N-Queens Solvers & Performance Analysis

Zualdi Luca, 2269867

Introduction

The objective of this homework is to explore, implement and compare two different approaches of AI to solve the N-queens problem. The problem requires to position N queens on a chessboard having NxN dimension so that the queens cannot attack each other. In practice, every couple of queens cannot position on the same row, column or diagonal.

The problem has been addressed using two different methods:

1. **A Star algorithm**: Use of a heuristic function to find the optimal path to a solution.
2. **Constraint Satisfaction Problem (CSP)**: A constraint-based approach, the z3 solver has been used to find solutions.

Task 1: Problem

To solve the N-queens problem, the problem has been implemented in a python class as it follows:

Using N as parameter, the class creates an empty NxN grid where a 0 cell means that it's empty and 1 that there is a queen in that cell. This is the initial state.

The action(row, col) function, adds a queen in the selected cell if there is not a queen in that column.

To check the number of conflicts between queens on the grid, the conflicts() function obtains the cells where queens are placed. After that, the function checks that for every couple of queens if there is a conflict.

The is_solved() function returns True if there are n queens on the grid and there are no conflicts.

A possible_actions() function has been implemented to check which cell are free to be occupied. The function returns an array of (row, col) couples.

In the end, there is a function to print the grid and one to load a different state (grid) in the problem.

Task 2.1: Implementation of A

The A star algorithm has been implemented as a class that takes a start state and the problem class to solve. Other values of the class are the frontier, implemented as a priority queue, the explored set, came_from, a dictionary that associate a parent to every explored state and two caches for heuristic functions and g, implemented as dictionary too.

In particular, the heuristic function implemented for this problem is the number of conflicts on the grid.

The a_star_search() function is responsible of the entire search: the functions start putting the first state in the frontier. For every state in the queue, the function checks that the problem is solved in that state. If not, the possible actions are calculated and every node expanded and placed in the frontier, if is not explored yet and is g in less than the g in cache for that state.

When goal is reached, the path is reconstrued using the specific function.

Is noted that the class is not very modular for other types of problem. To do that, just change the heuristic function and the method to load states and handle possible actions.

In the end, the class implement also some statistics value to execute experiments.

Task 2.2: Implementation of CSP

Like A star, CSP has been implemented as a class too. Using the z3 library, n variables are created. For each one, these constrains are added:

1. $0 \leq Row_i < n$
2. $Row_i \neq Row_j$
3. $|row_i - row_j| \neq |i - j|$

The class includes also a function to print the solution.

Task 3: Experimental results

The experiment has been conducted using different values of N.

For CSP N goes from 4 to 19. Due the computation time, for A star N goes only from 4 to 7.

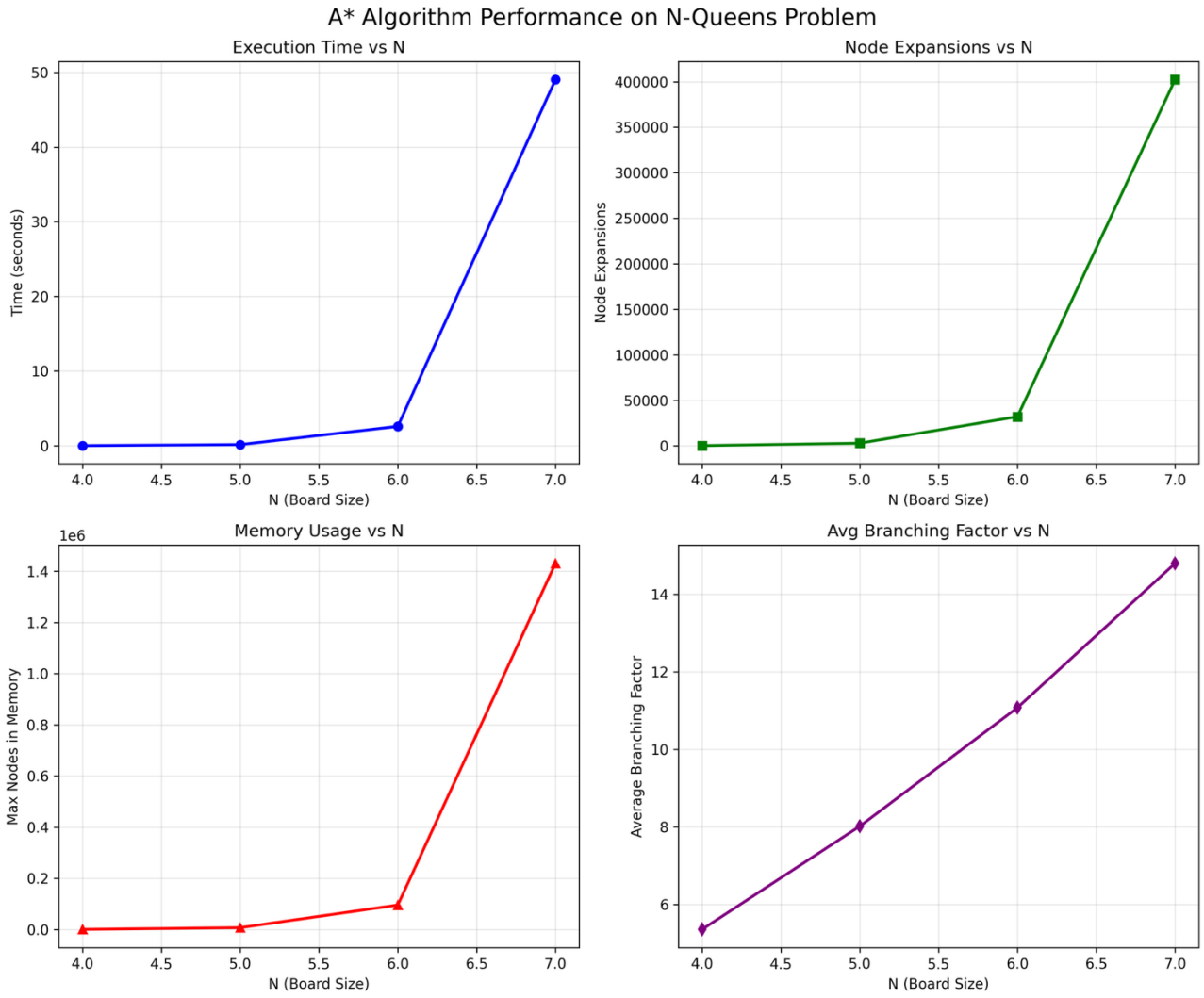
For A star the statistics include:

- Running time
- Expanded nodes
- Branching factor
- Maximum number of nodes in memory

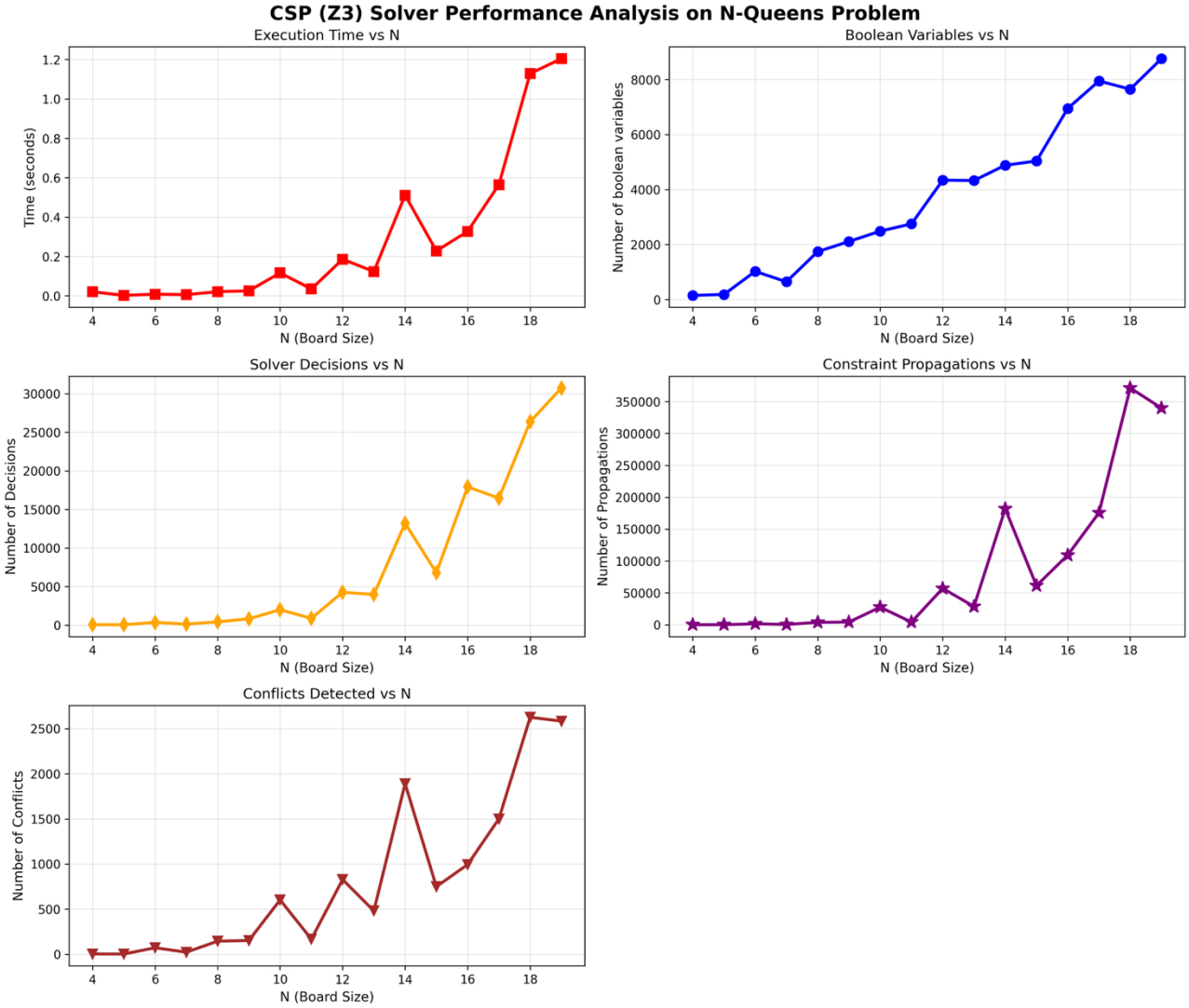
For CSP:

- Running time
- Decisions: number of hypotheses made
- Conflicts: number of impossible solutions found
- Propagations: number of constrain propagation
- Boolean variables: auxiliary variables created by solver

The following images show the plots for A star and CSP experiment respectively:



It's notable that time, node expansion and memory usage grows exponentially and average branching factor grows linearly.



Like in the A star experiment, in CSP the running time grows exponentially too. Also, constraint propagations, conflicts and decisions seem like to grow in that way. Boolean variables grow linear instead.

It is concluded that CSP performs better for this type of problems, handling larger grid in less time than A star algorithm.

How to run

Requirements:

The project is managed via **PDM** and requires:

- Python **3.11**
- **z3-solver** (>=4.15.4.0)
- **matplotlib** (>=3.10.8) for generating performance plots.

Installation:

Ensure you have PDM installed on your system.

1. Clone the repository.
2. Install dependencies:

```
pdm install
```

Usage:

You can run the solvers or the experiment suites directly via Python (or pdm run).

1. Run Individual SolversCSP Solver (Z3)

Solves a single instance (default N=20):

```
pdm run python csp.py
```

A Solver Solves a single instance (default N=4) and prints the steps.

```
pdm run python A_star.py
```

2. Run Performance Experiments

CSP Experiments Runs the Z3 solver for N=4 to 19.

```
pdm run python experiment_csp.py
```

- **Output:** Generates csp_z3_experiments.png containing 5 plots (Execution Time, Constraints vs N, Propagations, etc.).

A Experiments Runs the A* algorithm for N=4 to 7.

```
pdm run python experiment_Astar.py
```

- **Output:** Generates a_star_experiments.png containing 4 plots (Time, Node Expansions, Memory Usage, Branching Factor).