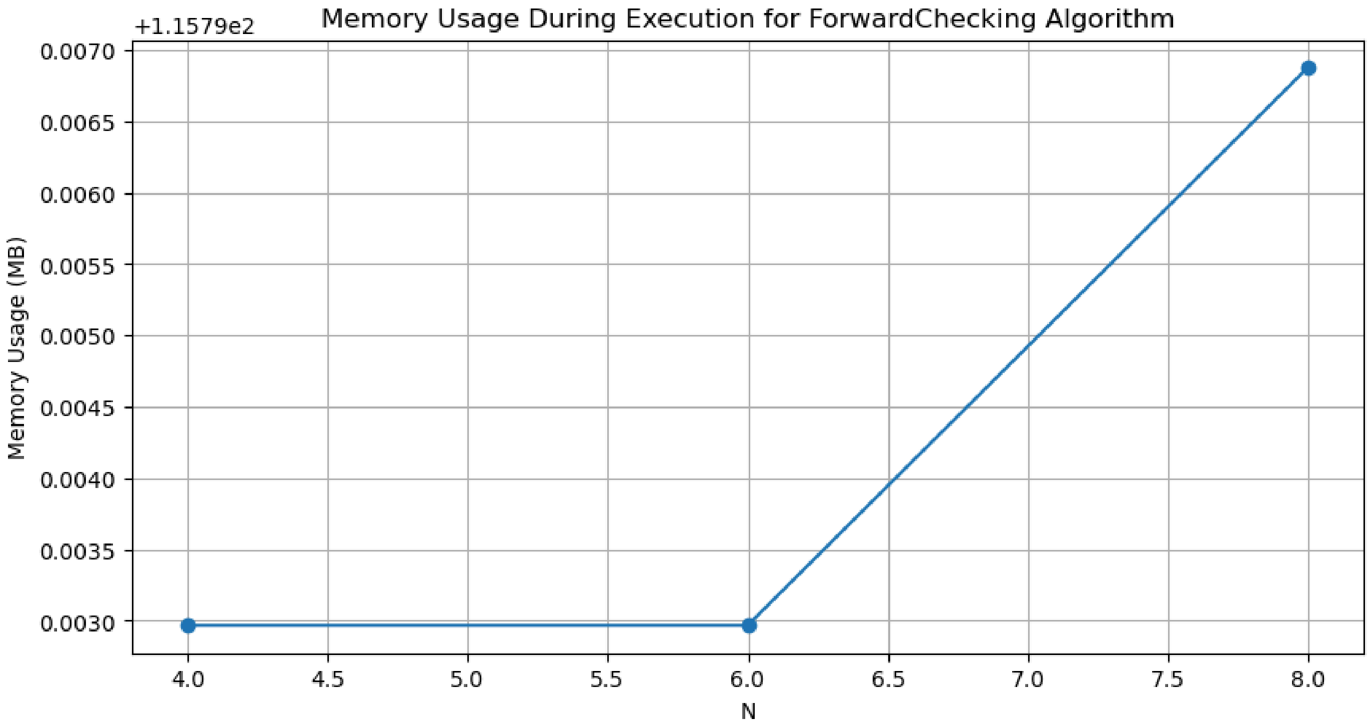
## Evaluation

In [12]:
```
def test_forward_checking_for_n(N):
    start_time = time.time()

    # Create NQueensCSP instance
    csp = NQueensCSP(N)
```

```python
    # Create ForwardChecking instance
    fc_solver = ForwardChecking(csp)

    # Run ForwardChecking algorithm
    solutions = fc_solver.forward_checking()

    end_time = time.time()
    execution_time = end_time - start_time

    end_memory = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2

    print(f"For N = {N}:")
    print("Number of solutions found:", len(solutions))
    print("Execution time:", execution_time, "seconds")
    print("Memory used during execution:", end_memory, "MB")

    return execution_time, end_memory

# Test for various values of N
Ns = [4, 6,8]
execution_times = []
memory_usages = []
for n in Ns:
    execution_time, memory_usage_during_execution = test_forward_checking_for_n(n)
    execution_times.append(execution_time)
    memory_usages.append(memory_usage_during_execution)

# Plot execution time
plt.figure(figsize=(10, 5))
plt.plot(Ns, execution_times, marker='o')
plt.title('Execution Time for ForwardChecking Algorithm')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

# Plot memory usage
plt.figure(figsize=(10, 5))
plt.plot(Ns, memory_usages, marker='o')
plt.title('Memory Usage During Execution for ForwardChecking Algorithm')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.grid(True)
plt.show()
```

```
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 115.79296875 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.0 seconds
Memory used during execution: 115.79296875 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.013200044631958008 seconds
Memory used during execution: 115.796875 MB
```




## DFS Using AC3

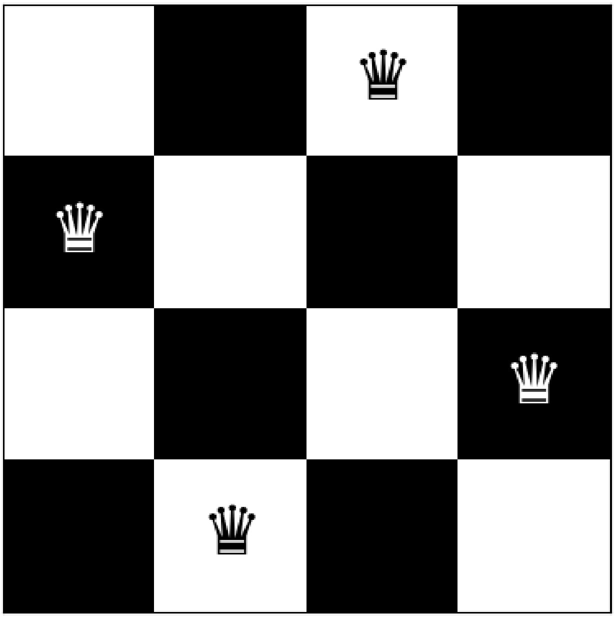
In [13]:
```
class DFSAC3:
    def __init__(self, csp):
        self.csp = csp
        self.solutions = []

    def solve_all(self):
        """
        Method to find all solutions using depth-first search with backtracking.

        Returns:
            list: A list of all solutions found.
        """

        self.solutions = []
        reduced_domains = self.apply_ac3()  # Apply AC3 to get reduced domains
        self.backtrack(0, [], reduced_domains)  # Start the backtracking process from column 0
        return self.solutions

    def apply_ac3(self):
        """
        Method to apply AC3 algorithm and obtain reduced domains for each variable.

        Returns:
```

```python
            dict: A dictionary containing reduced domains for each variable.
        """
        ac3(self.csp)  # Apply AC3
        reduced_domains = {var: self.csp.domains[var] for var in self.csp.variables}
        return reduced_domains

    def backtrack(self, col, solution, reduced_domains):
        """
        Recursive function to perform depth-first search with backtracking.

        Args:
            col (int): The current column being explored.
            solution (list): The current partial solution.
            reduced_domains (dict): Reduced domains for each variable.

        Returns:
            None
        """

        # If all columns are explored (base case)
        if col == self.csp.N:
            # Add the solution to the list of solutions
            self.solutions.append(solution[:])
            return

        # Iterate over each possible row in the current column
        for row in reduced_domains[col]:
            # Check if placing a queen in the current position is safe
            if self.is_safe(col, row, solution):
                # Place the queen in the current position
                solution.append(row)
                # Recursively move to the next column
                self.backtrack(col + 1, solution, reduced_domains)
                # Backtrack by removing the last queen placed
                solution.pop()

    def is_safe(self, col, row, solution):
        """
        Method to check if placing a queen in a given position is safe.

        Args:
            col (int): The column of the queen being placed.
            row (int): The row of the queen being placed.
            solution (list): The current partial solution.

        Returns:
            bool: True if placing the queen is safe, False otherwise.
        """

        # Iterate over previously placed queens
        for prev_col, prev_row in enumerate(solution):
            # Check if the new queen conflicts with any previous queen
            if not self.csp.checkConstraint(col, row, prev_col, prev_row):
                return False
            # Check if the new queen is on the same diagonal as any previous queen
            if abs(col - prev_col) == abs(row - prev_row):
                return False
        return True

    def plot_chessboard(self, board):
        N = len(board)
        chessboard = [[(i + j) % 2 for i in range(N)] for j in range(N)]  # Generate a chessboard pattern

        fig, ax = plt.subplots()  # Create subplots
        ax.imshow(chessboard, cmap='binary')  # Plot the chessboard pattern

        # Plot queens on the chessboard
        for i in range(N):
            for j in range(N):
                if board[j] == i:  # If there is a queen in the position
                    # Plot the queen symbol
                    ax.text(j, i, u'\u265B', fontsize=30, ha='center', va='center', color='black' if (i + j) % 2 == 0

        ax.set_xticks([])  # Hide x-axis ticks
        ax.set_yticks([])  # Hide y-axis ticks
        plt.show()  # Show the plot

    def plot_solutions(self):
        for i, solution in enumerate(self.solutions):
            print(f"Solution {i + 1}:")
            self.plot_chessboard(solution)
```

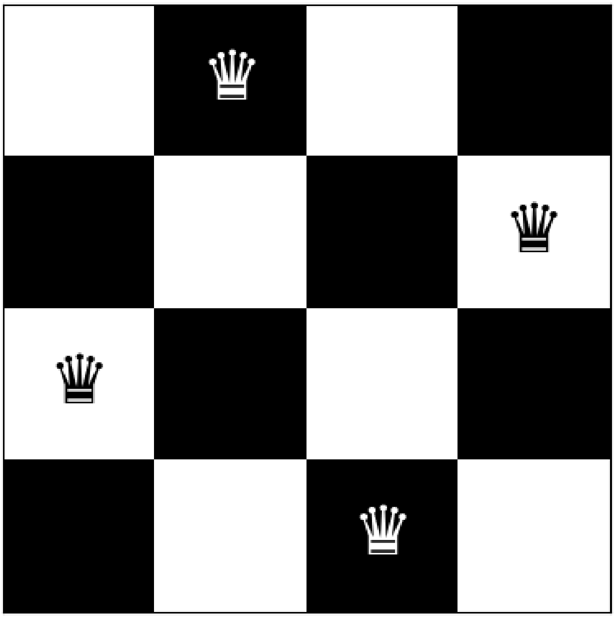
## Testing DFSAC3

In [14]:
```python
N = 4
csp = NQueensCSP(N)
dfsac3 = DFSAC3(csp)
solutions = dfsac3.solve_all()
dfsac3.plot_solutions()
```

```
Solution 1:
```



```
Solution 2:
```

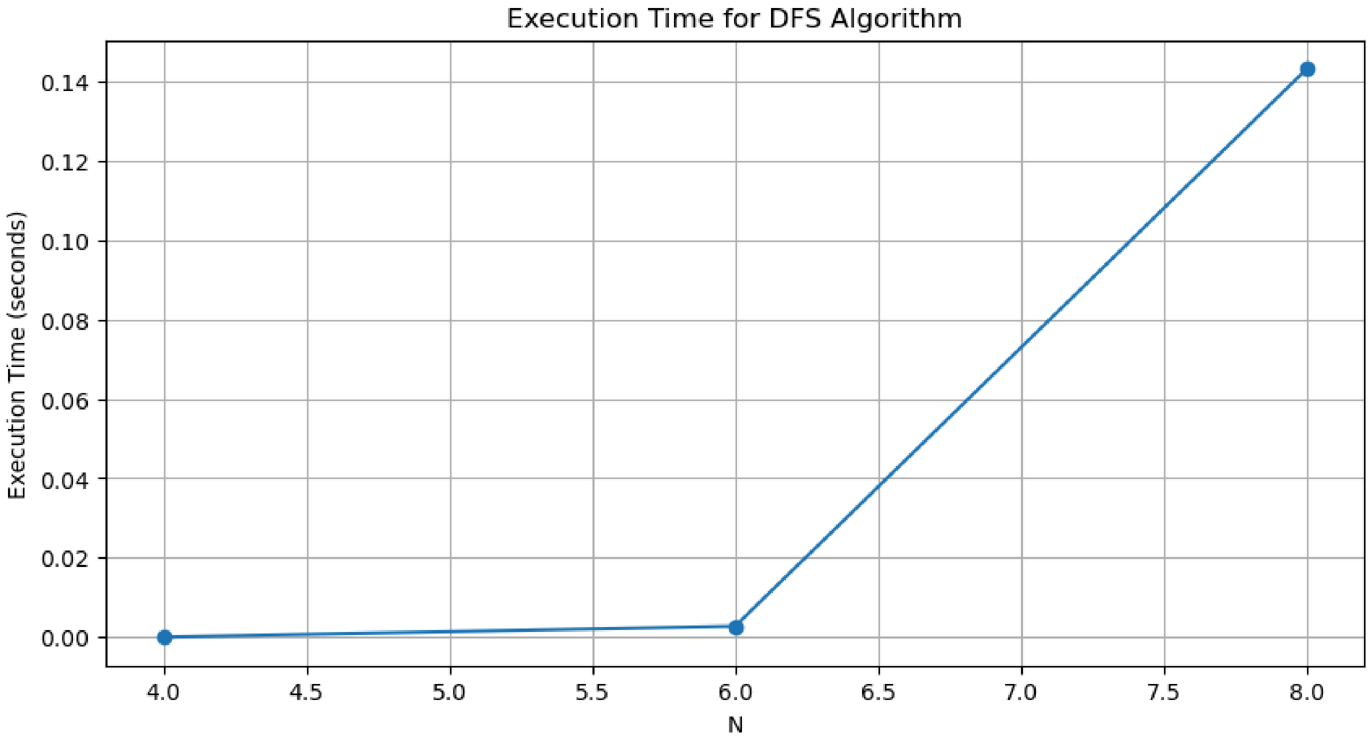


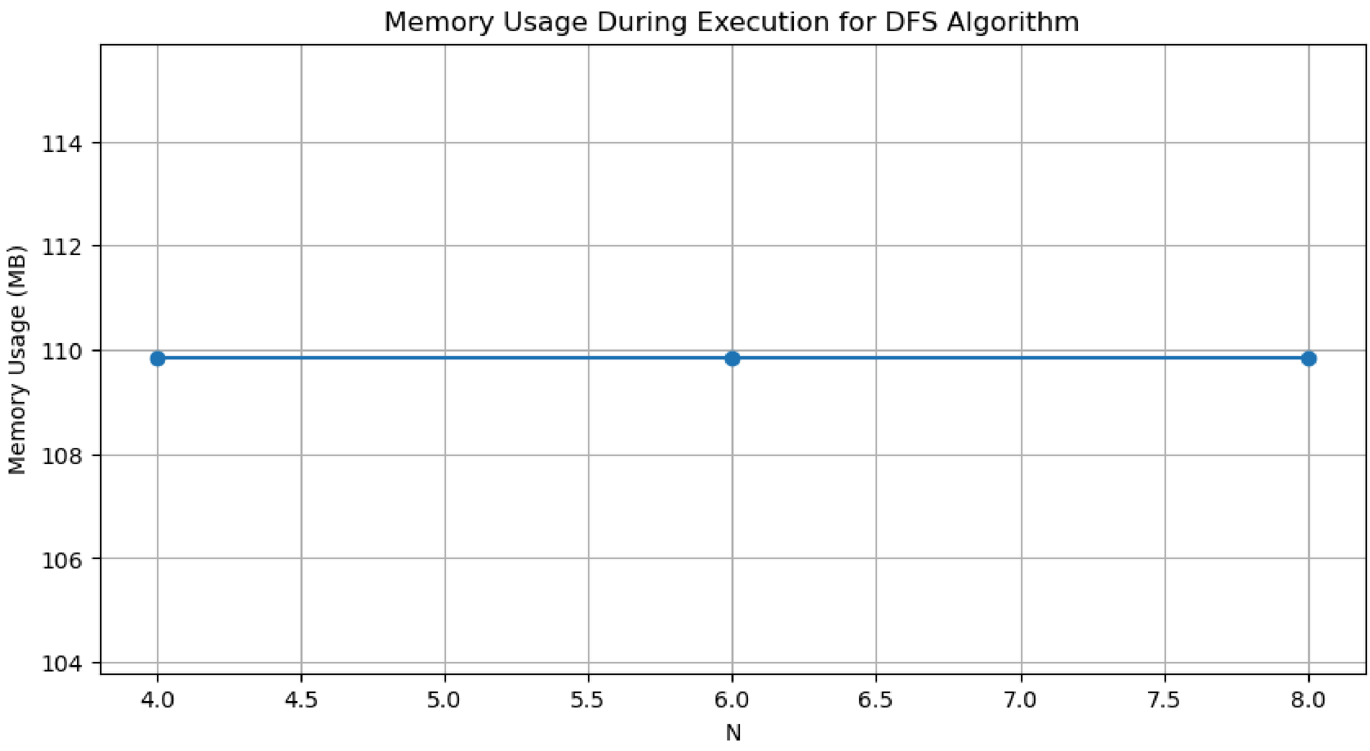
## Evaluation

In [15]:
```python
def test_dfsac3_for_n(N):
    start_time = time.time()

    # Create NQueensCSP instance
    csp = NQueensCSP(N)

    # Create DFS instance
    dfs_ac3 = DFSAC3(csp)

    # Run DFS algorithm
    solutions = dfs_ac3.solve_all()

    end_time = time.time()
    execution_time = end_time - start_time

    end_memory = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2

    print(f"For N = {N}:")
    print("Number of solutions found:", len(solutions))
    print("Execution time:", execution_time, "seconds")
    print("Memory used during execution:", end_memory, "MB")
```

```
    return execution_time, end_memory

# Test for various values of N
Ns = [4, 6, 8]
execution_times = []
memory_usages = []
for n in Ns:
    execution_time, memory_usage_during_execution = test_dfs_for_n(n)
    execution_times.append(execution_time)
    memory_usages.append(memory_usage_during_execution)

# Plot execution time
plt.figure(figsize=(10, 5))
plt.plot(Ns, execution_times, marker='o')
plt.title('Execution Time for DFS Algorithm')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()

# Plot memory usage
plt.figure(figsize=(10, 5))
plt.plot(Ns, memory_usages, marker='o')
plt.title('Memory Usage During Execution for DFS Algorithm')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.grid(True)
plt.show()
```

```
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 109.8359375 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.0027303695678710938 seconds
Memory used during execution: 109.8359375 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.14325475692749023 seconds
Memory used during execution: 109.8359375 MB
```



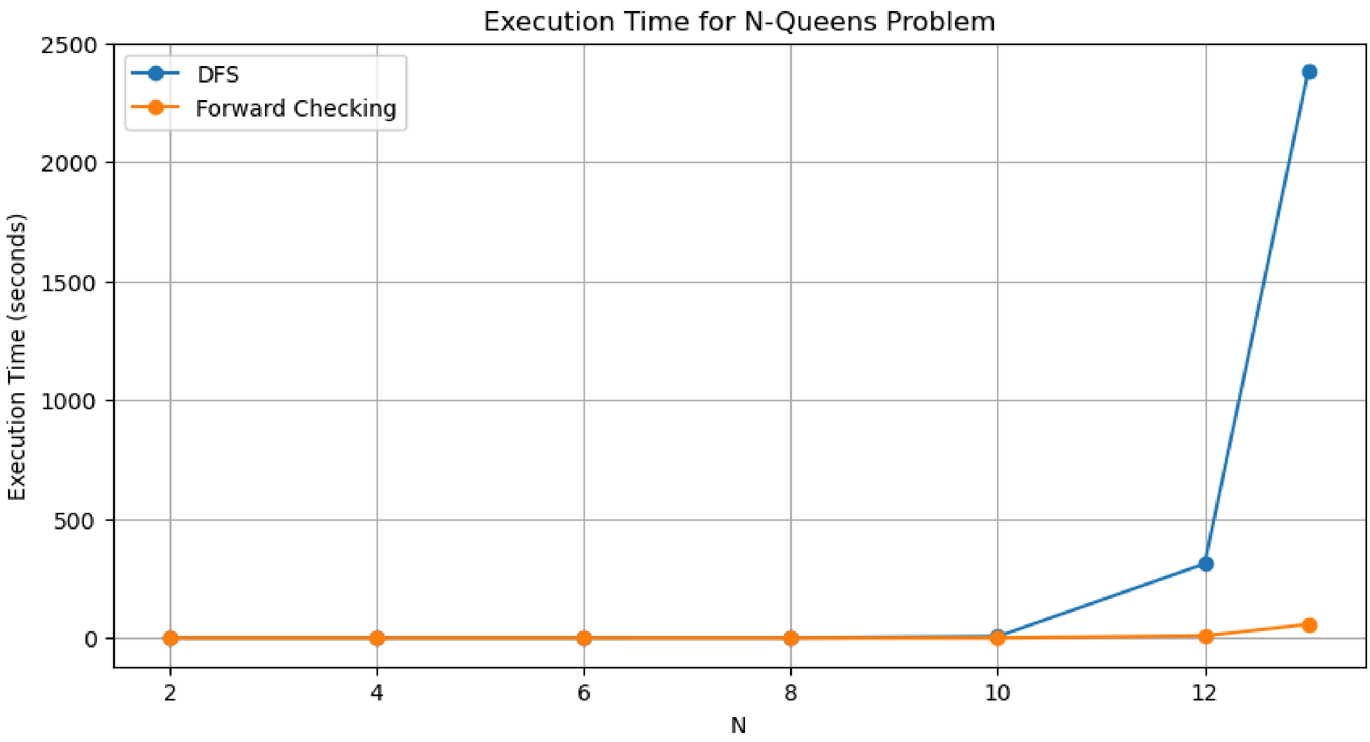


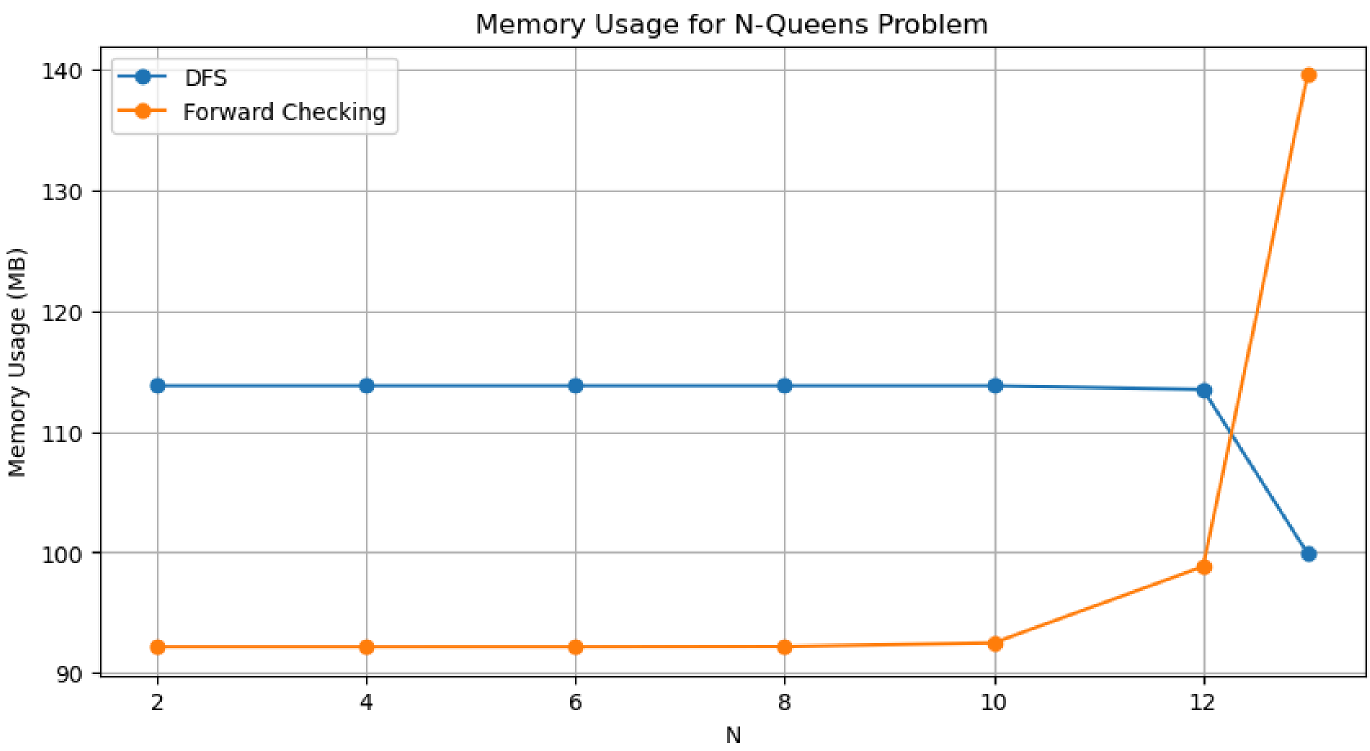

## Results (Simple DFS vs. FC)

In [16]:

```
Ns = [2,4,6,8,10,12,13]


print("DFS===========================")
dfs_execution_times = []
dfs_memory_usages = []
for n in Ns:
    execution_time1, memory_usage_during_execution1 = test_dfs_for_n(n)
    dfs_execution_times.append(execution_time1)
    dfs_memory_usages.append(memory_usage_during_execution1)
print("FORWAD CHECKING===============")
fc_execution_times = []
fc_memory_usages = []
for n in Ns:
    execution_time2, memory_usage_during_execution2 = test_forward_checking_for_n(n)
    fc_execution_times.append(execution_time2)
    fc_memory_usages.append(memory_usage_during_execution2)



plt.figure(figsize=(10, 5))
plt.plot(Ns, dfs_execution_times, marker='o', label='DFS')
plt.plot(Ns, fc_execution_times, marker='o', label='Forward Checking')
plt.title('Execution Time for N-Queens Problem')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(Ns, dfs_memory_usages, marker='o', label='DFS')
plt.plot(Ns, fc_memory_usages, marker='o', label='Forward Checking')
plt.title('Memory Usage for N-Queens Problem')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.legend()
plt.grid(True)
plt.show()
```

```
DFS===========================
For N = 2:
Number of solutions found: 0
Execution time: 0.0 seconds
Memory used during execution: 113.8046875 MB
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 113.8046875 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.004258155822753906 seconds
Memory used during execution: 113.8046875 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.14175128936767578 seconds
Memory used during execution: 113.8046875 MB
For N = 10:
Number of solutions found: 724
Execution time: 6.121318817138672 seconds
Memory used during execution: 113.8046875 MB
For N = 12:
Number of solutions found: 14200
Execution time: 311.94930124282837 seconds
Memory used during execution: 113.48828125 MB
For N = 13:
Number of solutions found: 73712
Execution time: 2381.305305957794 seconds
Memory used during execution: 99.90625 MB
FORWAD CHECKING===============
For N = 2:
Number of solutions found: 0
Execution time: 0.0 seconds
Memory used during execution: 92.17578125 MB
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 92.17578125 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.0010211467742919922 seconds
Memory used during execution: 92.1796875 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.012029170989990234 seconds
Memory used during execution: 92.19921875 MB
For N = 10:
Number of solutions found: 724
Execution time: 0.2842392921447754 seconds
Memory used during execution: 92.5078125 MB
For N = 12:
Number of solutions found: 14200
Execution time: 8.874612092971802 seconds
Memory used during execution: 98.82421875 MB
For N = 13:
Number of solutions found: 73712
Execution time: 57.06921458244324 seconds
Memory used during execution: 139.56640625 MB
```



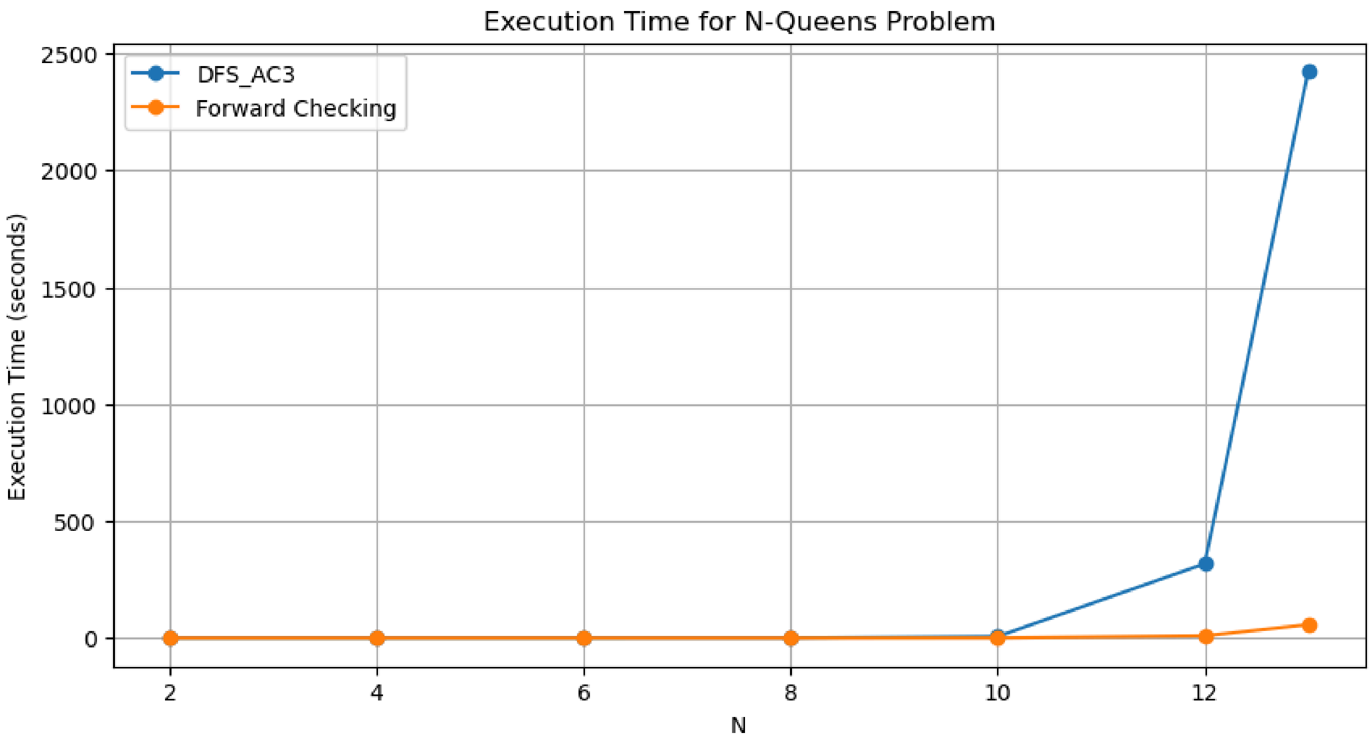

## Results ( DFS_AC3 vs. FC)

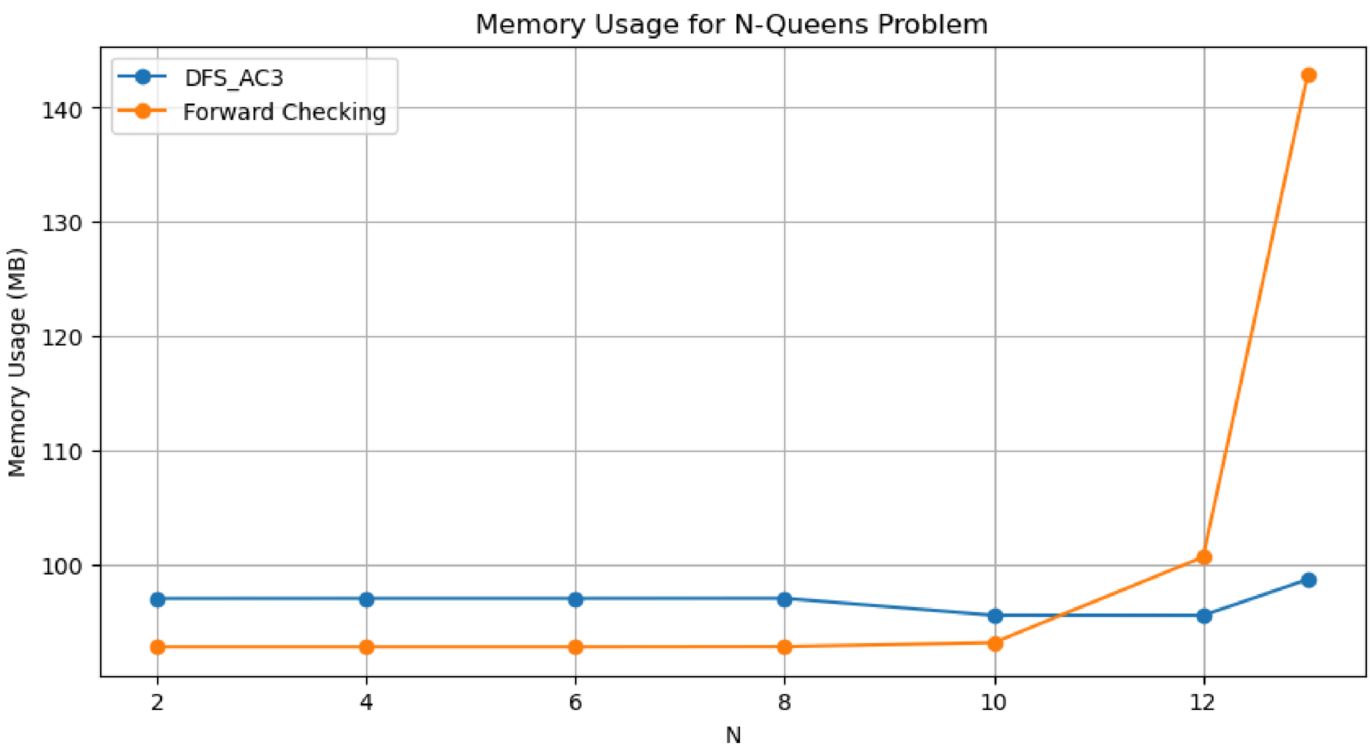
In [17]:

```
Ns = [2,4,6,8,10,12,13]



print("DFS WITH AC3====================== ")
dfsac3_execution_times = []
dfsac3_memory_usages = []
for n in Ns:
    execution_time3, memory_usage_during_execution3 = test_dfsac3_for_n(n)
    dfsac3_execution_times.append(execution_time3)
    dfsac3_memory_usages.append(memory_usage_during_execution3)

print("FORWAD CHECKING==================")
fc_execution_times = []
fc_memory_usages = []
for n in Ns:
    execution_time4, memory_usage_during_execution4 = test_forward_checking_for_n(n)
    fc_execution_times.append(execution_time4)
    fc_memory_usages.append(memory_usage_during_execution4)
```

```python
plt.figure(figsize=(10, 5))
plt.plot(Ns, dfsac3_execution_times, marker='o', label='DFS_AC3')
plt.plot(Ns, fc_execution_times, marker='o', label='Forward Checking')
plt.title('Execution Time for N-Queens Problem')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(Ns, dfsac3_memory_usages, marker='o', label='DFS_AC3')
plt.plot(Ns, fc_memory_usages, marker='o', label='Forward Checking')
plt.title('Memory Usage for N-Queens Problem')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.legend()
plt.grid(True)
plt.show()
```

```
DFS WITH AC3=====================
Removed value: 0 from domain of 0
Removed value: 1 from domain of 0
Revise done
For N = 2:
Number of solutions found: 0
Execution time: 0.0 seconds
Memory used during execution: 96.98046875 MB
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 96.98828125 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.005510807037353516 seconds
Memory used during execution: 96.98828125 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.14151787757873535 seconds
Memory used during execution: 96.99609375 MB
For N = 10:
Number of solutions found: 724
Execution time: 6.204867362976074 seconds
Memory used during execution: 95.49609375 MB
For N = 12:
Number of solutions found: 14200
Execution time: 317.35110306739807 seconds
Memory used during execution: 95.484375 MB
For N = 13:
Number of solutions found: 73712
Execution time: 2422.606921195984 seconds
Memory used during execution: 98.64453125 MB
FORWAD CHECKING==================
For N = 2:
Number of solutions found: 0
Execution time: 0.0 seconds
Memory used during execution: 92.73828125 MB
For N = 4:
Number of solutions found: 2
Execution time: 0.0 seconds
Memory used during execution: 92.73828125 MB
For N = 6:
Number of solutions found: 4
Execution time: 0.0 seconds
Memory used during execution: 92.73828125 MB
For N = 8:
Number of solutions found: 92
Execution time: 0.015026092529296875 seconds
Memory used during execution: 92.7578125 MB
For N = 10:
Number of solutions found: 724
Execution time: 0.28185272216796875 seconds
Memory used during execution: 93.10546875 MB
For N = 12:
Number of solutions found: 14200
Execution time: 9.088309288024902 seconds
Memory used during execution: 100.60546875 MB
For N = 13:
Number of solutions found: 73712
Execution time: 56.60212993621826 seconds
Memory used during execution: 142.90234375 MB
```

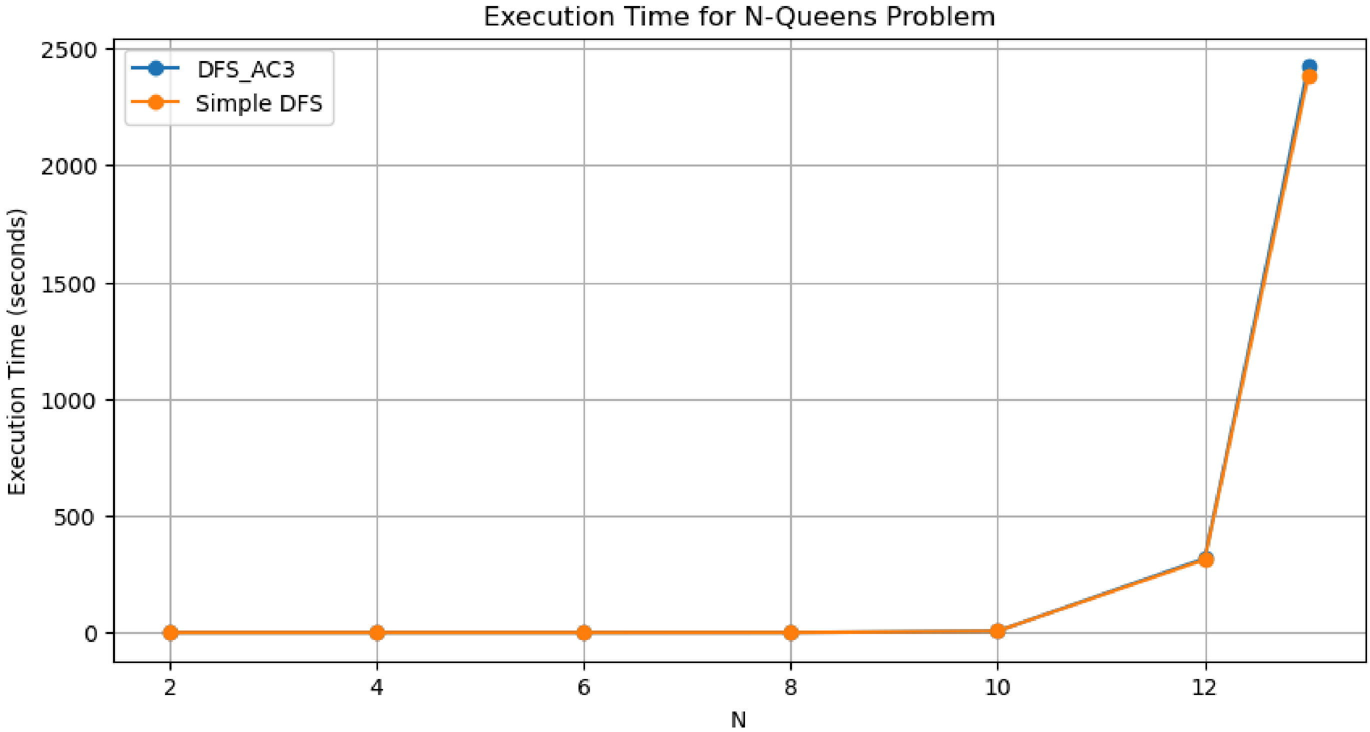


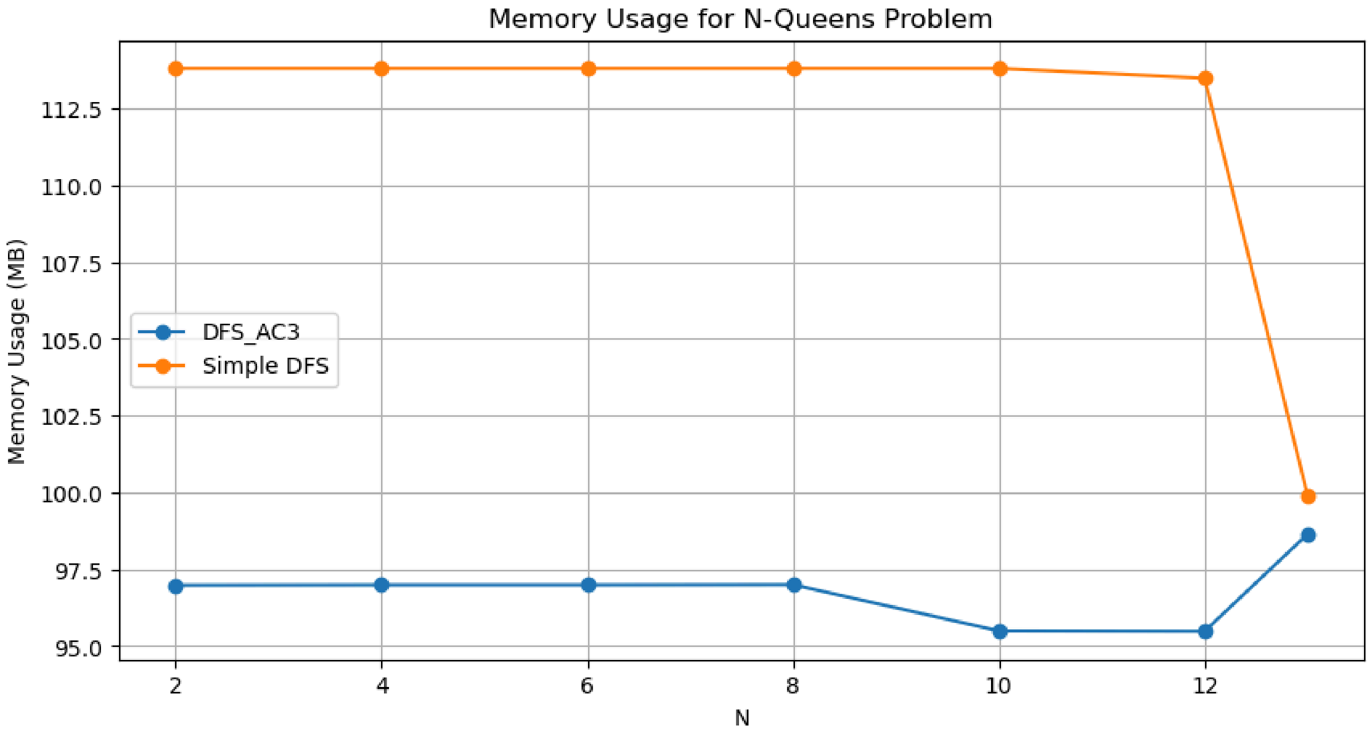




## DFS vs DFS_AC3

In [18]:

```
plt.figure(figsize=(10, 5))
plt.plot(Ns, dfsac3_execution_times, marker='o', label='DFS_AC3')
plt.plot(Ns, dfs_execution_times, marker='o', label='Simple DFS')
plt.title('Execution Time for N-Queens Problem')
plt.xlabel('N')
plt.ylabel('Execution Time (seconds)')
plt.legend()
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(Ns, dfsac3_memory_usages, marker='o', label='DFS_AC3')
plt.plot(Ns, dfs_memory_usages, marker='o', label='Simple DFS')
plt.title('Memory Usage for N-Queens Problem')
plt.xlabel('N')
plt.ylabel('Memory Usage (MB)')
plt.legend()
plt.grid(True)
plt.show()
```


Execution Time for N-Queens Problem
DFS_AC3
Simple DFS
Execution Time (seconds)
2500
2000
1500
1000
500
0
2
4
6
8
10
12
N


Memory Usage for N-Queens Problem
DFS_AC3
Simple DFS
Memory Usage (MB)
112.5
110.0
107.5
105.0
102.5
100.0
97.5
95.0
2
4
6
8
10
12
N