Artificial Intelligence Algorithms project report
Mathys BRONNEC, Lisa CHARUEL, Jules DEPLANCHON

In manufacturing, the problem of maximizing production while having different constraints is recurrent. This report delves into such a challenge within the context of biscuit production, where the primary task is to optimize the value of the biscuits made from a single roll of dough. The complexity of this problem comes from the presence of defects in the dough, which necessitates a strategic approach to biscuit placement and cutting.

This project showcases the development and application of two distinct algorithmic solutions to this challenge. We first though of implementing a Greedy Algorithm, trying to place the higher value/length ratio biscuits first. This approach, while straightforward and efficient in certain aspects, presented limitations in dealing with all the constraints of the problem.

In pursuit of a more robust solution, a Constraint Satisfaction Algorithm was then explored. This algorithm demonstrated superior performance by effectively managing the various constraints of biscuit sizes, values, and the distribution of defects in the dough. The transition from a simpler heuristic to a more complex constraint-based approach shows the evolution and depth of problem-solving strategies employed in this project.

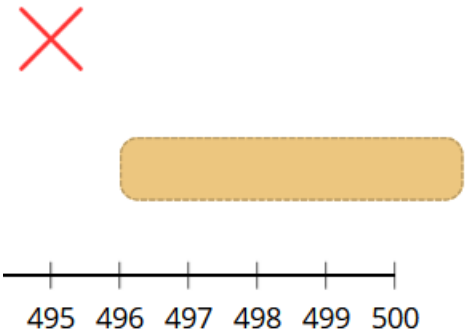
We have made a 'Biscuit' and a 'Dough' class, which helped us with the implementation of the two algorithms while helping us to guarantee no impossible placements were done. This report aims to detail the methodologies employed, focusing on the approach, and reasoning behind each algorithm.

The scenario of this biscuit factory is simplified. We have a 1-dimentional problem rather than an at least two-dimensional problem you would expect. The dough roll is 500 units long. We can place a biscuit at any integer unit of this roll and two biscuits obviously can't be overlapping.

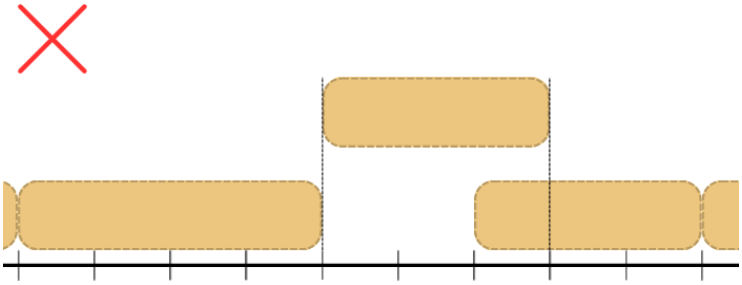
Now let's talk about the defects. All along the dough, there are 3 types of defects (type a, b, or c). We have a tab with the list of all the defects. In this tab, we have an "x" column, representing the position of the defect from the beginning of the roll, and a "type" column saying if it is a type a, b, or c defect. The defects are used to know whether a biscuit can be placed on a certain "x" position or not.

So, to be able to place a biscuit, you need these three conditions:

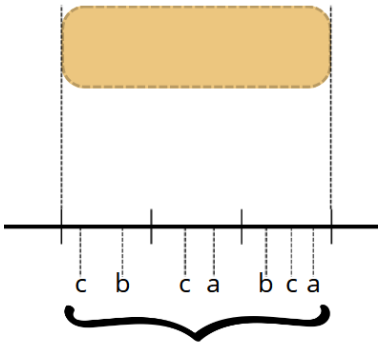
The biscuit should not get out for the roll



Two biscuits should not overlap

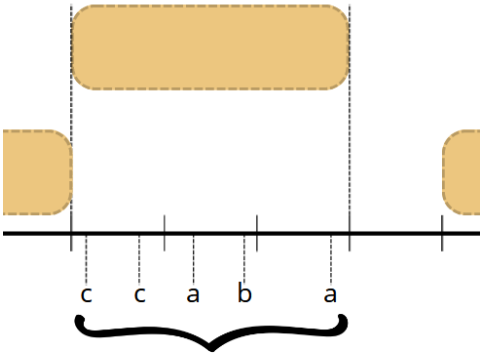


The number of defects of the dough on the area we want to have the biscuit on should be lower than the number of defects the biscuit can take for each type of defect. Let's see an example. The "biscuit 3" have a length of 5 and can take up to 2 type "a" defects, 3 type "b" defects and 2 type "c" defects. If you want to place a "biscuit 3" at position x, ignoring the other conditions, you need to be sure that on the interval [x, x+3[, there are 2 type "a" defects or less, 3 type "b" defects or less and 2 type "c" defects or less.



-2 type a
-2 type b
-3 type c

In the example above, there are too many type "c" defects



-2 type a
-1 type b
-2 type c

In the last example, the biscuit can be placed!

This problem mirrors real-world scenarios in manufacturing where resources are limited and must be utilized optimally. It exemplifies the challenges faced in production lines, where maximizing output while adhering to quality standards is a delicate balance. The project, therefore, not only addresses the specific problem of biscuit manufacturing but also is an introduction to optimization in production processes.

In summary, this project is a good way to have a glimpse of the production optimization process while allowing us to see the potential usage of AI algorithms in real situations.

Let's now see how we approached the problem. First, we wanted to have an environment to work on. That's why we created a Biscuit class. A Biscuit is composed of a length, a value, and a maximum allowed number of each type of defects. We also added a __str__ function to have a readable and understandable representation of a biscuit. There also is a function to create an "empty space biscuit" that is a placeholder having a length of 1 and a value of 0.

We have then made a dough class that represents the roll as a list of biscuit. For the dough length, there is a global variable set at the beginning of the code. We also made functions like "number_of_defects" and "canbeplaced" to facilitate biscuit placement according to our constraints as well as a "Value" function to get the value of the roll and a "printDough" function which name is pretty self-explanatory.

We have then created the 4 objects that will represent the 4 biscuits of the instructions and imported the defects list.

When studying the problem, we thought of how we would solve it if we needed to place the biscuits by ourselves. We thought we would just try to put the most biscuits with high value/length ratio since they take less space and optimize the value. We have noticed that this straightforward approach could be a good first algorithm to have an idea of the kind of score we should expect.

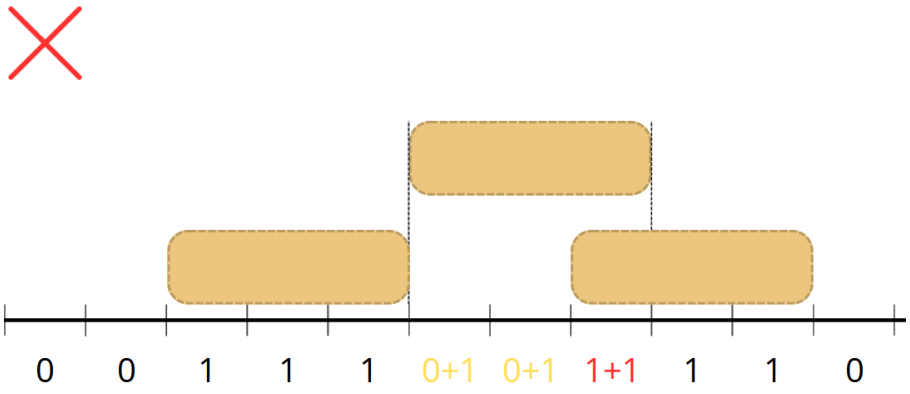
Our greedy algorithm was born. We have made it exactly the way we described it above. We made a list with our 4 biscuits; we sorted it by value/length ratio in descending order. We then created the algorithm. It tries placing the first biscuit of the list, if it can't, it tries placing the second one, if it still can't, it does the same with the third one then the fourth one. If it cannot place any of the biscuits in the list, it places an empty space. He does this until the end of the roll (until the total length reaches doughLenth).

This first algorithm gives better results than what we had expected. Let me explain what type of score we were expecting. Usually, a greedy algorithm doesn't give the best results possible, and is even quite far from it. Without the defect constraints, the maximum score we can expect is 800, we can do so by placing a 100 biscuit_3 (biscuit with the highest value/length ratio). However, with the defect constraint, this solution is not possible. With this in mind, we knew the score would be lower that 800. We just didn't know how much lower; it is pretty much guts feeling after this. We were expecting a score around maximum 700 for the greedy algorithm. But with this first algorithm (which we knew was not the best algorithm) we got the high score of 735! What a surprise. It was a good score, but we knew we could do better. We had a problem with clear constraints and a value maximization goal. A constraint satisfaction algorithm seemed to be the perfect solution.

We thought about how we could implement the different constraints using ORTools. After thinking of different possibilities like having a list of 500 elements and using placeholders or trying to use the same system as our dough class. We thought about doing a list of biscuit and their starting positions, this was easier for the implementation.
(example: [(Biscuit1 ,0), (Biscuit1,8), …, (Biscuit0,496)])

We started by setting up a bool var for each biscuit starting at each position. The bools variable goes from x=0 to doughtLenght – Biscuit.Length for each biscuit, so that the first constraint is respected (no biscuit cut out of the roll). We then have set up the overlapping constraint this way:

From each biscuit start pos to biscuit length, we add 1 to a list of 0, and every element of the list needs to be inferior or equal to 1



For the defects constraint, we listed all the “x” position where each biscuit cannot be placed on because of defects and we set their value to 0, which means, there is no biscuit_i at position j.

After setting up all the constraints, we added the Maximization objective to our function. Our goal was to maximize the values sum of the biscuits made.

Our model was ready, and gave us a solution, however, we wanted it to be compatible with our dough class. To do so, we sorted the biscuit list by start_position then we filled the empty spaces with empty_space_Biscuits (Biscuits of length 1 and value 0) and we created a new instance of dough and we added all of the biscuits in the list to the roll instance, making sure the solution is valid (since a the dough class verify if the biscuit can be placed before placing it).

We can now print our solution and see that we get a score of… 760! We may have found an optimal solution; we don’t think the score can go any higher than this since the status of our model is “OPTIMAL”.

We want to compare our algorithms:

The Greedy Algorithm was a good starting point, surpassing expectations with a score of 735. Its design, which prioritizes biscuits with the highest value/length ratio, demonstrates the algorithm's capability to yield a high-value output from the dough roll. However, focusing on immediate gains only can be counterproductive to get a really high score, since it can disturb the placement of the other biscuits.

In contrast, the Constraint Satisfaction Algorithm, using OR-Tools, was methodical in considering each constraint. By systematically accounting for biscuit placement, defect interference, and ensuring no overlaps, this algorithm took a more global view of the problem. This resulted in a better doing model, achieving a higher score of 760.

The performance of the Greedy Algorithm is highly dependent on the order and value-to-length ratio of the biscuits. While it can quickly generate a good solution, it may not consistently find the best solution, especially when faced with a complex distribution of defects.

The Constraint Satisfaction Algorithm's performance excelled due to its global perspective on the problem. By maximizing the overall value while strictly adhering to all constraints, it achieved a superior output quality. The ability to consider multiple factors simultaneously allowed for a more refined and potentially optimal solution.

Finally, while the Greedy Algorithm serves as an effective starting point and offers a satisfactory solution especially to get an idea of what kind of score to expect, the Constraint Satisfaction Algorithm emerges as a more robust choice for this problem. Its ability to navigate complex constraints and produce a higher-value score aligns with the goal of maximizing production quality and efficiency. The increase in complexity and computation time is worth compared to the optimality of the solution obtained.

Through this project, we learned that an AI Algorithm problem is a reflection problem, where you think of a solution, see the results it gives you and then adapt your solution, or change the way you do things because of the solution you got. Making a first algorithm was helpful for doing things step by step. We surely would not have created the classes if we had directly tried doing the constraint satisfaction algorithm, leading to a solution that could be impossible. This way of doing things helped us making sure that what we were doing made sense. The most challenging part was the implementation of the constraint satisfaction algorithm. At first, we didn’t expect to represent our solution that way in the second model. However, after multiples tests, it appeared to be the easiest way of doing it. This project has underscored the importance of adaptability and perseverance in the face of complex problem-solving, it has expanded our understanding of algorithm development and its potential to yield impactful solutions.

Thanks for reading our project reports.