n-Queens problem: modeling and solving

Team leaders: Ferre' D., Pizzoli D.

Academic year 2021/2022

Contents

1 Code and problem definition

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 Appendix A. All the relevant code is located at `src/main/java/nqueen`.

The n-Queens problem [5] 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.

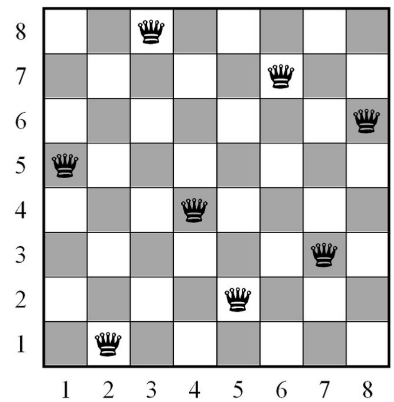


Figure 1: A possible solution to the 8-queens problem.

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 [3]).

2 Models

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 [2].

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 [5]). 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 [5], 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.

In order to have a visual representation of the work done by our custom propagator we have written the method `testPropagate` in the `Benchmark` class. This method takes the dimension n as a parameter and prints all of the steps taken by the propagator on a chessboard. You can see the output of the method `testPropagate`, with $n = 8$, in Appendix B. Note that:

- The methods `removeOne`, `removeTwo` and `removeThree` are called when the size of the domain of a variable is respectively one, two or three, and they implement the first three rules previously described.
- Some of the steps taken by the propagator are redundant, so in Appendix B we didn't print the chessboard for those steps.
- In Appendix B, notice that a variable in instantiated before every call of the propagate method (except for the first one).

3 Search strategies

Constraint propagation alone is almost never enough to build a solution, i.e. to instantiate all the variables. We need to explore the search space by using one or more search strategies. The task of a search strategy is to make decisions. A decision involves the selection of a variable and the selection of a value to assign to it. A decision taken by a search strategy may trigger constraint propagation.

We have tested several search strategies to see how they would perform:

- `domOverWDeg`: this is the default search strategy used by Choco for integer and boolean variables, and it is described in the work of F. Boussemart et al. [1]. The underlying idea is to select the variable that has the smallest $\frac{\text{DOMAIN\_SIZE}}{\text{WEIGHTED\_DEGREE}}$ ratio. A weighted degree is associated to each variable. During the search process, when a constraint is violated, the weighted degree associated with its variables is increased. As the search progresses, the weight of variables in hard constraints increases and this particularly helps the heuristic to select variables that belong to the "hard part" of the CSP. This heuristic follows the fail-first principle [4]: "To succeed, try first where you are most likely to fail".
- `minDomLB`: this is another search strategy based on the fail-first principle, but it is simpler then `domOverWDeg`. It selects the variable with the smallest domain and instantiates it with its lower bound.
- `minDomUB`: same as `minDomLB`, but instantiates the variable with its upper bound.
- `inputOrderLB`: this is very simple search strategy. It select the first variable that was added to the model and instantiates it with its lower bound.
- `inputOrderUB`: same as `inputOrderLB`, but instantiates the variable with its upper bound.
- `random`: this search strategy selects a random variable and instantiates it with a random value in its domain.

4 Results

In the first part of the analysis, 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. Note that here we have used the default `domOverWDeg` search strategy provided by Choco for int and bool variables.

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`;
- `search_stats.csv`;
- `nodes_stats.csv`.

For each chart of the first part, we have produced two variants that use different scales for the vertical axis, one is normal and the other is logarithmic. In the second phase, we provided only charts in logarithmic scale.

Testing with jUnit The number of solutions enumerated by each model was tested with jUnit. The `nQueenTester` class provides 4 simple tests: one for each category of models. In each test, we check the number of total solutions for $n \in [4; 12]$ and we compare it with the state of the art. Even though we do not check the validity of every solution, we think that this approach is enough to check the correctness of the models.

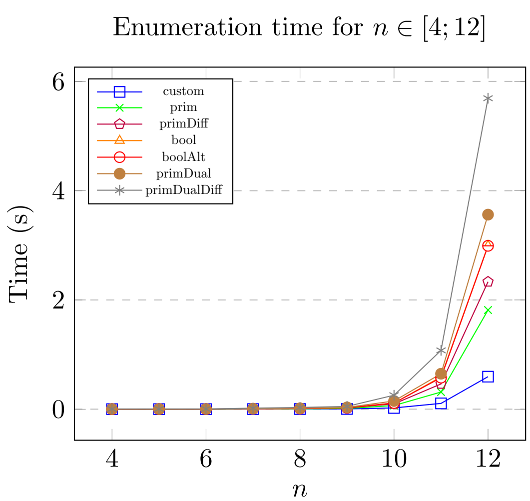
In the second part of the analysis, we took the fastest models and we tried several search strategies to see if there were some significant differences when looking for a single solution. The search strategies that we tested are those described in chapter 3.
We run these tests for $4 \leq n \leq 500$. Some search strategies became extremely slow for n greater than a certain value of k. When this happened, those search strategy were not considered for $n \geq k$, as you can see from the charts.

4.1 Enumeration - resolution time

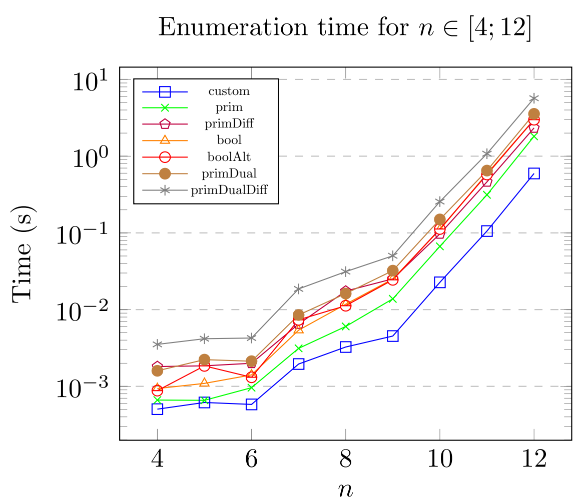
In this section, we show the results related to the time taken by the solver to enumerate all the solutions with different models.

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.

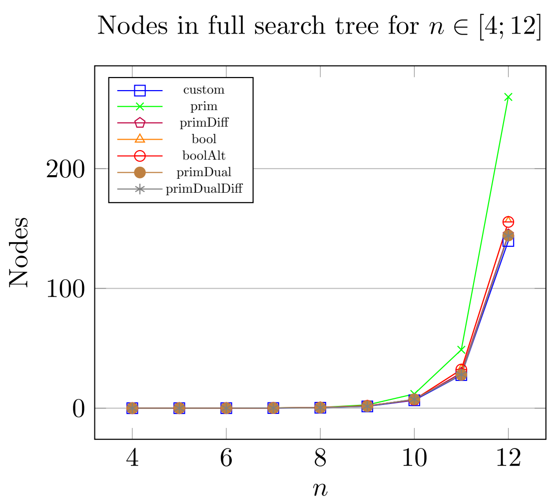
Figure 2: Charts for resolution times in Table 1 in seconds

4.2 Enumeration - number of nodes

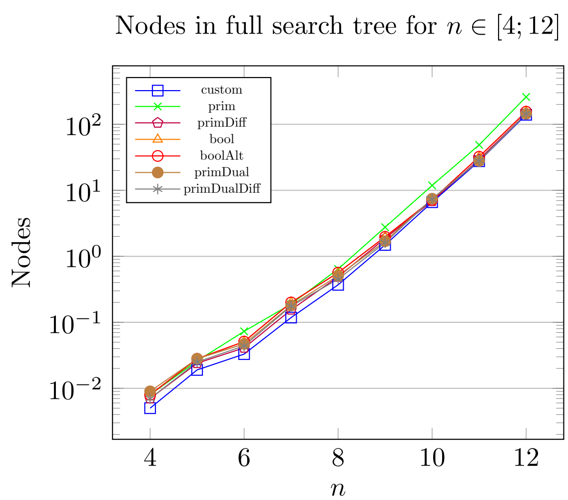
In this section, we show the results related to the number of nodes in the full search tree for each model.

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.

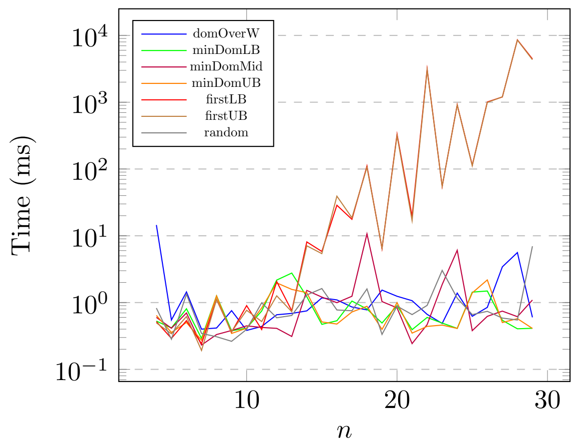
Figure 3: Number of nodes for $n \in [4; 12]$ and different models. Values expressed in K of nodes.

4.3 Search strategies - resolution time

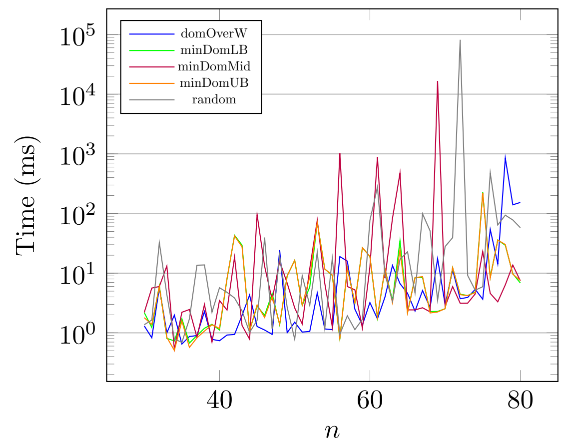
The results in the previous sections show that the Custom model and the Primal model are the ones that perform better. In this section, we show the results related to the time taken by these solvers to find the first solution, when using different search strategies.

4.3.1 Custom model

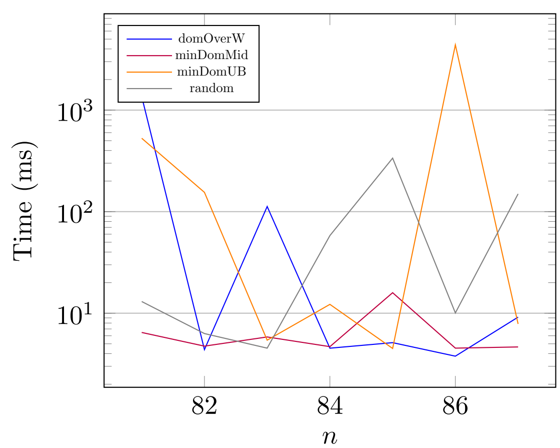
The custom model showed the smallest number of nodes and was the fastest to enumerate solutions.



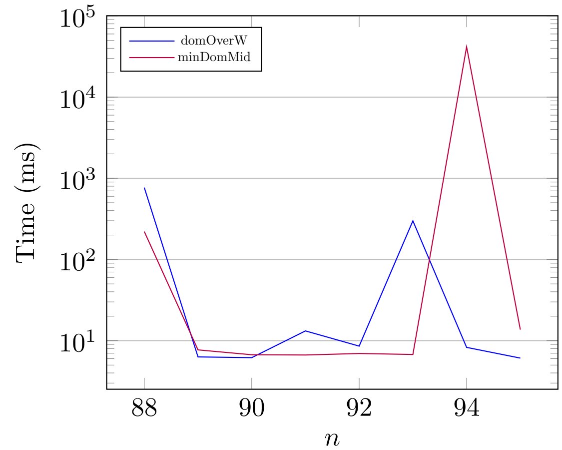
(a) Phase 1: executions of models with $n \in [4; 29]$.



(b) Phase 2: executions of models with $n \in [30; 80]$.



(c) Phase 3: executions of models with $n \in [81; 87]$.

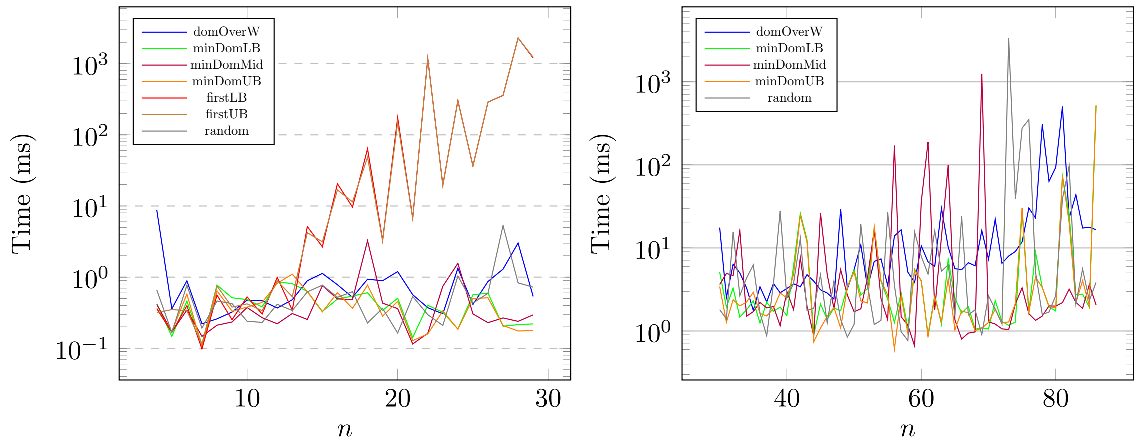


(d) Phase 4: executions of models with $n \in [88; 95]$.

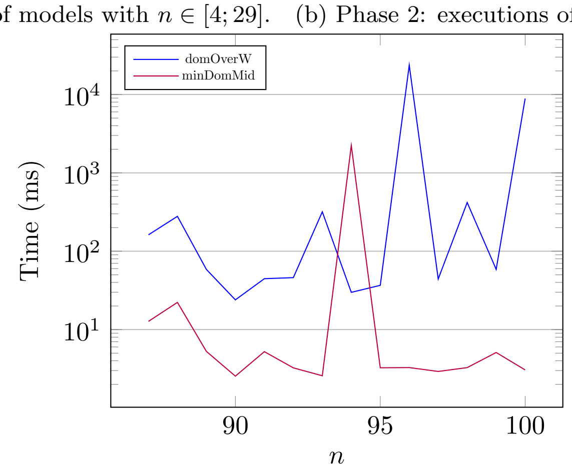
Figure 4: Charts for resolution times in milliseconds, to find the first solution. Execution has been splitted in 4 phases. After each phase, the slowest algorithms were removed. Times are expressed in a logarithmic scale.

4.3.2 Primal model

The primal model was overall the second best at enumerating solutions.



(a) Phase 1: executions of models with $n \in [4; 29]$. (b) Phase 2: executions of models with $n \in [30; 86]$.



(c) Phase 3: executions of models with $n \in [87; 100]$.

Figure 5: Charts for resolution times in milliseconds, to find the first solution. Execution has been splitted in 3 phases. After each phase, the slowest algorithms were removed. Times are expressed in a logarithmic scale.

5 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. Even though the boolean models use different types of variables, they showed similar performance. The Choco official documentation states that some filtering algorithms are optimized when using boolean variables, but this does not seem to be the case with the constraints used by this model. Note also that the constraints on the diagonals are expressed in the same way in both models; the one using boolean variables uses a conversion to int in order to use the *arithm* constraint provided by Choco. The results section shows that search trees have exactly the same number of nodes. It should be also noted that both models used the default search strategy provided by Choco, which is `domOverWDeg` for both int and boolean variables.

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.

The results in section 4.3 show that the choice of a good search strategy can be extremely important. When using the simple `firstLB` and `firstUB` search strategies, we can't even find a single solution in a reasonable time for some $n < 40$. On the other hand, when using the `minDomMid` strategy we can easily find a solution for n up to 500.

The default `domOverWDeg` strategy proved to be decent, but definitely inferior to `minDomMid`, which was not able to find a solution in a reasonable amount of time for several values near 100 and even greater. This makes sense, since in the Custom model we are only using one constraint and `domOverWDeg` is therefore not effective. Because of this, we wanted to try the `domOverWDeg` search strategy with the Primal model, which uses a lot of inequality constraints. However, the results presented in section 4.3.2 suggest that `minDomMid` still performs better.

References

[1] F. Boussemart et al. “Boosting systematic search by weighting constraints”. In: (2004).

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

[3] 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.

[4] G.L. Elliott R.M. Haralick. “Increasing tree search efficiency for constraint satisfaction problems”. In: (2004).

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

6 Appendix A

```
/
├── README_md
├── modelistation_iml
├── nodes_stats_csv
├── pom_xml
├── resolution_enum_stats_csv
├── search_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
    │           └── junit
    │               └── nQueenTester_java
    └── generated-sources
        └── annotations
```

7 Appendix B

```
Propagate method called
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07

Propagate method called
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07
00 -- -- -- -- -- -- --
00 01 02 03 04 05 06 07
00 01 02 03 04 05 06 07

RemoveOne() called by queen i=5
-- 01 02 03 04 -- 06 07
-- 01 02 03 -- 05 06 07
-- 01 02 -- 04 05 06 07
-- 01 -- 03 04 05 06 07
-- -- 02 03 04 05 06 07
00 -- -- -- -- -- -- --
-- -- 02 03 04 05 06 07
-- 01 -- 03 04 05 06 07

Propagate method called
-- 01 02 03 04 -- 06 07
-- 01 02 03 -- 05 06 07
-- 01 -- -- -- -- -- --
-- 01 -- 03 04 05 06 07
-- -- 02 03 04 05 06 07
00 -- -- -- -- -- -- --
-- -- 02 03 04 05 06 07
-- 01 -- 03 04 05 06 07

RemoveOne() called by queen i=2
-- -- 02 -- 04 -- 06 07
-- -- -- 03 -- 05 06 07
-- 01 -- -- -- -- -- --
-- -- -- 03 04 05 06 07
-- -- 02 -- 04 05 06 07
00 -- -- -- -- -- -- --
-- -- 02 03 04 -- 06 07
-- -- -- 03 04 05 -- 07

RemoveOne() called by queen i=5

Propagate method called
-- -- 02 -- 04 -- 06 07
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- 03 04 05 06 07
-- -- 02 -- 04 05 06 07
00 -- -- -- -- -- -- --
-- -- 02 03 04 -- 06 07
-- -- -- 03 04 05 -- 07

RemoveOne() called by queen i=1
-- -- -- -- -- -- 06 07
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- 06 07
-- -- 02 -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- 04 -- 06 07
-- -- -- -- 04 05 -- 07

RemoveTwo() called by queen i=0
-- -- -- -- -- -- 06 07
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- 06 07
-- -- 02 -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- 04 -- 06 07
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=1

RemoveOne() called by queen i=2

RemoveThree() called by queen i=3
-- -- -- -- -- -- 06 07
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- 06 07
-- -- 02 -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- 04 -- 06 07
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=5

RemoveThree() called by queen i=7
-- -- -- -- -- -- 06 07
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- 06 07
-- -- 02 -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- 04 -- 06 07
-- -- -- -- 04 05 -- 07

Propagate method called
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- 06 07
-- -- 02 -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- 04 -- 06 07
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=0
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- 07
-- -- -- -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- 04 -- -- 07
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=1

RemoveOne() called by queen i=2

RemoveTwo() called by queen i=3
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- 07
-- -- -- -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- 04 05 -- 07

RemoveThree() called by queen i=4
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- 07
-- -- -- -- 04 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=5

RemoveOne() called by queen i=6
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- 07
-- -- -- -- -- 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- 04 05 -- 07

RemoveTwo() called by queen i=4
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- 07
-- -- -- -- -- 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=5

RemoveOne() called by queen i=6
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- 07
-- -- -- -- -- 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- 04 05 -- 07

RemoveThree() called by queen i=7

Propagate method called
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- --
-- -- -- -- -- 05 -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- 04 05 -- 07

RemoveOne() called by queen i=0

RemoveOne() called by queen i=1

RemoveOne() called by queen i=2

RemoveOne() called by queen i=3
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- --
-- -- -- -- -- -- -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- -- 05 -- 07

RemoveOne() called by queen i=4
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- --
-- -- -- -- -- -- -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- -- 05 -- --

RemoveOne() called by queen i=5

RemoveOne() called by queen i=6

RemoveOne() called by queen i=7
-- -- -- -- -- -- 06 --
-- -- -- 03 -- -- -- --
-- 01 -- -- -- -- -- --
-- -- -- -- 04 -- -- --
-- -- -- -- -- -- -- 07
00 -- -- -- -- -- -- --
-- -- 02 -- -- -- -- --
-- -- -- -- -- 05 -- --
```