Team:
- Andrea Corradetti - andrea.corradetti2@studio.unibo.it
- Francesco Corigliano - francesco.coriglian2@studio.unibo.it

N-queens

n	AllDifferentGC		Decomposition	
	Fails	Time	Fails	Time
28	78,847	0,9 s	417,017	2,8 s
29	31,294	0,3 s	212,257	1,5 s
30	1,588,827	16,5 s	7,472,978	51,2 s

Comment briefly on how the solver performance changes from one (decomposed or non-global) model to the global model, along with a justification.

The alldifferent constraint can prune wrong assignments much earlier by comparing multiple variables at once.
When checking pair-wise, each pair might have legal assignments available but this does not imply that a legal assignment is present for multiple variables at once.
e.g. With 4 variables and domain {0, 1}, the first propagation won't detect any inconsistency because each pair does have 2 values available.
Alldifferent will detect that not enough values are available before assigning any variable.

Poster (table updated)

Instance	Naive model		Global model		Naive + Implied	
	Fails	Time	Fails	Time	Fails	Time
19x19	1,903,966	10,1 s	300649	1,44 s	461	0,82 s
20x20	2,756,416	17,5 s	2030	0,47 s	2	0,70 s

Comment briefly on how the solver performance changes from one (decomposed or non-global) model to the global model, along with a justification.

We don't know the internal workings of the global constraint but we suppose that the reason for the improved performance is analogous to the previous problem.
The naive constraint can only check for pairs. We expect the global constraint to check for multiple rectangles at the same time.
e.g. Say we have to place down 2 more posters but there's only enough space for one more.

-You talk about only the propagation strength of the global constraints. Looking at your results, we see time advantage too. Can you comment on this also?

-In general, global constraints tend to use ad-hoc implementations that exploit the specific semantics of the constraint. This might afford noticeable efficiency gains.
We can also expect that subsequent applications of the constraint will be more efficient due to incremental computation.
The bipartite graph algorithm for allDifferent is a clear example.
It's reasonable to think that diffn and global_cardinality also take advantage of clever implementations that achieve similar propagation with shorter execution times.

IMPLIED CONSTRAINT (indexes updated)
An additional solution with 2 implied constraints offers good performance with the naive no-overlap constraint.

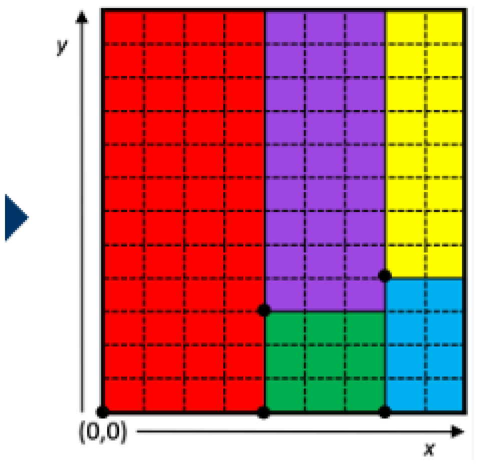
```
% these constraint significantly reduce the number of failures
% posters don´t exceed vertical space, implied
constraint
  forall (i in 1..w) (
    sum(j in 1..n where Xs[j] <= i /\ i < Xs[j] + ws[j]) (hs[j]) <= h
  );
% posters don't exceed horizontal space, implied
constraint
 forall (i in 1..h) (
    sum(j in 1..n where Ys[j] <= i /\ i < Ys[j] + hs[j]) (ws[j]) <= w
 );
```

-nice observation about the implied constraints, though I believe you should use <= for i< Xs[j] + ws[j] and j< Ys[j] + hs[j].

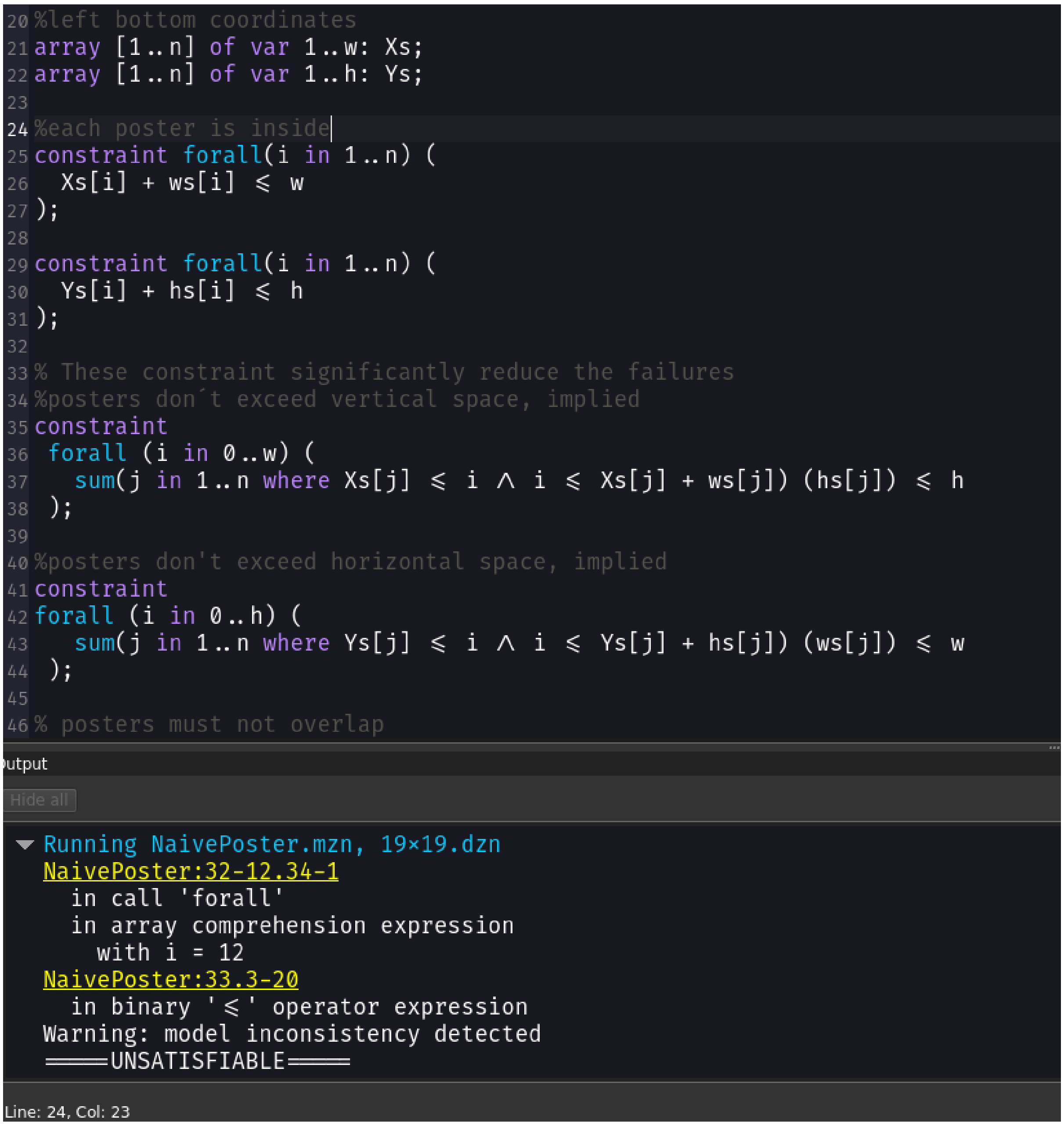
I think you should use <= but iterate i over 1..w and 1..h (it does not make sense to consider the case where i = 0).

We think that the coordinates can't be 1-indexed. (the array itself is indexed 1..n, but the domain should start from 0)
If the total width of the grid is 19 and we have a 19-wide poster, starting from 1 would yield 20. It makes sense that if a poster is 2 squares wide, and starts from the left margin, it should end at 2.

This picture in the slides is also starting from 0,0



The following is a screenshot of the result with coordinates 1..w and 1..h.
Minizinc warns us of an inconsistency.



```
20 %left bottom coordinates
21 array [1..n] of var 1..w: Xs;
22 array [1..n] of var 1..h: Ys;
23
24 %each poster is inside
25 constraint forall(i in 1..n) (
26   Xs[i] + ws[i] ≤ w
27 );
28
29 constraint forall(i in 1..n) (
30   Ys[i] + hs[i] ≤ h
31 );
32
33 % These constraint significantly reduce the failures
34 %posters don´t exceed vertical space, implied
35 constraint
36  forall (i in 0..w) (
37    sum(j in 1..n where Xs[j] ≤ i ∧ i ≤ Xs[j] + ws[j]) (hs[j]) ≤ h
38  );
39
40 %posters don't exceed horizontal space, implied
41 constraint
42 forall (i in 0..h) (
43    sum(j in 1..n where Ys[j] ≤ i ∧ i ≤ Ys[j] + hs[j]) (ws[j]) ≤ w
44  );
45
46 % posters must not overlap
```

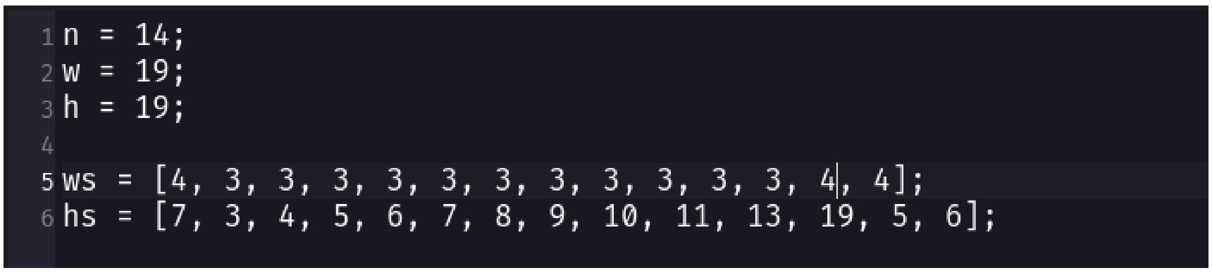
utput

Hide all

```
Running NaivePoster.mzn, 19×19.dzn
NaivePoster:32-12.34-1
  in call 'forall'
  in array comprehension expression
    with i = 12
NaivePoster:33.3-20
  in binary '≤' operator expression
Warning: model inconsistency detected
=====UNSATISFIABLE=====
```

Line: 24, Col: 23

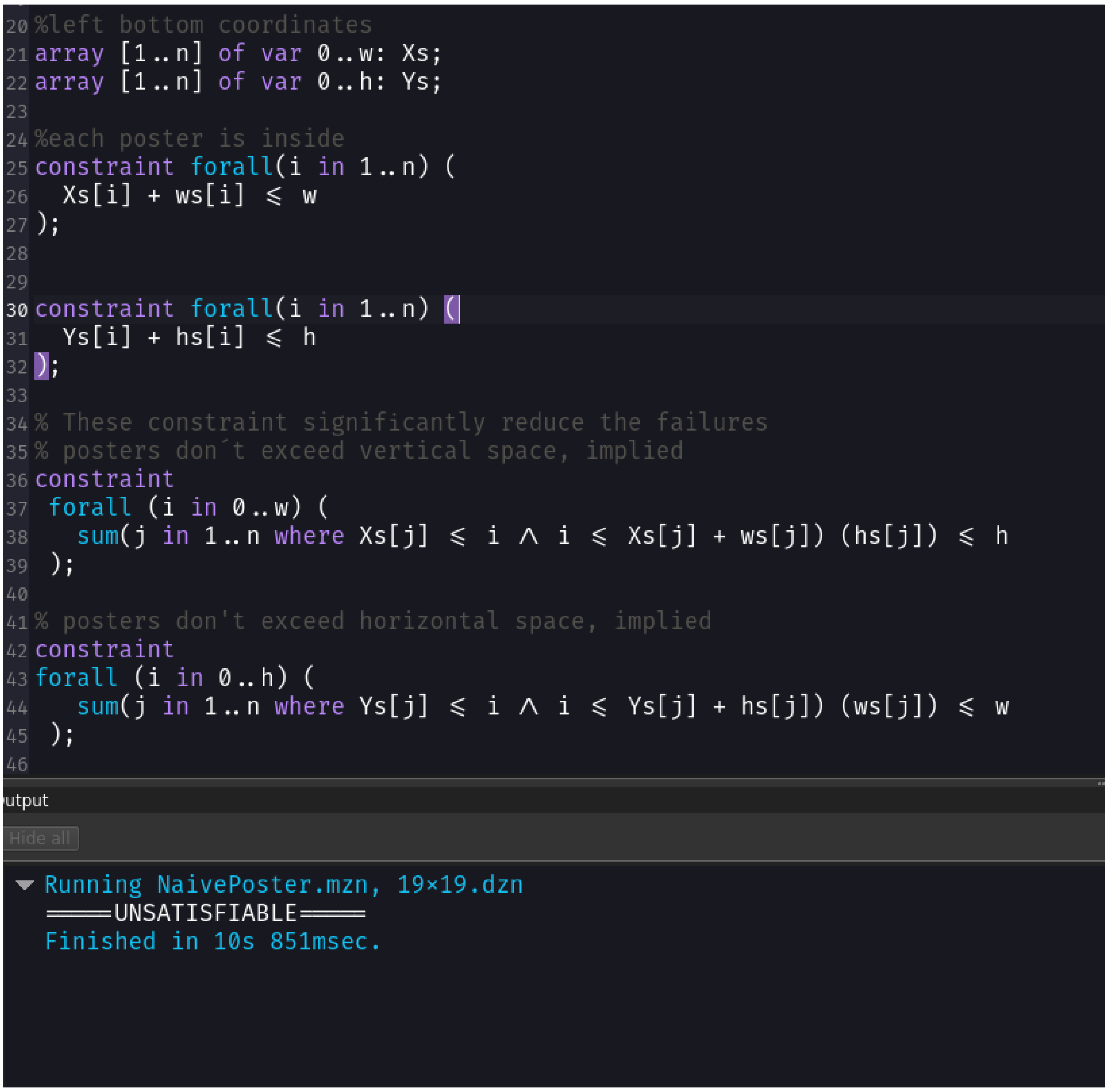
At position 12 of the input array we have a poster with a height of 19 which causes the inconsistency at lines 30-31



```
n = 14;
w = 19;
h = 19;

ws = [4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4];
hs = [7, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 19, 5, 6];
```

A coordinate system starting from 0 solves the problem. We could of course subtract one and get the same result, but to us it makes more sense to say that a 19 height poster starting from the bottom will reach height 19 (0 + 19).

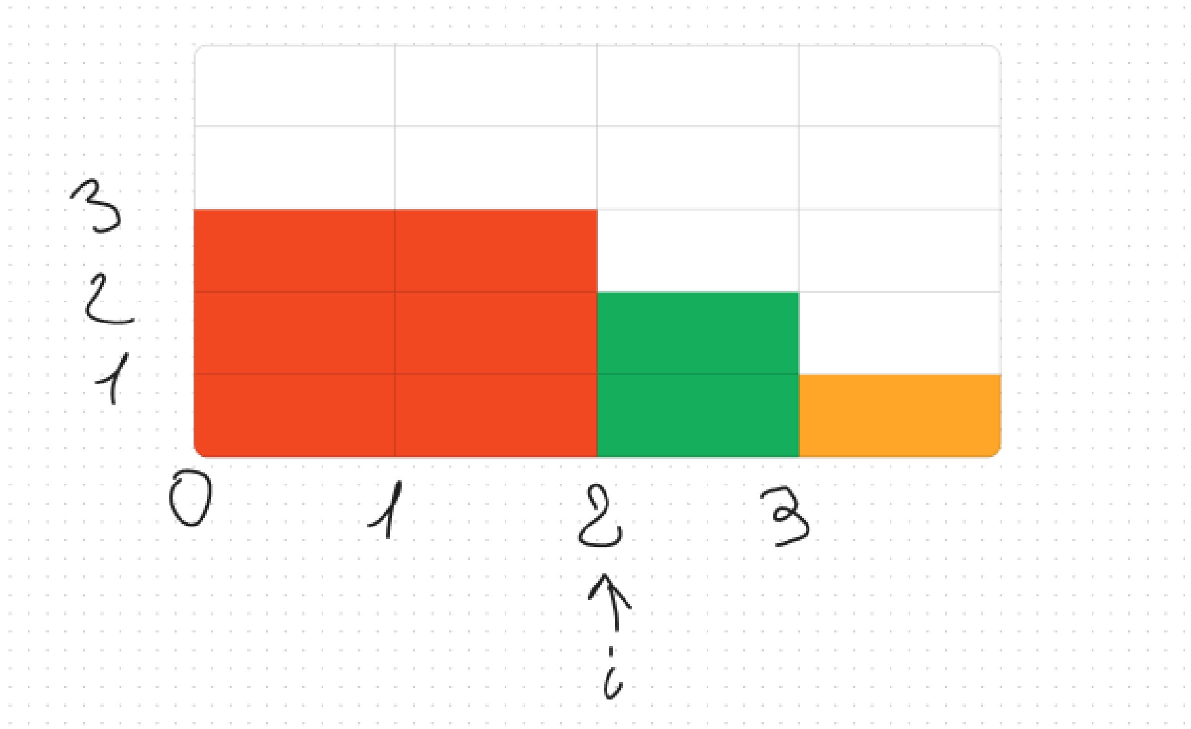
If we fix the above issue we get to the second point. Here you can see we are using 0..w and 0..h
If we try to use <= on both sides the problem is unsatisfiable. A Possible explanation is after the screenshot.



```
%left bottom coordinates
array [1..n] of var 0..w: Xs;
array [1..n] of var 0..h: Ys;

%each poster is inside
constraint forall(i in 1..n) (
  Xs[i] + ws[i] ⩽ w
);


constraint forall(i in 1..n) (
  Ys[i] + hs[i] ⩽ h
);

% These constraint significantly reduce the failures
% posters don´t exceed vertical space, implied
constraint
 forall (i in 0..w) (
   sum(j in 1..n where Xs[j] ⩽ i ∧ i ⩽ Xs[j] + ws[j]) (hs[j]) ⩽ h
 );

% posters don't exceed horizontal space, implied
constraint
forall (i in 0..h) (
   sum(j in 1..n where Ys[j] ⩽ i ∧ i ⩽ Ys[j] + hs[j]) (ws[j]) ⩽ w
 );

```

```
Running NaivePoster.mzn, 19×19.dzn
 =====UNSATISFIABLE=====
 Finished in 10s 851msec.
```

We think this makes sense because we would be counting some posters twice. Let me explain with another picture.



Say we have i = 2.
We are between red and green in this example.
If we are using Xs[j] <= i and i <= Xs[j] + ws [j], the sum for i = 2 will be 5.
Both red and green satisfy the condition.
Indeed, i = 2 ,Xs[red] = 0, Xs[red] + ws[red] = 2.
So red is included in the sum. because 0 <= i <= 2

But also, i= 2, Xs[green] = 2, Xs[green] + ws[green] = 3
So green is included in the sum. because 2 <= i <= 3.

In this case The sum for i = 2 would be 5. But we can see from the picture that that's not the case. No column reaches height 5. So one side has to be strictly less to avoid being included.

We'd like to point out that we are indexing the points of the grid between the squares and not the squares themselves.
This solution has the nice consequence of allowing for 0-wide (or 0 height) posters.

Please let us know if you still don't agree. We'd really like to understand if we got something wrong.

Here are the new benchmarks that **compare the global model with the naive + implied for one solution**:

Benchmark - Global vs Naive + implied, 1 solution

Instance	Naive model		Global model		Naive + Implied	
	Fails	Time	Fails	Time	Fails	Time
19x19	1,903,966	10,1 s	300649	1,44 s	461	0,82 s
20x20	2,756,416	17,5 s	2030	0,47 s	2	0,70 s

Eventually the global model is still better than the naive + implied model? You didn't comment on that.

The 20x20 seems to be easier to solve. It's plausible that the default solving strategy is especially convenient for those particular starting parameters.
The global constraint is much faster in the 20x20 grid and still marginally faster in the 19x19 grid.
Strangely, the global constraint fails much more in the 19x19 grid but is still faster.
This might point to a more efficient algorithm, as suggested during lectures.
Also, and this is an educated guess, the global algorithm might be "failing faster": it might be trying more routes but also figuring out early that the route is not viable.
Maybe the implied constraint is failing deeper in the tree.

I did not understand what you are referring to when you say "benchmark with 3000 solutions" and "benchmark with 5000 solutions". I had given you two instances to work on. Can you run your experiments of naive+implied on the same instances? You can extend the first table under the Poster title with an additional model.

benchmark with 3000 solutions				
	global constraint 19x19	naive + implied 19x19	global constraint 20x20	naive + implied 20 x 20
failures	352434	91925	10449	47780
total solvetime	3.6548	4.42633	0.450067	2.81521

benchmark with 5000 solutions				
	global constraint 19x19	naive + implied 19x19	global constraint 20x20	naive + implied 20 x 20
failures	1008366	233651	16740	98082
total solvetime	10.6618	11.1544	0.729471	5.50406

As explained in our emails, those tables refer to searching for 3000 and 5000 solutions for the satisfaction problem with the given parameters. We have updated the top table as requested and we are replicating at the top of this page for conveniency

Sequence

n	Base		Base+Implied		Global		Global+Implied	
	Fails	Time	Fails	Time	Fails	Time	Fails	Time
500	617	12,2 s	495	7,8 s	989	0,22 s	493	0,11 s
1000	1247	1m 24 s	995	33 s	1989	0,8 s	993	0,32 s

Going from Base → Global, and going from Base + Implied → Global + Implied: what is the main advantage of using a global constraint? Why?

As expected, the solve time is much shorter in both cases.
We suppose once again that the global constraint might be able to check for multiple variables simultaneously instead of having to check each constraint separately.
The constraints in the quantification might all be satisfiable if taken singularly, but they might be unsatisfiable if considered jointly.

-Again you are talking mainly about the propagation strength. Sometimes the global constraints are useful mainly due their efficiency in propagation rather than in the amount of propagation as seen in this example (no gains in search space). Please justify this (like I said above).
You talk about only the propagation strength of the global constraints. Looking at your results, we see time advantage too. Can you comment on this also?

-In general, global constraints tend to use ad-hoc implementations that exploit the specific semantics of the constraint. This might afford noticeable efficiency gains.
We can also expect that subsequent applications of the constraint will be more efficient due to incremental computation.
The bipartite graph algorithm for allDifferent is a clear example.
It's reasonable to think that diffn and global_cardinality also take advantage of clever implementations that achieve similar propagation with shorter execution times.

-Why do you think the base models (even the base+implied) are too costly timewise?

~~It's O(n^2) constraint checks with 4 comparisons for each propagation when using the naive constraint. Not sure about gcc but we assume it's going to be fewer.~~

Can you explain better how you calculated O(n^2)? I did not understand what you mean by 4 comparisons either...
HINT: if you look at the solution statistics, you will see that there are many more variables in the base model. First understand where they come from and quantify them in terms of n, then quantify how many constraints are posted on those variables, again in terms of n. Hopefully the answer will tell you why meta-constraint based propagation is costly.

In the initial discussion, I was thinking of the poster model and of n*n/2 pairs.
It of course does not directly apply to the sequence model. Sorry for the confusion.

After considering the correct problem it turns out we are checking for O(n^2) constraints. Explanation following the data.

Global+implied model statistics:

```
%%%mzn-stat: flatIntConstraints=3
%%%mzn-stat: flatIntVars=500
%%%mzn-stat: flatTime=0.0316181
%%%mzn-stat: method=satisfy
%%%mzn-stat: paths=0
%%%mzn-stat-end

%%%mzn-stat: failures=493
%%%mzn-stat: initTime=0.004866
%%%mzn-stat: nodes=991
%%%mzn-stat: peakDepth=251
%%%mzn-stat: propagations=6891
%%%mzn-stat: propagators=4
%%%mzn-stat: restarts=0
%%%mzn-stat: solutions=1
%%%mzn-stat: solveTime=0.061242
%%%mzn-stat: variables=500
%%%mzn-stat-end
%%%mzn-stat: nSolutions=1
%%%mzn-stat-end
Finished in 119msec.
```

Base+implied model statistics:

```
%%%mzn-stat: evaluatedReifiedConstraints=250000
%%%mzn-stat: flatBoolVars=250000
%%%mzn-stat: flatIntConstraints=500502
%%%mzn-stat: flatIntVars=250500
%%%mzn-stat: flatTime=2.59371
%%%mzn-stat: method=satisfy
%%%mzn-stat: paths=0
%%%mzn-stat-end

%%%mzn-stat: failures=495
%%%mzn-stat: initTime=1.97569
%%%mzn-stat: nodes=994
%%%mzn-stat: peakDepth=252
%%%mzn-stat: propagations=4187607
%%%mzn-stat: propagators=8876
%%%mzn-stat: restarts=0
%%%mzn-stat: solutions=1
%%%mzn-stat: solveTime=0.179191
%%%mzn-stat: variables=500500
```

```
%%%mzn-stat-end
%%%mzn-stat: nSolutions=1
%%%mzn-stat-end
Finished in 7s 903msec.
```

The global model makes use of 500 flatInt vars, which we interpret as one per digit. So O(n) vars. It also only has 3 constraints on those variables O(1).

The base model on the other hand is using 250,000 boolean variables; we suppose this is a boolean variable for each combination of index and value (O(n^2)).
Of the 500,502 flatIntConstraint, 2 are the implied constraints, and the remaining 500,500 are generated by the first meta-constraint. This is O(n^2) constraints.
That tells us why it is costlier than the global model.
We are not sure where the 250,500 flatIntVars are coming from. We suppose minizinc might define one for each bool variable when casting and one for each sum.

Is there an implied constraint that now becomes redundant in the Global + Implied model? Why?

The first implied constraint seems to be redundant. Looking at the code, we used the global cardinality constraint in this way: `constraint global_cardinality(x, array1d(0..n-1) , array1d(x));`
To have a self-describing sequence of n digits, the sum of the digits themselves must be n, because the digits specify the number of occurrences and the total occurrences must be n. This can't be easily checked from multiple count constraints; in this case, specifying a condition on the sum can be useful to reason on all digits at the same time.
When using the global constraint as shown above, we pose that digit i in range 0..n-1 must appear x[i+1] times.
Assuming the global constraints can keep track of all counts at the same time, we don't need to check for the sum. The property is going to be implicitly maintained simply by tracking all counts simultaneously.