CP Third Assignment

Students: Massimo Rondelli e Gianmarco Gabrielli

N-Queens problem

In the Table 1, we can see the results of the input order min and random value computation. It is visible that, in this case, the random value approach gives us a better solution. The variables in this case are choosen in order from the array q and we assign its smallest domain value. This cause that all the variables are associated the smallest values in the domain, which means that, at the beginning, the queens are grouped to the left side of the cheesboard. It's necessary to backtrack a lot in order to find a solution. That's wht we have a lot of failures. With the random approach, instead, we assign the variable a random value from its domain. In this case, we have more probabilty that the queen is already in a solution's position or, at least, close to it. Rather than the min value approach. This is also why we have less failures. We need to backtrack less.

Looking at the results of Table 1, we see that they are the same, even though we are usign two different approaches. **Domain size** and **domWdeg**. Why's that?

Using **dom**, we choose the next variable with minimum domain size. Choosing variables with the minimum domain size allow us to create the minimum search tree after propagation. Propagation's effect is more powerful. With **weighted degree heuristic**, instead, we combine min domain size and the weighted degree. In this model, we give more weight to the constraints that are more difficult to satisfy, and we calculate the degree of the variable's according. During the propagation of a constraint c, its weight $w(c)$ is incremented by 1 if the constraint fails. The weighted degree of a variable X_i is the following:

$$w(X_i) = \sum_{\text{c s.t. } X_i \in X(c)} w(c)$$

In the heuristic **domWdeg** we choose the variable X_i with the minimum domain size and the maximum weighted degree:

$$\frac{\mid D(X_i) \mid}{w(X_i)}$$

Now it's more clear that the two approaches are different, but why we get the same statistics? If we think about it, in the domWdeg we are always going to take the variable with the minimum domain size despite how many times the constraint failed. All the variables are associated to the three constraints in the model.

```
constraint alldifferent(q)::domain;
constraint alldifferent([q[i]+i | i in 1..n])::domain;
constraint alldifferent([q[i]-i | i in 1..n])::domain;
```

So, even if one constraint fails more than the other, all the variables are associated to the same $w(c)$, so we can imagine the denominator of the fraction as a constant, which means that the **domWdeg** is

$$\frac{\mid D(X_i) \mid}{w(X_i)} = \mid D(X_i) \mid$$

which is equal to the domain size model. That's why we get the same results even if the heuristic is different.

		Fails			
		30	35	45	50
input order	**min value**	1.588.827	2.828.740	-	-
	random value	9	10	6	42
min domain size	**min value**	15	21	6	123
	random value	1	0	1	10
domWdeg	**min value**	15	21	6	123
	random value	1	0	1	10

Table 1: N-Queens Problem statistics

Poster Placement problem
Looking at the results obtained, a few observations can be made. It is not clear which heuristic is the best one at first look of this problem. It depends. The best results are those ones in bold. With both the "min domain size" and "domWdeg" approaches, we go to choose as variables those that have smaller domains. They are assigned their own smallest value in their domain. Using this approach, we assign the variables with the smallest domain, that is, the largest pieces, their smallest values, so, starting with the largest pieces, we begin to place them all to the left of the chessboard. By doing so, by placing the larger pieces first and all to the left, we are more likely to place future pieces. Why is random assignment not good in this problem? By assigning values to variables randomly, that is, placing pieces randomly on the board, there would be a risk of placing a large piece in the center of the board itself, then making it impossible to place other large pieces. Instead, starting the insertion of pieces from the left side of the chessboard leaves more free space for future pieces. In fact, it is possible to see the statistics. Random value assignment takes about 20 times longer than min value assignment. min value and domWdeg behave about the same way. That is why the results are very similar.

		19x19		20x20	
		Fails	Time	Fails	Time
input order	**min value**	1.315.598	11s 35ms	26.063.823	3m 12s
	random value	-	-	-	-
min domain size	**min value**	**239.954**	**1s 796ms**	**1.873**	**244ms**
	random value	2.929.153	19s 172ms	5.797.312	35s 987ms
domWdeg	**min value**	**236.024**	**1s 820ms**	**1.873**	**244ms**
	random value	2.929.030	19s 30ms	5.797.456	35s 957ms

Table 2: Poster Placement using 19x19.dzn and 20x20.dzn (unsorted)

In Table 3, we can see how a static heuristic and with the array of ordered chunks, markedly improve the statistics for this problem. In this case the variables are chosen according to their order within the array. However, since they are sorted in descending order, the largest pieces are assigned first and inserted starting from the left of the chessboard. Again it can be seen that random assignment of values leads to no solution.

In conclusion, we can say that the choice of a static heuristic rather than a dynamic one depends on the type of problem we have and what the data at our disposal look like. Dynamic heuristics, in this problem, tries to mimic the behavior of static with the ordered array. However, it fails to achieve its excellent results.

n	Input Order - Min Value		Input Order - Random Value	
	Fails	**Time**	**Fails**	**Time**
19x19	**62**	**417ms**	52.181.151	-
20x20	**323**	**315ms**	47.654.745	-

Table 3: Sorted array: Input Order - Min and Random value

Quasigroup Placement problem

In the quasigroup problem, we are going to use three different solution methods. Default search, domWdeg and domWdeg with Luby restart. Looking at the results we obtained, we can say that while domWdeg with restarts appears to function well overall, the optimal search method varies depending on the specific instance. We note that in instance **qc30-05**, domWdeg random without restarts, performs significantly better than both default and domWdeg with restarts. Restarts may not be effective in this situation because they occur too soon, which prevents them from thoroughly examining the relevant part of the search tree at an early stage. Instance **qc30-08** has relatively low search times for all strategies, suggesting that it may be simple instance to solve. In this case, the default search works best. It is possible that the default method outperforms the more detailed domWdeg methods in simple instances. This may be because domWdeg follow the fail-first principle, trying where failure is most likely to occur. To reduce the size of the search tree, the solver will therefore first fail a number of branches.

		default	**domWdeg - random**	**domWdeg - random + Luby**
qc30-03	Fails	3.648.841	1.495.755	**642.427**
	Time	-	1m 55s	**1m 28s**
qc30-05	Fails	1.167.530	**7.163**	303.205
	Time	1m 36s	**714ms**	42s 542ms
qc30-08	Fails	**1250**	4.398	11.990
	Time	**191ms**	454ms	1s 927ms
qc30-12	Fails	230.082	47.036	**21.986**
	Time	16s 549ms	4s 92ms	**3s 593ms**
qc30-19	Fails	773.526	3.185.314	**48.244**
	Time	1m 1s	-	**7s 427ms**

Table 4: Quasigroup problem statistics

All things considered, we may conclude that using restart techniques is a good idea when the problem utilized Depth First Search because you have a better chanche of discovering the solution more quickly that if you carry out the traditional search.