n-Queens problem: modeling and solving

Team leaders: Ferre' D., Pizzoli D.

Academic year 2021/2022

Contents

1 Problem definition

The n-Queens problem [3] is a well-known problem in constraint programming. The problem involves placing n queens on a chessboard in such a way that none of them can capture any other using the allowed moves.

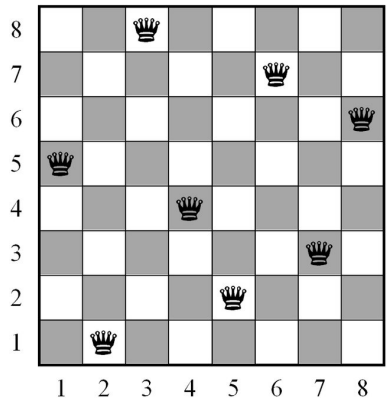


Figure 1: A possible solution to the 8-queens problem.

In other words, the problem is to select n squares on a chessboard so that any pair of selected squares is never aligned vertically, horizontally, nor diagonally (Figure 1 [2]).

2 Code and models

The public GitHub repository at https://github.com/Boufalah/ModelisationPPC-PL hosts all the code for this assignment. You can see a visual representation of the structure of the repository in section 5. All the relevant code is located at `src/main/java/nqueen`.

Several models were used to solve the n-Queens problem.

2.1 Primal model

Instance of the problem

The input is an array X of n integer variables, one for each queen. The index i of the i-th variable in X is the row of the queen in the matrix, while the value X_i is the column of that queen.

The output of this solver is a list of column indexes X_i, $\forall i \in [1; n]$ such that each X_i represent the column of the queen on row i.

We considered 2 variants of this strategy: the first one uses the built-in `allDifferent` global constraint, whereas the second one explicitly declares a group of binary inequality relations. The implementation of both models was provided by the Choco-solver documentation [1].

2.2 Row-Column model

Instance of the problem

In this case we have 2 arrays R, C each of n integer variables. For each index i, we have to find the row R_i and the column C_i of the i-th queen. For this model we used the global constraint `allDifferent` on both R and C and some additional constraints to take care of diagonals. These constraints are:

$$R_i - C_i \neq R_j - C_j \quad \wedge \quad R_i + C_i \neq R_j + C_j, \quad \forall\, i, j \in [1, n]$$

The output of the solver is a list of coordinates $(R_i, C_i),\ \forall i \in [1; n]$ that satisfy the constraints of the n-Queens problem.

2.3 Boolean model

Instance of the problem

The input is a matrix X of $n \times n$ variables in $[0, 1]$, one for each cell of the chessboard. A variable is *true* if there is a queen in that specific cell. For each variable, we define if the position contains a queen (*true*) or if it is empty ($false$). The output of the solver is a matrix where a cell is *true* if there is a queen in it.

We considered 2 variants of this strategy: the first one uses integer variables in $[0; 1]$, the second one uses boolean variables.

2.3.1 Variant with IntVar variables

In this variant we used the sum of values to count the number of queens for each row and column. We used an auxiliary matrix Y, which is basically X rotated by 90°, to write less complex constraints. Finally, here are the constraints for the diagonals:

$$X_{i,j} + X_{i+k,j+k} \leq 1 \quad \wedge \quad Y_{i,j} + Y_{i+k,j+k} \leq 1$$

$$\forall\, i, j, k \in [1, n] \quad | \quad i + k \leq n \ \wedge \ j + k \leq n$$

2.3.2 Variant with BoolVar variables

Choco-solver provides some handy methods for expressing constraints on boolean variables (this kind of constraints are referred to as SAT constraints in the Choco documentation). In this implementation we used the built-in methods `addClausesAtMostOne` and `addClausesBoolOrArrayEqualTrue`, which both take a boolean array as input. The former allows at most one *true* value, while the latter allows at least one *true* value. When combined they only allow a single value to be *true*. Diagonal constraints are implemented with the same logic as the previous variant.

2.4 Primal-Dual model

Instance of the problem

In a similar way to what we did with the primal model, we could define a dual model that is based on columns instead of rows. In the dual model we would have an array of queens Y, where Y_i is the row of the queen on column i. The idea is to declare both the primal and the dual models and then "link" them to find common solutions.

Let's suppose that X is the array used by the primal model and Y the one used by the dual model. After declaring both the models we can "link" their solution by using a constraint that enforces:

$$Y_j = i \leftrightarrow X_i = j$$

The solver returns a solution for both the arrays X and Y such that the constraint is satisfied.

As with the primal model, we considered 2 variants of this strategy: the first one uses the built-in `allDifferent` global constraint for both the primal and the dual model, whereas the second one explicitly declares a list of binary inequality relations.

2.5 Model with custom propagator

Instance of the problem

We used the same setting as the primal model, but with a single constraint (described in [3]). The input is therefore an array X of n integer variables, one for each queen. The index i of the i-th variable in X is the row of the queen in matrix, the value X_i is the column of that queen.

For the implementation of the filtering algorithm, we followed the rules described in [3], which we briefly report here. Let i be a queen and $D(j)$ the domain of the j-th queen. We have 4 rules:

1. $D(i) = \{a, b, c\} \mid a < b < c \wedge b = a + k \wedge c = b + k \implies$ Remove b from $D(i-k)$ and $D(i+k)$.
2. $D(i) = \{a, b\} \mid a < b \implies$ Remove a, b from $D(i-(b-a))$ and $D(i+(b-a))$.
3. $D(i) = \{a\} \implies$ Remove $a + j$ from $D(i+j)$ and $a - j$ from $D(i-j)$.
4. If a queen has more than 3 values in its domain, we cannot deduce anything.

Note: We think that rule 3 should also remove a from all of the other domains to avoid having two queens in the same column:

3. $D(i) = \{a\} \implies$ Remove $a + j$ from $D(i+j)$, $a - j$ from $D(i-j)$ and a from $D(k), \forall\ k \neq i$.

This model uses a single constraint that encapsulates all of the rules above. For this reason, it is crucial that the propagator associated with the constraint is idempotent. A propagator p is idempotent if $p(D) = p(p(D))$ for all domains D. A idempotent propagator basically ensures constraints to be satisfied.
If you want to know more about how our custom propagator ensures idempotence, take a look at the code and the comments of the `CustomProp` class.

3 Results

We evaluated and compared the different models to test their efficiency. We focused on two metrics in particular: the resolution time for enumerating all solutions and the number of nodes in the full search tree.

We didn't consider the Row-Column model in this experimental section. This is because it is highly inefficient as a consequence of all the unwanted symmetries that it introduces in the solution space. Every solution is represented as a set of n pairs (row, col); thus, for every legitimate solution, there are actually $n!$ permutations of these pairs that are considered different by the solver.
The total number of solutions returned by the solver is then $n! * k$, where k is the number of legitimate solutions.
Example: if $n = 6$ there are $k = 4$ legitimate solutions, with this model the solver returns $6! * 4 = 2880$ solutions. This is definitely not a good modeling choice for this problem.

Each model was tested ten times for each value of $n \in [4; 12]$ and we calculated the average resolution time. If you would like to run the tests yourself, the `Benchmark` class was designed for that. The results are stored in two files at `src/main/java/nqueen`:

- `resolution_enum_stats.csv`;
- `nodes_stats.csv`.

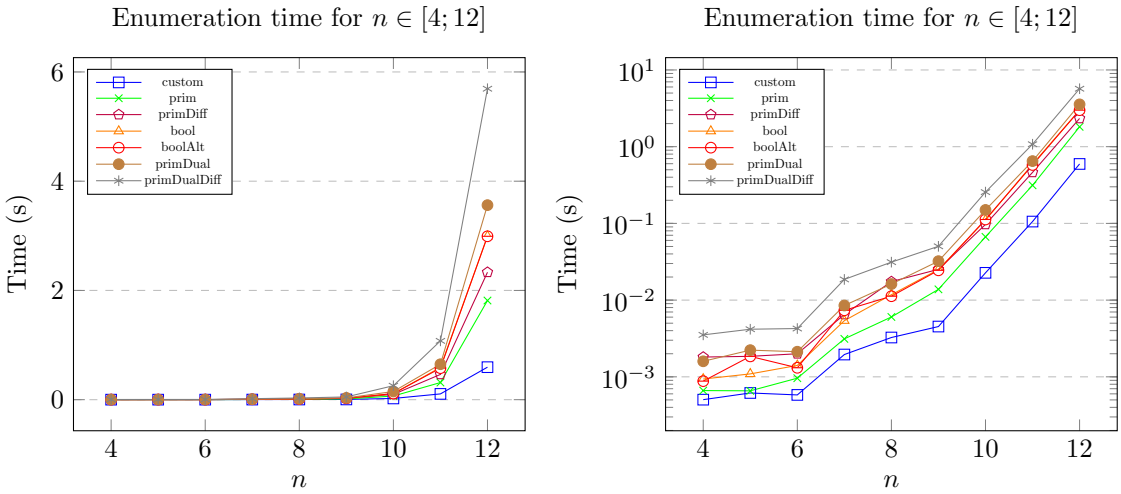
For each chart we have produced two variants that use different scales for the vertical axis, one is normal and the other is logarithmic.

3.1 Resolution time for enumeration

In this section, we show the results related to the time taken by the solver to enumerate all the solutions with different models.

n	custom	prim	primDiff	bool	boolAlt	primDual	primDualDiff
4	0.5	0.7	1.8	0.9	0.9	1.6	3.5
5	0.6	0.7	1.9	1.1	1.8	2.2	4.2
6	0.6	1.0	2.0	1.4	1.3	2.1	4.3
7	2.0	3.1	6.5	5.4	7.4	8.5	18.6
8	3.3	6.0	17.4	11.8	11.2	16.2	31.3
9	4.5	13.8	25.2	25.0	24.4	32.2	50.3
10	22.7	66.8	97.0	110.3	112.4	149.8	255.8
11	105.7	313.7	462.3	591.4	577.1	648.7	1'076.6
12	595.7	1'815.3	2'332.5	2'992.4	2'991.1	3'561.6	5'692.8

Table 1: Resolution times for enumerating all solutions, with $n \in [4; 12]$. Every value is the average of 10 executions. Times are expressed in milliseconds.



(a) Data from Table 1 with a normal scale.

(b) Data from Table 1 with a logarithmic scale.

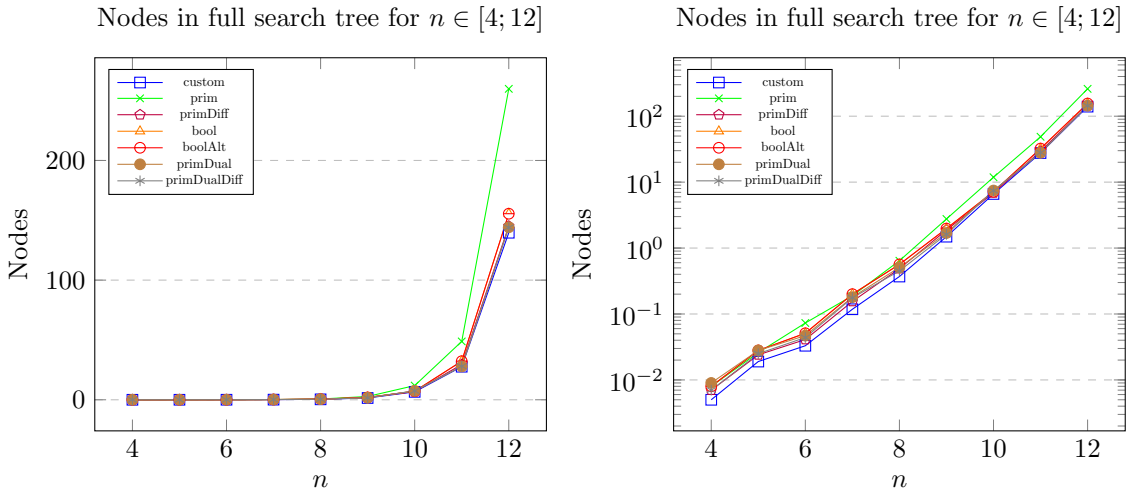
Figure 2: Charts for resolution times in Table 1 in seconds

3.2 Number of nodes

In this section, we show the results related to the number of nodes in the full search tree for each model.

n	custom	prim	primDiff	bool	boolAlt	primDual	primDualDiff
4	5	8	7	8	8	9	7
5	19	26	24	28	28	28	25
6	33	73	41	51	51	47	44
7	118	189	155	200	200	180	176
8	370	642	488	565	565	509	451
9	1'489	2'761	1'873	1'974	1'974	1'691	1'615
10	6'612	11'863	6'871	7'159	7'159	7'458	7'417
11	27'638	48'764	29'478	32'252	32'252	27'852	28'053
12	139'604	259'853	145'128	155'503	155'503	144'170	143'978

Table 2: Number of nodes of the full search tree for $n \in [4; 12]$.



(a) Data from Table 2 with a normal vertical scale.

(b) Data from Table 2 with a logarithmic scale.

Figure 3: Number of nodes for $n \in [4; 12]$ and different models. Values expressed in K of nodes.

4 Conclusions

By looking at the results, it is clear that the worst performing model is the Primal-Dual, in both its variants. When compared with the Primal model (especially the non-allDifferent variants), we can see that, even though the search tree has fewer nodes, the resolution time is approximately double. The huge number of contraints of the Primal-Dual model is probably slowing down the filtering algorithm too much.

The Boolean models are not the best performance-wise, but they still proved to be faster then the Primal-Dual models.

It is interesting to look at the results of the Primal and the Primal-allDifferent models; even though the Primal is faster, the number of nodes in the Primal-allDifferent is significantly lower. This is expected since there are situations in which the global `allDifferent` constraint can prune some values that a simpler clique of inequality constraints is not able to, because it lacks a "global" view.

The Primal model performed really well, especially the non-allDifferent variant.
The real winner, though, is definately the Custom model. It is considerably faster than the second best (the Primal model) and we can also notice that its search tree has the lowest number of nodes, which suggests that it has a strong filtering algorithm.

This small project allowed us to take our first steps into constraint programming and to better understand the theoretical concepts presented during the lectures.

References

[1] *Choco-solver n-queens tutorial*. URL: `https://choco-solver.org/tutos/first-example/`.

[2] Nasrin Mohabbati Kalejahi, Hossein Akbaripour, and Ellips Masehian. "Basic and Hybrid Imperialist Competitive Algorithms for Solving the Non-attacking and Non-dominating n-Queens Problems". In: vol. 577. Jan. 2015, pp. 79–96. ISBN: 978-3-319-11270-1. DOI: `10.1007/978-3-319-11271-8_6`.

[3] Jean-Charles Régin. "Global Constraints and Filtering Algorithms". In: (Jan. 2003). DOI: `10.1007/978-1-4419-8917-8_4`.

5 Repository structure

```
/
├── README_md
├── modelistation_iml
├── nodes_stats_csv
├── pom_xml
├── resolution_enum_stats_csv
├── src
│   └── main
│       └── java
│           └── nqueen
│               ├── BaseQueenModel_java
│               ├── Benchmark_java
│               ├── BooleanModel_java
│               ├── BooleanModelAlt_java
│               ├── Callable_java
│               ├── CustomModel_java
│               ├── CustomProp_java
│               ├── PrimalDiffModel_java
│               ├── PrimalDualDiffModel_java
│               ├── PrimalDualModel_java
│               ├── PrimalModel_java
│               ├── RowColumnModel_java
│               └── Utilities_java
└── target
    ├── classes
    │   └── nqueen
    │       ├── BaseQueenModel$Stats_class
    │       ├── BaseQueenModel_class
    │       ├── Benchmark$EnumModels_class
    │       ├── Benchmark_class
    │       ├── BooleanModel_class
    │       ├── BooleanModelAlt_class
    │       ├── Callable_class
    │       ├── CustomModel_class
    │       ├── CustomProp_class
    │       ├── PrimalDiffModel_class
    │       ├── PrimalDualDiffModel_class
    │       ├── PrimalDualModel_class
    │       ├── PrimalModel_class
    │       ├── RowColumnModel_class
    │       └── Utilities_class
    └── generated-sources
        └── annotations
```