Oscar Pastural
Richard Goudelin
Capucine Boudin
DIA2

Artificial Intelligence Algorithms - MESIIN476023

Table des matières

Problem Description

The problem we're trying to solve is how to make the most profitable biscuits using a single roll of dough, considering defects and different biscuit specifications. Here are the specific aspects and challenges of the problem:

Maximizing Space and Value: The dough is a limited resource that we want to use efficiently. We need to place biscuits of different sizes and values on the dough in a way that maximizes the total value without exceeding the length of the dough. This involves careful calculations to determine the arrangement to make the best use of the available dough.

Considering Defects: Defects in the dough can affect the quality and salability of the biscuits. Each biscuit has a tolerance for specific types and amounts of defects. When placing a biscuit on a section of the dough, we need to take into account the defects in that section and ensure they don't exceed the biscuit's tolerance for defects.

Non-Overlap and Integer Positions: Biscuits cannot overlap with each other, and their positions along the length of the dough need to be whole numbers (integers). Finding an optimal arrangement that satisfies these constraints is challenging, especially as the number of possible positions increases with the length of the dough.

Python Implementation Steps

First, we decided to use OOP for various reasons including encapsulation. Indeed, OOP allows us to encapsulate related behaviors within classes. By creating separate classes for Biscuit and Dough, we can manage their respective properties and functionalities in a structured manner. This makes it easier to understand and maintain the codebase.

Class Definitions and Data Structures: As previously introduced, **we created classes for Biscuit and Dough** to organize information and actions:

- The Biscuit class represents different types of biscuits with specific characteristics, such as length, value, and maximum allowed defects.

- The Dough class represents the dough on which biscuits are placed, considering defects and constraints. The Biscuit class initializes biscuit properties based on its type, and it provides methods to retrieve information about the biscuit, such as its type, length, value, and maximum allowed defects. This class encapsulates the details of each biscuit type and facilitates easy access to its attributes. On the other hand, the Dough class represents the dough's properties, including its length, a list of defects, and a list of placed biscuits. It allows the addition of defects, placement of biscuits, and validation of biscuit placement against constraints. It serves as a instance of our problem, and helps us visualize it.

Reading Defects and Initialization: We use a tool called pandas to read a file named 'defects.csv' that contains defects data: their positions and types. This information is used to populate the defects in the Dough class. This step involves processing the file and creating a list of defects along with their positions and types.

Calculating Total Value: After the biscuits are placed, a function of the Dough class calculates the overall value of the dough roll. The calculation is based on the successfully placed biscuits and their respective values.

Proposed solutions

Dynamic programming as a benchmark

The commons algorithms do not apply here, but **dynamic programming** is suitable for tackling this problem. Our initial goal was to establish an optimal solution that could serve as a benchmark for evaluating other approaches. The dynamic programming approach boasts a time complexity of **O(n)** and a memory complexity of **O(n^2)**, which is highly efficient.

Here's the core concept: At each position i, denoted as **V(i)**, we aim to determine the maximum attainable value by considering all possible biscuit placements up to that point. This entails calculating the highest value achievable by inserting a biscuit of length **L** at the current position and adding it to **the best value** at the position **i–L**.

Having applied this method, we identified an optimal solution with a score of **760**. This result now **provides a basis** for comparison with subsequent approaches.

Genetic Algorithm (GA)

We first decided to use Genetic Algorithm. Indeed, these algorithms are very effective solutions for maximizing profitability which is perfect for our problem.

In fact, considering we have a single dough, we need to optimize limited resources. GA can find the best arrangement of biscuits, making the most of the available. We also need to handle defects which are possible with GA. By accounting for the specific types and amounts of defects that biscuits can tolerate, it can place biscuits in sections of the dough that minimize the impact of defects.

Moreover, the problem involves constraints like non-overlapping biscuits and the need for integer positions along the dough's length. Through crossover and mutation operations, GA made exploration of different combinations of possible solutions until the best was found. This allows for the discovery of optimal arrangements that satisfy all requirements.

By using this algorithm, we managed to maximize our profitability. With this approach, we have gotten a total value of around 530.

Here is a explanatory of our Genetic Algorithm Class:

Function	**Role**	***Time complexity***	***Space complexity***
initialize_population()	A population of solutions is randomly generated, representing different arrangements of biscuits on the dough.	*O(population_size)*	*O(population_size)*
random_solution()	Solutions are selected based on their fitness, with a bias towards higher-value arrangements.	*O(dough.length)*	*O(1)*
fitness()	Calculates the maximum fitness by summing up the fitness values of all individuals in a solution. Randomly selects a value and iterates through the population	*O(len(individual))*	*O(1)*
selection()	The selection function uses a roulette wheel method to choose an individual from the population. It calculates the highest fitness by adding up the fitness values of all individuals. Then, it randomly picks a value between 0 and the highest fitness and goes through the population, adding the fitness of each individual to a running total until it surpasses the chosen value.	*O(population_size)*	*O(1)*
crossover()	Pairs of solutions exchange information to create new solutions, mimicking genetic crossover with parents and children (we use single point crossover in this one).	*O(min(len(parent1), len(parent2)))*	*O(len(parent1) + len(parent2))*
mutate()	Random changes are introduced to some solutions to maintain diversity.	*O(len(individual))*	*O(1)*
evolve()	Creates a new population by choosing parents, combining their genetic material through crossover, and introducing random changes through mutation. The new population is created by adding the resulting offspring. We check the population size to keep it constant. At each iteration the population is updated.	*O(population_size)*	*O(population_size)*
Main (not inside the Class but used to compile the GA algorithm)	Initializes the dough and biscuits objects, creates a GeneticAlgorithm instance, initializes the population, and runs the evolution loop for a specified number of generations. After the evolution loop, it finds the best solution in the final population based on fitness and prints the total value and the best solution.	*O(generation * population_size)*	*O(population_size)*

Genetic Algorithm (GA) with Elitism Roulette

In our first Genetic Algorithm, each generation is replaced entirely by the offspring. In this second approach, we Implement an elitism by selecting a certain number of top individuals from the current population to survive unchanged into the next generation.

Therefore, we keep the same functions as before, but we implement elitism in our evolution strategy. This allows the algorithm to run faster, even though the top result is often the same as our first GA algorithm. As it is written from the GA Class (GeneticElitism(GeneticAlgorithm)), only one functions has changed:

Function	Role	*Time complexity*	*Space complexity*
evolve()	Ensures that the best solutions from the current population have a higher chance of being carried over to subsequent generations.	*O(population_size)*	*O(population_size)*

With this approach, we got a result around 540.

Genetic Algorithm with Tournament and One Point Crossover

This Genetic Algorithm we used provided several improvements compared to the previous ones. As the last one, elitism is implemented that is to say we preserve the best individuals, ensuring that they are directly carried over to the next generation without changes. This helps maintain high-quality genetic material throughout the evolution process. But that's not it, we also implemented a tournament selection. It is used to select individuals for the new generation. This approach introduces diversity by randomly selecting individuals and choosing the best among them. Finally, the mutation operation is applied to the offspring population, which helps introduce variability and explore different areas of the solution space.

As it is a sub-class from GeneticAlgorithm class, these two functions are overwritten:

Function	Role	*Time Complexity*	*Space Complexity*
evolve()	The best individuals from the current population are preserved and directly carried over to the next generation without any changes.	*O(population_size)*	*O(population_size)*
Selection()	Combines elitism and tournament selection. The top individuals (elites) are selected based on their fitness, and for the remaining slots in the new generation, tournament selection is used. Tournament selection randomly chooses a subset of individuals and selects the best one as a parent.	*O(population_size)*	*O(population_size)*

With this approach, we got higher total value, around 555.

Genetic Algorithm with Tournament and Uniform Crossover

By replacing the One Point Crossover by a Uniform Crossover, which means that instead of taking the half of two parents to make a child, we take one value of each alternating through the Dough, we have performed the best we had and got a higher value of 600.

Genetic Algorithm Conclusion

Overall, the uniform crossover coupled with tournament and elite selection appears to strike a better balance between exploration and exploitation and produces the best results.

First, we can see that implementing elitism, present in the second and third algorithms, helps in preserving the best solutions and accelerating convergence.

The second algorithm was an improvement, but the crossover and mutation operators remain basic, potentially limiting exploration capabilities. Finally, the last genetic algorithm includes a more advanced selection mechanism that produces more accurate results.

To conclude on the genetic algorithm aspect, the trade-off between exploration and exploitation is required for effective optimization. In our case, the fourth one is the second faster and the better algorithm.

Greedy Search

We decided to implement a greedy search in three different ways. Greedy search algorithms are often straightforward to implement and understand compared to more complex optimization techniques. They also have lower time complexity compared to more sophisticated optimization algorithms. They make quick local decisions without extensive computation or exhaustive search.

We tested this algorithm with four different approaches:

Function	Role	Time Complexity	Space Complexity
greedy_biscuit_placement()	The function takes the dough object and a dictionary of biscuits as input. It sorts the biscuits in descending order based on their value per unit length. Then it iterates over the sorted biscuits and tries to place them on the dough while considering the remaining space and defect requirements. The function keeps track of the total value of placed biscuits and the positions of the placed biscuits. It returns the total value and a list of placed biscuits.	O(n log n + dough.LENGTH)	O(n + dough.LENGTH)

First Heuristic: Value

We tested a first heuristic based on the value only which means the algorithm selects the biscuits with the highest value first. That can be done by sorting the biscuits based on their absolute value in descending order. With this approach, we got a total value of **658**.

Second Heuristic: Value / Length

The second heuristics is based on the best value of a biscuit in relation to their length the algorithm prioritizes biscuits with the highest value relative to their length. That can be done by sorting the biscuits in consideration of their value/length results in descending order. With this approach, we got a solution of **663**.

*Third Heuristic: Value / (Length*Max number of defects)*

The third heuristic emphasizes biscuits with high value relative to both length and the maximum number of defects allowed. That can be done by sorting the biscuits based on the value-to-(length * max defects) ratio in descending order. With this approach, we got a solution of **674**.

Greedy Search Conclusion

Overall, when comparing the three heuristics, we can conclude that the third one tends to have higher quality solution. However, the performance of these algorithms depends on the specific characteristics of the input data and the problem requirements. In our case, the third heuristic would suit best.

Constraint Satisfaction Problem

We decided to end our project by using a **constraint satisfaction problem algorithm** because it is perfectly suited for our problem. Indeed, this approach allows maximization of space and value.

CSPs are designed to handle problems with multiple constraints, and in this case, the goal is to maximize the total value of biscuits while respecting the limited resource (dough). CSPs are also able to manage defects because they excel at incorporating complex constraints into the problem definition.

The **tolerance** for defects in each biscuit can be easily represented as constraints in the model. What also helped us choosing CSP is the requirement that biscuits cannot overlap and must be placed at integer positions. In fact, it matches with the natural representation and handling of constraints in CSPs. Overall, the biscuit placement problem shows concrete characteristics that make it suitable for a CSP approach.

Piece of code	**Role**	***Time Complexity***	***Space Complexity***
(ii) Create variables	Initializes the CP model and creates a list of biscuit objects (biscuit_types). The list comprehension creates biscuit objects with unique identifiers. Creates boolean decision variables (biscuit_vars) representing whether a biscuit is placed at a specific position.	*O(Number of Biscuit* maximum possible starting position)*	*O(Number of Biscuit* maximum possible starting position)*

(iii) Create Constraints No overlapping biscuits	Adds constraints to the model to ensure that no two biscuits overlap on the dough	*O(Number of Biscuit* (maximum possible starting position)^2)*	*O(Number of Biscuit* maximum possible starting position)*
(iii) Create Constraints Defects within acceptable limits	Ensures that the number of defects in each biscuit's placement is within acceptable limits.	*O(Number of Biscuit* (maximum possible starting position)^2)*	*O(Number of Biscuit* maximum possible starting position)*
(iv) Objective function to maximize	Defines the objective function to maximize the total value of the biscuits placed on the dough.	*O(Number of Biscuit* (maximum possible starting position)^2 * Length)*	*O(Number of Biscuit* maximum possible starting position)*
(v) Solve the mode	Initializes the CP solver, solves the model, and prints the time taken for the solution.	*O(Number of Biscuit* (maximum possible starting position)^2 * Length)*	*O(Number of Biscuit* maximum possible starting position)*

This approach is the algorithm that gives us the highest value quality. In fact, we get a total value around **760**. Its time and space complexity may be higher compared to heuristic approaches like greedy search, but it guarantees the satisfaction of all specified constraints during the optimization process.

Here is a recap of our different approaches:

Method	**Mesured Time**	**Time Complexity**	**Space Complexity**	**Score**
Best Genetic Algorithm	3 minutes	O(population_size·(population_size· number of biscuits in dough +max_attempts· number of biscuits in dough)) / O(population_size·number of biscuits in dough)		600
Greedy Search	2 seconds	O(4*log*4+2000) / O(500)		674
CSP Ortools librairy	0.15 seconds	O(Number of Biscuit* (maximum possible starting position)^2 * Length) / O(Number of Biscuit* maximum possible starting position)		760 (optimum)
Dynamic programming	0.01 seconds	O(Length of dough) /O(Length of dougth^2)		760 (optimum)

Conclusion and Reflections

This project helped us learn about solving problems more effectively and using programming skills to make real solutions. It highlights the crucial role of method selection in finding effective solutions.

Addressing real-world problems involves navigating various challenges, such as optimizing for both speed and quality while managing multiple constraints. In addition to the genetic algorithm, the project allowed us to explore the application of the greedy search and constraint satisfaction problem (CSP). The greedy search emphasizes making locally optimal choices at each step, while the CSP provides a systematic approach to handle constraints. These methods, alongside the genetic algorithm, showcase different problem-solving strategies with different advantages and disadvantages.

For this problem with this data dynamic programming is in fact the best approach but in others conditions with a big memory complexity it might not be.

Reflecting on the project, it becomes evident that the careful choice of the solving method significantly influences the development process and the final outcome that we can get. Therefore, having the skills to understand and choose the right methods is essential for successful problem-solving.

Thank you for reading.