REPORT AI ALGO : Biscuit manufacturing factory

KRIKA Jinane DIA 4
KRIKA Camila DIA 4
BOYER Hugo DIA1

December 24, 2023

1 Introduction

The problem can be formulated as a combinatorial optimization problem where the objective is to maximize the total value of the produced biscuits while respecting constraints related to size, defects, and the arrangement of biscuits on the dough roll.

The roll can contain defects of different classes at different positions. The task is to find an optimal assignment of biscuits on the roll, while adhering to certain constraints such as integer positions, non-overlapping placement, and compliance with defect thresholds. The value of a solution is determined by the sum of the values of individual biscuits placed on the dough roll.

2 Data for problem-solving

2.1 Some numerical information

- Length of the roll = 500 units
- The roll has three classes of defects ('a','b','c')
- The biscuit manufacturing factory aims to produce 4 types of biscuits, which are :
  - Biscuit 0 with a length of 4, a value 6, and maximum allowed defects as ('a':4,'b':2,'c':3)
  - Biscuit 1 with a length of 8, a value 12, and maximum allowed defects as ('a':5,'b':4,'c':4)
  - Biscuit 2 with a length of 2, a value 1, and maximum allowed defects as ('a':1,'b':2,'c':1)
  - Biscuit 3 with a length of 5, a value 8, and maximum allowed defects as ('a':2,'b':3,'c':2)

We have also a CSV file named "defects.csv". This file contains two columns:

- "x" column refers to the positions of defects on the roll.
- "class" column refers to the class of the defect.

2.2 Constraints of the problem

- Overlap Constraint: Ensure that no biscuits overlap on the dough roll.
- Defect Constraint: The number of defects of each type in a biscuit must not exceed the allowed threshold.
- Roll Size Constraint: The sum of the sizes of the biscuits must not exceed the length of the dough roll

3 Example

The 'dough roll' represents a one-dimensional line of length 500 units. 'Defects' are imperfections that can occur at certain positions along this line. 'Defect classes' indicate different types of imperfections ('a,' 'b,' 'c') along with their positions, which are specified in the CSV file.
As an example, we can consider 'Biscuit 0' with the following characteristics:

- Length: 4 units
- Value: 6
- Maximum allowed 'a' class defects: 4
- Maximum allowed 'b' class defects: 2
- Maximum allowed 'c' class defects: 3

The goal is to position this biscuit on the dough roll in a way that maximizes the total production of Biscuits while adhering to the constraints. Here is how this could be achieved:

Placement of "Biscuit 0" :

Let's assume the Biscuit starts at position $x = 10$ on the dough roll. The Biscuit has a length of 4 units, so it extends from $x = 10$ to $x = 13$.
During this interval, we consult the 'defects.csv' file to check if there are any 'a,' 'b,' or 'c' class defects within this range. If the number of defects of each class adheres to the defined thresholds, the placement is valid.

Placement Constraints :

The placement of Biscuits must be at integer positions on the dough roll. Biscuits cannot overlap. If a biscuit is placed at 'x=10,' no other biscuit can be placed in the range $x = 11$ to $x = 14$.
Defects within a Biscuit's range must adhere to the defined thresholds for each class.

Calculation of Solution Value :

The value of the solution is the sum of the values of Biscuits placed on the dough roll.
Segments of the dough roll without Biscuits have a value of 0.

4 Mathematical aspect

We can formulate this problem mathematically, we can consider it as a combinatorial optimization problem where we want to maximize the value of biscuits produced from a dough roll of fixed length while considering the constraints related to defects and biscuit sizes.

4.1 Variables

- L: Total length of the dough roll (500 units).
- B_i: Type of biscuit i, where $i \in \{0, 1, 2, 3\}$.
- l_i: Length of biscuit i.
- v_i: Value of biscuit i.
- $d_{i,j}$: Maximum allowed defects of class j for biscuit i, where j is the defect class.
- $x_{i,k}$: Starting position of the k-th biscuit of type i on the dough roll.
- n_i: Number of biscuits of type i placed on the roll.

4.2 Defects

- Defect classes are a, b, c.
- Position and class of each defect are given in 'defects.csv'.

4.3 Objective Function :

Maximize the total value of biscuits produced:

$$\text{Max} \sum_{i=0}^{3} v_i \cdot n_i$$

4.4 Constraints :

1. Biscuits must not overlap:
$$\forall i, \forall k, x_{i,k} + l_i \leq x_{i,k+1}$$
2. Biscuits must fit within the roll:
$$\forall i, \forall k, x_{i,k} + l_i \leq L$$
3. Defects constraints for each biscuit type must be respected:
$$\forall i, \forall k, \text{the number of defects of class } j \text{ within } [x_{i,k}, x_{i,k} + l_i] \leq d_{i,j}$$
4. The sum of all biscuit lengths cannot exceed the total length of the roll:
$$\sum_{i=0}^{3} n_i \cdot l_i \leq L$$

5 Algorithmic resolution

In the context of our study, we divided it into three parts. First, we tested whether our Python implementation correctly validated the constraints. Once this verification was done, we used two algorithms to solve this problem: the greedy algorithm and the genetic algorithm.

5.1 Greedy Algorithm

Definition

- **Nature** : The greedy algorithm is a sequential decision-making method that, at each step, chooses the option that appears to be the best at that moment.
- **Objective** : Find an optimal or near-optimal solution for optimization problems.

Characteristics

- **Local vs. Global** : The algorithm makes local optimal decisions without considering the global state.
- **Simplicity and Efficiency** : Often easy to implement and quick in terms of execution time.
- **No Backtracking** : Once a choice is made, the algorithm does not backtrack on its previous decisions.

Advantages

- **Fast and Efficient** : Can provide solutions quickly for some problems. Easy to Understand and Implement.

Limitations

- **Non-Optimality** : Does not always guarantee the optimal solution.
- **Problem Dependent** : Effective for some problems but inefficient or incorrect for others.

```
procedure GreedyAlgorithm()
1.  S ← ∅;
2.  Evaluate the incremental cost of each element e ∈ E;
3.  while S is not a complete feasible solution do
4.      Select the element s ∈ E with the smallest incremental cost;
5.      S ← S ∪ {s};
6.      Update the incremental cost of each element e ∈ E \ S;
7.  end_while;
8.  return S;
end.
```

Figure 1: Greedy Algorithm

Greedy Algorithm for our study

- Sort the biscuits by value per unit of length: This prioritizes biscuits that offer the best ratio between value and space occupancy.
- Place one biscuit at a time: Try to place the most 'efficient' biscuit (best value/length ratio) each time.
- After each successful placement, recalculate the best biscuit to place next.
- Continue until space is exhausted: Stop the algorithm once no more biscuits can be placed.

Explanation of the *BiscuitPlacementProblem* Class

Purpose

The `BiscuitPlacementProblem` class is designed to solve a problem of placing biscuits on a roll with defects. The goal is to place the biscuits in a way that maximizes total value while considering the length of the roll and the present defects. **Class Structure**

Attributes

- `roll_length`: The length of the roll on which the biscuits are to be placed.
- `defects`: A list of defects on the roll, loaded from a file.
- `placed_biscuits`: A list of biscuits that have been placed on the roll.
- `biscuit_types`: Types of biscuits available for placement.

Methods

- `__init__`: Initializes an instance with the roll length and loads defects from a file.
- `load_defects`: Loads defects from a CSV file and organizes them by class.
- `add_biscuit_type`: Adds a type of biscuit to the list of available biscuits.

- `is_valid_placement`: Checks if the placement of a biscuit is valid.
- `place_biscuit`: Places a biscuit on the roll if the placement is valid.
- `total_value`: Calculates the total value of the biscuits placed on the roll.
- `display_rolls_and_biscuits`: Displays the biscuits placed on the roll and the total value.

Utility of the Class and Methods

- **Complexity Management**: The class encapsulates the complexity of the biscuit placement problem, making the code more organized and maintainable.
- **Reusability**: The methods provide a clear interface for manipulating the biscuit placement problem, allowing the code to be reused in different scenarios.
- **Extensibility**: It's easy to add new types of biscuits or modify the placement rules thanks to the class structure.
- **Testability**: Each method can be tested individually, facilitating debugging and code validation.

Output of our Greedy Algorithm

```
Biscuits placés :
Biscuit 3 à la position 2, Longueur: 5, Valeur: 8
Biscuit 2 à la position 7, Longueur: 2, Valeur: 1
Biscuit 0 à la position 9, Longueur: 4, Valeur: 6
Biscuit 1 à la position 13, Longueur: 8, Valeur: 12
Biscuit 2 à la position 21, Longueur: 2, Valeur: 1
Biscuit 3 à la position 24, Longueur: 5, Valeur: 8
```

Figure 2: Overview of the table of biscuits placed on the roll

The image above illustrates the table of biscuits placed on the roll, displaying the following information for each biscuit after using Greedy Algorithm :

- **Position**: Specifies the exact location on the roll where the biscuit has been placed.
- **Length**: The size of the biscuit, which is critical to consider to avoid overlapping with other biscuits and not to exceed the total length of the roll.
- **Value**: The value assigned to each type of biscuit, reflecting its significance or quality.

The table shows that biscuits are strategically placed at various positions to maximize the use of available space and to avoid defects present on the roll.

```
Valeur totale: 662
```

Figure 3: Accumulated total value

The image above presents the total value achieved after all the biscuits have been placed. This value is the sum of the individual values of each placed biscuit and represents the final outcome that the algorithm sought to maximize. The total value is a key indicator of the algorithm's performance, showing its efficiency in optimizing biscuit placement given the constraints of defects and available space.

662 is a good score because we can prove that the maximum theoretical score to achieve is 750 if we do not take into account the defects of the roll.

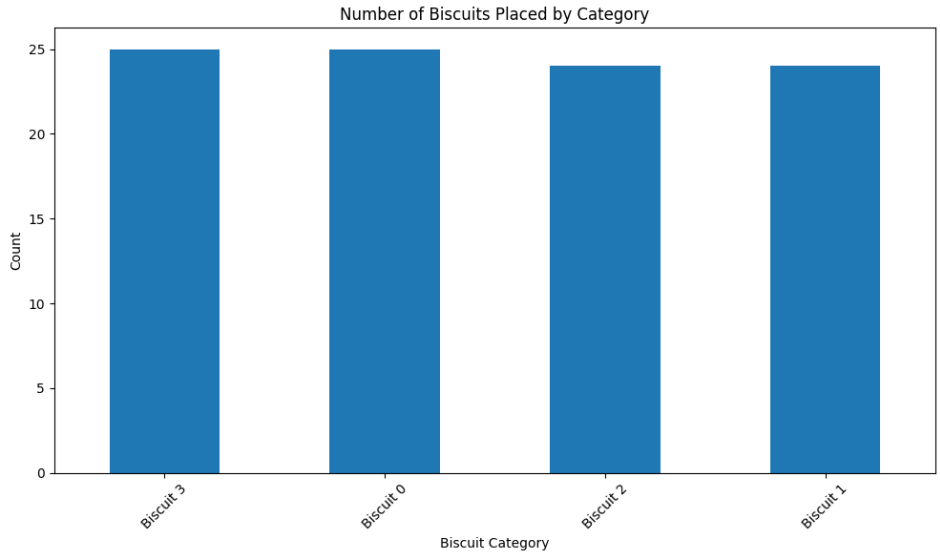

Figure 4: Number of biscuits per category with Greedy Algorithm

We can also visualize the number of biscuits per category with Greedy Algorithm :

- **Local Optima**: The near-equal distribution of biscuits suggests that the Greedy Algorithm consistently found placements for all types of biscuits without prioritizing one type excessively over another. This might indicate that at each step, the choice of which biscuit to place next did not result in a significant difference in local value, leading to a balanced outcome.
- **Value Maximization**: Given the characteristics of the biscuits, the Greedy Algorithm has not necessarily maximized the total value, as Biscuit 1, with the highest individual value, does not have a significantly higher count. This could be due to the fact that the algorithm places biscuits based on immediate value per unit length without forecasting future placements.
- **Algorithm Efficiency**: The Greedy Algorithm's efficiency in this context can be assessed by its ability to place biscuits quickly and without much foresight into future placements. It appears to have performed well in terms of distributing different types of biscuits across the roll length.
- **Implications for Greedy Algorithm**: The even distribution hints that the Greedy Algorithm might not always yield the absolutely most optimized solution in terms of total value but can still provide a satisfactorily balanced solution that meets multiple criteria efficiently.

In conclusion, this histogram indicates that the Greedy Algorithm has produced a balanced distribution of biscuit types across the roll. However, it may not have fully optimized for the total value due to its nature of making decisions based on local optima without considering the overall impact of these decisions.

Mathematical proof

The length of the roll is 500. The biscuit with the highest value has a value of 12 and a length of 8. Theoretically, we can therefore achieve a maximum score of :

$$\boxed{(500/8) \times 12 = 750}$$

5.2 Genetic Algorithm

Problem Definition

- **Population**: A set P of candidate solutions, often referred to as individuals or chromosomes.
- **Fitness Function**: A function $f : P \to R$ that evaluates the quality or fitness of each individual in the population.

Algorithm Process

1. **Initialization**:
   - Generate an initial population P of random solutions.
2. **Evaluation**:
   - Calculate the fitness for each individual in P.
3. **Selection**:
   - Select individuals for reproduction. Individuals with higher fitness are more likely to be chosen.
4. **Crossover**:
   - Randomly pair selected individuals and exchange parts of their structures to create new individuals (offspring).
5. **Mutation**:
   - Introduce random variations in some individuals to maintain genetic diversity.
6. **Replacement**:
   - Replace some individuals from the old population with new individuals to form a new population.
7. **Iteration**:
   - Repeat steps 2 to 6 for several generations until a stopping criterion is reached (e.g., a fixed number of generations or a fitness plateau).

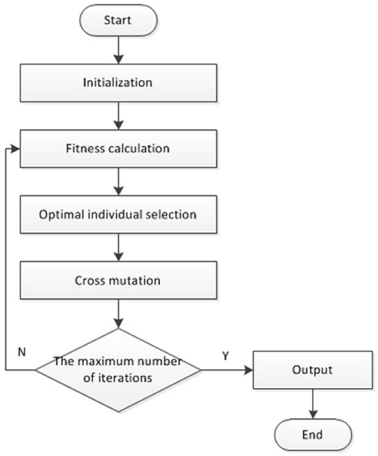

Figure 5: Genetic Algorithm

Example of Applications

- **Function Optimization**: Finding values of variables that maximize or minimize a given function.
- **Search Problems**: Solving complex problems like route optimization, scheduling, or network design.

Key Properties

- **Exploration and Exploitation**: The algorithm balances between exploring new areas of the search space (mutation) and exploiting good solutions already found (crossover).
- **Population Diversity**: Genetic diversity within the population helps avoid local optima and favors the discovery of globally optimal solutions.

Limitations

- **Convergence**: There is no guarantee of finding the optimal solution.
- **Sensitivity to Parameters**: The performance of the algorithm can be heavily influenced by the choice of parameters (mutation rate, population size, etc.).
- **Computational Complexity**: Can be computationally expensive, especially for large populations and many generation cycles.

Genetic Algorithm Implementation for Biscuit Placement

We use the previous ***BiscuitPlacementProblem*** Class for our Genetic Algorithm and here we detail the other methods for this Algorithm :

- `create_random_arrangement`: This helper function generates a random arrangement of biscuits on the roll. It randomly selects a biscuit type and places it if the position is valid, iterating until the end of the roll is reached.
- `fitness`: The fitness function calculates the total value of a given arrangement of biscuits by placing each biscuit on the roll and summing their values.
- `initialize_population`: Initializes the population for the genetic algorithm with a given size by creating many random arrangements.
- `select_fittest`: Selects the top-performing individuals from the population based on their fitness scores to be the parents for the next generation.
- `crossover`: Combines two parent arrangements at a random crossover point to create offspring, incorporating traits from both parents.
- `mutate`: Applies random changes to an individual's arrangement at a specified mutation rate, altering the position of biscuits to introduce variability.
- `genetic_algorithm`: The main function that runs the genetic algorithm. It initializes a population and iteratively applies the selection, crossover, and mutation processes to create new generations, aiming to improve the fitness of the population with each generation.

The image below illustrates the table of biscuits placed on the roll, displaying the following information for each biscuit after using Genetic Algorithm :

- **Position**: Specifies the exact location on the roll where the biscuit has been placed.
- **Length**: The size of the biscuit, which is critical to consider to avoid overlapping with other biscuits and not to exceed the total length of the roll.
- **Value**: The value assigned to each type of biscuit, reflecting its significance or quality.

```
Biscuits placés :
Biscuit 0 à la position 1, Longueur: 4, Valeur: 6
Biscuit 1 à la position 5, Longueur: 8, Valeur: 12
Biscuit 3 à la position 13, Longueur: 5, Valeur: 8
Biscuit 1 à la position 18, Longueur: 8, Valeur: 12
Biscuit 2 à la position 26, Longueur: 2, Valeur: 1
Biscuit 3 à la position 28, Longueur: 5, Valeur: 8
Biscuit 1 à la position 40, Longueur: 8, Valeur: 12
Biscuit 0 à la position 51, Longueur: 4, Valeur: 6
Biscuit 1 à la position 55, Longueur: 8, Valeur: 12
```

Figure 6: Overview of the table of biscuits placed on the roll

```
Valeur totale: 684
```

Figure 7: Overview of the table of biscuits placed on the roll

The image above presents the total value achieved after all the biscuits have been placed. This value is the sum of the individual values of each placed biscuit and represents the final outcome that the algorithm sought to maximize. The total value is a key indicator of the algorithm's performance, showing its efficiency in optimizing biscuit placement given the constraints of defects and available space.

We achieved a score of 684, which is slightly higher than the score obtained with the greedy algorithm (662).

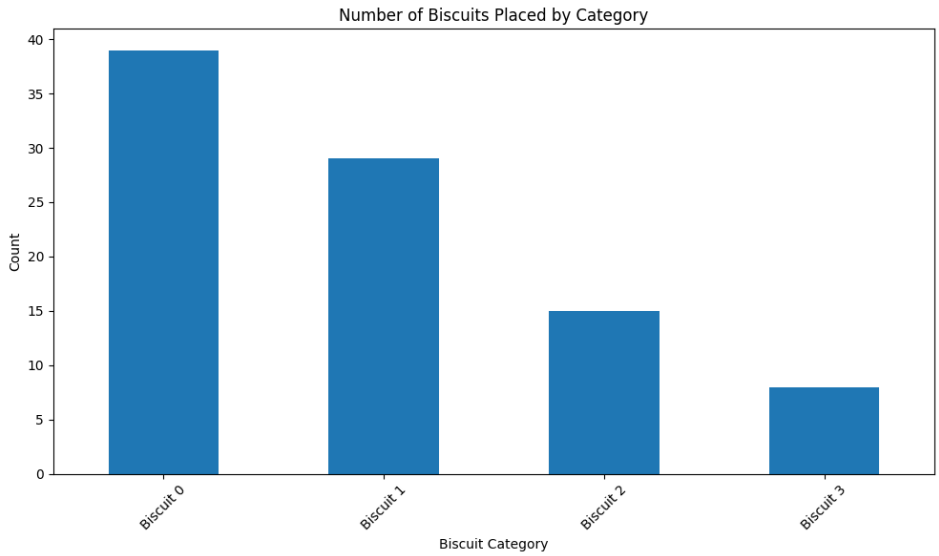

Figure 8: Number of biscuits placed by category with Genetic Algorithm

- **Dominance of Biscuit 0**: The most frequently placed biscuit is Biscuit 0, which has a moderate length and value. Its higher count could be due to its shorter length allowing more of them to fit within the roll or due to a balance between its length, value, and defect constraints.
- **Biscuit 1 Placement**: Biscuit 1, although having the highest value, is not the most frequently placed. This could be because its longer length makes it harder to fit as many on the roll, or the genetic algorithm might be optimizing for other factors as well as value.
- **Lower Counts for Biscuit 2 and 3**: Biscuit 2 and 3 have significantly fewer placements. For Biscuit 3, this might be due to its larger length and defect constraints, even though it has a

high value. Biscuit 2 has the lowest value, which might explain its reduced placement, as the algorithm likely prioritizes higher-value biscuits.

- **Genetic Algorithm Behavior**: Unlike a Greedy Algorithm, which makes decisions based solely on immediate considerations, a genetic algorithm uses crossover and mutation to explore the solution space more broadly. The differences in count could reflect an evolved balance between placing as many biscuits as possible (favoring shorter ones) and maximizing value (favoring longer, more valuable ones), within the constraints of defects and roll length.
- **Evaluation of Genetic Algorithm**: If the genetic algorithm's objective was to maximize the total value, the distribution suggests it may not have fully optimized for value, potentially due to competing constraints or the stochastic nature of genetic algorithms which include random mutations and crossover events that can introduce diversity but may not always improve the fitness function.
- **Potential for Optimization**: Given the genetic algorithm's potential to explore a wide range of solutions, there could be room for further optimization. Adjustments to the algorithm's parameters, such as mutation rate or selection pressure, could potentially lead to a distribution that places more high-value biscuits without violating defect or length constraints.

6 Conclusion and Reflections

Through the implementation and comparison of the Greedy and Genetic algorithms in the context of the Biscuit Placement Problem, we have gained a deeper understanding of optimization strategies and their suitability for different types of problems.

One of the significant challenges encountered was balancing the various constraints—such as the roll length, biscuit dimensions, and defect tolerances—while also striving to maximize the total value of the biscuit arrangement. Addressing this challenge required a thoughtful consideration of the algorithms' decision-making processes and their parameter tuning, which was especially pertinent for the Genetic algorithm with its population size, mutation rates, and selection criteria.

The Greedy algorithm, with its simplicity and speed, provided a solid benchmark and a clear illustration of how local decisions can quickly lead to a good solution. However, the Genetic algorithm, despite its complexity and slower performance, was particularly insightful. It highlighted how an iterative and exploratory approach could potentially yield superior solutions by considering a broader set of possibilities and escaping local optima.

In conclusion, this project has not only bolstered our technical knowledge and skills in algorithm development and data visualization but has also enhanced our problem-solving abilities. It underscored the importance of matching the algorithm to the problem's nature and the value of persistence and creativity in overcoming computational challenges. These insights will undoubtedly be of great benefit in our future endeavors in the field of computer science and analytics.