Our problem involves optimizing biscuit production from a single roll of dough in a biscuit manufacturing setting. The key challenge is to maximize the total value of biscuits produced while considering various constraints, such as biscuit sizes, values, and the maximum number of defects of each class that a biscuit can contain. The objective is to strategically place different types of biscuits on the dough roll, taking into account these constraints to achieve the highest possible total value. The complexity of the problem lies in the combinatorial nature of biscuit placement and the limitations imposed by defects and biscuit specifications

To choose an adapted solution for this problem, we looked for similar classical problems and their resolutions. The problem that was the closest that we found is the Knapsack problem. The similarity between those two situations is that they are both optimisation problems with respect to some constraints. Like the Knapsack problem, our biscuit situation is an optimization problem with constraints, and it is NP-complete which means that there is no known optimal algorithm. However, there is some algorithm that get closer to the optimal situation than a simple random repartition or a greedy algorithm, these algorithms are called metaheuristic optimizations.

We will explore two of these solutions, the first one will be the Ant Colony Optimization algorithm (ACO), and the second one will be a Genetic Algorithm (GA).

ACO:

The goal is to simulate an ant colony, to do so we create a class ant. Each ant will first explore a random possible solution and gives it a “pheromone” value depending on the final score. This pheromone score is registered in a matrix common to all the ants. Concretely our pheromone matrix is composed of 4 rows (one for each category) and 500 columns (one per node of our dough). We initialize this matrix with np.ones((4,500)), next every time an ant must choose a new biscuit to put on a specific spot, it will make the next calculus on every types of biscuit that are available on the x spot: . This operation gives the probability that we use a biscuit of type a on the spot x. With the value of the pheromone matrix for biscuit a on x (pheromons[a,x]), the heuristic value of this move (biscuit value/length). We also have α and β which are coefficient that control the influence of the pheromone value and the heuristic value in the finale probability.

So that is how our pheromone matrix help us to take decisions, but as we said, it is initially a matrix of ones, therefore we must build it from training. To build a proper matrix we ask to multiple ants to build a correct dough using our matrix. Once they are done we look at the scores of all the new dough and for each node used in each dough we add the normalized score (divided by the highest score and the number of time this specific node is used with this specific type of biscuit) to its spots in the pheromone matrix and we normalize all the nodes in the matrix where

. And we redo this operation as many times as necessary. But in order to allow ants to take new decisions we have to apply an evaporation factor, meaning we reduce equally all the scores of the pheromone matrix. This process will lead to the convergence of the pheromone matrix.

This is the logical process for the ACO, but we also had to create an environment to execute this algorithm, so we made a class Dough and a class Biscuit. The Biscuit class is the simplest, it only takes in argument the type of biscuit (0,1,2,3) and then we store in the Biscuit object the informations about this biscuit (value, length, number of accepted defects, etc…). The Dough class is a bit more complicated, its arguments are the length of the dough (for us we put it by default at 500), the surface which will be the container of the information about the biscuits that we use, concretely what we store in the surface variable are tuple (x, type) where x indicates the beginning node of our biscuit and x the type of biscuit, and it also have a defects argument that we use to build a dictionary that stores the number of defaults of each types in each units. The most important functions in the class Dough are the possible(T, x) function which return True if we can begin a biscuit of type T at x, and action((x,T)) that modifies the surface to register the biscuit of type T at x.

After we implemented all of this we made some measures, our score is usually around 710, the surface of the dough is always fully exploited, this score isn’t great because a random repartition that follows the rules is around 680, so we only won 20 points, but the maximum value without minding the defects is 750, so it could be near the optimal repartition.

We tested our algorithm with several hyperparameters, first we wanted to test the influence of the evaporation rate. It appears that a highest value of evaporation rate lead our matrix to converge faster, however a too great value would discriminate greatly some values after the very first iteration.

We observed some small change in the final values of the matrix for different α and β, but we couldn’t fully understand why these changes occurred, so we choose to keep a value of 1 for both. We initially tested with 100 iterations, but we constated that the convergence was already very strong at this point (only one and zeros values) so we reduced it to 50, which is enough and takes much less time. Finally, we obtained the attached pheromone matrix where represents the probability of having a biscuit of category *i* in the *jth* position.

GA:

A genetic algorithm (GA) is an optimization technique inspired by the process of natural selection. It simulates evolution by iteratively improving a population of candidate solutions. Each solution, is an individual in a generation, it is evaluated by a fitness function. The best-performing solutions are more likely to be selected for reproduction, where crossover and mutation operations create new solutions (offspring). Over successive generations, the population evolves towards an optimal or near-optimal solution. GAs are effective for complex optimization problems where traditional methods might be less efficient.

The genetic algorithm for our biscuit distribution problem operates through these technical steps:

1. **Population Initialization**: Generates a diverse initial population of Dough instances using the create function. Each instance represents a potential solution, encapsulating a specific arrangement of biscuits.
2. **Fitness Evaluation**: Determines the fitness of each Dough instance based on its score, which quantifies the total value of biscuits, adjusted for defects and type compatibility.
3. **Parent Selection**: We choose to reproduce every individual, to do so we associated to each individual its score and we applied an eugenics logic. Meaning that if you have a higher score you are more likely to be reproduced with another high-value individual.
4. **Crossover**: For our crossover function we took the parents and looked for the common points, for example, if they both have a biscuit starting at point 134, then point 134 is considered as a common point. Next choose at random on of these points and our children are the concatenation of the parents on this point (child1=parent1[:134]+parent2[134:]).
5. **Mutation**: Our function choose a random sample of the dough and create another random dough with the same dimensions and constraints, then we simply exchange the original sample with the new one. This operation is done after the Crossover, and only according to a mutation rate that we choose which decrease by 5% at each generation.
6. **Generational Iteration**: Repeats the process across multiple generations. Each generation involves selection, crossover, and mutation, progressively evolving the population towards better solutions.

In our application of the GA we created a starting population of 100 dough and we iterated over 70 generations with a starting mutation rate of 6%. The results of this method are quite good, we have a finale score of 735, which is more interesting than the ACO for a much smaller time of calculus (maybe because of a mistake on our side concerning the ACO implementation). We observed that a higher number of generations often brings a better score, but after 70 iterations it doesn’t really improves. Finally, our best dough was the one attached.