Costas Array as a CSP

Ruben Teimas

Universidade de Évora, Évora, Portugal
February 6, 2021
`m47753@alunos.uevora.pt`

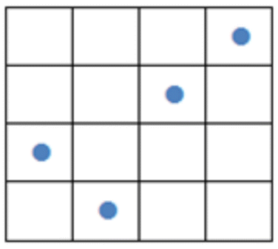
Abstract: Costas Array is a mathematical problem introduced in 1965 by John Costas. It has a some real-world applications, like configuring sonar frequencies. In this experiment the problem was modeled as a Constraint Satisfaction Problem (*CSP*) and later translated to code for benchmarking and checking the model's validity. The results confirm Costas Array as a good candidate for *CSP* but also its complexity as the size, n, grows.

Keywords: Costas Array · Constraints · CSP · OR-Tools

1 Introduction and Motivation

Costas Array problem can be described as a set of n points, each at the center of a square in an $n \times n$ square tiling such that each column contains only one point, and all of the *n(n-1)/2* displacement vectors between each pair of dots are distinct. [1]

Fig. 1: Valid *4*x*4* Costas Array



This problem is not only very interesting from a theoretical perspective but it also has real world applications in setting radars and sonars frequencies by avoiding noise.

Looking at the well specified conditions that make a valid Costas Array, it is easy to picture it formulated as a Constraint Satisfaction Problem (*CSP*). The base formulation of the problem is similar to the *N-Queens* [2] with some different constraints.

*CSP*s can be translated to real examples by using *SAT* solvers. These type of solvers are quite fast as they are primarily used in optimization problems. Some of the most notable are *Choco Solver*, *Gecode*, *OR-Tools*

2 State of the Art

The concept was introduced in 1965, by *John P. Costas*, and since there began gaining traction. [1]

Nonetheless, Costas Array is a very complex problem that grows exponentially with n. We can think of it as an *n*x*n* matrix permutation and (*n-1*)! distance and slope calculation, therefore the search space and processing power needed will increase as n increases.

In order to make the problem more approachable several construction methods were proposed such as *Welch* and *Lempel–Golomb* methods. Even though they allowed to discover a great number of Costas Arrays they could not compute any valid Costas Array for $n \geq 32$.

Since then, several papers have been written proposing new construction methods and experiments in order to speed-up this problem's resolution. One of them is "Constraint-Based Local Search for the Costas Array Problem" by Diaz, Richoux, Codognet, Caniou and Abreu. [3]
In it, the authors explore a Costas Array solution based on *Adaptative Search*, first using a sequential implementation and then by making it parallel using *OpenMPI*.

3 Problem Representation

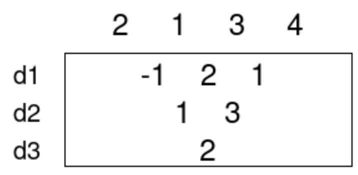
Even though the Costas Array problem is an nxn matrix, just like *N-Queens*, it can be represented as an one dimensional array.

Each index of the array, i, corresponds to a column of the matrix and the value of the index, V_i, corresponds to a row.

Following this representation of the problem, the ***Fig.1*** is given by [2, 1, 3, 4].

To calculate the distance and the slope of the displacement vectors a difference triangle is used. The triangle can be represented as an array that treats each row as an array.

Fig. 2: Difference Triangle for [2,1,3,4] Costas Array



For instance, the difference triangle from ***Fig.2*** can be represented as [[-1,2,1],[1,3],[2]].

Any Costas Array's difference triangle will be represented by an array holding (*n-1*) sub-arrays each with (*n-row_index*) elements. The array has a total of (*n-1*)! variables.

4 Problem Formulation

4.1 Naive

The Costas Array problem can be expressed as a *CSP*. The problem, *P*, is given by the equation:

$$P = (V, D, C)$$

Which means the problem, *P*, is given by the set of variables *V* with domain *D* and respective constraints *C*.

Given the size, *n*, of the array the set o variables, *V*, is:

- r_i: variable that holds the value of one row in column *i*.
- array[n]: collection of n r_i variables, $[r_1, r_2, \ldots, r_n]$.
- $d_j i_k$: variable that holds a value of distance and orientation of array[i] at distance *d*.
- difference_triangle[n-1]: collection of (*n-1*) sub-arrays and a total of (*n-1*)! $d_j i_k$ variables, [[$d_1 i_1$, $d_1 i_2$,. . . ,$d_1 i_{n-1}$],. . . , [$d_{n-1} i_1$]].

The domain,*D*, of the set of variables, *V*, is:

- $r_i \in$ [1, *n*].
- $d_j i_k \in$ [(*1-n*), (*n-1*)].

r_i domain is self-explanatory: since the value represents a row, its domain will be between 1 and the number of rows existent, *n*.

The $d_j i_k$ variable holds a difference between r_i variables, therefore, the lower bound of its domain will be the smallest value of said variables, 1, minus the highest value, n. The upper bound will be the opposite.

The variables *array[n]* and *difference_triangle[n-1]* were not mentioned in the domains because they are simply collections of the variables above.

The set of constraints, *C*, applied to the variables, *V* can are the following:

- All the r_i values must be different. For a Costas Array to be valid, the points cannot be on the same row and column. We assure that there are no points in the same column just by using the above representation. To guarantee that there are no points on the same row an *alldifferent* constraint must be applied to the variable *arrays[n]*.
- The difference triangle has another constraint to assure that its variables correspond to the difference of values at the distance *d*. The constraint can be expressed as: the d^{th} row of the triangle must contain the differences $V_{i+d} - V_d$ for all *i=1*,. . . , n-1.
- In order to have a valid Costas Array we need to apply a last constraint to ensure that all the elements, *e*, in each row, *r*, of the triangle are different. This is possible by applying an *alldifferent*(r_i) for each *r*=1,. . . , *n-1*.

4.2 Improved

For the improved formulation of the problem I maintained the overall naive formulation while trying to improve some aspects of the problem, such as constraints and search strategies.

The first improvement that came to mind was to add redundant constraints in order to increase propagation.

Unfortunately the only redundant constraint class I could think of was a constraint to ensure that all values of the *array[n]* are different. This constraint states that no variable on the *difference_triangle* can have a value of zero. Since the variable's value are a subtraction of 2 positions of the *array*, saying that the value can't be 0, means that all variables in *array* have different values.

There was not chosen any particular search strategies.

5 Tools

The model's translation to code was made using *Google OR-Tools* [4] for Python.

This tool is an open-source software suite, developed by Google, in order to solve and optimize problems such as routing, linear programming and constraint programming. It was developed in C++ but, fortunately, it provides wrappers for other programming languages like Python, Java and C#.

Some of the best solvers are open-source as well, like *Choco* and *Gecode*, but the real reason that led me to chose this tool is how recent, and good [5] it is.

The only drawback I found is the fact that the *Search* in the *Solver* class is not very configurable for the *CP-SAT* module (which is recommended over the previous *CP Sovler*).

6 Results

To test the validity of the model we proceeded to benchmarking. The tests were made on a machine with a *Intel i7-7700HQ* with 8 cores of 3.800GHz, 16GBs of *DDR4* RAM and a *NVIDIA GeForce GTX 1050* with 4GBs of VRAM.

The first benchmark made was when I tried to find all solution for each n on both versions. This process was repeated until the time to find all solutions was no longer acceptable.

Size	**Time(s)**	**Solutions**
1	0,07	1
2	0,04	2
3	0,07	4
4	0,07	12
5	0,07	40
6	0,09	116
7	0,17	200
8	0,44	444
9	1,64	760
10	10,76	2160
11	71,27	4368
12	570,31	7852

Naive

Size	**Time(s)**	**Solutions**
1	0,05	1
2	0,04	2
3	0,04	4
4	0,05	12
5	0,07	40
6	0,08	116
7	0,16	200
8	0,40	444
9	1,40	760
10	7,52	2160
11	42,07	4368
12	367,77	7852

Improved

Table 1: Time to find all the solutions, s, for each n and respective s

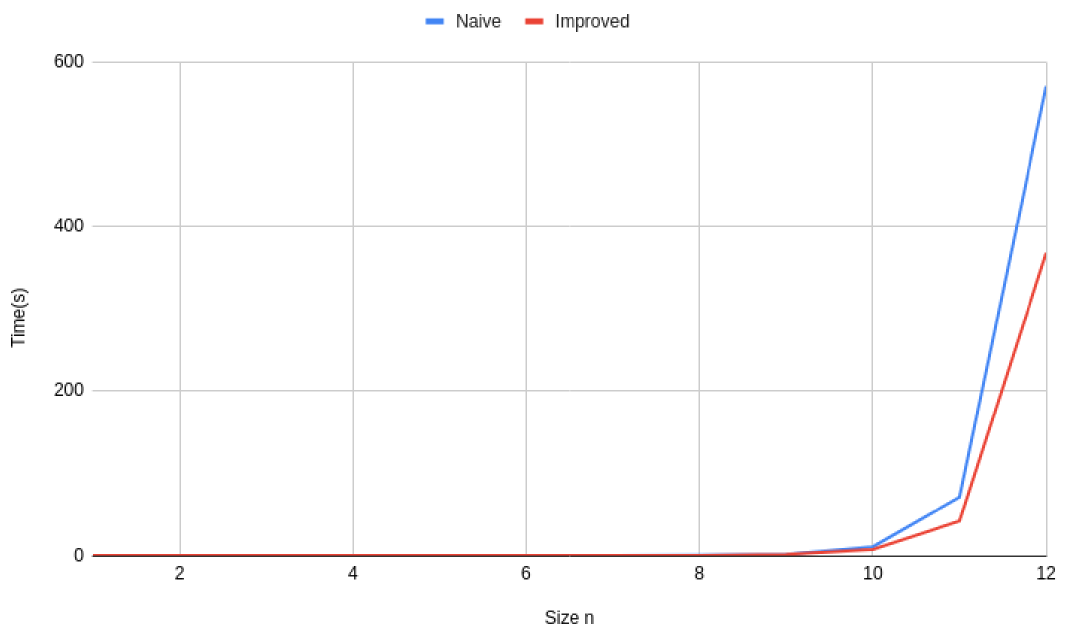
The first thing to notice is that both versions stopped having good results at the same order of n, $n = 12$.

Another interesting fact is that the improved model's performance is pretty similar to the naive one until the order $n = 9$, however, after that the performance increase is visible, with a speed up of around 65%.

On the tables the number of solutions found are also displayed, the results are similar to the ones obtained from *Wikipedia* [1]

For a better comparison of the time between the 2 versions a graphic was made.

Fig. 3: Representation of *Table 1* in a graphic



I believe the program would be able to find all the solution for $n = 13$, however, when the time, t, was $t = 571s$ it had only found 4695 out of 12828 know Costas Array for that n, so it would take a while.

Another experiment that was benchmarked was how much time it took to find one solution, given n. Just like the previous experiment, this one stopped when the time to find a solution stopped being acceptable.

Size	Time(s)
1	0,05
2	0,04
3	0,05
4	0,05
5	0,05
6	0,05
7	0,04
8	0,07
9	0,06
10	0,07
11	0,07
12	0,09
13	0,08
14	0,72
15	0,91
16	27,52
17	83,71
18	2,58

Naive

Size	Time(s)
1	0,05
2	0,05
3	0,05
4	0,04
5	0,05
6	0,05
7	0,06
8	0,06
9	0,06
10	0,07
11	0,10
12	0,12
13	0,13
14	0,13
15	0,32
16	1,36
17	72,4
18	182,96

Improved

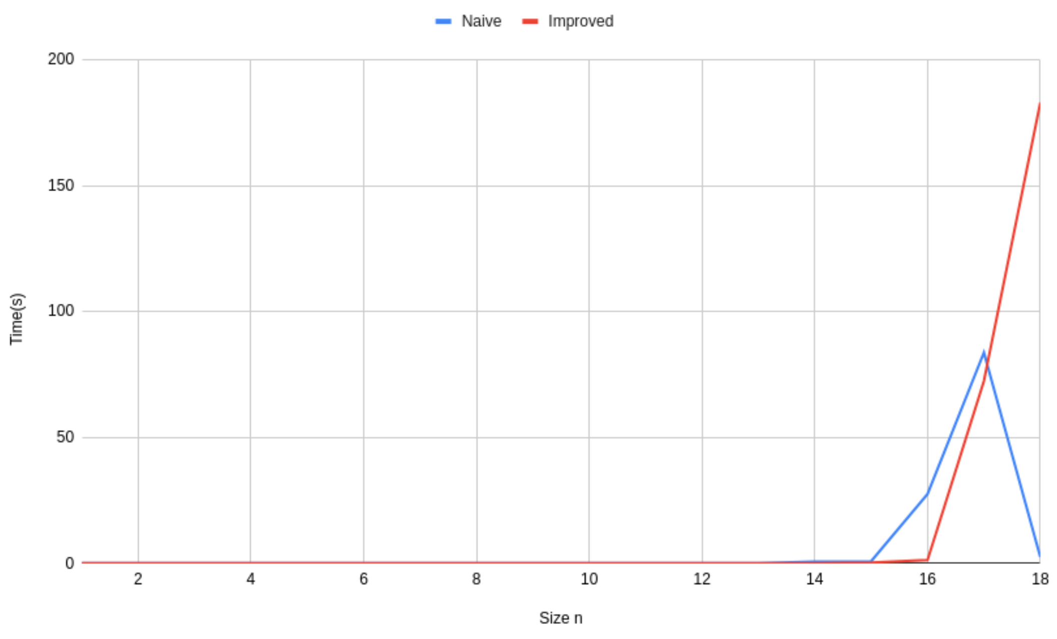
Table 2: Time to find one solution for each n

Just like the previous benchmark, the improved version only starts to get better as the order of n starts to get bigger, $n = 14$. From that order of n onward the improved version has always better results than the naive one, except for $n = 18$.

The result for this n can be considered an outlier on the Naive version, since it performs considerably better than the same n for the Improved version, and better than the previous n for the Naive version.

The benchmark was stopped at $n = 18$ because for $n = 19$, after 400s, no solution had been found.

Fig. 4: Representation of *Table 2* in a graphic



The information described above, from ***Table 2***, can be observed at this graphic.

7 Conclusions and future work

Overall, the results of this experiment were quite good and can confirm how complex this problem can get as the size, n, increases.

Even with a fast solver like *OR-Tools* it started to become infeasible to find all solutions for n around n=*12*. In the future, the benchmark could be done for greater instances of n, to check how the time really grows exponentially, as there was only opportunity to watch this growth for a couple of n.

A good way to improve the results is by removing eventual symmetries from the problem or by using different search strategies. Just like I said before, I was not able to experiment the last option with *OR-Tools*, so, It would probably be a good idea to port this model to another *CP Solver*, like *Choco*.

References

1. Costas array `https://en.wikipedia.org/wiki/Costas_array` (Visited on 23/01/2021)
2. Eight Queens Puzzle `https://en.wikipedia.org/wiki/Eight_queens_puzzle` (Visited on 23/01/2021)
3. Diaz D., Richoux F., Codognet P., Caniou Y., Abreu S. (2012) Constraint-Based Local Search for the Costas Array Problem. In: Hamadi Y., Schoenauer M. (eds) Learning and Intelligent Optimization. LION 2012. Lecture Notes in Computer Science, vol 7219. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-34413-8_31
4. OR-Tools `https://developers.google.com/optimization` (Visited on 23/01/2021)
5. MiniZinc Challenge 2020 Results `https://www.minizinc.org/challenge2020/results2020.html` (Visited on 23/01/2021)