Roll Dough Optimization

Matteo Chianale - Charles Terrier - DIA5

2023 - 2024

Contents

Chapter 1

Introduction

1.1 Problem explanation

Remainder of the problem
- We had to answer to an optimization problem which is to place different types of biscuit on a roll of length 500.

- Perhaps, there are different constraints placed on the roll which can be 'a', 'b' or 'c'.
- Types of biscuits:
biscuit0 = 'symbol': '0', 'value': 6, 'length': 4, 'a': 4, 'b': 2, 'c': 3
biscuit1 = 'symbol': '1', 'value': 12, 'length': 8, 'a': 5, 'b': 4, 'c': 4
biscuit2 = 'symbol': '2', 'value': 1, 'length': 2, 'a': 1, 'b': 2, 'c': 1
biscuit3 = 'symbol': '3', 'value': 8, 'length': 5, 'a': 2, 'b': 3, 'c': 2
- So we can't place biscuit on a part of the roll where the biscuit doesn't respect its constraint. Also we can't overlapping biscuits.
- Moreover, information about defects was in 'defects.csv'.

1.2 Representation of the problem

In a first part we needed to create a class to represent the problem.
You can find this class in **roll.py**.

attributes of roll class:
- **Size of the roll** (in our case its 500).

- **An array to contain defects**, each case of the array is a dict with key equals to the defect's names 'a','b' and 'c' and initialized for each defect to 0. So each case of index i represent the position i of the roll and tells us about defects at this position.
- **An array to contain biscuits**, same idea that defects array, but we just represent each case of the array by the symbol of the biscuit:
example : biscuit0 is place from index 1 of the roll, you will have the roll like this = [_,'[0','0','0','0','0','0]' .....]. - **a Value** which is the sum of values of each biscuit in the roll.

We added different functions use full to answer to the problem:
- For example get the total number of each defects which are placed on a partition of the roll.
- A function to add a biscuit if the conditions are respected, else does nothing and return False. And more !

1.3 Initialization of the problem

- Before creating algorithms to answer the problem, we initialized our main roll of size 500 and we had defects.
- To add them, we read 'defects.csv', perhaps defects in these file had a float position x. However in our case, each biscuit needs to start at an integer position, so we took integer parts of each defect and added them to the roll.

View of a part of the main roll after placed defects:

Total Value : 0

	0	1	2	3	4	5	6	7	8	9	10	11
0	{'a': 2, 'b': 0, 'c': 0}	{'a': 1, 'b': 0, 'c': 0}	{'a': 1, 'b': 1, 'c': 0}	{'a': 1, 'b': 1, 'c': 0}	{'a': 0, 'b': 0, 'c': 0}	{'a': 0, 'b': 0, 'c': 0}	{'a': 0, 'b': 0, 'c': 0}	{'a': 0, 'b': 0, 'c': 0}	{'a': 0, 'b': 0, 'c': 1}	{'a': 0, 'b': 0, 'c': 2}	{'a': 0, 'b': 1, 'c': 1}	{'a': 0, 'b': 0, 'c': 0}
1	_	_	_	_	_	_	_	_	_	_	_	_

- index 0 is for defects.
- index 1 is for biscuits but its currently empty.

Chapter 2

How to place biscuits ?

Before to code our final algorithms, we want to manage how to update a roll with just one iteration with the more appropriate way. First we will just place randomly biscuits on the roll and check the minimum value to reach with other methods.

2.1 Random method

function in 'roll.py' : randomized_roll_rec, which takes in argument the index where to start on the roll, but for all project we will just start at the index 0. We coded this random method recursively.

How it works ?
- we just start at index 0, we pick a random biscuit, if the chosen biscuit can be place on the roll we place it else we chose randomly remain biscuits. perhaps if zero biscuit can be place we recalled the function at the index + 1. - Else, if we success to place a biscuit we recalled the function at index + length of the placed biscuit.

Results:
- Mean value of result : **664**. This mean will be our landmark, objective to move with uses of other algorithms. - Mean time to run : **0.0019**.

2.2 Random Method with Scoring

In this section, we refine our approach by introducing an algorithm that assigns a score to each type of biscuit. This score directly influences the probability of placing a biscuit on the dough.

2.2.1 Assigning Scores to Biscuits

For each biscuit, we calculate a 'benefit' by dividing its value by its length. We also consider its tolerance to defects: the higher the defect tolerance, the higher the score.

Next, we normalize these features so that the scores range between 0 and 1.
We apply importance rates to each feature, which must sum up to 1. The final score is the sum of each feature's value multiplied by its importance rate.
The sum of importance rates (weight) musts be equal to 1.

Finally, to determine the probability of placing each biscuit, we normalize the final scores.

2.2.2 Quantitative Results

- current scores with importance rates : [0.4, 0.2, 0.2, 0.2]

	benefit	a	b	c	score	probabilitie
0	0.363636	0.15	0.0	0.133333	0.647070	0.290514
1	0.363636	0.20	0.2	0.200000	0.963736	0.432665
2	0.000000	0.00	0.0	0.000000	0.000100	0.000090
3	0.400000	0.05	0.1	0.066667	0.616767	0.276911

The mean value achieved by this algorithm is **720**, which is an improvement over the mean value of 665 obtained by the random algorithm.
This represents a gain of +55. However, it is important to note that the time to run this algorithm has also increased.

2.2.3 Variations of importance rates

We compare the results of this scoring method with those of the previous random method, focusing on the average value achieved and the execution time. We use charts and tables for a visual presentation of these comparisons.

2.2.4 Reflections on the Method

We discuss the effectiveness of the scoring algorithm compared to the random method. We explore how the scores and probabilities affect the placement of biscuits and the total value achieved.

2.3 Variation of Scoring

Now, we apply different scores to our biscuits. Although it is tempting to focus solely on the benefits in the score, considering constraint rates is also crucial as running the algorithm multiple times with varied probabilities can be more time-efficient, especially for biscuits with more constraints.

2.3.1 Defining Scores and Probabilities

We introduce a function `getScore` which takes importance rates as input and returns a score for each biscuit. This score is used to calculate the probability of placing a biscuit on the dough.

2.3.2 Analysis of Importance Rates

We conduct a first analysis varying the importance rate for the benefit, keeping the constraint rates static. The importance rates are adjusted from 0.1 to 1.

2.3.3 Results and Observations

Upon running the algorithm with varying importance rates, we record the mean, maximum values achieved, and execution times.

As anticipated, improved results are noted with an increase in the benefit importance rate. This is evident from the mean value of the algorithm, which shows an increase when the benefit importance rate is given more weight relative to the constraint rates.

Regarding time complexities, better outcomes are observed with higher constraint rates, suggesting a trade-off between computational efficiency and the quality of results.

2.4 Using Partitions to Define Constraint Rates

2.4.1 Global Vision

In the previous part, we assigned an equal importance rate for each constraint, corresponding to the remaining value after accounting for the benefit rate. However, we can refine these rates by considering the actual distribution of defects on the dough roll.

For instance, if we have a significant number of defects 'a' compared to 'b' and 'c', we might choose to allocate a higher importance rate to 'a'.

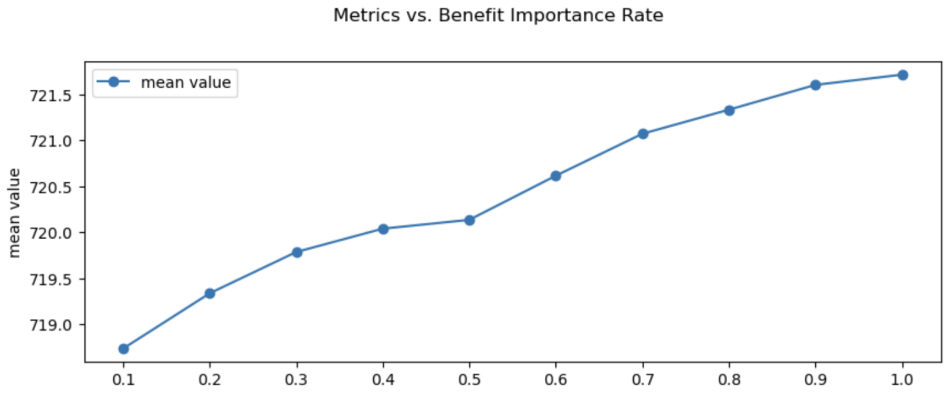


Figure 2.1: Mean Value vs Benefit Importance Rate

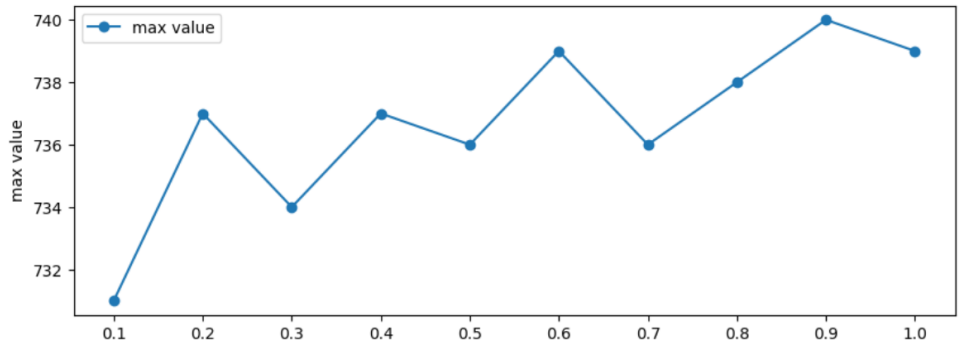


Figure 2.2: Max Value vs Benefit Importance Rate

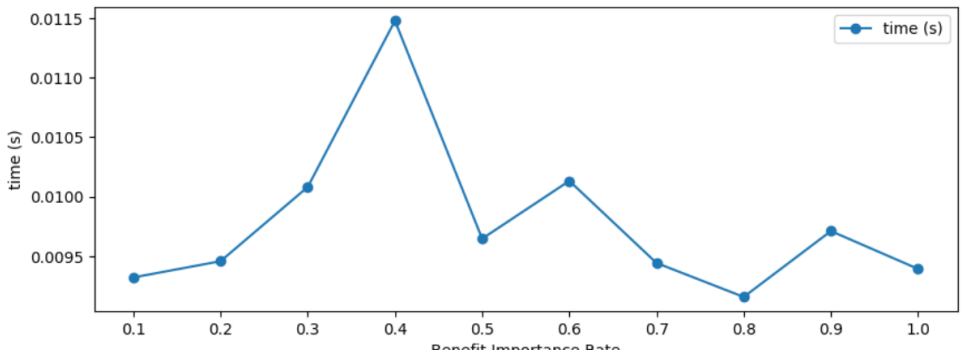


Figure 2.3: Mean Time vs Benefit Importance Rate

This approach is expected to provide a more accurate representation of the constraints.

2.4.2 Auto-Constraint Rate Experimentation and Analysis

The introduction of the `auto_constraint_rates` function represents a significant enhancement in our approach to scoring biscuits. This function dynamically adjusts the importance rates of constraints based on the distribution of defects throughout the dough roll. By analyzing the number of each type of defect, we can assign a weighted importance to each constraint, potentially leading to a more refined and efficient biscuit placement strategy.

In our experiments with auto-adjusted constraint rates, we conducted extensive simulations to quantify the effectiveness of this method. These simulations were iterated numerous times to capture a robust set of data points, allowing for a comprehensive analysis of the algorithm's performance.

2.4.3 Refined Approach with Dough Partitioning

To further our optimization efforts, we investigated the use of dough partitioning as a method to define constraint rates with higher precision. Partitioning the dough allowed us to apply constraints locally, which could lead to a more tailored and potentially more effective placement of biscuits, especially in areas with a high concentration of specific types of defects.

This granular approach to constraint application was tested to see if a localized consideration of defects would translate into higher overall scores. However, initial results indicated that the mean score achieved with partitions did not surpass that of the previous algorithm, suggesting that further refinement might be necessary.

2.4.4 Summary of Findings

Our exploration into the placement of biscuits using randomly generated probabilities and auto-adjusted constraint rates has led to a nuanced understanding of the interplay between benefit rates and constraints. The `random_probabilities_roll_rec` function, which utilizes these dynamic probabilities, indicates that the optimal choice of benefit rate is a delicate balance. It is not solely the highest benefit rate that yields the best results; instead,

it is the rate that acknowledges the distribution of defects and the computational time required for the algorithm to run.

In essence, while an increase in the benefit rate generally correlates with better results, this must be balanced against the time efficiency of the algorithm. Finding the sweet spot where benefit rates and constraint rates align optimally is critical for maximizing the outcome of the biscuit placement strategy. This ongoing process is a matter of fine-tuning parameters and adapting our approach based on the insights gained from each simulation run.

Chapter 3

Artificial Bee Algorithm

We coded an adapted artificial bee algorithm for our problem. This algorithm computes 3 phases during max iteration input value.

3.1 Abstract explanation of the algorithm

Employee Bee Phase: In this phase, each "employee bee" represents a potential solution to the optimization problem. These bees explore the vicinity of their current solutions, seeking to find better ones. This process is akin to bees searching for nectar near their hive. The quality of each new solution (nectar amount) is evaluated, and better solutions are remembered for further exploration.

Onlooker Bee Phase: Onlooker bees watch the dance of the employee bees in the hive. The dance is a form of communication used by bees to convey the quality and location of the nectar source they have found. In the ABC algorithm, the onlooker bees choose which employee bee to follow based on the quality of the solution they represent (the more successful the employee bee's dance, the more likely an onlooker bee will follow it). This selective process allows the algorithm to focus more on promising areas of the solution space.

Scout Bee Phase: Scout bees are responsible for diversification. In nature, when a food source is exhausted, some bees will go out to find new sources. Similarly, in the ABC algorithm, if a solution cannot be improved

upon further (indicating a potential local optimum), it is abandoned, and the corresponding bee becomes a scout. The scout bee then randomly explores the solution space for entirely new solutions, preventing the algorithm from becoming overly concentrated in one area and potentially stuck in a local optimum.

Together, these phases enable the ABC algorithm to balance exploration (finding new solutions) and exploitation (refining existing solutions), key components in solving complex optimization problems efficiently.

3.2 Parameters of our ABC

To create an ABC instance we give few parameters:

- a roll : In our case, we just directly use the main roll.
- colony size : The number of 'bees', in our case the number of different rolls from the main one. Increase this parameter allows to gain in variability of our potential solutions.
- max iteration : The total number of run of the 3 previous phases. More is bigger, more we gain in values, perhaps, also increases complexities times. Increase it is not necessary the best solution, because we will see after in our results that a limit value is reached.
- limit : This value defines how many times a roll can avoid improvements. When trial of a roll reach's this limit, it resets the roll to a totally new solution. This parameter allows to avoid flat zone or to keep trying to update a good solution which can't be improved more.

3.3 Adaption of ABC for our problem

In our case, it was complicate to define what is a food source for the ABC, we decided to define a food source as a roll with a benefit rate for scoring method.

So when we initialize an ABC instance, it will generate n : colony_size of food sources with a random benefit rate between 0 and 1 for each one.
The idea, is the variable to update with ABC is not directly the roll but the benefit rate.

So during employee or onlooker phases, to update a roll we generate several partitions of the roll with a minimum size parameter and a probability for each one. From all this partitions we pick one using probabilities. We give higher probabilities to partition with a smaller score which is equal to the value of the partition divided by its length.

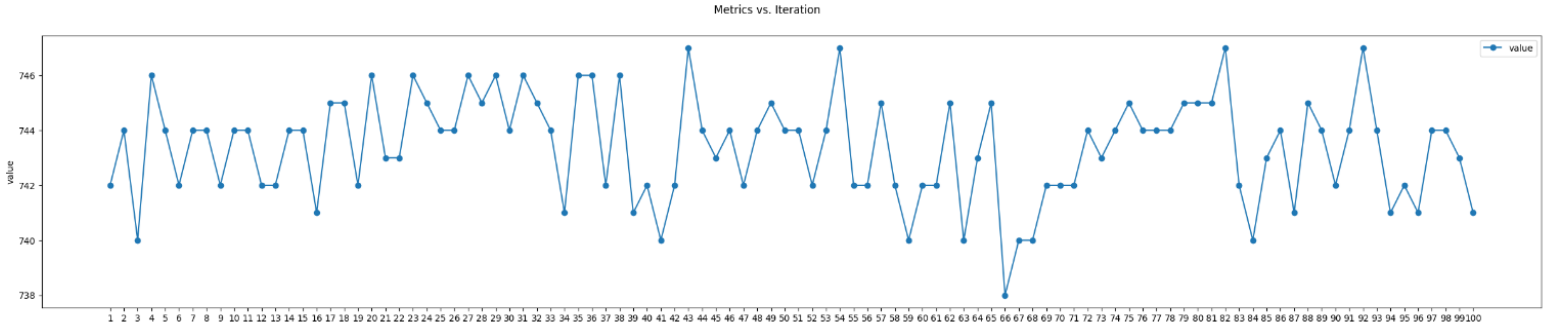
From the chosen partition, we run random_scoring_roll_rec_ using a new benefit rate which is randomly choose between the current roll's benefit rate and a selected partner roll's benefit rate.

For scout phase, we just run random_scoring_roll_rec_ on the roll with a totally new benefit rate.

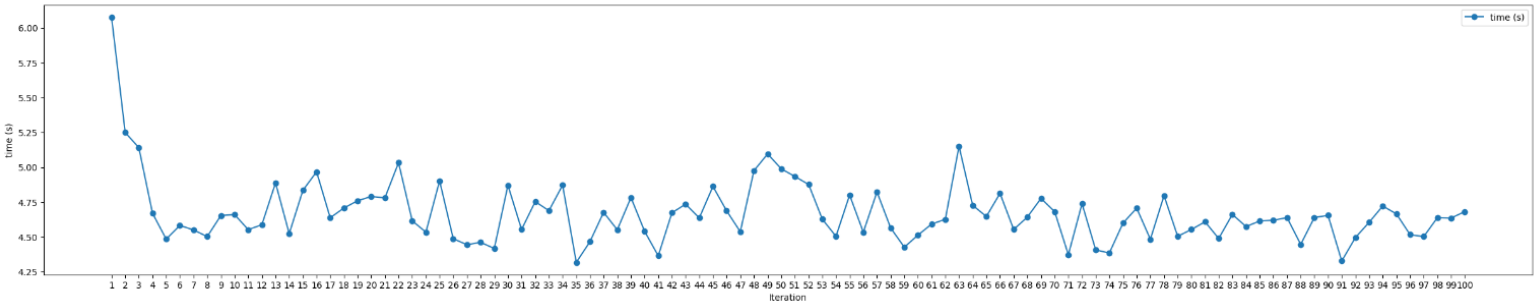
3.4 Analysis of ABC

We run 100 times ABC on the main roll with:
- colony size : 10
- max iteration : 100
- limit : 50

Value vs Iteration :



Time vs Benefit Importance Rate :



3.5 Conclusion

We can see that a bound is reached : 149. So we want to know by considerably increasing parameters to reach this bound. But it didn't made improvement. So we want to create a new algorithm based on our ABC.

Chapter 4

Artificial Bee Algorithm, recursive version

4.1 Our idea

- Now we want to apply previous algorithm but recursively.
- To made this we start ABC on the main roll.
- **The idea is to recall ABC on smaller roll (partitions of main_roll).**
- Give the possibilities to go back when you reach the smallest partitions or when you don't make improvement.
- When you apply ABC on smaller partitions, **limit value needs to decrease** because you have less possibilities of biscuit's placements.
- Minimum limit is set to 5, we had good results with this value.
- To avoid unnecessary runs of ABC, we added an attribute **max_update**, so when no improvements are made we decrease max_update, and if it reaches 0 we stop the algorithm.

4.2 Conclusion

Finally, it was a success, we had new best solutions with a value between 750 to 755, in depends of the run. Perhaps it increases the complexities time a lot more than simple ABC, from 3 to 7 minutes !

We can improved this new algorithm in the future, by, for example, compute

the bound values of partitions and stop ABC for partitions with a found value greater than this bound. Or by finding partition partners on the roll and use them to avoid to run several times ABC for sames partitions.