Optimization project: Biscuit Optimizer

Roberto Basla

Politecnico di Milano
School Of Industrial and Information Engineering
Computer Science and Engineering
A.Y. 2021/2022

Contents

1 Introduction

This document reports the project developed for the Optimization course at Politecnico di Milano. Input data, models and solutions are available at the GitHub repository https://github.com/rb-sl/biscuit_optimizer.

1.1 The problem

The problem consists in choosing how to place cookie cutters on some available dough in order to maximize the total value given by each shape. Moreover, the most frequent shape must be selected no more than 3 times the most infrequent to avoid selecting too many biscuits of one type only.

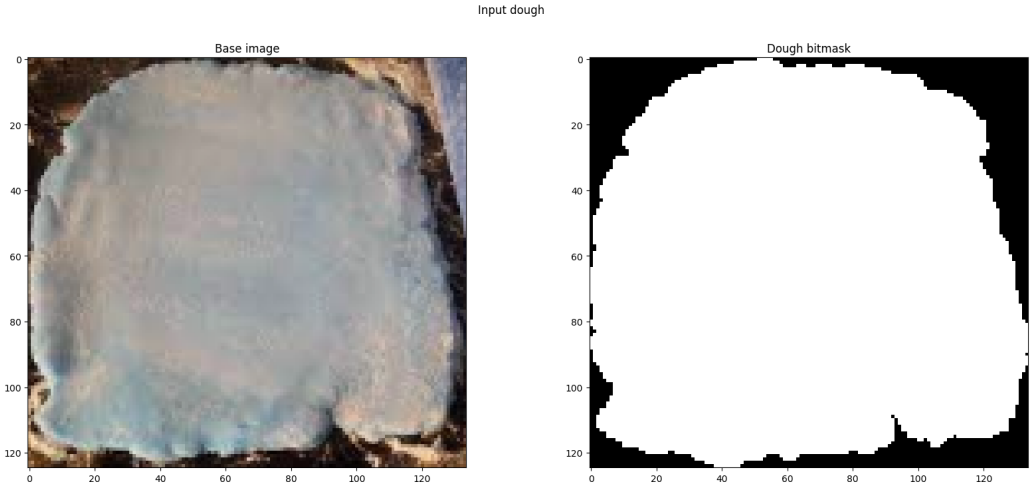
This is a case of nesting problem, i.e. a two-dimensional cutting and packing problem involving irregular shapes; however, differently from classical formulations that define the elements to be placed as polygons, this project tackles the problem by considering images and binary masks over their pixels. The input data is preprocessed as follows.

1.1.1 Available dough

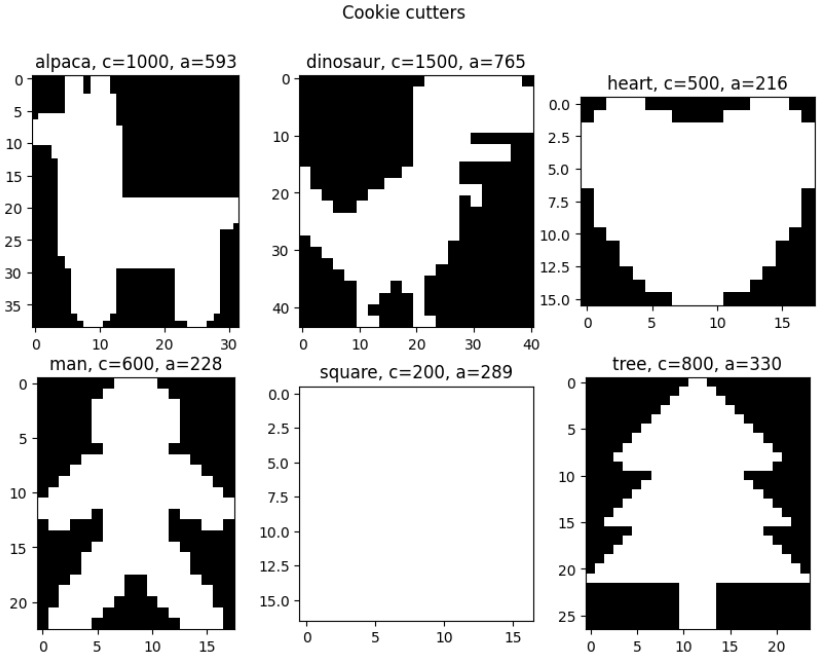
The available dough is represented by an image $P \in \mathbb{R}^{n \times m \times 3}$, where n and m are respectively the number of rows and columns, while 3 refers to the color channels. The photo used for this project (source) was cropped and subsampled (from the initial $n = 180$, $m = 192$ to $n = 125$ and $m = 134$) in order to simplify the problem. From this image, a binary mask encoding the usable area was manually drawn, thus obtaining the Dough bitmask $B \in \{0, 1\}^{n \times m}$ in Figure 1a.

1.1.2 Cookie cutters

Cutter masks were hand drawn and assigned arbitrary values as per Figure 1b; they will be the cookie cutters that need to be overlayed to the input image to "cut" it and obtain the final biscuits. These bitmasks will be referenced as b_i, while each pixel corresponds to $b_{i,h,k} \quad \forall h \in R_i, \forall k \in C_i$, where R_i and C_i represent respectively the set of rows and the set of columns of bitmask i.



(a) Input dough and its binary mask (134×125 pixels)



(b) Cutters' binary masks

1.2 Technology

This project was developed with both AMPL and Jupyter Notebooks using the following technologies:

- AMPL with CPLEX solver
- Python with Google's OR-Tools optimization suite and CP-SAT solver
- Python with Gurobi's API and solver
- Pure Python for heuristics

All algorithms were run on a desktop setup with an Intel Core i7-6700 processor and 16GB DDR3 RAM; Python 3.9.12 was installed inside a Windows 10 Anaconda Environment. Only the nesting model with OR-Tools was run remotely on the Kaggle platform.

2 Modeling

This section covers modeling approaches to the problem using AMPL, Google OR-Tools and Gurobi.

2.1 Integer knapsack model

The first approach consists in a relaxation of the problem that doesn't consider the placement of cutters on the dough: in this case only areas are considered, and the input image's mask B is reduced to the number of available pixels (namely, $B = 13929$). Additional parameters are the cutter values $\underline{c}$ and cutter areas $\underline{a}$, while the integer vector $\underline{x}$ represents the number of times each cutter is selected (and is therefore the problem's variable).

This simpler problem consists in an integer knapsack that can be modeled as follows:

$$\max \quad \sum_{i \in I} c_i x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in I} a_i x_i \leq B \tag{2}$$

$$x_i \leq 3x_j \quad \forall i \in I, \forall j \in I \tag{3}$$

$$x_i \geq 0 \quad \forall i \in I \tag{4}$$

It maximizes the biscuit value multiplied by the number of times the cutter is used (1), subject to the knapsack's budget constraint (2) and the frequency constraints (3). Obviously, cutters must be selected a non-negative number of times (4).

As with any relaxation, the solution value is expected to be an upper bound for the complete problem.

2.1.1 Results

The model was solved very quickly by all three softwares, achieving a final value of 29000 and a dough usage of 99.94% of its pixels with the solution

$$\underline{x} = \begin{bmatrix} 4 \\ 4 \\ 4 \\ 11 \\ 4 \\ 12 \end{bmatrix}$$

Figure 2 shows the types of biscuits resulting from the optimization (39 in total). Models and results are available in the GitHub repository rb-sl/biscuit_optimizer/src/01-knapsack.

Figure 2: Visualization of biscuits resulting from the knapsack problem

2.2 Nesting model

The complete nesting model takes into account the initial mask and the cutter masks (so that they do not overlap). The solution is achieved through a set of binary variables y over sets I (the set of cutters), N (the set of rows in the dough mask) and M (the set of columns in the dough mask). $y_{i,n,m}$ is equal to 1 if a cutter i is placed on the dough mask such that its top left corner is on the (n, m) pixel in the mask and 0 otherwise.

In addition to sets and parameters already defined, the additional sets IN_i and OUT_i need to be defined as

$$\text{IN}_i = \{(n, m) : n \in N, m \in M, n + R_{i,max} \leq N_{max} \wedge m + C_{i,max} \leq M_{max}\}$$
$$\text{OUT}_i = \{(n, m) : n \in N, m \in M, n + R_{i,max} > N_{max} \vee m + C_{i,max} > M_{max}\}$$

Naturally, it follows from the definition that $\text{IN}_i \cap \text{OUT}_i = \emptyset$ and $\text{IN}_i \cup \text{OUT}_i = B$, where B represents the dough bitmask. These sets need the additional parameters:

- N_{max}: number of rows of the dough mask
- M_{max}: number of columns of the dough mask
- $R_{i,max}$: number of rows of cutter i

- $C_{i,max}$: number of columns of cutter i

Figure 3 shows an example of these sets and parameters for $i = 1$.

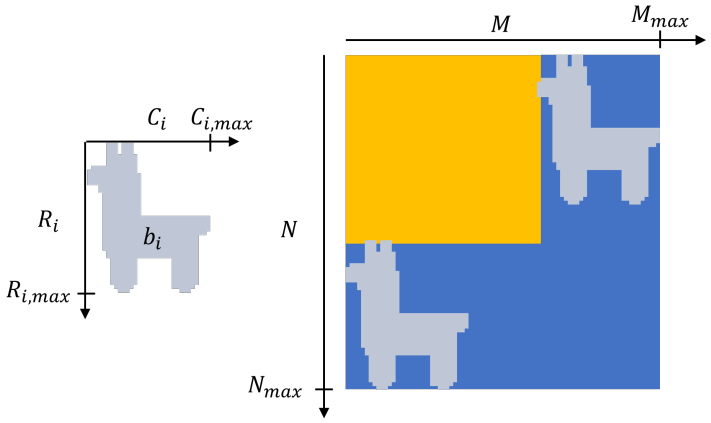


Figure 3: Examples of sets IN_i (yellow) and OUT_i (blue) for $i = 1$

Finally, in order to help the formulation of the non-overlapping constraints, the integer variable z is introduced over I, N and M. $z_{i,n,m}$ represents the number of cutter masks i that cover pixel (n, m) (obviously, only the active part of a cutter mask influences z). Therefore, the complete nesting model is:

$$\max \sum_{i \in I} c_i \sum_{n \in N} \sum_{m \in M} y_{i,n,m} \tag{5}$$

$$\text{s.t.} \quad y_{i,n,m} = 0 \qquad \forall i \in I, \forall (n,m) \in \mathrm{OUT}_i \tag{6}$$

$$y_{i,n,m} \leq B_{n+h-1,m+k-1} + (1 - b_{i,h,k}) \quad \forall i \in I, \forall (n,m) \in \mathrm{IN}_i, \; \forall h \in R_i, \forall k \in C_i \tag{7}$$

$$z_{i,n,m} = \sum_{h \in R_i} \sum_{k \in C_i} y_{i,n-h+1,m-k+1} b_{i,h,k} \quad \forall i \in I, \forall n \in N, \forall m \in M \tag{8}$$

$$\sum_{i \in I} z_{i,n,m} \leq 1 \qquad \forall n \in N, \forall m \in M \tag{9}$$

$$\sum_{n \in N} \sum_{m \in M} y_{i,n,m} \leq 3 \sum_{n \in N} \sum_{m \in M} y_{j,n,m} \qquad \forall i \in I, \forall j \in I \tag{10}$$

$$y_{i,n,m} \in \{0, 1\} \qquad \forall i \in I, \forall n \in N, \forall m \in M \tag{11}$$

$$z_{i,n,m} \geq 0 \qquad \forall i \in I, \forall n \in N, \forall m \in M \tag{12}$$

The objective function (5) is equivalent to the knapsack objective (1) (as $\sum_{n \in N} \sum_{m \in M} y_{i,n,m} = x_i \quad \forall i \in I$) and thus maximizes the overall value of chosen cutters.

Constraint set (6) enforces that the optimizer cannot place cutters that would result in partially cut-out biscuits due to the dough mask not being able to host them whole (thus operates on OUT_i); on the other hand, constraint set (7) operates on IN_i sets and enforces that a $y_{i,n,m}$ may be selected only if the active pixels of cutter mask i on (n, m) fall on active pixels of the dough mask starting from (n, m) up to $(n + R_i, m + C_i)$; it consists in the linearization of an AND operation.

The set of linking constraints (8) defines z in function of y so that the value of a $y_{i,n-h+1,m-k+1}$ (i.e., in the range defined by the cutter mask's dimension) is added to $z_{i,n,m}$ only if the cutter bitmask $b_{i,h,k}$ is active. Constraints (9) enforce the non-overlapping property of the model by limiting the sum of active masks at each pixel (n, m) in the dough mask.

Finally, constraints (10) enforce the maximum difference between the most frequently chosen cutter and the most infrequent. (11) and (12) define the variables.

2.2.1 Results

Due to the NP-Hard nature of the nesting problem, all solvers needed to be limited in time: the execution limit was set to 40000 seconds (some minutes less than 12 hours) for all instances. This decision may be the cause of the three optimizers finding different (locally optimal) solutions, unlike in the knapsack case where they were able to solve the problem exactly. It should be also noted that Python solutions added a small preprocessing, effectively merging constraints (6) and (7) in the form

```
for i in I:
  for n in N:
    for m in M:
      if not can_host(dough_mask, cutter_mask, n, m):
        model.addConstr(y[n, m, i] == 0)
```

Effectively reducing the number of constraints by checking beforehand with the `can_host` function if a cutter i could be placed in (n, m) and, if not, just forcing $y_{i,n,m}$ to 0.

Models and results are available in the GitHub repository rb-sl/biscuit_optimizer/src/02-nesting.

AMPL solution The solution found by the CPLEX optimizer over the model defined in AMPL reached the objective of 16300 and is displayed in Figures 5a and 5b.

OR-Tools solution The CP-SAT optimizer solving the model defined in Google OR-Tools found a solution with value 17800, shown in Figures 5c and 5d. Thanks to the possibility of adding callbacks every time a new solution is found, for this optimizer Figure 4 shows the plot of time against the best solution found.

Gurobi solution Gurobi was able to reach the objective value of 17800 (using the same cutters) as well, but with a different configuration as shown in Figures 5e and 5f.

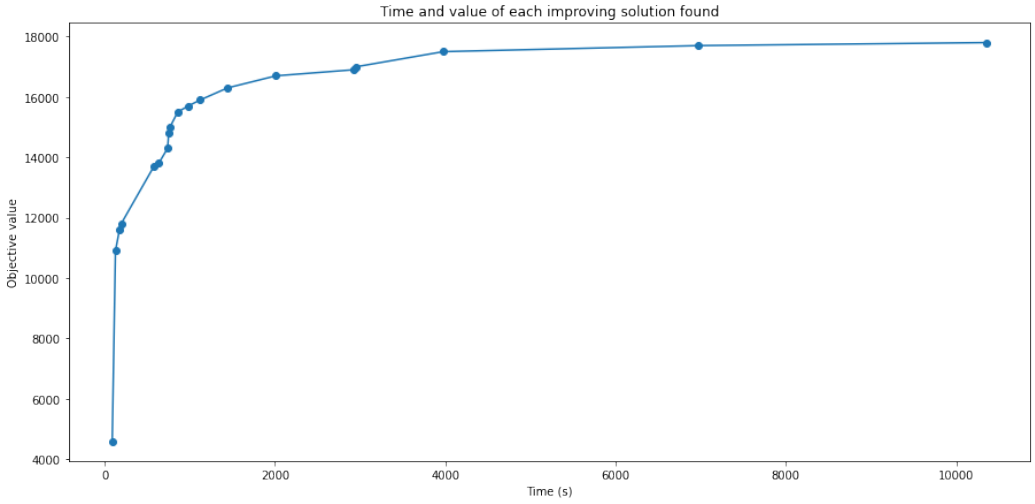


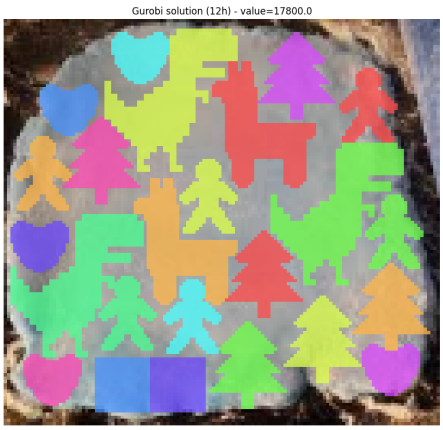
Figure 4: Plot of the best current solution found by CP-SAT against time

(a) AMPL/CPLEX solution

(b) AMPL biscuits result

(c) OR-Tools/CP-SAT solution

(d) OR-Tools biscuits result



(e) Gurobi solution

(f) Gurobi biscuits result

Figure 5: Nesting results

3 Heuristic solutions

As the solvers were not able to find a solution in a reasonable time, some heuristics were devised to tackle the problem. Python notebooks ot the experiments are available in the GitHub repository rb-sl/biscuit_optimizer/src/03-heuristics.

3.1 Greedy heuristics

The first set of heuristics tries to solve the nesting problem greedily. In order to do so, a series of cost map encoding the cost of activating a variable `y[i, n, m]` (i.e., placing a cutter of type i at coordinates (n, m)) was defined:

- Value-based cost map: assigns the cost $c_{i,n,m} = -v_i$ (i.e. the negative value of cutter i) to each `y[i, n, m]` independently of the position (n, m)
- Area-based cost map: assigns the cost $c_{i,n,m} = a_i$ (i.e. the area consumption of cutter i) to each `y[i, n, m]` independently of (n, m)
- Occupancy-based cost map: assigns to each `y[i, n, m]` a cost proportional to the number of `y` variables that would be made infeasible by activating `y[i, n, m]` at the first step, $c_{i,n,m} = \text{len}(o_{i,n,m})$ with o representing the list of ys made infeasible by each (i, n, m)
- Composition of Value and Area cost maps: $c_{i,n,m} = -\frac{v_i}{a_i}$
- Composition of Value and Occupancy cost maps: $c_{i,n,m} = -\frac{v_i}{o_{i,n,m}+1}$
- Composition of Area and Occupancy cost maps: $c_{i,n,m} = (o_{i,n,m}+1) \cdot a_i$

These cost maps define the quantities to be minimized by the greedy procedures, while the actual results refer to the sum of biscuit values.

3.1.1 Results

Heuristics were run 100 times to find average results for each strategy. Figure 6 shows their distribution: only strategies involving the occupancy were consistent in their results, as that was the only cost map that defined values for (i, n, m) instead of for i only. Other heuristics had intrinsic randomness, either explicitly defined (as in the case of greedy/random) or implicitly (e.g., the area case would be greedy only in choosing the cutter and random in selecting the position).

Table 1 reports average values and time for the simulations; it introduces a completely random baseline to compare the results. In general, results were worse than the ones provided by complete nesting models; in particular, heuristics including areas performed worse than the other cases (area-based

only was even worst than the random baseline). The best performing strategy (greedy combining value and occupancy) is kept for the next section.

Heuristic	Average value	Time for 100 simulations (s)
Random (baseline)	9151	277.8
Greedy/Random	13489	331.79
Greedy (value)	13249	334.69
Greedy (area)	9000	260.93
Greedy (occupancy)	15600	329.51
Greedy (value+area)	9555	258.44
Greedy (value+occupancy)	16900	353.11
Greedy (area+occupancy)	15600	378.72

Table 1: Heuristics results

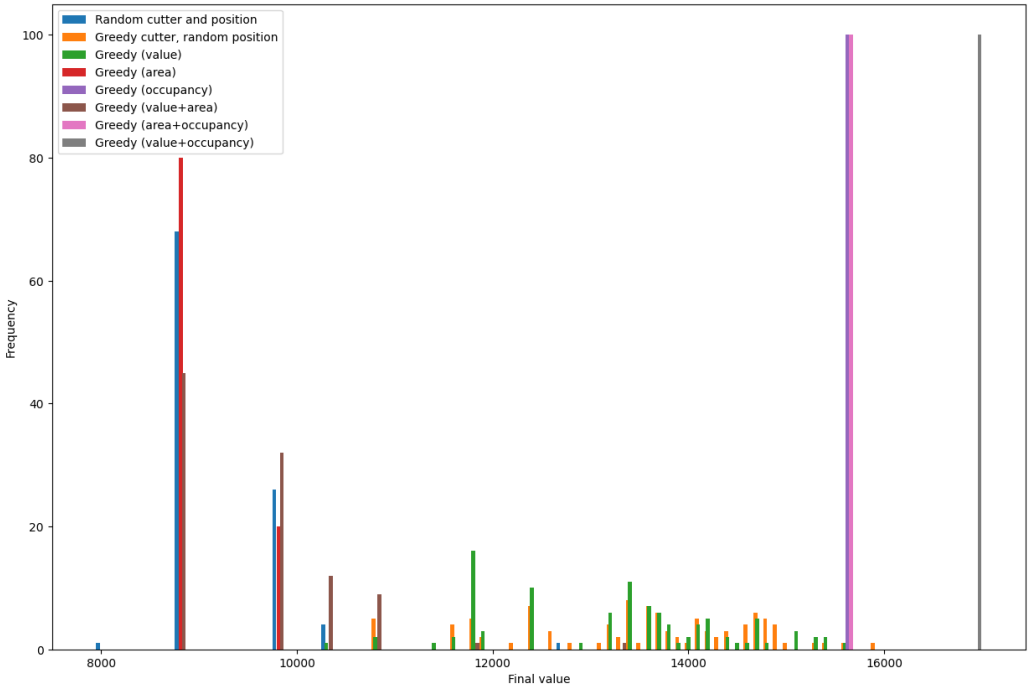


Figure 6: Distribution of heuristic results over 100 simulations for each strategy

3.2 GRASP

The last class of implemented heuristics is Greedy Randomized Adaptive Search Procedures as described by Resende and Ribeiro[2]. It is composed

of two main phases that are iterated a number of times.

Greedy Randomized Construction This phase builds a solution by first creating a Restricted Candidate List (RCL) containing elements with the smallest incremental cost as defined by the cost map. Elements $e = (i, n, m)$ are chosen to be part of the RCL if their cost $c(e)$ is

$$c(e) \in [c^{min}, c^{min} + \alpha(c^{max} - c^{min})]$$

For some choice of α. The next element is chosen randomly from the RCL.

Local search This phase searches the neighborhood of the solution found by the previous phase as in Hill Climbing. The neighborhoods are built according to the scheme defined by Delorme et al.[1] using k-p exchanges, which consist in varying k variables from 1 to 0 and p variables from 0 to 1. In particular were used 0-1, 1-1, 1-2 and 2-1 exchanges.

While previous heuristics encoded solutions as human-readable Python dictionaries, results are now 3D NumPy arrays of shape $n_{cutters} \times n_{rows} \times n_{columns}$ that correspond to variables y in Section 2.2:

$$y_{i,n,m} = \begin{cases} 1 & \text{if the solution has a cutter } i \text{ in position } (n, m) \\ 0 & \text{otherwise} \end{cases}$$

With this definition, the symmetric distance becomes just

$$\Delta(s, g) = \sum_{i \in I} \sum_{n \in N} \sum_{m \in M} |s_{i,n,m} - g_{i,n,m}|$$

i.e. the number of different elements between the starting solution s and the guiding solution g.

This project's implementation made some changes with respect to the paper's description, namely:

- The Greedy Randomized Construction phase doesn't stop in order to avoid impeding the construction of a feasible solution with the remaining cutters for performance purposes: this change improved the construction time to 2s from 21s (on average) without impacting the quality of solutions
- The Greedy Randomized Construction phase doesn't build infeasible solutions, so the repair procedure is not needed

- The Local Search uses two additional parameters limiting the maximum total time and the maximum time between improvements, as the neighborhood exploration is the slowest phase (even if it can be faster when starting from a good solution[1])

3.2.1 Path Relinking

Path Relinking aims at exploring trajectories connecting solutions obtained by the GRASP procedure. It keeps an elite pool of the best solutions found and relinks them to new GRASP solutions, potentially obtaining better results. The choice of which (initial) solution is relinked to the other (guiding) determines the type of Path Relinking. This project implements the following alternatives, as described in [2]:

- Forward Path Relinking: the GRASP output is the initial solution while the guiding solution is chosen from the elite pool according to its distance
- Backward Path Relinking: the roles of initial and guiding solutions are reversed with respect to the previous case
- Mixed Path Relinking: the origin of the two solution is changed at each iteration
- Evolutionary Path Relinking: every e steps of any of the previous variants the solutions in the elite pool are relinked among themselves

The only changes with respect to the algorithm description in the paper were the addition of a tabu list to the local search phase containing the initial and guiding solutions (otherwise many solutions tend to fall back into local optima found by GRASP), and limiting the minimum distance of new elite solutions also when the pool is not full.

3.2.2 Results

GRASP and its Path Relinking variant were able to consistently reach and exceed the values obtained by nesting models of section 2.2. Table 2 contains the best results for each of the implemented strategies.

GRASP GRASP heuristics were run 100 times to assess the average value of solutions, which is close to that of the best heuristic in Section 3.1 (being 16834, obtained in 17315.07s). However, it was able to reach higher values as shown in Figure 8a, and considering that GRASP's result are actually given by the best one among a number of iterations (instead of just one as in the test), it is able to consistently outperform the previously considered heuristics. An example using 5 iterations is reported in Table 2.

Path Relinking Path Relinking proved to be a good enhancement: the following output shows an example where solutions with values 16900 and 17900 generated a solution with value 17400 (the best one along the path), which led to the best solution after local search.

```
PATH RELINKING: 6/10
GRASP: 1/3
Starting local search from solution v=15900
Best solution after local search (5 improvements / 13095 checked): v=16900
GRASP: 2/3
Starting local search from solution v=15300
Best solution after local search (5 improvements / 19186 checked): v=16100
GRASP: 3/3
Starting local search from solution v=15400
Best solution after local search (4 improvements / 18033 checked): v=16400
>>> [17900, 17500, 17900, 17200, 17900]
Starting relinking of solution v=16900 to v=17900 (distance = 40.0)
Best relinking solution in path (40 steps / 328 checks): v=17400 at step=39
Best solution after local search (2 improvements / 2335 checked): v=18300
```

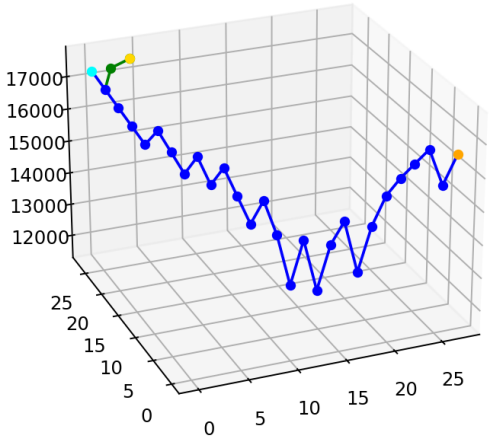
An example of Path Relinking and Local Search is displayed in Figure 7. This enhancement was not tested for 100 steps as the output is not independent from previous iterations. However, it was able to find a solution with value 18500 (shown in Figure 8b), performing better than the Python models.

Finally, the following output shows an example of successful evolution, where the new elite pool consists in the result for `k=4`:

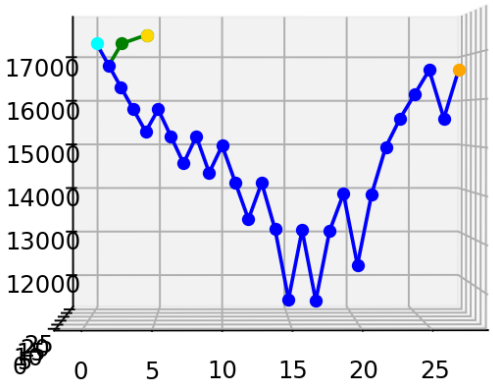
```
k=1: [16800,17100,16900]             => [17300,16900,16800,17000,16600]
k=2: [17300,16900,16800,17000,16600] => [17200,17500,17500,17600,17500]
k=3: [17200,17500,17500,17600,17500] => [17900,17600,17800,17600,17500]
k=4: [17900,17600,17800,17600,17500] => [17800,17900,17800,17800,17700]
```

Heuristic	Best value	Iterations	Time (s)
GRASP	17800	5	317.77
Forward Path Relinking	18500	10	2765.12
Backward Path Relinking	18300	10	2761.82
Mixed Path Relinking	18300	10	2776.01
Evolutionary Mixed Path Relinking	18300	20	9687.86

Table 2: GRASP and Path Relinking results

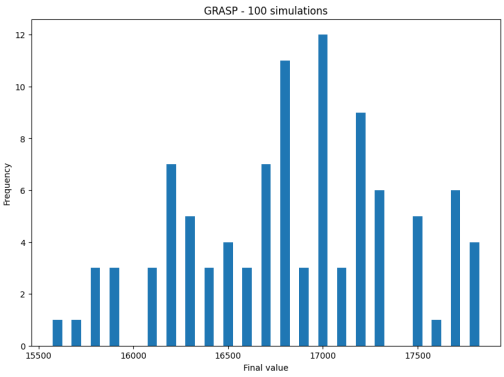


(a) 3D Path Relinking plot

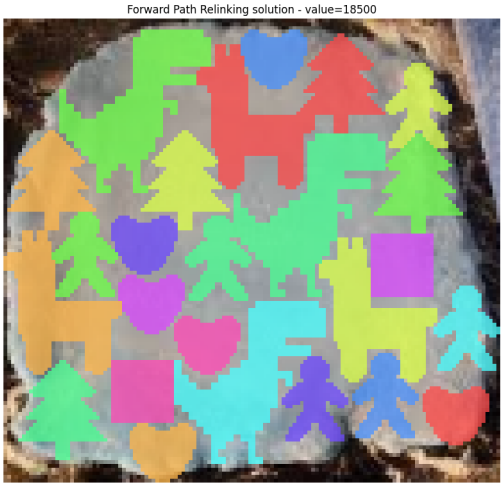


(b) Projection over distance from the starting solution and value

Figure 7: Path Relinking plot. Shows the starting solution (cyan), the guiding solution (orange) and the relinking path (blue). The Local Search path is in green, while the best final solution is gold. Axes correspond to distance from the starting solution, distance from the guiding solution and value.



(a) 100 single GRASP iterations



(b) Best Path Relinking solution

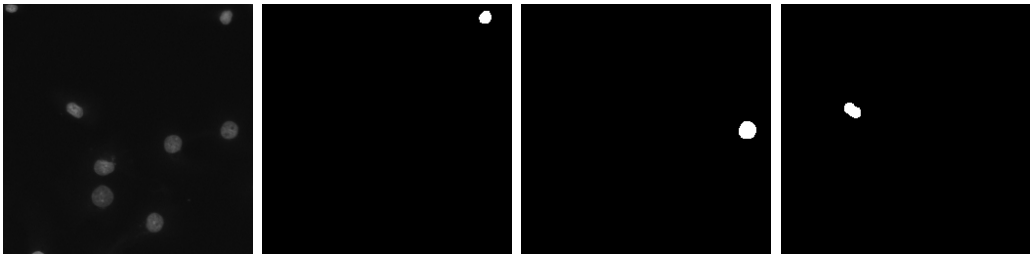
Figure 8: GRASP results

4 Further applications: histological images augmentation

This section shows a parallel between the biscuit nesting problem and a method of data augmentation for biomedical images. Images are taken from a public dataset available on kaggle.com.

4.1 The augmentation problem

Data augmentation for vision tasks in machine learning is a series of techniques used to increase the amount of data by adding slightly modified copies of already existing images or newly created synthetic data. This is an important process especially in the biomedical field, where obtaining many samples can be hard or very expensive. Figure 9a shows an image used for instance segmentation, where the objective of the learning algorithm (usually a neural network) is detecting objects of interest in an image and delineate their boundaries. The other images in Figure 9 show some of the binary masks that represent the ground truth segmentation of one nucleus (i.e. what the algorithm has to learn).



(a) Base image (b) Ex. mask 1 (c) Ex. mask/2 (d) Ex. mask/3

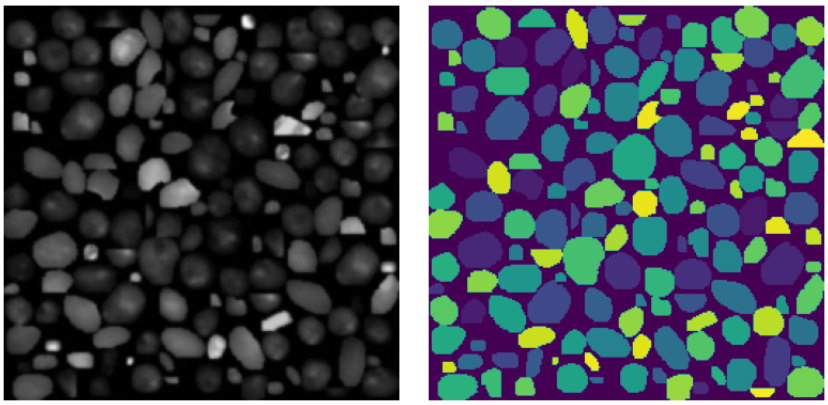
Figure 9: Example image and its masks for instance segmentation

The parallel with the problem solved by this project consists in considering the possibility of creating new (artificial) images starting from individual nuclei masks: the ground truth acts as the cookie cutters while an empty image is the biscuit dough.

4.2 Heuristic examples

The notebook available at src/04-nuclei shows how an heuristic can be used to generate new nuclei images. Unfortunately, the algorithms devised in Section 3 were not able to handle large quantities of nuclei (in the order

of hundreds), therefore an ad-hoc algorithm was developed. Figure 10 shows a result of this process.

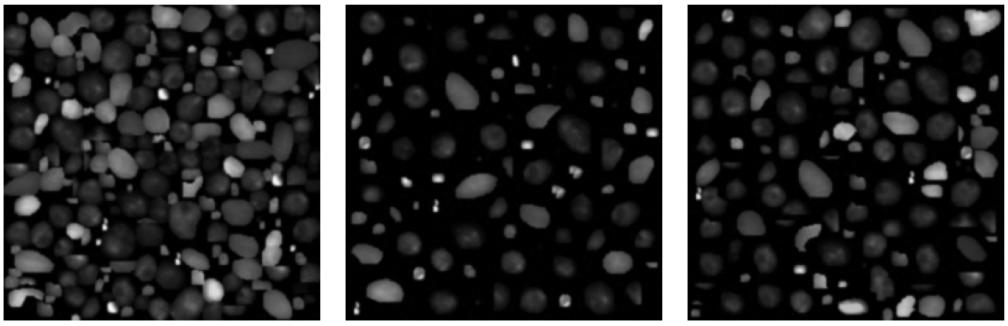


(a) Augmented image (b) Augmented mask

Figure 10: Example of augmentation

4.3 Margin modification

Augmentations usually rely on some randomization in order to create different types of images. In this context, the usage of the mask's gradient can help in enlarging or shrinking the actual nucleus mask, allowing to pack them differently and, if necessary to teach the network, overlap them. Figure 11 shows three examples of augmented images obtained by modifying the mask and using the previous heuristic.



(a) Lower margin (-1) (b) Higher margin (+5) (c) Random margin

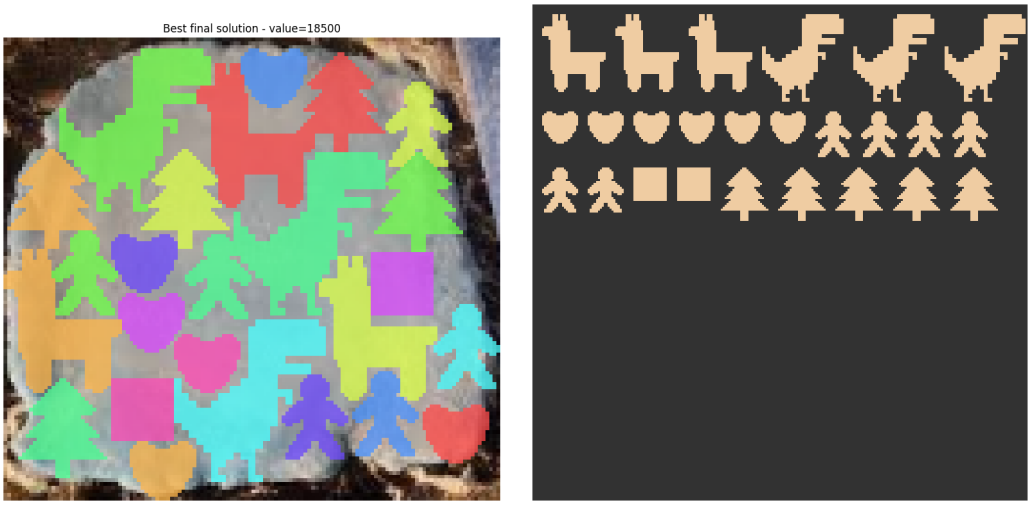
Figure 11: Example of augmentation with modified margins

5 Conclusions

This project implemented models and heuristics in order to solve the nesting problem of biscuit cutters given the available cookie dough. Due to the NP-Hard nature of this class of problems, optimization models were not able to find the optimal solution in a reasonable time; the devised heuristics, especially in the case of GRASP with Path Relinking, were able to outperform the models' partial results. Table 3 reports the best values for each of the technologies and strategies described in this report, while Figure 12 displays the best solution found.

Problem	Technology / Algorithm	Best value	Time
Integer Knapsack	AMPL (CPLEX)	29000	0.11
Integer Knapsack	OR-Tools (CP-SAT)	29000	0.562
Integer Knapsack	Gurobi	29000	0.2
Nesting	AMPL (CPLEX)	16300	40000
Nesting	OR-Tools (CP-SAT)	17800	40000
Nesting	Gurobi	17800	40000
Nesting	Greedy heuristics	16900	3.45
Nesting	GRASP	17800	317.77
Nesting	GRASP (Path Relinking)	18500	2765.12

Table 3: Final results for each problem and solution



(a) Best solution placement (b) Best solution biscuits result

Figure 12: Best solution

References

[1] Xavier Delorme, Xavier Gandibleux, and Joaquin Rodriguez. Grasp for set packing problems. *European Journal of Operational Research*, 153:564–580, 03 2004.

[2] Mauricio Resende and Celso Ribeiro. Greedy randomized adaptive search procedures: Advances and applications. *Handbook of Metaheuristics*, 01 2010.