Decision Making with Constraint Programming

Exercise 3

Dessì Leonardo [0001141270]

Spini Riccardo [0001084256]

Corso di laurea magistrale in Informatica

Alma Mater Studiorum
Università di Bologna

October, 2023

1 | nQueens

	n			
	30	35	45	50
input order – min value	1 588 827	2 828 740	-	-
input order – random value	9	10	6	42
min domain size – min value	15	21	6	123
min domain size – random value	**1**	**0**	**1**	**10**
domWdeg – min value	15	21	6	123
domWdeg – random value	**1**	**0**	**1**	**10**

Table 1.1: nQueens results

Table 1.1 shows the time and number of failures of the nQueens problem using different heuristics. The heuristics are divided into:

- *input order*: we are using the method to specify how MiniZinc should explore the solution space. *input_order* indicates that variables should be selected in the order in which they were declared in the input section.
- *minimum domain size*: chooses the variable with the smallest domain.
- *domWdeg*: this heuristic estimates a ratio between the domain size and the weighted degree, which represents the number of times it failed in a constraint earlier in the search. Then select the variable X_i with lowest $dom(X_i)/w(X_i)$. The weighted degree for each variable starts from 1 and changes dynamically during search.

The 3 heuristics were used together with the following 2 search methods.

- *minimum value*: this method assigns the variable its smallest domain value;
- *random value*: this method assigns the variable a random value in its domain.

From the table, we can see that *input order - min value* has the highest number of failures and in the case where there were 45 and 50 queens, it failed to even finish execution in the required time (5 minutes). Choosing variables in the input order, using the minimum value search method, leads to the most unfavorable results in terms of failures. This is due to the rigid adherence to input-ordered variable selection, which restricts the choice of the most advantageous variables. Therefore, if an incorrect variable is chosen initially, the solver may traverse a vast sub-tree, resulting in prolonged execution times.
In some cases, the solver requires notably more time to reach a solution, possibly due to a particular variable ordering that increases the difficulty of the problem. However,

adopting a randomised approach improves the situation. By exploring various parts of the tree, the probability of finding a solution rises, decreasing the likelihood of getting stuck in a lengthy subtree.
The results highlight that the primary factor influencing failures is the heuristics used for variable selection. Specifically, both *min domain size* and *domWdeg* consistently result in the same failures, irrespective of the chosen search method. This pattern arises because each variable encompasses all constraints initially. Consequently, the weights associated with variables are initially equal. When the solver encounters a failure, it increases the weight assigned to the constraint. However, since all variables have identical constraints, each variable's weight simultaneously increments by 1. Consequently, the weights remain constant throughout the execution, and the only variable selection determinant becomes the size of the domain.

2 | Posters Placement

	19x19		**20x20**	
	Failures	**Time**	**Failures**	**Time**
input order - min value	1 362 457	14.763 s	-	-
input order - random value	-	-	-	-
smallest domain - min value	239 954	3.3 s	**1 873**	**0,453 s**
smallest domain - random value	2 929 153	26.152 s	5 797 312	49.551 s
domWdeg - min value	**236 024**	**3.011s**	**1 873**	**0.452 s**
domWdeg - random value	2 929 030	26.990 s	5 797 456	58.02 s

Table 2.1: Result of Poster problem without reordering data files

	19x19 reordered		**20x20 reordered**	
	Failures	**Time**	**Failures**	**Time**
input order – min value	**62**	**0.188 s**	**323**	**0.278 s**
input order – random value	-	-	-	-

Table 2.2: Result of Poster problem with reordered data files

The table 2.1 and 2.2 shows respectively the results from different search methods used for the Poster Placement problem, both with and without rearranging the rectangles.
From the data, it's clear that heuristics based on input order consistently result in long execution times and frequent failures. This is because this technique is inflexible and struggles to identify advantageous variables early in the search.

In contrast to the previous problem, using randomized search methods doesn't lead to better performance in the Poster Placement problem. The problem's strong constraints significantly affect the optimal placement of posters. As a result, using a deterministic method that follows constraints tends to perform better than a randomized approach. In this scenario, it's more effective to prioritize finding posters that violate constraints rather than starting with a random placement that might initially satisfy constraints but later disrupt the overall poster arrangement on the paper.
Using the *input order* heuristic on a decreasingly ordered data set, a significant reduction in failures is observed. This phenomenon occurs because the first posters are those with the most stringent constraints, so it is advantageous to try to place them initially. In this way, if they are to fail, they will do so soon, thus preventing subsequent constraints from causing further failures once more posters have been placed. Similarly, smaller posters, i.e. those with less tight constraints, are more easily placed later in the execution.
From the results, it can be seen that *smallest domain* and *domWdeg* show very similar, sometimes identical results, but in some circumstances *domWdeg* requires fewer failures to find a solution. This is due to the fact that, unlike the previous situation with nQueens, in which all variables were uniform with the same constraints, the rectangles now have different constraints. In particular, placing a larger rectangle is more complex, as the associated constraints are more stringent than for smaller rectangles.
This variation in the constraints and dimensions of rectangles contributes to a significant difference. Rectangles that are more difficult to position tend to generate more frequent failures due to the greater rigidity of their constraints, resulting in a higher weight. Consequently, they are preferred in the initial choices. In certain scenarios, this dynamic makes *domWdeg* slightly more effective than *smallest domain*, although both are outperformed by static heuristics with reordered data.
Finally, using the `vis_geost_2d` function that is present in our code, we created a graphical representation of a solution of problem with a 20x20 paper.

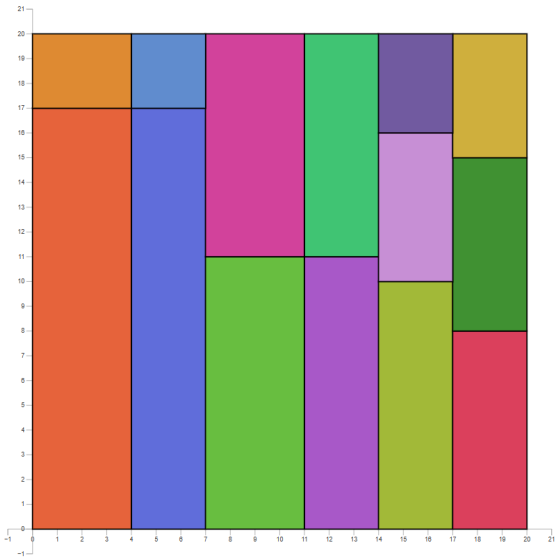

Figure 2.1: Paper 20x20

3 | QuasiGroup

	Default search		domWdeg with random value		domWdeg – random value with restarting	
	Time	**Fails**	**Time**	**Fails**	**Time**	**Fails**
qc30-03	-	-	110.96 s	1 061 184	**101 s**	**642 427**
qc30-05	71 s	657 955	**1.23 s**	**5 885**	49.33 s	303 205
qc30-08	**0.37 s**	**627**	1.16 s	6 403	2.21 s	11 990
qc30-12	25.91 s	259 082	6.27 s	53 200	**3.64 s**	**21 986**
qc30-19	43.53 s	381 330	-	-	**8.66 s**	**48 244**

Table 3.1: Results of QuasiGroup problem in Default search, domWdeg - random value and domWdeg - random value with restarting

The table 3.1 reports the number of failures and total time requested by different types of searches for the QuasiGroup problem.
There is a case (qc30-08) in which the *default search* method proves to be the most efficient in terms of time and failures. This occurrence may be attributed to several poor decisions at the start of the tree as seen in previous exercises. The solver completely governs the default search. This may cause erroneous decisions at the beginning and subsequent search difficulty. By chance, it happened that the initial random configuration of the instance is very good using the input order and therefore the default search took less time. By chance, the first assignments provided by the default search are good. Therefore,

by selecting the best choice at the beginning of the search, you decrease the chances of failure and speed up the process of discovering a resolution afterwards.
Below, we have illustrated this situation graphically.

	Default search	**Heuristic search**	**Heuristic search + restart**
Failures	627	6 403	11 990
Nodes	1 316	12 874	25 608
Propagations	99 972	905 898	1 574 148

Table 3.2: Statistically explaining the performance results of the different search methods for instance qc30-08.

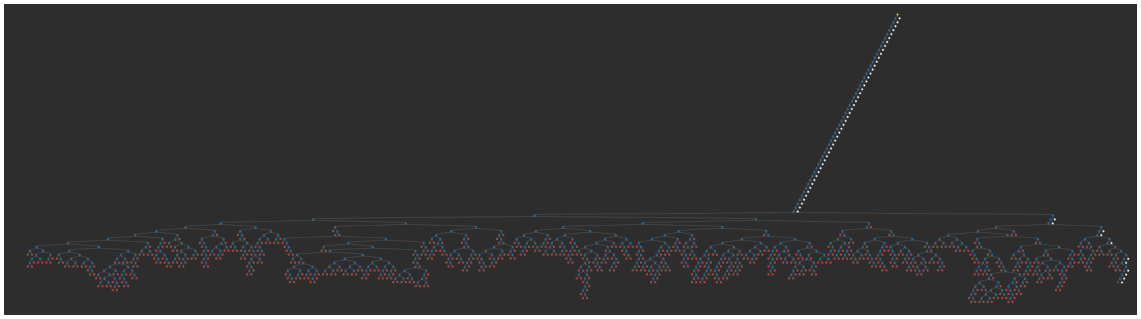

Figure 3.1: qc30-08 default search

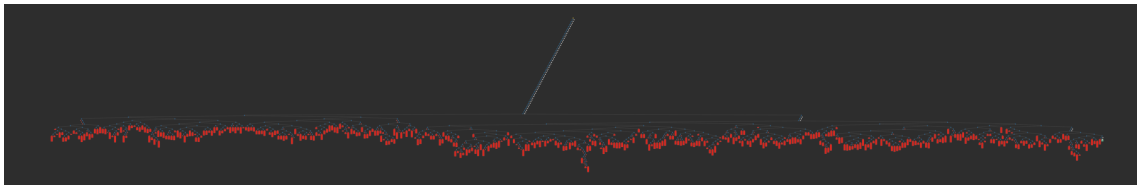

Figure 3.2: qc30-08 domWdeg without restart

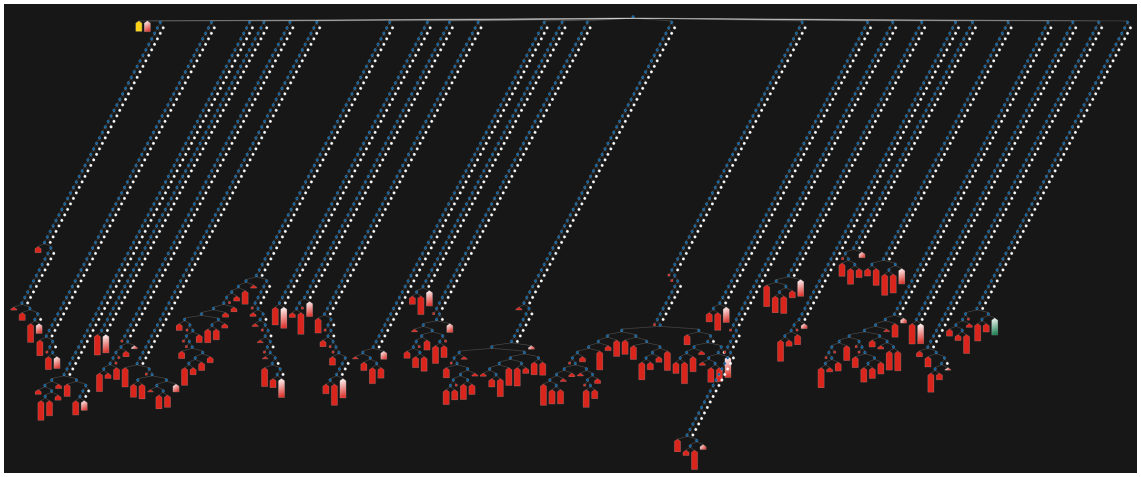

Figure 3.3: qc30-08 domWdeg with restart

Another thing that can be seen from the search tree images is the typical restart behaviour as in the following image

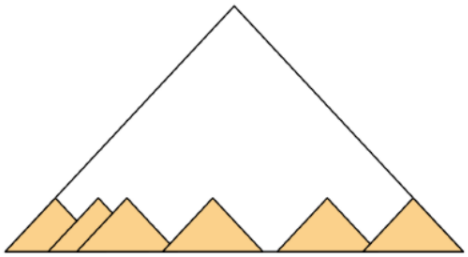

Figure 3.4: Typical search tree with randomization and restarts

In general, the *domWdeg* method with restart turns out to be less fallacious and time-consuming than the other approaches. This method generates additional information, represented by weights, during execution. When the search is restarted from the root node, the solver selects the next variables based on the previously generated information, improving the search and trying to minimize the initial errors in the decision tree.
However, in instances qc30-05 and qc30-08, this method does not seem to be the most effective. This could be attributed to the fact that, coincidentally, the order of the variables is already optimal and the solver, at first, does not spend the necessary time to explore the first branch. Instead, he begins the search again, losing the benefit of an already optimal ordering of the variables. This situation, however, occurs in only a few instances, highlighting the heavy tail behavior of this type of problem.
In addition, we experimented with variations in the `restart_luby` parameter and noticed that in these specific instances, different results can be obtained by changing the

parameter. Unfortunately, there is no defined criterion for choosing the optimal value of this parameter.
Finally, using the `vis_geost_2d` function that is present in our code and slightly modifying the Minizinc html template by adding more colors, we created a graphical representation of problem qc30-19 before and after resolution with Minizinc. Initially, all instances have 374 occurrences of 0, which corresponds to 41.55% white cells.

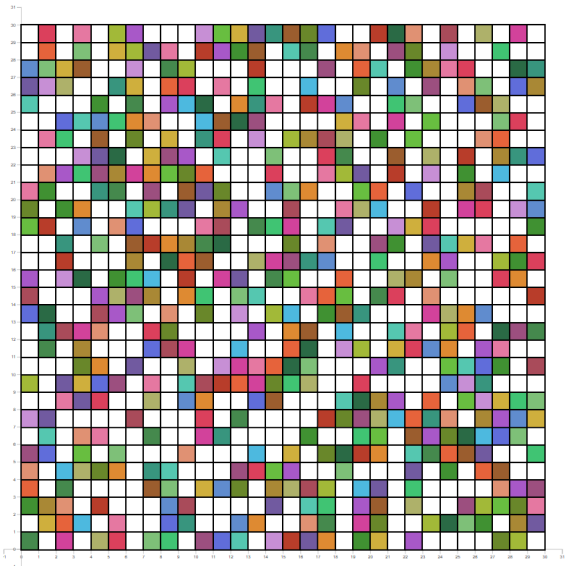

Figure 3.5: qc30-19 grid initial status

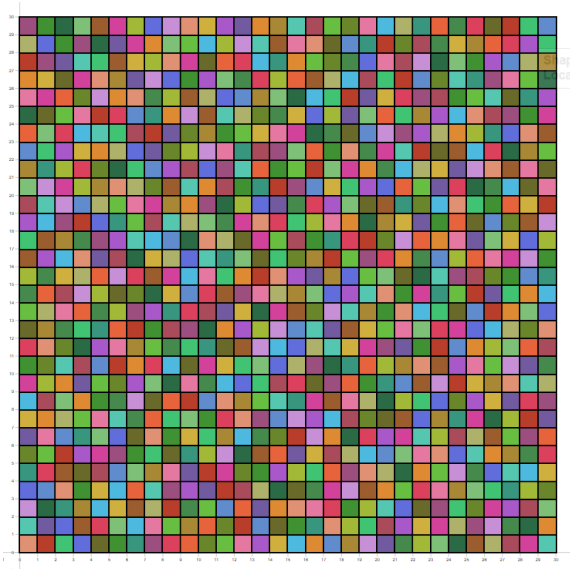

Figure 3.6: A qc30-19 solution

4 | Conclusions

4.1 | When are random decisions (not) useful? Why?

Assigning a random value to a variable may or may not be beneficial depending on the context. For example, in the nQueens problem, the use of randomness proves beneficial. Randomly placing a queen within the chessboard may be more advantageous than systematically placing it in the top left-hand corner. This is because the random placement of the first few queens increases the probability that they will control a greater number of squares than the least (top left) placement. This approach makes the placement of subsequent queens more complex, reducing the search space in case of failure.
In contrast, in the 'Poster' problem, the use of randomisation proves less useful. Randomly placing a poster in the grid increases the probability of failure in the first steps, especially if the poster is large. Finally, in the "quasiGroup" problem, the use of randomisation proves to be advantageous, especially when combined with the restart of search.

4.2 | Are dynamic heuristics always better than static heuristics? Why?

No, dynamic heuristics are not always better than static ones. In fact, we have noticed that when instance variables are reordered using some problem-specific heuristics and solving with static heuristics, better results are achieved. However, in cases where we do not have problem-specific heuristics, dynamic heuristics help us more.

4.3 | Is programming search and/or restarting always a good idea? Why?

There is no universal answer, but we have noted a few factors:

When it may **be useful** to restart the search:

- **Search stagnation**: If the algorithm seems to be stuck in a search region without making significant progress, restarting may be useful to allow the algorithm to explore new portions of the solution space.
- **Diversification of search**: Restarting can help diversify the search by introducing a new initialisation or exploration strategy, allowing the algorithm to explore alternative solutions.
- **Presence of many local minima**: In problems with multiple local minima, restarting may help to avoid prematurely converging to a suboptimal solution.

When it may **not be useful** to restart the search:

- **Good progression**: If the algorithm is making steady progress without getting stuck, it may not be necessary to restart the search.
- **Negative impact on execution time**: Restarting may come at a cost in terms of execution time, especially if the problem takes a long time to solve. In some cases, the benefit of the restart may not justify the additional time required.
- **Good initialisation**: If the algorithm is already well initialised with good heuristics, restarting may not lead to significant improvements.

Ultimately, the decision to restart the search depends on the complexity of the problem and the dynamics of the search.