AI Course Project #2. Constraint Satisfaction Problem

N-Queens Problem Solver

Abdulla Akhundzada

March 4, 2025

1. Introduction

This document explains the implementation of an N-Queens Problem solver. The goal is to find a valid placement of N queens on an N x N chessboard such that no two queens threaten each other. This means no two queens can be in the same row, column, or diagonal. The solution is implemented using **Backtracking Search** enhanced with **Arc Consistency (AC3) constraint propagation** and the **Minimum Remaining Values (MRV) heuristic**.

This document will cover:

- **Algorithm Choice Justification:** Why Backtracking Search with AC3 and MRV is suitable for this problem.
- **Approach Explanation:** A detailed breakdown of the algorithm implementation, including state representation, constraints, heuristics, constraint propagation, and the search process.
- **Code Structure Explanation:** Overview of the project codebase and file descriptions.
- **Test Cases Explanation:** Description of implemented unit tests and their purpose.
- **How to Run:** Instructions on executing the solver and using command-line arguments.

2. Algorithm Choice: Backtracking Search with AC3 and MRV

Backtracking Search was chosen as the core algorithm for solving the N-Queens problem as it is a classic and effective method for Constraint Satisfaction Problems (CSPs). To enhance its efficiency, especially for larger board sizes, we incorporated **AC3 constraint propagation** and the **Minimum Remaining Values (MRV) heuristic**. Its key advantages in this context are:

- **Completeness:** Backtracking search is guaranteed to find a solution if one exists, by systematically exploring the search space. For the N-Queens problem, it will either find a valid queen placement or determine that no solution is possible.
- **Systematic Exploration:** It explores potential solutions in a depth-first manner, trying to place queens column by column. When a placement leads to a conflict, it backtracks and tries a different placement.
- **Efficiency Enhancements:** Plain backtracking can be inefficient for larger N values due to a large search space. To mitigate this, we integrate:
  - **AC3 Constraint Propagation:** This algorithm proactively reduces the search space by removing inconsistent values from the domains of variables. By enforcing arc consistency, we can detect failures earlier and prune branches of the search tree that are guaranteed to lead to conflicts, significantly improving performance.
  - **MRV Heuristic:** The Minimum Remaining Values (MRV) heuristic is used for variable ordering. It selects the variable (row in our case) with the fewest remaining valid choices (columns) in its domain. This heuristic aims to fail-fast, pruning the search space more effectively by focusing on the most constrained variables first.

3. Approach Explanation: Backtracking Search, AC3, and MRV Implementation

The implementation of the N-Queens problem solver combines Backtracking Search, AC3 constraint propagation, and MRV heuristic. Here are the key components:

3.1. State Representation

A **state** in this problem is implicitly represented during the recursive backtracking process. However, we can consider a *partial assignment* of queens to rows as the evolving state. Specifically:

- **Assignment (`self.assignment` in `NQueensSolver`):** A dictionary where keys are rows (integers from 0 to N-1) and values are columns (integers from 0 to N-1). `self.assignment` stores the column in which a queen is placed for each assigned row. An empty dictionary initially represents no queens placed. As the algorithm progresses, this dictionary is populated.
- **Domains (`self.domains` in `NQueensSolver`):** A dictionary where keys are rows (integers from 0 to N-1) and values are lists of possible columns (integers from 0 to N-1) where a queen can be placed in that row without immediately violating constraints *with respect to already placed queens*. Initially, for each row, the domain is all possible columns (0 to N-1). AC3 is used to reduce these domains by removing inconsistent values.

3.2. Constraints

The N-Queens problem has the following constraints:

- **Row Constraint:** Only one queen can be placed in each row (implicitly handled by the backtracking approach, as we assign queens row by row).
- **Column Constraint:** No two queens can be in the same column.
- **Diagonal Constraint:** No two queens can be on the same diagonal (both main and anti-diagonals).

These constraints are checked in the `Position.is_attacking(other)` method and the `NQueensSolver._is_valid_placement(po` method.

3.3. Minimum Remaining Values (MRV) Heuristic

The MRV heuristic is implemented in the `NQueensSolver._select_unassigned_variable(domains)` method.

Explanation of MRV Heuristic:

1. **Identify Unassigned Variables:** It finds rows (variables) that are not yet in the `self.assignment`.
2. **Select Variable with Smallest Domain:** From the unassigned rows, it selects the row that has the fewest remaining valid column choices in its `domains`. The intuition is that choosing the most constrained variable first is more likely to lead to a conflict sooner if a solution is not possible, allowing for earlier pruning of the search space.

3.4. AC3 Constraint Propagation

The AC3 algorithm is implemented in the `NQueensSolver._enforce_arc_consistency(domains)` and `NQueensSolver._revise_arc(row1, row2, domains)` methods.

Explanation of AC3 Implementation:

1. **Arcs:** In the N-Queens problem, an "arc" is a directed constraint between two rows (variables). For every pair of rows (row1, row2), we consider the arc from row1 to row2.
2. **Arc Consistency:** An arc from row1 to row2 is consistent if for every value in the domain of row1, there is at least one value in the domain of row2 that does not violate the constraint between row1 and row2 (i.e., the queens are not attacking each other).
3. **AC3 Algorithm (`_enforce_arc_consistency`):**
   - **Initialization:** A queue is initialized with all arcs (all ordered pairs of rows (row1, row2) where row1 != row2).
   - **Queue Processing:** While the queue is not empty:
     - An arc (row1, row2) is dequeued.
     - **`_revise_arc(row1, row2, domains)`:** This method checks if the arc from row1 to row2 is consistent. For each value in the domain of `row1`, it checks if there is a consistent value in the domain of `row2`. If for a value in `row1` there is no consistent value in `row2`, that value is removed from the domain of `row1`. The `_revise_arc` method returns `True` if the domain of `row1` was revised (values removed), and `False` otherwise.

- **Domain Wipeout Check:** If `_revise_arc` removed values from the domain of `row1`, and the domain of `row1` becomes empty (domain wipeout), it means there is no solution from the current partial assignment, and AC3 returns `None`.
- **Queue Update:** If the domain of `row1` was revised, all arcs pointing to `row1` (i.e., (row3, row1) for all row3 != row1 and row3 != row2) are added back to the queue, as the revision of `row1`'s domain might affect the consistency of these arcs.

4. **`_revise_arc(row1, row2, domains)`:** This method iterates through each column (`col1`) in the domain of `row1`. For each `col1`, it checks if there exists *any* column (`col2`) in the domain of `row2` such that placing queens at `(row1, col1)` and `(row2, col2)` does not result in an attack. If no such `col2` exists in the domain of `row2` for a given `col1`, it means `col1` is inconsistent with respect to `row2` and the current domains, so `col1` is removed from the domain of `row1`.

3.5. Backtracking Search Process (`_backtrack_with_ac3`)

The backtracking search algorithm, enhanced with AC3, is implemented in the `NQueensSolver._backtrack_with_ac3(current_do` function:

1. **Base Case:** If the number of assigned rows (`len(self.assignment)`) equals the board size (`self.size`), it means a complete and valid assignment has been found. The function returns `self.assignment`.
2. **Variable Selection (MRV):** Select an unassigned row using the MRV heuristic via `self._select_unassigned_variable(c` If no unassigned row remains (all rows are assigned, but base case not met - should not happen due to base case check), return `None`.
3. **Domain Iteration:** Iterate through each column (`column`) in the domain of the selected row (`row`) from `current_domains[row]`.
4. **Validity Check:** For each `column`, check if placing a queen at `(row, column)` is valid with respect to already placed queens using `self._is_valid_placement(Position(row, column))`.
5. **Assignment and Constraint Propagation:**
   - If the placement is valid:
     - Assign the column to the current row: `self.assignment[row] = column`.
     - **AC3 Propagation:** Create local domains by copying `current_domains` and fixing the domain of the current row to just the assigned column. Then, call `self._enforce_arc_consistency(local_domains)` to propagate constraints and revise domains.
     - **Recursive Call:** If AC3 does not lead to domain wipeout (`revised_domains` is not `None`), recursively call `self._backtrack_with_ac3(next_domains)` with the revised domains to try to extend the assignment to the next row.
     - **Solution Found:** If the recursive call returns a valid assignment (`result` is not `None`), it means a complete solution has been found. Return `result`.
   - **Backtracking:** If the placement is invalid or the recursive call fails to find a solution:
     - Backtrack: remove the assignment for the current row: `del self.assignment[row]`. This undoes the current assignment and the algorithm will try the next column in the domain of the current row in the loop.
6. **No Solution:** If all columns in the domain of the selected row have been tried, and no solution is found, it means no solution exists for the current partial assignment. Return `None`.

4. Code Structure Explanation

The project codebase is organized as follows:

```
nQueens_project
|-- nQueens.py
|-- utils.py
|-- main.py
|-- test.py
```

- **`nQueens.py`:** This file contains the core algorithm implementation for solving the N-Queens problem:
  - `Position` class: Represents a position on the chessboard and includes the `is_attacking` method to check for attacks between queens.
  - `NQueensSolver` class:

        - `__init__(self, board_size)`: Constructor to initialize the solver with the board size, domains, and assignment.
        - `solve(self)`: The main function to initiate the solving process and return the solution assignment.
        - `_backtrack_with_ac3(self, current_domains)`: Implements the recursive backtracking search algorithm with AC3 constraint propagation and MRV heuristic.
        - `_select_unassigned_variable(self, domains)`: Implements the MRV heuristic to select the next variable (row) to assign.
        - `_is_valid_placement(self, position)`: Checks if placing a queen at the given position is valid with respect to already placed queens.
        - `_enforce_arc_consistency(self, domains)`: Implements the AC3 algorithm to enforce arc consistency.
        - `_revise_arc(self, row1, row2, domains)`: A helper function for AC3 to revise the domain of `row1` based on the domain of `row2`.
- **`utils.py`:** Contains utility functions:
    - `print_board_representation(assignment)`: Prints a simple text-based representation of the chessboard with queen placements.
    - `is_valid_assignment(assignment)`: Verifies if a given queen assignment is valid (no two queens attack each other). Used for testing.
- **`main.py`:** The main entry point of the program.
    - Parses command-line arguments using `ArgumentParser`:
        - `--n`: The size of the N x N chessboard (default: `16`).
        - `--dont_print_result`: Flag to suppress printing the raw solver output (default: `False`).
        - `--dont_print_representation`: Flag to suppress printing the board representation (default: `False`).
        - `--dont_print_timing`: Flag to suppress printing the elapsed time (default: `False`).
    - Creates an `NQueensSolver` instance with the specified board size.
    - Calls `solver.solve()` to find a solution.
    - Prints the elapsed time, the raw solver output, and the board representation based on command-line flags.
- **`test.py`:** Contains unit tests for verifying the correctness of the `NQueensSolver`. Uses the `unittest` framework.
    - `TestNQueensSolver` class:
        - `test_random_1(self)`, `test_random_2(self)`, `test_random_3(self)`: Test cases that randomly generate board sizes within a range (`N_MIN` to `N_MAX`) and assert that the solver finds a valid solution using `is_valid_assignment`.
        - `test_case(n)`: A helper function used by test cases to run the solver for a given board size `n` and check the validity of the solution.

5. Test Cases Explanation

The `test.py` file implements unit tests to ensure the `NQueensSolver` function works correctly. The test cases cover:

- **Randomized Board Sizes:** The tests `test_random_1`, `test_random_2`, and `test_random_3` each generate a random board size `n` between `N_MIN` (10) and `N_MAX` (150). This is designed to test the solver's performance and correctness over a range of problem sizes, including relatively large boards.
- **Solution Validity Check:** For each test case, after running the solver, the `is_valid_assignment` function from `utils.py` is used to verify that the returned queen assignment is indeed a valid solution to the N-Queens problem. This ensures that no two queens in the returned assignment are attacking each other.
- **Timing Information:** Each test case also prints the time taken to solve the N-Queens problem for the given board size. This provides a basic performance metric and allows observing how the solving time scales with increasing board size.

These tests use the `unittest` framework in Python and assert that for each random board size, the `is_valid_assignment` function returns `True`, confirming that the solver found a valid solution. Running `python test.py` executes all defined tests and reports any failures.

6. How to Run

To run the N-Queens Problem Solver, follow these steps:

1. **Save the project files:** Ensure you have saved all the provided Python files (`nQueens.py`, `utils.py`, `main.py`,

`test.py`) in the same directory.

2. **Execute `main.py`:** Run the `main.py` script from the command line using `python main.py`.

   Command-Line Arguments:

   The `main.py` script accepts the following optional arguments:

   - `--n <N>` (default: `16`): Specifies the size of the N x N chessboard.
     - Example: `python main.py --n 8` (solves for an 8x8 board)
   - `--dont_print_result`: Suppresses printing the raw solver output (the assignment dictionary).
     - Example: `python main.py --dont_print_result`
   - `--dont_print_representation`: Suppresses printing the text-based board representation.
     - Example: `python main.py --dont_print_representation`
   - `--dont_print_timing`: Suppresses printing the elapsed solving time.
     - Example: `python main.py --dont_print_timing`

   You can combine these arguments as needed. For example, to solve for an 8x8 board and only see the board representation:

   ```
   python main.py --n 8 --dont_print_result --dont_print_timing
   ```

   To see the full list of arguments and their descriptions, use the `--help` flag:

   ```
   python main.py --help
   ```

   Output:

   By default, the program will output:

   - The elapsed solving time.
   - The raw solver output (the assignment dictionary mapping rows to columns).
   - A text-based representation of the chessboard showing queen placements ('Q') and empty squares ('.').

   The output can be controlled using the command-line arguments described above.

Testing

To execute the tests, run the following command in the terminal in the project directory:

```
python test.py
```

Output must give a result similar to the following output example:

```

Assigned N: 75

Solved in 19.424 seconds.

.
Assigned N: 12

Solved in 0.020 seconds.

.
Assigned N: 51

Solved in 2.124 seconds.

.
----------------------------------------------------------------------
Ran 3 tests in 21.568s

OK
```

The exact runtime will vary depending on your system and the randomly chosen board sizes in the tests. If the implementation is successful, all test cases will pass and indicate "OK". Failures will indicate potential issues in the code that need to be debugged.

7. Conclusion

In this project, an N-Queens Problem Solver has been implemented using Backtracking Search enhanced with AC3 constraint propagation and the MRV heuristic. This approach effectively finds solutions for the N-Queens problem, leveraging constraint propagation to prune the search space and the MRV heuristic to guide the search efficiently, especially for larger board sizes. One thing to note that too large board sizes, for example, 900, take too much time on a personal computer, without implementation of paralellization, which shows there is still room for improvement in the implementation case. The provided test cases ensure the correctness of the implementation and demonstrate its capability to solve the N-Queens problem for various board dimensions.