Facultat d'Infromàtica de Barcelona (UPC) November 25, 2023
Master in Innovation and Research in Informatics AMMM
Group member name(s): Adrià Lisa Bou and Lucas Estevao Bazilio
Group member email(s): adria.lisa@estudiantat.upc.edu, lucas.estevao@estudiant.upc.edu

COURSE PROJECT

1 Linear Programming Solution

We define the following constants for the problem, which we will need to take as an input ($0 \notin \mathbb{N}$).

- $n \in \mathbb{N}$: number of orders for the bakery. Consider each order is indexed by $i \in N = [1, \dots, n]$.
- $t \in \mathbb{N}$: number of time slots available for serving. Each one is indexed by $j \in T = [1, \dots, t]$.
- $p_i \in \mathbb{R}^+$: profit we make for order $i \in N$.
- $0 < l_i \leq T$: length in time slots required to bake order $i \in N$.
- $0 < mind_i \leq maxd_i \leq T$: earliest and latest time slot the order $i \in N$ must finish baking so it can be picked up.
- $c_i \in \mathbb{R}^+$: surface in the oven that order $i \in N$ requires, measured in m^2
- $\max_{i \in N} c_i \leq maxsur$: maximum surface capacity for the oven at every time slot $j \in T$.

In our first formulation, we used the following variables:

- $y_i \in \{0, 1\}$: binary variable indicating whether order $i \in N$ is served by the bakery.
- $f_i \in T \cup \{0\}$: integer variable indicating the time-slot when an order finishes baking and is consequently delivered. If an order i is not taken ($y_i = 0$), we assign $f_i = 0$.
- $x_{ij} \in \{0, 1\}$: binary variable indicating if order $i \in N$ is being baked at time slot $j \in T$.
- $z_{ij} \in \{0, 1\}$: auxiliary indicator variable that only equals 1 when a change in value occurs between $x_{i(j-1)}$ and x_{ij}. If an order i starts baking from the beginning, we will consider $z_{i1} = 1$ (see eq. (7)).

ILP formulation

$$
\begin{aligned}
\textbf{maximize} \quad & \sum_{i=1}^{N} p_i y_i && && (1)\\
\textbf{s.t.:} \quad & \sum_{j \in T} x_{ij} \geq y_i && \forall i \in N && (2)\\
& \sum_{i \in N} c_i x_{ij} \leq maxsur && \forall j \in T && (3)\\
& \sum_{j \in T} x_{ij} = l_i \cdot y_i && \forall i \in N && (4)\\
& f_i \geq y_i \cdot mind_i && \forall i \in N && (5)\\
& f_i \leq y_i \cdot maxd_i && \forall i \in N && (6)\\
& z_{i1} = x_{i1} && \forall i \in N && (7)\\
& z_{ij} \geq x_{ij} - x_{i(j-1)} && \forall (i,j) \in N \times T \setminus \{1\} && (8)\\
& z_{ij} \geq x_{i(j-1)} - x_{ij} && \forall (i,j) \in N \times T \setminus \{1\} && (9)\\
& z_{ij} \leq x_{ij} + x_{i(j-1)} && \forall (i,j) \in N \times T \setminus \{1\} && (10)\\
& z_{ij} \leq 2 - x_{ij} - x_{i(j-1)} && \forall (i,j) \in N \times T \setminus \{1\} && (11)\\
& 1 + f_i \geq j z_{ij} && \forall (i,j) \in N \times T && (12)\\
& f_i \geq (2y_i - \sum_{j \in T} z_{ij})t && \forall i \in N && (13)\\
1 + f_i \leq j z_{ij} + & (2 - \sum_{k=1}^{j} z_{ik})(t+1) + (1 - z_{ij})(t+1) && \forall (i,j) \in N \times T && (14)
\end{aligned}
$$

Clearly, the objective function in eq. (1) aims to maximize the profit, thus, a lower bound on y_i is unnecessary as it is well defined using only the upper bound in eq. (2). The constraints defined in eqs. (3) to (6) are self-explanatory.

The constraints defined in eqs. (8) to (11) are used to define z_{ij}. They were derived having a XOR gate in mind, $z_{ij} = x_{ij} \oplus x_{i(j-1)}$.

Consider the quantities $\sigma_i = \sum_{j \in T} z_{ij}$. At first, we had an additional constraint $\sigma_i \leq 2\ \forall i \in N$ to ensure that the orders are baked continuously, but we will later see that said condition is implicitly defined in eq. (14).

Defining f_i was the hardest part. In eq. (12) we ensure that the lower bound matches the index j when the order finishes.

However, it is possible that the order i finishes at the last time slot $j = t$, setting us in the case where $\sigma_i = 1$. That is when the constraint in eq. (13) becomes meaningful, effectively setting $f_i = t$, as we assume that $maxd_i \leq t\ \forall i \in N$. Also note that the addition of the term y_i was necessary, because in the case where $\sigma_i = y_i = 0$ we want $f_i = 0$.

To understand the last constraint in eq. (14), one has to realize that if any of the coefficients preceding $(t+1)$ is greater than 0, it will become meaningless as already $1 + f_i \leq t + 1$. That happens when $z_{ij} = 0$ or if we have not yet seen a second change in the values of x_{ij}, that is, $\sum_{k=1}^{j} z_{ik} = 1$. Thus, eq. (14) will only take the meaningful form $1 + f_i \leq j z_{ij}$ exactly at the second (and final) instance of $z_{ij} = 1$.

Finally, we can also see that if $\sigma_i > 2$, for some value of j at eq. (14) we would get $1 + f_i \leq j - t - 1 \implies f_i < 0$ which would not be feasible due to eq. (5).

2 Solutions via Greedy Heuristics

All three algorithms (Greedy, Greedy with Local Search, and GRASP) utilize a class `Sol` defined in `solution.py`. The `Sol` class encapsulates the solution to the optimization problem, represented as a collection of pairs (i, f) indicating the orders $i \in N$ that are taken and their corresponding finishing time f. Moreover, it contains the constants for the problem instances (n, t, p, l, c, $mind$, $maxd$, and $maxsur$), and various convenient methods and inner variables, such as:

- `usedCapacities`$[j]$: floating-point array that keeps track of the surface capacity used by the orders already inserted into `Sol` at every time slot $j \in T$.
- `totalProfit`: variable storing the total sum of the profits p_i of each order i in the solution.
- `insertion`(): method for inserting a new order (i, f) into `Sol`, and updating `usedCapacities` and `totalProfit` accordingly.
- `deletion`(): same functionalities as `insertion`() but for removing elements.
- `isFeasible`(i): returns the pair (i, f) if order i can be processed satisfying all the constraints at finishing time f, or $(-1, \cdot)$ if it is not feasible to process the order given all the orders already inserted into `Sol`. The implementation is essentially the same as in the Greedy algorithm 1. It is worth noting that, if the order can finish at several time slots $f \in T$, this method will return the earliest.

2.1 Greedy Constructive Algorithm

Our greedy algorithm processes orders in descending order by their profits (p) and adds them into the schedule if it is feasible to do so. Thus, we are always prioritizing the insertion of tasks with the highest cost/objective value:

$$\text{Cost}(i) = p_i \quad \forall i \in N \tag{15}$$

If taking a new candidate order $i \in N$ would satisfy the surface constraint in eq. (3) for all the time slots required by its length l_i, finishing at some time f between $mind_i$ and $maxd_i$, the pair (i, f) will be a <u>feasible</u> aggregate to our solution, and we will just insert it. Given a partial solution $\mathcal{S}$, we can think about the quality of a candidate task i as:

$$q(i, \mathcal{S}) = \begin{cases} p_i & \text{if } \mathcal{S} \cup \{(i, f)\} \text{ is feasible for some } f \in T \\ -\infty & \text{otherwise.} \end{cases} \qquad \forall i \in N \tag{16}$$

Thus, in our greedy algorithm, we will be inserting the order with the highest quality q at every given time.

In algorithm 1 you can see the pseudo-code of our approach, which we conceived having algorithmic efficiency in mind. We used the iterator k to indicate the number of time slots we have yet to check for the furnace surface capacity constraint.

2.2 Local Search

To improve on the solution provided by the greedy algorithm, we wondered if by eliminating one order from $\mathcal{S} \in$ `Sol` we could then add more orders into it and increase the `totalProfit`. More formally, we conducted a first-improving local search, moving from $\mathcal{S}$ to the first neighbouring solution $\mathcal{S}'$ that improves the objective function.

Notably, after removing one order i from the initial greedy solution $\mathcal{S} \leftarrow$ Greedy, we can only hope to improve it if two or more orders with combined profit higher than p_i.

To speed things up, we introduced a set of remaining candidate tasks R to insert into $\mathcal{S}'$, with the purpose of not checking the feasibility of any task more than once in the constructive phase of the neighbouring solution. The implementation is based on algorithm 2.

Algorithm 1 Greedy Constructive Algorithm

```
procedure GREEDY
    $\mathcal{S} \leftarrow$ Sol$(n, t, p, l, c, mind, maxd, maxsur)$
    $I \leftarrow$ SORTINDICES$(p)$    ▷ Sort indices by values in $p$ in descending order
    for $i$ in $I$ do
        $j \leftarrow mind_i - l_i$
        $k \leftarrow l_i$
        while $j + k \leq maxd_i$  and  $k > 0$ do
            if $\mathcal{S}$.usedCapacities$[j] + c_i \leq maxsur$ then
                $j \leftarrow j + 1$
                $k \leftarrow k - 1$
            else
                $j \leftarrow j + 1$
                $k \leftarrow l_i$
            end if
        end while
        if $k = 0$ then
            $\mathcal{S}$.insertion$(i, j)$
        end if
    end for
    return $\mathcal{S}$
end procedure
```

Algorithm 2 Greedy Local Search Algorithm

```
procedure LOCALSEARCH($\mathcal{S}$)
    $R \leftarrow \{i \in N \mid (i, \cdot) \notin \mathcal{S}\}$    ▷ Assume $R$ is sorted in descending order by $p$, i.e.:
    best $\leftarrow \mathcal{S}$.totalProfit    ▷ $R = \{i_1, \ldots, i_r\} \implies p_{i_1} \geq \cdots \geq p_{i_r}$
    for all $(i, f) \in \mathcal{S}$ do
        $\mathcal{S}' \leftarrow \mathcal{S}$
        $\mathcal{S}'$.deletion$((i, f))$
        for $j \in R$ do
            $(k, f) \leftarrow \mathcal{S}'$.isFeasible$(j)$
            if $k \neq -1$ then
                $\mathcal{S}'$.insertion$(j, f)$
            end if
        end for
        if $p < \mathcal{S}'$.totalProfit then
            $\mathcal{S} \leftarrow \mathcal{S}'$
            best $\leftarrow \mathcal{S}'$.totalProfit
        end if
    end for
    return $\mathcal{S}$
end procedure
```

2.3 GRASP

The Greedy Randomized Adaptive Search Procedure (GRASP) is a heuristic that aims to escape from local maxima in the local search procedure. We explore a controlled number (m) of solutions $\mathcal{S}'$ in the solution space that are constructed by a non-deterministic variant of the greedy algorithm and are subsequently improved by "LocalSearch($\mathcal{S}'$)" [alg. 2] In the end, we will just keep the best solution out of any that improve on the deterministic "Greedy()" [alg. 1] with local search.

In the constructive phase, we will not just pick the highest quality order (eq. (16)). Instead, we will randomly pick it from a random candidate list RCL, that can be though of as a set "top quality" orders. Let $p_{\text{min}} = min_{i \in N}\{p_i\}$ and $\alpha \in [0, 1)$. Since at any given moment we will simply know the value i of the highest quality order, given a partial solution $\mathcal{S}$ it is convenient to define $RCL(i)$ as follows:

$$RCL(i) = \{j \in N \mid j \text{ is a feasible addition to } \mathcal{S} \wedge p_j \geq p_{\text{min}} + \alpha \cdot (p_i - p_{\text{min}})\} \tag{17}$$

If we consider the indices of the orders to be order by p in descending order, it is clear that $j \geq i \; \forall j \in RCL(i)$, and the expression for the lower bound on p_j in eq. (17) becomes:

$$p_{\text{cota}} = p_n + \alpha \cdot (p_i - p_n) \tag{18}$$

If $\alpha = 1$, this algorithm will be equivalent to running the deterministic greedy with local search m times, achieving no improvement whatsoever. On the contrary, if $\alpha = 0$, the algorithm will be picking new feasible solutions completely at random.

algorithm 3 shows the pseudocode of our implementation. As in algorithm 2, we used a set of remaining candidates R to discard all the orders that are found unfeasible to insert into the solution.

Algorithm 3 GRASP Algorithm

```
procedure GRASP($m, \alpha$)
    $\mathcal{S} \leftarrow$ LocalSearch(Greedy())
    best $\leftarrow \mathcal{S}$.totalProfit
    for $iter = 1, \ldots, m$ do
        $i \leftarrow 1$                                ▷ Assume indices are ordered by profit, i.e., $p_1 \geq \cdots \geq p_n$
        $R \leftarrow \{1, \ldots, n\}$
        while $i \leq n$ do                                      ▷ Constructive phase
            $RCL \leftarrow \varnothing$
            $p\_cota \leftarrow p_n + \alpha \cdot (p_i - p_n)$
            $j \leftarrow i$
            while $j \leq n$ and $p_j \geq p\_cota$ do
                $(k, f) \leftarrow \mathcal{S}'$.isFeasible($j$)
                if $k \neq -1$ then
                    $RCL \leftarrow RCL \cup \{(j, f)\}$
                    $j \leftarrow j + 1$
                else
                    $R \leftarrow R \setminus \{j\}$
                end if
                if $R = \varnothing$ then break
                end if
                while $j \notin R$ and $j \leq n$ do
                    $j \leftarrow j + 1$
                end while
            end while
            if $RCL \neq \varnothing$ then
                $(i^*, f^*) \leftarrow$ random($RCL$)        ▷ Choose an element from $RCL$ at random
                $\mathcal{S}'$.insertion($i^*, f^*$)
                $R \leftarrow R \setminus \{i^*\}$
            end if
            while $i \notin R$ and $i \leq n$ do
                $i \leftarrow i + 1$
            end while
        end while
        $\mathcal{S}' \leftarrow$ LocalSearch($\mathcal{S}'$)
        if $\mathcal{S}'$.totalProfit $> best$ then
            $best \leftarrow \mathcal{S}'$.totalProfit
            $\mathcal{S} \leftarrow \mathcal{S}'$
        end if
    end for
    return $\mathcal{S}$
end procedure
```

2.4 Tuning of the α parameter

The α parameter plays a pivotal role in striking a balance between the algorithm's greedy and randomized components, significantly influencing its capacity to explore a diverse set of solutions. To optimize this parameter, we conducted experiments involving the tuning of α. Specifically, we selected a set of $M = 100$ instances, each of size $n = 5000$, and computed the average profit for varying values of α. As elaborated in Figure 1, the choice of $n = 5000$ was justified by the observation that CPLEX demonstrated acceptable execution times up to a maximum of 5100 orders.

We opted for a comprehensive exploration of the α space, choosing a range of [0.05, 0.95] with increments of 0.05. Simultaneously, we scrutinized the impact of the number of iterations m in the GRASP algorithm on the α tuning process. Two distinct values, namely 20 and 60, were employed for m in this investigation.

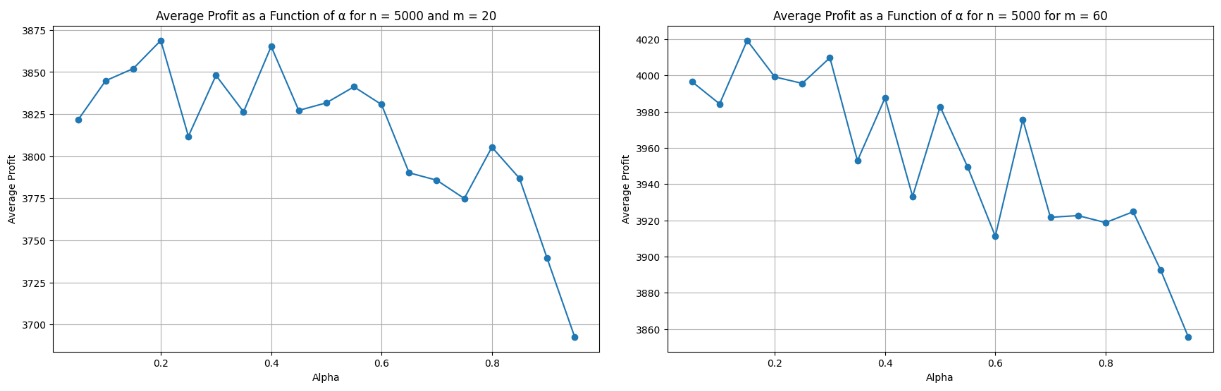

Figure 1: Alpha tuning experiments with $m = 20$ and $m = 60$

It is clear that the expectation for the average profit increases with m, and also that higher values for α make the GRASP algorithm more deterministic and less prone to improve on the initial local maxima.

By intuition, a lower value for alpha will have a greater expected value for the objective function as the size of RCL also increases. However, that will only happen if m is appropriately large, as with more possible random choices more trials will be needed to find a good new order to insert in our solution, like finding a needle in a haystack. We did not have time to mathematically prove these claims but from our empiric observations, we are convinced that they hold.

Even though we are aware that the highest values on figures (1) are by no means *optimal* choices, and can be mostly attributed to *luck*, we felt comfortable choosing $\alpha^* = 0.2$ for $m = 20$ and $\alpha^* = 0.15$ for $m = 60$.

We also noticed that lower values for alpha yielded higher computation times, but we did not take that fact into consideration as, with such values for m, GRASP ran quite fast in our computers.

3 Performance Comparison: CPLEX vs. Heuristic Algorithms

3.1 Introduction

This section investigates the performance of IBM CPLEX, a commercial optimization solver, in comparison with each heuristic algorithm for solving our optimization problem. We aim to discern the strengths and weaknesses of each approach in terms of solution quality and computation time.

The instances for evaluation were systematically chosen to cover the range of sizes $n = 100, 200, \ldots, 5000, 5100$. The upper boundary of 5100 was chosen simply because CPLEX took more than 30 minutes to evaluate that instance.

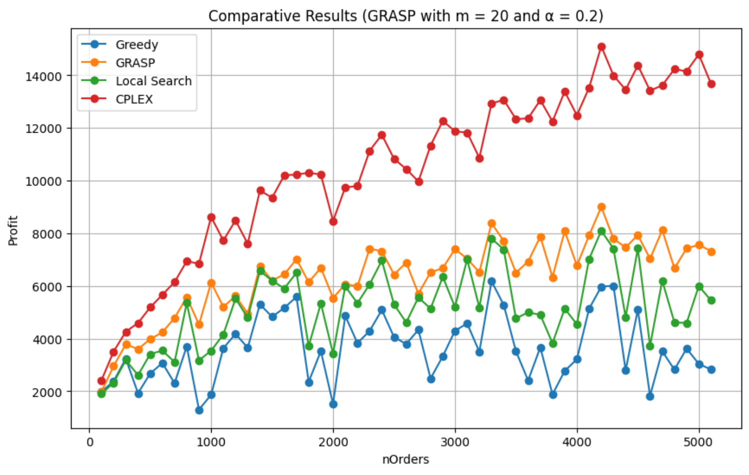

Figure 2: Comparative Results regarding the quality of the solution among CPLEX and heuristic algorithms. (GRASP with $m = 20$ and $\alpha = 0.2$)

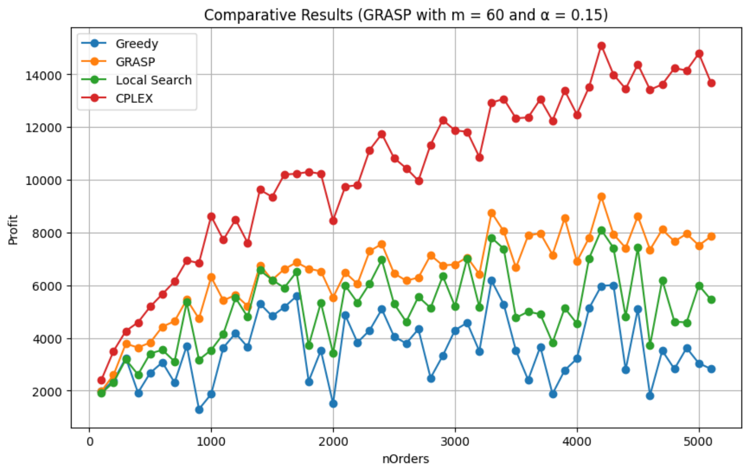

Figure 3: Comparative Results regarding the quality of the solution among CPLEX and heuristic algorithms. (GRASP with $m = 60$ and $\alpha = 0.15$)

In light of the analyses presented in Figure 2 and Figure 3, a discernible pattern emerges regarding the ranking of solution quality. Evidently, the algorithms can be hierarchically ordered as follows: CPLEX secures the foremost position, followed by GRASP, local search, and, ultimately, the greedy algorithm. This discernment not only underscores the efficacy of CPLEX as the leading solution provider but also delineates a nuanced hierarchy among the algorithms, thereby contributing valuable insights to the understanding of their respective performances.

3.1.1 Computation Time

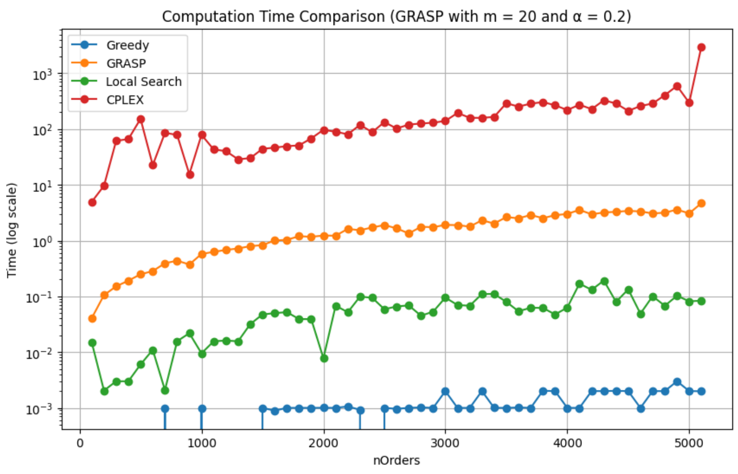

Figure 4: Comparative Results regarding the computation time of the solution among CPLEX and heuristic algorithms. (GRASP with $m = 20$ and $\alpha = 0.2$)

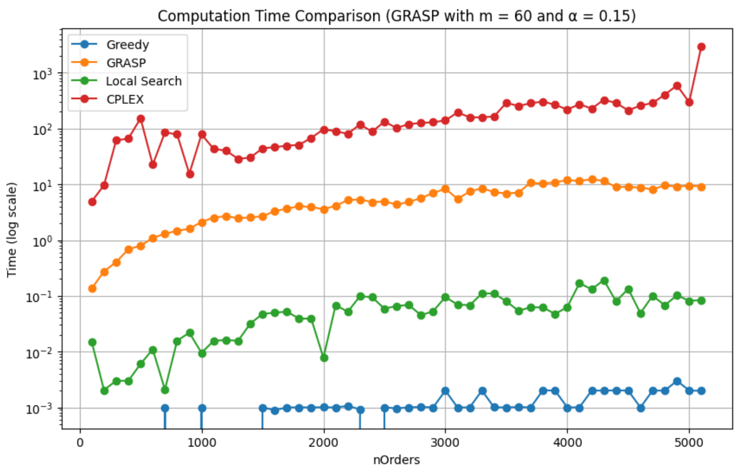

Figure 5: Comparative Results regarding the computation time of the solution among CPLEX and heuristic algorithms. (GRASP with $m = 60$ and $\alpha = 0.15$)

Upon meticulous examination of the computational efficiency across the algorithms, as evidenced by the data presented in Figure 4 and Figure 5, a distinctive hierarchy emerges in terms of computation time. Notably, the computational speed is hierarchically organized as follows: the greedy algorithm exhibits the swiftest computational performance, succeeded by local search as the second fastest, followed by GRASP, and ultimately, CPLEX represents the algorithm with the most extensive computation time.

This observed hierarchy in computational efficiency not only provides valuable insights into the temporal dynamics of algorithmic execution but also underscores the nuanced trade-offs between solution quality and computational expeditiousness.