Algorithmic Methods for Mathematical Models Course Project

Tim Wichelmann
tim.wichelmann@estudiantat.upc.edu

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

December 13, 2023

Contents

List of Figures

1 Introduction

In this course project, we examine and optimize the order selection and scheduling of a bakery. To this end, we formalised a model and employed CPLEX to solve the resulting Linear Integer Program (ILP). Although it delivers an optimal solution, obtaining the order selection in this manner can take a long time for large instances. Therefore, we developed a greedy constructive algorithm that delivers a feasible and sufficient selection many orders of magnitude faster. The report includes our iterative process of further closing the optimality gap by developing and benchmarking more sophisticated greedy cost functions, employing local search, and by implementing the GRASP meta-heuristic. Finally, we summarise the achieved results.

2 Formal Problem Statement

In this section, we summarise the given problem and its respective input data, the output as well as the objective.

Given

- n : The number of orders.
- t : The latest time slot we consider.
- $profit_i$, $(1 \leq i \leq n)$: Profit associated with order i in €.
- $length_i$, $(1 \leq i \leq n)$: Required number of time slots to process order i.
- $min\_deliver_i$, $(1 \leq i \leq n)$: First time slot in which order i can be picked up by the customer.
- $max\_deliver_i$, $(1 \leq i \leq n)$: Last time slot order i can be picked up by the customer.
- $surface_i$, $(1 \leq i \leq n)$: Oven surface required to bake order i in m^2.
- $surface\_capacity$: Oven surface available in the bakery in m^2.

If an order is processed, it has to be picked up in the same time slot in which it will be finished. Due to the limited surface capacity, the oven surface occupied by the scheduled orders can never exceed $surface\_capacity$ for any given time slot. At any time, the total surface of the bread that is being baked cannot exceed this capacity. An order i has to be picked up in the time range given its $min\_deliver_i$ and $max\_deliver_i$ times, including the boundaries.

Output

The output consists of a matrix of size $n \times t$. It represents a timetable indicating if and when an order should be processed (see figure 3).

Objective

Maximize the total profit, calculated as the sum of profits of each order that is successfully scheduled for the next period.

3 Integer Linear Programming Model

This section includes a description of our Linear Integer Program.

3.1 Decision variables

- y is a matrix of size $n \times t$ consisting of binary variables which indicates if the order i will be baked in time slot j.

3.2 Auxiliary variables

- x_i is a binary variable indicating whether order i has the right amount of time slots assigned to it. We use it as an indicator of whether an order is part of the schedule or not.
- $start_i$, $(1 \leq i \leq n)$denotes the time slot in which the baking process of order i is started.
- end_i, $(1 \leq i \leq n)$ denotes the time slot in which the baking process of order i will be finished.
- *geq_start* is an $n \times t$ matrix of binary variables where geq_start_{ij} is 1 iff. j is greater or equal to the first time slot of order i.
- *leq_end* is an $n \times t$ matrix of binary variables where leq_end_{ij} is 1 iff. j is less or equal to the last time slot of order i.
- min_start_i, $(1 \leq i \leq n)$, which is the earliest possible time slot for order i. This can be done independently of other variables:

$$min\_start_i = min\_deliver - length_i + 1$$

- max_start_i, $(1 \leq i \leq n)$, which denotes the latest possible time slot for order i, given by

$$max\_start_i = max\_deliver - length_i + 1$$

3.3 Objective function

We want to maximize the total profit obtained from the selected orders

$$\max \sum_{i=1}^{n} profit_i \, x_i.$$

3.4 Constraints

However, a possible solution is subject to the following constraints:

- In every time slot, the space capacity is respected:

$$\sum_{i=1}^{n} surface_i \, y_{ij} \leq surface\_capacity, \qquad (1 \leq j \leq t) \qquad (1)$$

- Each order is started in a time slot that allows it to finish in the delivery window:

$$start_i \leq max\_start_i, \qquad (2)$$
$$min\_start_i \leq start_i, \qquad (1 \leq i \leq n) \qquad (3)$$

- If an order i is part of the schedule, it is assigned the $length_i - 1$ contiguous time slots from $start_i$ to end_i:

$$j \geq start_i - (t+1) * (1 - geq\_start_{ij}) \quad (4)$$
$$start_i \geq j - (t+1) * geq\_start_{ij} \quad (5)$$
$$end_i \geq j - (t+1) * (1 - leq\_end_{ij}) \quad (6)$$
$$j \geq end_i - (t+1) * leq\_end_{ij} \quad (7)$$
$$0 \leq geq\_start_{ij} + leq\_end_{ij} - 2 * y_{ij}, \qquad (1 \leq i \leq n, min\_start_i \leq j \leq max\_deliver_i) \quad (8)$$

- If an order i is part of the schedule, its end time end_i is equal to $start_i + length_i - 1$. If order i is not part of the schedule, this is indicated by the fact that $start_i > end_i$.

$$end_i = start_i + x_i\ length_i - 1, \qquad (1 \leq i \leq n) \quad (9)$$

- If an order is part of the schedule, we assign the correct amount of time slots to it. Otherwise, we assign zero time slots to it:

$$\sum_{j=min\_start_i}^{max\_deliver_i} y_{ij} = x_i\ length_i, \qquad (1 \leq i \leq n) \quad (10)$$

- There are no orders assigned to time slots that are infeasible for this order because of the delivery window:

$$\sum_{j=1}^{min\_start_i - 1} y_{ij} = 0, \qquad (1 \leq i \leq n) \quad (11)$$
$$\sum_{j=max\_deliver_i + 1}^{t} y_{ij} = 0, \qquad (1 \leq i \leq n) \quad (12)$$

4 Greedy Algorithm and Meta-Heuristics

In this section we describe and reason about the development of our greedy algorithm. This includes a straight-forward approach as a baseline which we continuously improve toward a a reliably sufficient solution compared to the optimal solution. Later, we apply local search and GRASP and tune the α parameter.

4.1 Greedy constructive algorithm

```
Input: orders
solution ← ∅
sortedOrders ← sort(orders, $\lambda\, x$ : orderRating($x$), DESC)
foreach order in sortedOrders do
    candidateSlots ← findFeasibleAssignments(order, solution)
    if $|candidateSlots| = 0$ then
        continue with next iteration
    end
    sortedCandidates ← sort(candidateSlots, $\lambda\, x$ : assignmentRating($x$, solution), DESC)
    candidate ← sortedCandidates[0]
    assign(order, startTime(candidate), solution)
end
return solution;
```

Our *greedy cost function* is split into two separate calculations, one for the orders and one for the time slots. We developed different rating criteria which are further described in section 6. We found the following combination to consistently produce good results:

- For an order i, a function weighing its positive and negative qualities against each other:

$$orderRating(i) = \frac{profit_i(max\_deliver_i - min\_deliver_i)}{(length_i \cdot surface_i)}$$

- For a possible starting time slot j (for a fixed order i), the average percentage of space used during the baking

$$assignmentRating(i,j) = \frac{\sum_{l=j}^{j+length_i} (\sum_{k=1}^{n} y_{kj}\, surface_k / surface\_capacity)}{length_i}$$

4.2 Local search

Next, we present the local search procedure which uses the *first improvement* strategy to improve a solution towards a local optimum.

```
Input: Solution S
iteration ← 0
while $time < maxTime$ do
    oldOrders ← sort(attendedOrders(S), $\lambda\, x$ : reassignmentRating(x), DESC)
    oldOrder ← oldOrders[mod(iteration, length(oldOrders))]
    currentProfit ← fitness(S)
    newOrders ← sort(unattendedOrders(S), $\lambda\, x$ : profit(x), DESC)
    oldStart ← startsAt(S, oldOrderId)
    unassign(S, oldOrderId)
    foreach $newOrder$ in $newOrders$ do
        assignments ← findFeasibleAssignments(S, newOrder)
        if $|assignments| = 0$ then
            continue with next iteration
        end
        move ← (oldOrder, newOrder, start(assignments[0]))
        neighborHighestProfit ← evaluateNeighbor(S, move)
        if $currentProfit < neighborHighestProfit$ then
            neighbor ← createNeighborSolution(solution, move)
            if $neighbor = NULL$ then continue with next iteration;
        end
    end
    if $neighbor = NULL$ then
        solution.assign(oldOrderId, oldStart)
        iteration ← iteration + 1
        continue with next iteration
    else
        neighborFitness ← getFitness(neighbor)
        incumbent ← neighbor
        incumbentFitness ← neighborFitness
        iteration ← 0
    end
end
return $incumbent$
```

The *reassignmentRating* of a given order is equal to the surface it occupies. Removing an order with a high surface frees up more space for local search to fit in previously unused orders.

Once an old order is removed, the new orders are tried in order of profit. This ensures that local search can quickly increase the objective function value.

4.3 GRASP

Due to its greedy nature, the formally described heuristic only considers a limited set of candidates, even while other paths may lead to more profit. By employing GRASP, we achieve a configurable variability in the candidate set. By picking a random candidate within a limit, we can obtain more solutions and pick the best one while still being considerably faster compared to solving the ILP.

```
GRASP constructive phase
Input: orders
solution ← $\emptyset$
foreach order in orders do
    candidates ← feasibleAssignments(order, solution)
    if $|candidates| = 0$ then
        continue with next iteration
    end
    Evaluate $q(c)$ for all $c \in$ candidates
    $q^{\min} \leftarrow \min\{q(c) \mid c \in \text{candidates}\}$
    $q^{\max} \leftarrow \max\{q(c) \mid c \in \text{candidates}\}$
    $RCL_{\max} \leftarrow \{c \in C \mid q(c) \geq q^{\max} - \alpha(q^{\max} - q^{\min})\}$
    Select $c \in RCL_{\max}$ at random
    solution ← solution $\cup \{c\}$
end
return solution
```

Here $q(i, j)$ denotes the assignment rating we have seen before in the greedy construction phase, which is the average space usage if the order i were to be scheduled at time slot j.

5 Parameter Tuning

By increasing and decreasing the α-parameter, we can tune the variability of candidates we consider. The following plot was obtained by running our GRASP implementation on the same instance for one minute while increasing the α-parameter, thus adding more candidates for every respective run. Compared to the deterministic greedy solver, we consistently achieve a significantly better result by also considering previously unexplored paths. As can be seen in the plot, an alpha value of 0.8 turned out to be quite efficient while a fully randomized approach probably spends too much execution time trying out bad orders and assignments.

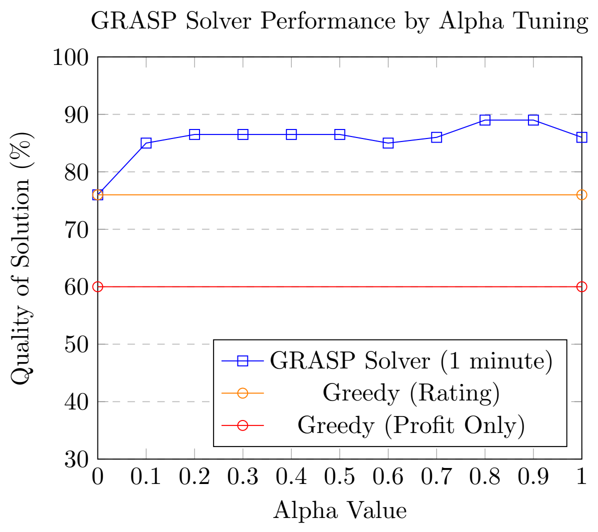

Figure 1: This plot compares the quality of solutions when tuning the α-parameter compared to the optimal solution and the one obtained using the Greedy solver. Each run was granted a maximum period of one minute.

As a practical example, we computed the optimal solution of a relatively large instance of 125 pickable orders and 125 available timeslots which took more than two hours. In the following plot, we see the relative profits obtained by applying GRASP with an increasing amount of granted time compared to the optimal solution. For an alpha value of 0.8, we reach 89% of the optimal solution within 60 seconds.

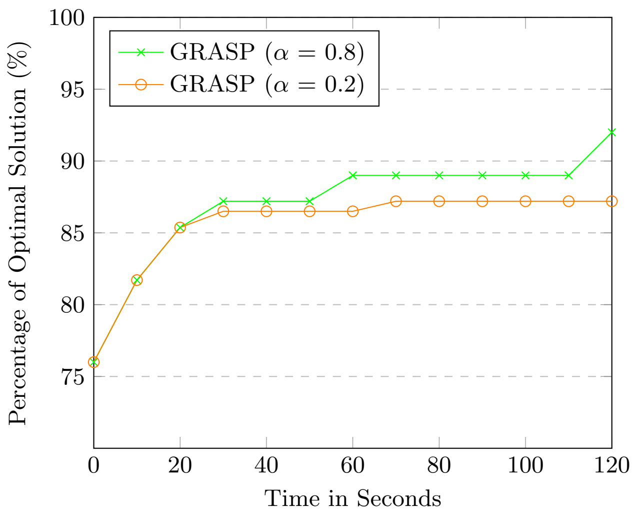

Figure 2: At $\alpha = 0.8$, 60 seconds are usually enough to almost reach 90% of the optimal profit.

6 Comparison

In this section, we compare the results of the previously described implementations and visualise the obtained order selections and schedules. To present the effects of our optimizations, we created solutions for the previously described instance of 125 pickable orders and 125 available timeslots using different implementations. In the following figure, we see an example of an optimal solution that consists of 24 picked orders which would achieve a profit of 164 €. Although it is an optimal schedule, computing it took over two hours in our local development environment.

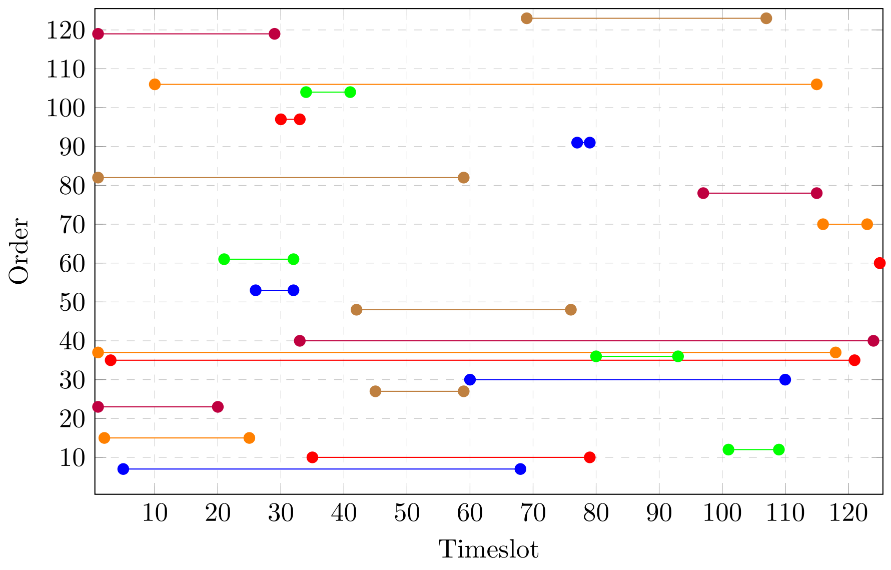

Figure 3: This schedule represents an optimal solution for a given data set of 125 orders and 125 time slots. The profit of this optimal solution is 164 €.

The next schedule was created using the Greedy solver with a cost function tuned to only consider the profit of an order. In this case, we would only pick 13 orders which will result in a total profit of 97 €. We can account a significant portion of the optimality gap to high-profit orders which may occupy a lot of surface for a long amount of time. For example, order 14 (orange) adds 10 €to our total profit, yet it needs to be processed for 114 time slots while requiring 2 m^2 of our total capacity of 15 m^2. Although its individual profit is high (in this instance the highest possible profit), the order blocks the surface capacity for too long.

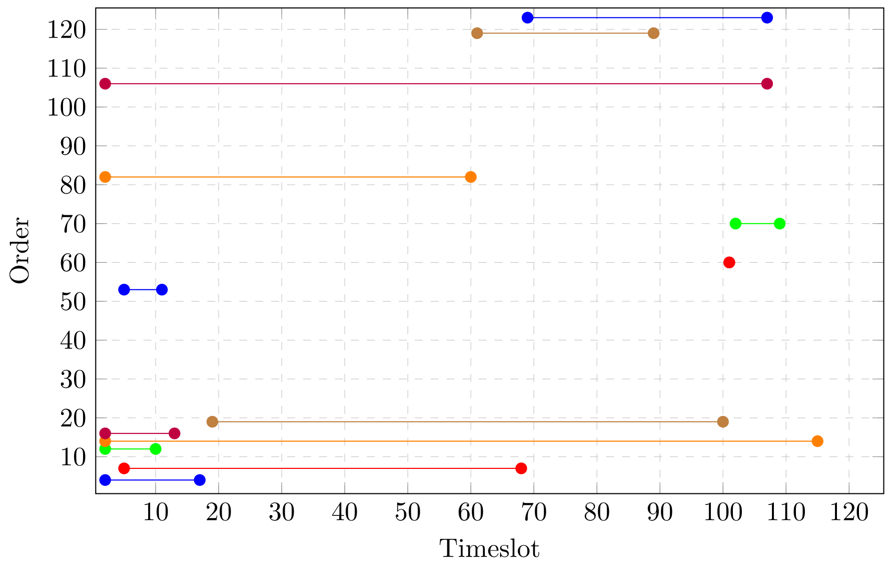

Figure 4: This schedule was created for 125 possible orders, resulting in an optimized profit of 97 €. The greedy algorithm was tuned towards preferring orders with high profit.

This schedule was created using the refined greedy cost function seen in section 4.1. Besides the profit, we now also take the required surface and time of an order into account. Additionally, we value flexible orders with large delivery time windows. As a result, we scheduled 20 orders which led to a significantly improved profit of 126 € leaving an optimality gap of 23,2%. If we compare it to the previous schedule, we see how order 14 left the schedule in favor of multiple smaller orders which accumulate more profit than order 14 alone. Generally, we can say that this approach works much better for the majority of instances.

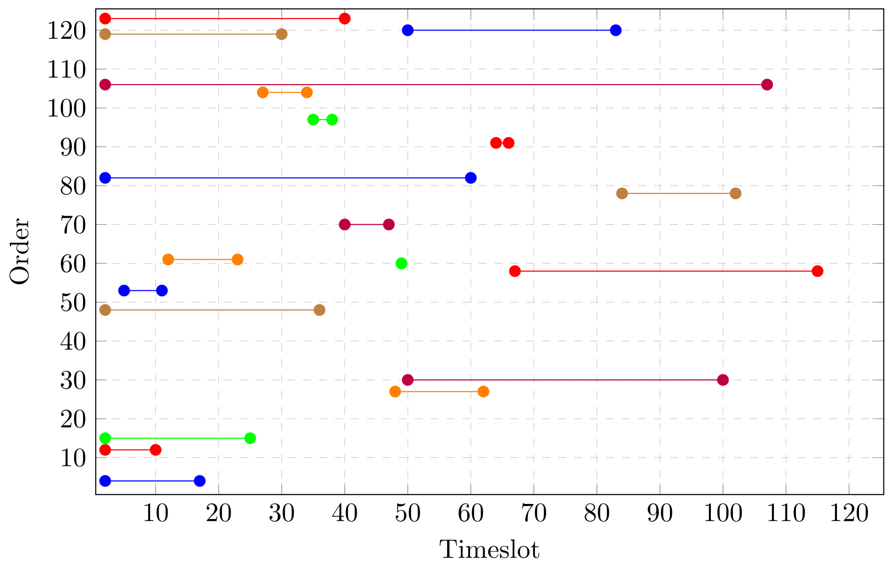

Figure 5: Computing this schedule took one second. It would result in an optimized profit of 126 €. The greedy algorithm was tuned towards preferring short orders with little surface requirement, but also considering the profit.

The last schedule was created by running the GRASP solver for one minute at an α-value of 0.8. Randomizing the candidate selection to a certain extent allows us to explore a variety of paths and achieve an optimized profit of 146 € which leaves us with an optimality gap of 11% for this given instance.

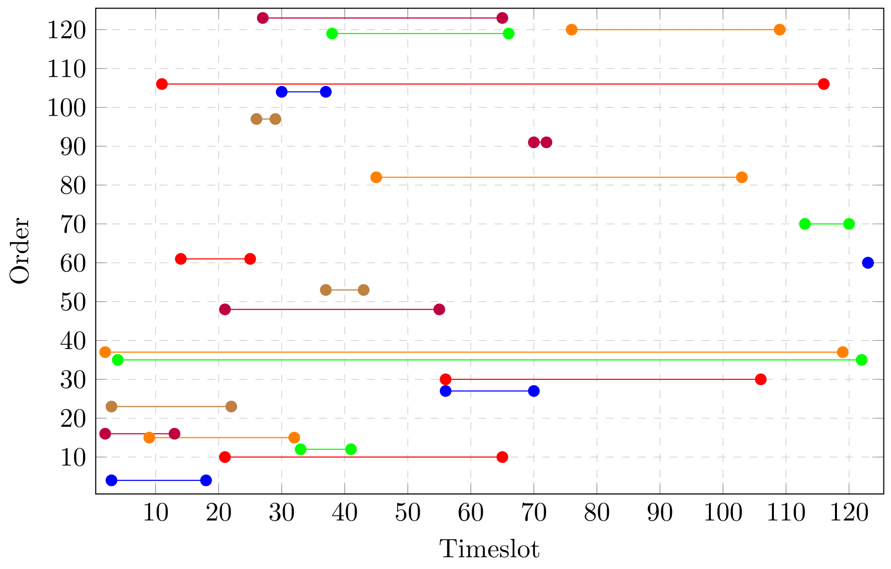

Figure 6: By running the GRASP meta-heuristic for one minute, we get a schedule that will result in a profit of 146 €.

In the following plot, we can see that our refined greedy cost function works well for most instances. We can consistently enhance the greedy solution by additionally running local search for 30 seconds. Using GRASP, we can reliably obtain a very good solution within one minute.

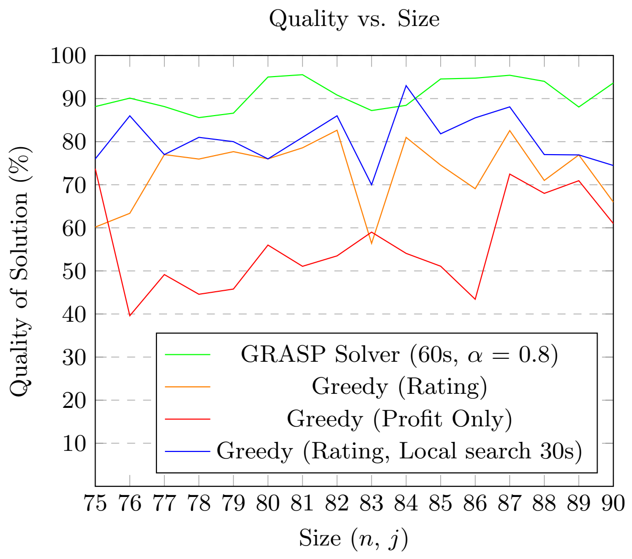

Figure 7: The quality of the solution obtained with different solvers compared to an optimal solution.

Since we need to submit our order selection in time, an increasing instance size quickly establishes a border at which it makes no sense to compute an optimal solution. However, we can still generate a sufficiently good selection for much larger instances in a fraction of the time by using our heuristic approach. For example, with our given computing resources, we would be able to reliably compute an optimal solution in time for instances with more than 150 orders and 150 available timeslots. The greedy solver allows us to go far beyond this limit while still delivering good results, especially in combination with local search and GRASP.

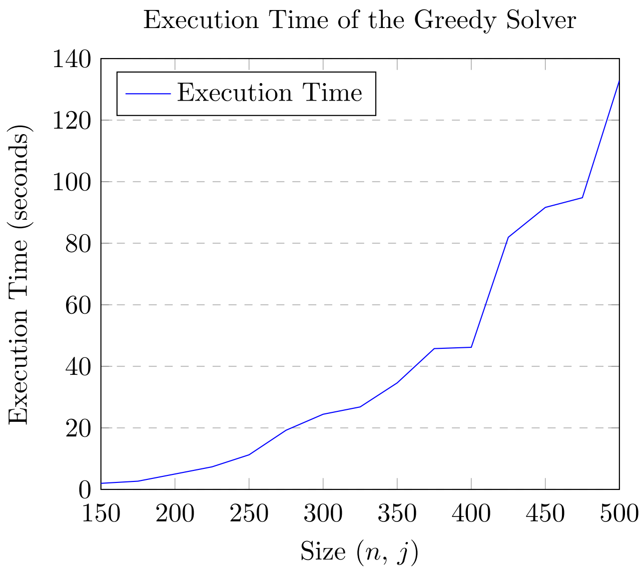

Figure 8: The Greedy implementation can provide feasible and profitable schedules within very little time compared to CPLEX.

The time needed to compute the optimal solution is very sensitive to the instance size but also depends on the input data itself. Therefore, obtaining an optimal solution using CPLEX does not scale well, as can be seen in the following plot.

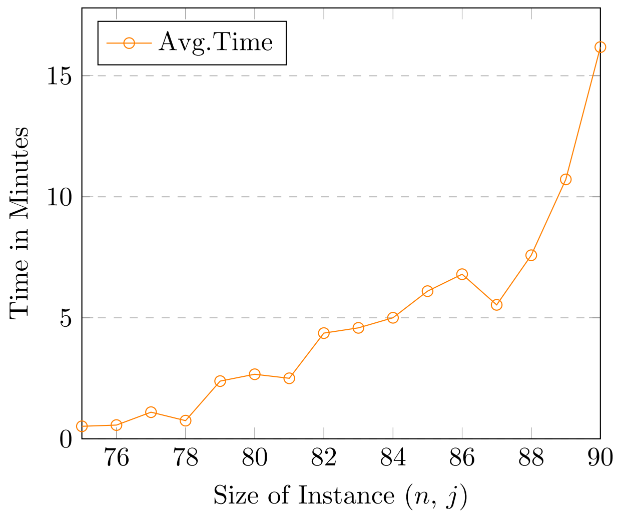

Figure 9: Avarage time needed to obtain an optimal solution for an increasingly large instance using CPLEX.

7 Future Prospects

For the greedy algorithm, there is the possibility to try different combinations of order/time slot greedy cost functions. In the case of GRASP, we could apply the RCL to the list of sorted orders to explore more diverse solutions.

8 Summary

In this project, we formally modeled the problem of picking and scheduling orders to maximize the profit of the bakery and used CPLEX to find optimal solutions for generated data sets. Since obtaining an optimal solution for a large instance can take a long time, we also developed heuristic approaches to generate feasible schedules that result in a good profit. To this end, we first implemented a greedy algorithm which is tuned towards picking high-profit orders. As described in figure 4, this approach tends to cloak the schedule with a few profitable orders which can result in a rather high optimality gap for most instances. Hence, we developed a greedy cost function that favors flexible, small, and short orders that still add a reasonable profit (see 4.1) which results in much better profits. Local search was used as an improvement to the greedy algorithm to include feasible high-profit orders that the greedy algorithm occasionally misses. Lastly, we applied the GRASP meta-heuristic to our greedy solver and tuned the alpha value to further close the optimallity gap towards a consistent single-digit percentage in a short amount of time.