Algorithmic Methods for Mathematical Models
Course Project

Marco Bartoli, Ilan Vezmarovic

UPC Universitat Politècnica de Catalunya

May 26, 2024

1 Formal Problem Definition

1.1 Inputs

- n: number of products
- x: height of the suitcase in millimeters
- y: width of the suitcase in millimeters
- c: limit to the total weight of the suitcase in grams
- p_i: price of the i-th product in euros
- w_i: weight of the i-th product in grams
- s_i: side length of the i-th product's (square) box in millimeters

1.2 Outputs

- Indices of the products chosen to maximize the accumulated price
- Arrangement of the products in the suitcase

2 Mathematical Formulation

2.1 Decision Variables

- Chosen_i: binary variable that is 1 if object i is chosen, and 0 otherwise.
- PointsX_i: the x-coordinate of the bottom-left corner of object i.
- PointsY_i: the y-coordinate of the bottom-left corner of object i.
- $\text{Overlap}_{i,j,d}$: binary variable indicating if objects i and j do not overlap in direction d, where $d \in \{1, 2, 3, 4\}$.

2.2 Objective Function

Maximize the total price of the chosen objects:

$$\text{maximize} \sum_{i=1}^{n} p_i \cdot \text{Chosen}_i$$

2.3 Constraints

2.3.1 Max Weight Constraint

Ensure the total weight of the chosen objects does not exceed the suitcase's capacity:

$$\sum_{i=1}^{n} w_i \cdot \text{Chosen}_i \leq c$$

2.3.2 Coordinate Bounds Constraints

Ensure each object lies entirely within the suitcase's boundaries:

$$\forall i \in \{1, \ldots, n\}, \quad \text{PointsX}_i \geq 1$$

$$\forall i \in \{1, \ldots, n\}, \quad \text{PointsY}_i \geq 1$$

$$\forall i \in \{1, \ldots, n\}, \quad \text{PointsX}_i + s_i - 1 \leq x$$

$$\forall i \in \{1, \ldots, n\}, \quad \text{PointsY}_i + s_i - 1 \leq y$$

2.3.3 Non-Overlapping Constraints

Ensure no two chosen objects overlap within the suitcase using the big-M method:

1. Horizontal non-overlapping to the right:

$$\forall i, j \in \{1, \ldots, n\}, i \neq j, \quad \text{PointsX}_i - \text{PointsX}_j + s_i \leq -M \cdot (\text{Chosen}_i + \text{Chosen}_j + \text{Overlap}_{i,j,1} - 3)$$

2. Horizontal non-overlapping to the left:

$$\forall i, j \in \{1, \ldots, n\}, i \neq j, \quad \text{PointsX}_j - \text{PointsX}_i + s_j \leq -M \cdot (\text{Chosen}_i + \text{Chosen}_j + \text{Overlap}_{i,j,2} - 3)$$

3. Vertical non-overlapping upwards:

$$\forall i, j \in \{1, \ldots, n\}, i \neq j, \quad \text{PointsY}_i - \text{PointsY}_j + s_i \leq -M \cdot (\text{Chosen}_i + \text{Chosen}_j + \text{Overlap}_{i,j,3} - 3)$$

4. Vertical non-overlapping downwards:

$$\forall i, j \in \{1, \ldots, n\}, i \neq j, \quad \text{PointsY}_j - \text{PointsY}_i + s_j \leq -M \cdot (\text{Chosen}_i + \text{Chosen}_j + \text{Overlap}_{i,j,4} - 3)$$

2.3.4 At Least One Not Overlapping Constraint

Ensure that for any two objects, at least one of the non-overlapping conditions is satisfied:

$$\forall i, j \in \{1, \ldots, n\}, i \neq j, \quad \sum_{d=1}^{4} \text{Overlap}_{i,j,d} \geq 1$$

3 Heuristic

3.1 Greedy Algorithm

A greedy algorithm makes the optimal choice at each step by selecting products based on $Qvalue = \frac{price}{side \times weight}$.

Algorithm 1 Greedy Algorithm

```
Input: A problem instance with $n$ products, suitcase dimensions $(x, y)$, and weight capacity $c$.
Output: A feasible solution with selected products that maximize the total price.
Initialize solution as empty
Sort products by $(price/side/weight)$ in descending order
for each product in sorted products do
    if product can fit in the suitcase and does not exceed weight limit then
        Add product to solution
        Update suitcase dimensions and weight capacity
    end if
end for
return solution
```

3.1.1 Pseudocode

4 Meta-heuristics

4.1 Local Search Algorithm

After constructing an initial solution using the greedy algorithm, a local search is applied to improve it by exploring neighboring solutions.

4.1.1 Pseudocode

Algorithm 2 Local Search Algorithm

```
Input: An initial solution, mode (FIRST_IMPROVEMENT or BEST_IMPROVEMENT), strategy (EXCHANGE or REASSIGNMENT)
Output: An improved solution
bestSolution = initialSolution
improved = true
while improved do
    improved = false
    for each product in bestSolution do
        if strategy == EXCHANGE then
            newSolution = exchangeProduct(bestSolution, product)
        else if strategy == REASSIGNMENT then
            newSolution = reassignProduct(bestSolution, product)
        end if
        if newSolution is better than bestSolution then
            bestSolution = newSolution
            improved = true
            if mode == FIRST_IMPROVEMENT then
                return bestSolution
            end if
        end if
    end for
end while
return bestSolution
```

4.2 GRASP (Greedy Randomized Adaptive Search Procedure)

GRASP consists of a construction phase where a feasible solution is generated using a randomized greedy algorithm, followed by a local search to iteratively improve the solution. By default, GRASP uses `FIRST_IMPROVEMENT` and `EXCHANGE` for local search.

Algorithm 3 Exchange Product

```
Input: A solution, a product to be exchanged
Output: A new solution with the product exchanged
bestSolution = initialSolution
newProductList = copy of the selected product list from the solution
remove the selected product from newProductList
for each unselected product do
    add unselected product to newProductList
    newSolution = findSolutionForProductList(newProductList)
    if newSolution is better than bestSolution then
        bestSolution = newSolution
        if mode == FIRST_IMPROVEMENT then
            return bestSolution
        end if
    end if
    remove unselected product from newProductList
end for
return bestSolution
```

Algorithm 4 Reassign Product

```
Input: A solution, a product to be excluded for re-evaluation
Output: A new solution with products reassigned
bestSolution = initialSolution
for each selected product do
    newProductList = copy of the all product list
    remove the selected product from newProductList
    newSolution = findSolutionForProductList(newProductList)
    if newSolution is better than current solution then
        bestSolution = newSolution
        if mode == FIRST_IMPROVEMENT then
            return bestSolution
        end if
    end if
end for
return bestSolution
```

4.2.1 Pseudocode

Algorithm 5 GRASP Algorithm

```
Input: A problem instance, maxIterations, alpha (for RCL threshold)
Output: The best solution found
bestSolution = null
for iteration = 1 to maxIterations do
    greedySolution = constructGreedyRandomizedSolution(problem, alpha)
    localOptimalSolution = LOCAL_SEARCH(greedySolution, FIRST_IMPROVEMENT)
    if bestSolution is null or localOptimalSolution is better than bestSolution then
        bestSolution = localOptimalSolution
    end if
end for
return bestSolution
```

Algorithm 6 Construct Greedy Randomized Solution

```
Input: A problem instance, alpha (for RCL threshold)
Output: A feasible solution
Initialize solution as empty
Sort products by ($price/side/weight$) in descending order
while there are remaining products do
    $qMax$ = maximum $Q$ value in remaining products
    $qMin$ = minimum $Q$ value in remaining products
    threshold = $qMax - alpha \times (qMax - qMin)$
    RCL = {products with $Q$ value >= threshold}
    selectedProduct = randomly select a product from RCL
    if selectedProduct fits in the suitcase and does not exceed weight limit then
        Add selectedProduct to solution
        Update suitcase dimensions and weight capacity
    end if
end while
return solution
```

5 Tuning Local Search

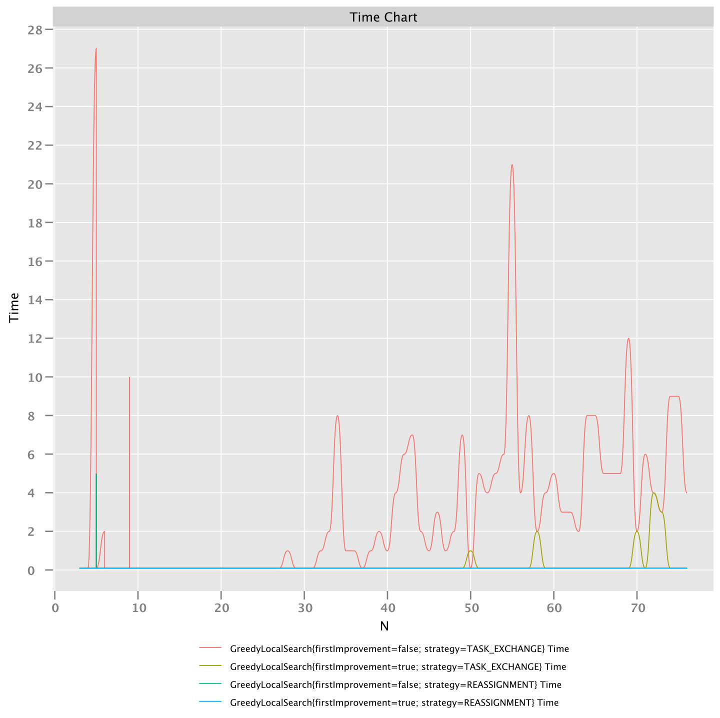

Figure 1: Time Parameters Local Search

The results show a clear trade-off between execution time and solution quality. The Best Improvement strategy, especially with Task Exchange, produced the highest objective values but took much longer to run. On the other hand, Task Reassignment methods were the fastest but resulted in slightly lower objective values.

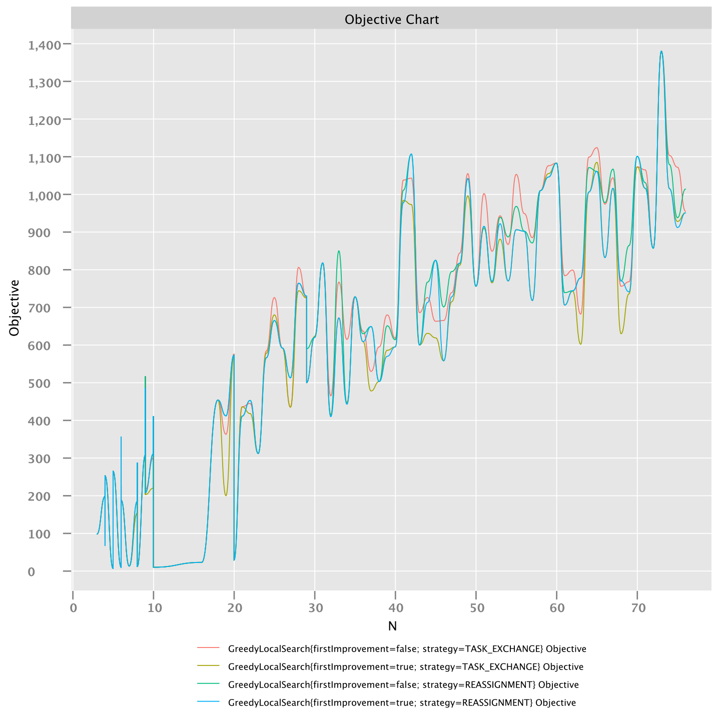

Figure 2: Objective Parameters Local Search

The First Improvement strategy, while faster, often settled for less optimal solutions compared to the Best Improvement strategy because it stops early upon finding the first better neighbor.

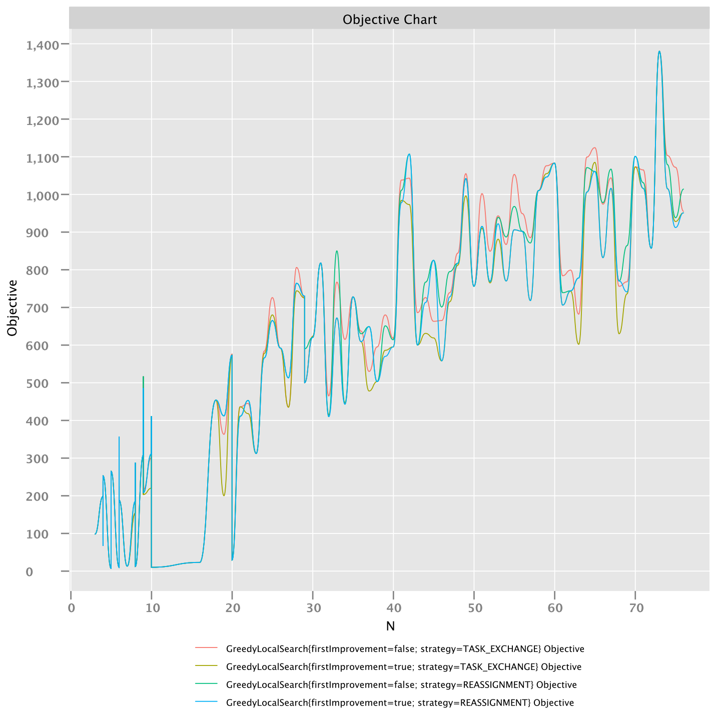

Figure 3: Objective Parameters Local Search

From the experiments, we can draw the following conclusions:

- **Best Improvement + Task Exchange**: Best for scenarios where solution quality is most important, and higher execution times are acceptable.
- **First Improvement + Task Reassignment**: Best for scenarios requiring quick solutions with reasonably good quality.
- **Best Improvement + Task Reassignment**: Offers a balanced trade-off, providing good quality solutions within an acceptable time frame.
- **First Improvement + Task Exchange**: Not recommended due to less optimal solutions and slower performance compared to task reassignment.

Conclusion

The Best Improvement strategy with Task Exchange is best for scenarios focusing on solution quality, despite longer execution times. On the other hand, the First Improvement strategy with Task Reassignment is preferable for scenarios needing quick solutions, balancing speed and solution quality well. The Best Improvement strategy with Task Reassignment offers a middle ground, providing good solutions within reasonable times. Therefore, the choice of strategy should match the specific needs for execution time and solution quality of the application. In this study, First Improvement and Reassignment were chosen to minimize time loss while maintaining satisfactory solution quality.

6 GRASP Parameter Tuning

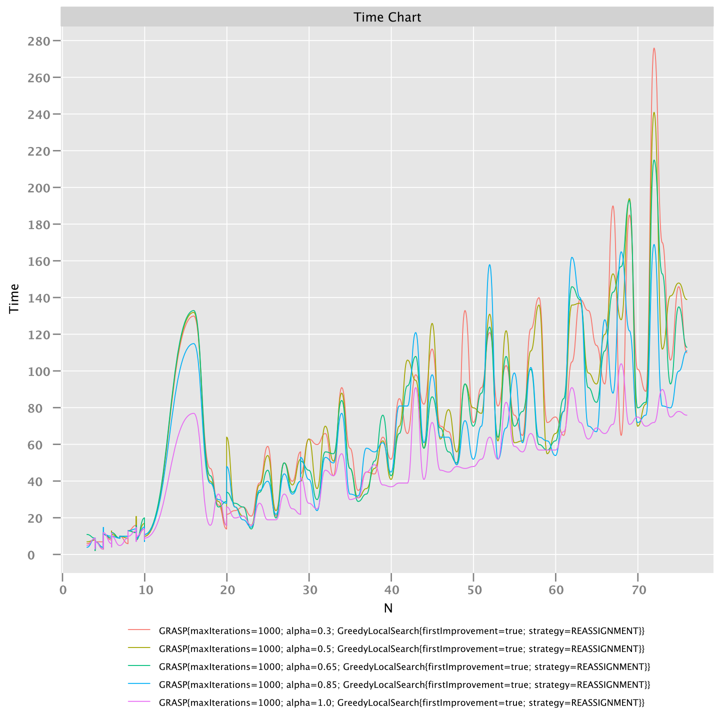

Figure 4: Execution Time for Different Alpha Values in GRASP

The results indicate that the execution time for the GRASP algorithm is generally faster with higher alpha values. Specifically, alpha values closer to 1 (corresponding to purely random selection) resulted in the fastest execution times, while lower alpha values, like 0.3, resulted in slower execution times.

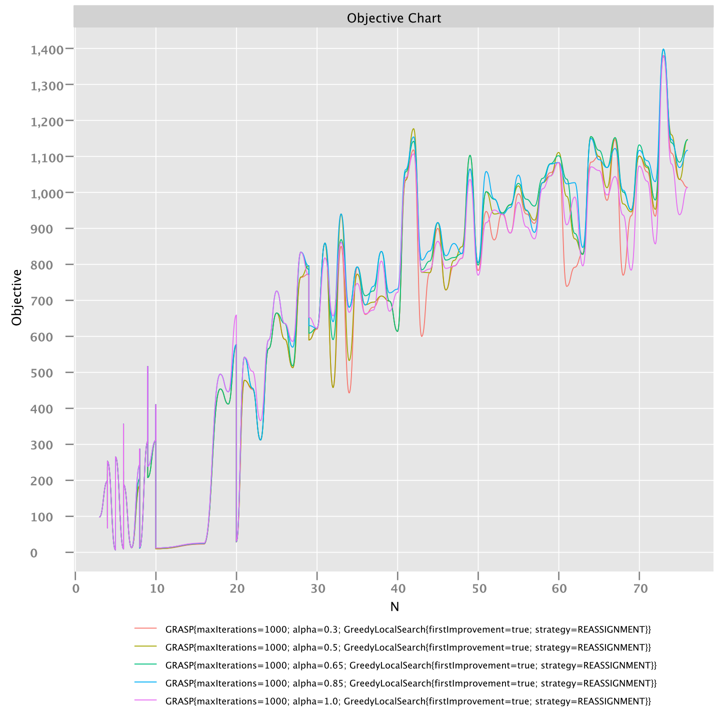

Figure 5: Objective Values for Different Alpha Values in GRASP

In terms of objective values, both the lowest (alpha = 0.3) and the highest (alpha = 1) alpha values produced the lowest objective values. The intermediate alpha values, particularly alpha = 0.65, resulted in the best objective values, suggesting a balanced approach between greedy and random selection generates better results.

Conclusion

The experiments reveal that using a higher alpha value in the GRASP algorithm speeds up the execution time, but both very low and very high alpha values result in lower objective values. The best solution quality was obtained with an alpha value of 0.65, which strikes a balance between greedy and random selection. For practical applications, an alpha value around 0.65 is recommended to achieve a good balance between execution time and solution quality.

7 Overall Performance Graphs

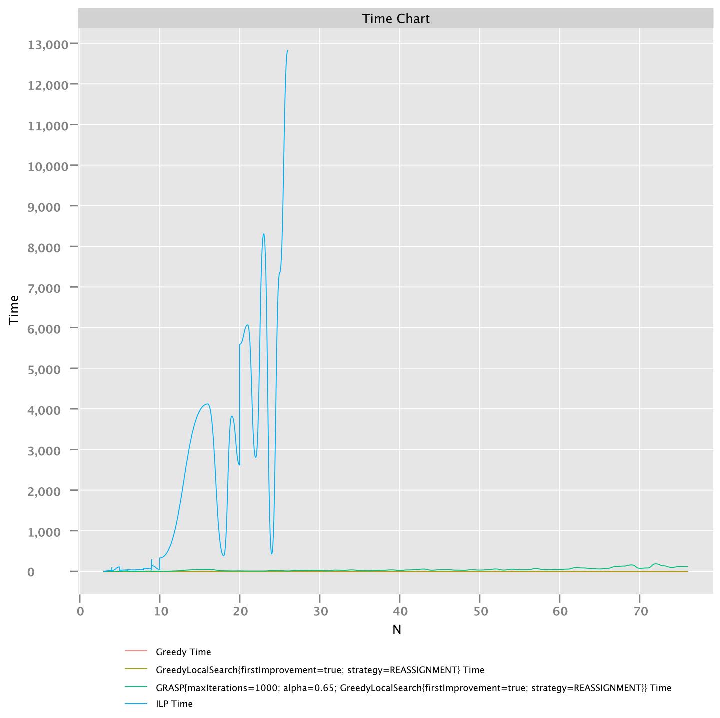

Figure 6: Execution Time for All Algorithms

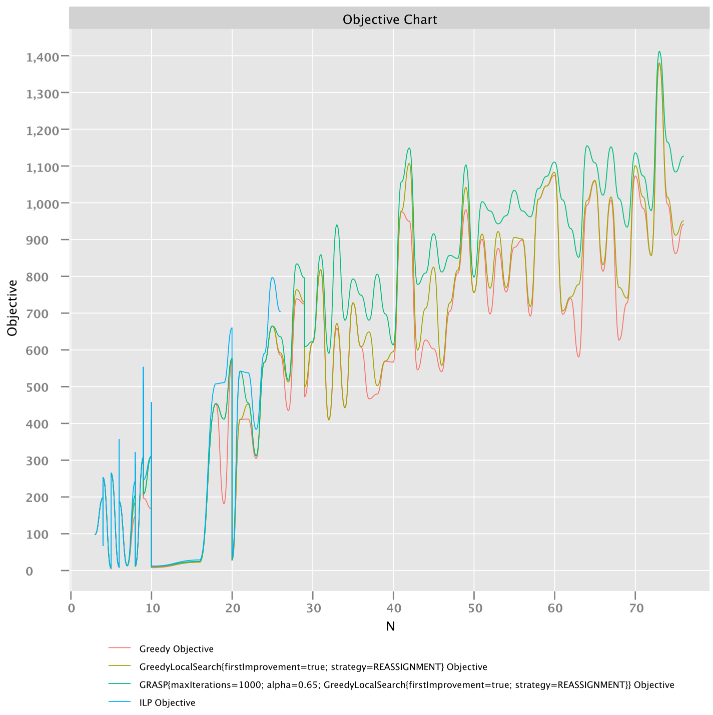

Figure 7: Objective Values for All Algorithms

8 Conclusion

The overall performance analysis reveals significant differences among the algorithms in both execution time and solution quality. The ILP algorithm always finds the best solution because it explores all feasible solutions. But the trade-off is that it consistently performed the worst, failing to provide a solution for instances where $N > 27$ within a 30-minute time-frame. This limitation makes ILP unsuitable for larger problem sizes.

In contrast, the Greedy algorithm provided quick solutions but often at the expense of solution quality. The addition of Local Search to the Greedy algorithm improved the solution quality considerably, particularly with the Best Improvement strategy combined with Task Exchange, although it required more time to execute.

The GRASP algorithm, with its different alpha parameters, demonstrated a good balance between execution time and solution quality. While higher alpha values (purely random selection) resulted in faster execution times, an intermediate alpha value, particularly around 0.65, yielded the best objective values. This suggests that a balanced approach between greedy and random selection is most effective.

In summary, for small to medium-sized problems where execution time is critical, the Greedy algorithm with Task Reassignment is preferable. For scenarios requiring high-quality solutions, the Best Improvement strategy with Task Exchange or GRASP with an alpha value of 0.65 offers the best results. ILP is impractical for larger instances due to its prohibitive execution times.

9 How to run

- Install IBM CPLEX
- Install Java 22
- Install Maven

Those are the commands to run it from command line without using an IDE

This command will run both OPL and Heuristic and output benchmark values in CSV.

```
cd heuristic
mvn compile exec:java -Dexec.mainClass="edu.upc.fib.ammm.Main" -Dexec.args="../opl"
```

Make plots of a run.

```
mvn compile exec:java -Dexec.mainClass="edu.upc.fib.ammm.Plot" -Dexec.args="output.csv"
```

Instance generator.
Change InstanceGenerator.java parameters and then run.

```
mvn compile exec:java -Dexec.mainClass="edu.upc.fib.ammm.InstanceGenerator"
```