UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

AMMM-MIRI

# Course Project
# Strategic Packing for a Jewelry Fair

Vincent Olesen
vincent.olesen@estudiantat.upc.edu
Joakim Svensson
joakim.svensson@estudiantat.upc.edu

Barcelona, Spain
May 17, 2024

## 1 Introduction

In this project, we consider the problem of packing square boxes with a fixed price, weight, and size into a suitcase, a rectangle of fixed size with a maximum weight capacity. Our objective is to maximize the price of the items in the suitcase under the maximum weight contract and ensure no overlap. Conceptually, this problem is a generalization of the classic knapsack problem and the 2D bin packing problem, both of which are known to be NP-complete, as both these well-known problems can be formulated as specific instances.

To solve this problem, we explore two main approaches: (1) deriving optimal solutions through Mixed Integer Linear Programming (MILP) formulations and (2) seeking sub-optimal solutions via meta-heuristic techniques. Further, we examine the balance between achieving optimal solutions and the computational effort required.

## 2 Problem Description

In this section, we formalize the problem. We consider a suitcase of height $x$ and width $y$ (in millimeters) and with a maximum weight capacity $c$ (in grams). For convenience, we define $S = \{1, \ldots, x\} \times \{1, \ldots, x\}$ to be set of all suitcase cells. We consider $n$ products and let $P = \{1, \ldots, n\}$ denote the product indices. Then for each item we have

- $p_i$: 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 box (in millimeters).

Let $X$ denote the set of product chosen, subject to the weight constraint $\sum_{i \in X} w_i \leq c$ and non-overlapping constraint, then we aim to maximize $\sum_{i \in X} p_i$, that is the total value of the items in the suitcase

## 3 Methodology

In this section, we will present the methodology used to investigate the trade-offs in the optimality of solutions compared to the computational feasibility of MILP and meta-heuristic techniques. The methodology is outlined in fig. 1, and an in-depth description of the approach is given in the following.

### 3.1 Instance Generation

In this study, we focus on generating instances for three specific types of problems: a 2D bin packing problem, a knapsack problem, and a hybrid model that combines elements of both, aiming to replicate the reference instances.

**Bin Packing** We configure a suitcase of dimensions $n \times 2n$ and populate it with a randomized arrangement of squares that fill the space. For each square, the weight $w_i$, price $p_i$, and side length $s_i$ are defined as $s_i^2$, ensuring that the optimal solution yields a total objective value of $2n^2$.

**Knapsack** We define a suitcase of dimensions $n \times 1$, containing $m > n$ squares, each with side length $s_i = 1$ and weight $w_i = 1$. The price $p_i$ for each square is randomly assigned following a uniform distribution between 0 and 100.

**Mix** We set up a $n \times 2n$ suitcase with a pre-determined random arrangement of squares that fill the space. This scenario includes $m > n$ squares, where each square's weight $w_i$ is $s_i^2$ and its price $p_i$ follows a uniform distribution from 0 to 100.

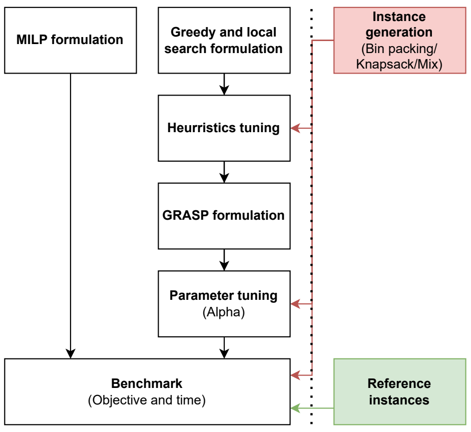

MILP formulation
Greedy and local search formulation
Instance generation (Bin packing/ Knapsack/Mix)
Heurristics tuning
GRASP formulation
Parameter tuning (Alpha)
Benchmark (Objective and time)
Reference instances

**Figure 1:** *Flow diagram of the proposed methodology for comparing the performance of MILP and metaheuristic techniques.*

These instances allow for examining solution strategies across varying dimensions and under the emphasis of different constraints.

## 3.2 MILP Implementation

We consider two different formalization of the problem, arising from different ways of enforcing non-overlap of items. We hypothesize, that the formalization have different performance characteristics, in e.g. the special cases of the knapsack and 2D bin-packing problems.

### 3.2.1 Implementation I

In this formalization of the problem, we base the formalization around the following principle: *If two items are in the suitcase, one should be to the left or below the other - or vice versa.*

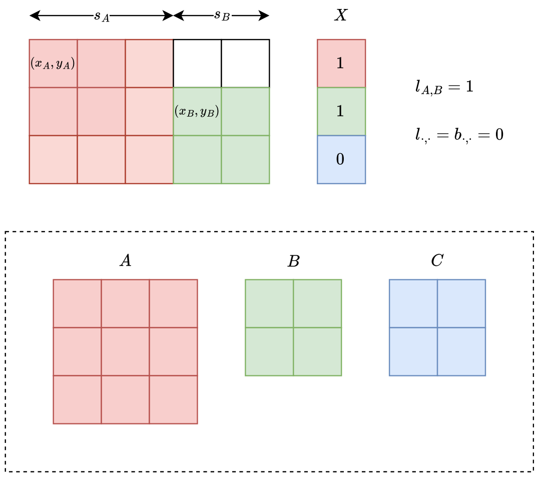


**Figure 2:** *Illustration of Formalization I - Depicts the principle that if two items are placed in the suitcase, one must be positioned to the left or below the other - or vice versa.*

We consider the following decision variables:

- $X_i$: Binary variable where $X_i = 1$ if $i$-th product is chosen, 0 otherwise.
- $x_i, y_i$: Coordinates of the bottom-left corner of the $i$-th product's box within the suitcase.

Then the problem can be stated as follows.

$$
\begin{aligned}
\text{max} \quad & \sum_{k \in P} p_k \cdot X_k & & (1) \\
\text{s.t.} \quad & \sum_{(i,j) \in S,\ k \in P} w_k \cdot X_i \leq c & & (2) \\
& x_k \geq 1 & \forall k \in P \quad & (3) \\
& y_k \geq 1 & \forall k \in P \quad & (4) \\
& x_k + s_k \leq x & \forall k \in P \quad & (5) \\
& y_k + s_k \leq y & \forall k \in P \quad & (6) \\
X_i = X_j = 1 \implies & (x_i + s_i \leq x_j) & \forall i \in P, \forall j \in P : i < j \quad & (7) \\
& \vee (x_j + s_j \leq x_i) & & \\
& \vee (y_i + s_i \leq y_j) & & \\
& \vee (y_j + s_j \leq y_i) & &
\end{aligned}
$$

To elaborate, eqn. 2 expresses the maximum weight constraint, eqn. 3, 4, 5 and 6 constraints the corner position for all products, such that the product is within the suitcase, and eqn. 7 expresses the non-overlapping constraint.

**Linearization of Implication** We will now linearize eqn. 7, such that we have a mixed integer linear program (MILP).

In addition to the above decision variables, we will consider decision variables $l_{i,j}$ and $b_{i,j}$ for all pairs of distinct products $\forall i \in P, \forall j \in P : i < j$. Here, $l_{i,j}$ and $b_{i,j}$ are equal to one if product $i$ is located respectively to the left or below of product $j$. Then the last constraint can be replaced by the following linear constraints.

$$
\begin{aligned}
x_i + s_i &\leq x_j + M \cdot (1 - l_{i,j}) & \forall i \in P, \forall j \in P : i < j \\
x_j + s_j &\leq x_i + M \cdot (1 - l_{j,i}) & \forall i \in P, \forall j \in P : i < j \\
y_i + s_i &\leq y_j + M \cdot (1 - b_{i,j}) & \forall i \in P, \forall j \in P : i < j \\
y_j + s_j &\leq y_i + M \cdot (1 - b_{j,i}) & \forall i \in P, \forall j \in P : i < j \\
l_{i,j} + b_{i,j} + l_{j,i} + b_{j,i} &\geq X_i + X_j - 1 & \forall i \in P, \forall j \in P : i < j
\end{aligned}
$$

This is a application of the big-M method. We note, that tight (optimal) bounds are implemented in code, but $M$ was chosen to deliver the point more clearly.

## 3.3 Implementation II

In this formalization of the problem, we base the formalization around the following principle: *Two items can not occupy the same cell in the suitcase.*

We consider the following decision variables.

- $x_{i,j,k}$: Binary variable, is one if product $k \in P$ has the bottom left corner in position $(i, j)$, zero otherwise.
- $y_{i,j,k}$: Binary variable, is one if product $k \in P$ occupies position $(i, j)$.

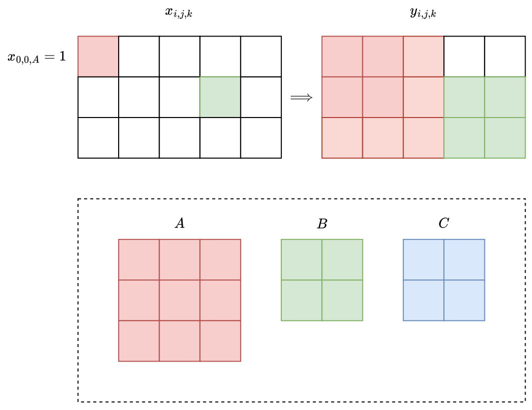


**Figure 3:** *Illustration of Formalization II - Shows the principle that no two items can occupy the same cell in the suitcase.*

Figure 3, illustrates the link between $x_{i,j,k}$ and $y_{i,j,k}$.

Then the problem can be stated as MILP as follows.

$$\max \quad \sum_{(i,j)\in S, k\in P} p_k \cdot x_{i,j,k} \tag{8}$$

$$\text{s.t.} \quad \sum_{(i,j)\in S,\ k\in P} w_k \cdot x_{i,j,k} \leq c \tag{9}$$

$$\sum_{k\in P} x_{i,j,k} \leq 1 \qquad \forall (i,j) \in S \tag{10}$$

$$\sum_{k\in P} y_{i,j,k} \leq 1 \qquad \forall (i,j) \in S \tag{11}$$

$$\sum_{(i,j)\in S} x_{i,j,k} \leq 1 \qquad \forall k \in P \tag{12}$$

$$x_{i,j,k} = 0 \qquad \forall k \in P, \forall (i,j) \in O_k \tag{13}$$

$$x_{i,j,k} \leq y_{i',j',k} \qquad \forall (i,j) \in S, \forall k \in P, \forall (i',j') \in S_{i,j,k} \tag{14}$$

where $S_{i,j,k} = S \cap \{i, .., i+s_k-1\} \times \{j, \ldots, j+s_k-1\}$ denotes the cells occupied by product $k \in P$, when placed at $(i,j)$ and $O_k = \{(i,j) \mid i+s_k-1 > x \vee j+s_k-1 > y\}$ denotes invalid corner placements for product $k \in P$, that is product $k$ would be out-of-bounds.

To elaborate, eqn. 9 expresses the maximum weight constraint, eqn. 10 and 11 express the constraint of at most one corner and product per cell respectively, eqn. 12 expresses the constraint of at most one corner per product, eqn. 13 constrains infeasible corner placements and finally eqn. 14 expresses the non-overlapping constraint.

In this second formalization, we established a relationship between the decision variables $x_{i,j,k}$ and $y_{i,j,k}$ through implications, where $x_{i,j,k} = 1$ implies $y_{i',j',k} = 1$ for all $(i',j')$ within the bounds defined by $s_k$, the side length of product $k$.

## 3.4 Metaheuristic Implementation

This section presents our proposed heuristic approaches, specifically a greedy search, a greedy search with local search and a Greedy Randomized Adaptive Search Procedure (GRASP) [1], to the problem and explains the rationale behind our design decisions.

In all our metaheuristic implementations, we evaluate a candidate set by selecting the best-performing candidate according to the greedy function $q$. Initially, we considered a greedy function $q((p,(x,y)),S)$, where $p$ is a product and $S$ is a partial solution. This function assessed the quality of placing product $p$ at coordinates $(x,y)$ within the partial solution $S$. However, preliminary tests indicated that using a bin-packing heuristic for product placement—thereby making $q$ independent of the specific placement of product $p$—yielded better results. This approach also proved to be more efficient, as formulating computationally efficient heuristics for placement was challenging. For instance, optimizing for the largest square of zeros in a matrix is a well-known $\mathcal{O}(n^2)$ problem.

Consequently, in our implementation, the greedy function does not include product placement. Instead, we employ the `MaxRect` heuristic, as described in [2]. This heuristic has demonstrated empirical effectiveness in 2D bin packing and is used to determine the placement of products.

Additionally, we treat the greedy function $q$ as a tunable parameter in our metaheuristic implementations, allowing for further optimization in subsequent stages.

### 3.4.1 Greedy Algorithm

The pseudocode for the greedy algorithm is presented in Algorithm 1. In this implementation, we can combine the creation of the candidate set with the extraction of the candidate *maximizing* the greedy function. Explicit generation of the candidate set is presented in Algorithm 2, which is used in the local search and the GRASP implementations. Note that for convience, we exclude infeasible (partial) solutions, already in the generation of the candidate set.

**Algorithm 1** Pseudocode for the Greedy Search

```
function GreedySearch(products, q)
    suitcase ← EmptySuitcase($x, y$)
    for product in Sort(products, $q$) do                                  ▷ Sort by $q(\cdot,\cdot)$
        if suitcase.weight + product.weight $\leq c$ then                ▷ Check weight limit
            if CanFit(product, suitcase) then          ▷ Try to add square w. MaxRect algo.
                suitcase ← AddProduct(suitcase, product)
            end if
        end if
    end for
    return suitcase
end function
```

### 3.4.2 Local Search

The pseudocode for the local search is presented in Algorithm 3. In this implementation, we utilize a first-improvement strategy. To generate neighborhood solutions, we perform the following steps: (1) we remove a product from the suitcase, (2) optionally, we repack the suitcase, and (3) we add each product that is not in the suitcase.

The following sections will discuss the influence of repacking solutions in local search.

### 3.4.3 GRASP

The pseudocode for the GRASP implementation is presented in Algorithm 4.

**Algorithm 2** Pseudocode for the creation of the candidate set

```
function CandidateSet($products$, $suitcase$)
    $candidates \leftarrow \{\}$
    for $product \in products$ do
        if $product \notin suitcase$ then
            if $suitcase.weight + product.weight \leq c$ then
                if CanFit($product$, $suitcase$) then ▷ Try to add square w. MaxRect algo.
                    $candidate \leftarrow$ AddProduct($suitcase, product$)
                    $candidates \leftarrow candidates \cup \{candidate\}$
                end if
            end if
        end if
    end for
    return $candidates$
end function
```

**Algorithm 3** Pseudocode for the Local Search algorithm

```
function LocalSearch($products$, $suitcase$, $repack$)
    $improved \leftarrow$ false
    while $\neg improved$ do
        $improved \leftarrow$ false
        for $neighbour \in$ NeighborHoodSolutions($products, suitcase$) do
            if $neighbour.value > suitcase.value$ then
                $suitcase \leftarrow neighbour$
                $improved \leftarrow$ true
                break
            end if
        end for
    end while
    return $suitcase$
end function
function NeighborhoodSolutions($products$, $suitcase$, $repack$)
    $N \leftarrow \{\}$
    for $productToRemove \in products$ do
        $neighborhoodSuitcase \leftarrow$ RemoveProduct($suitcase, productToRemove$)
        if $repack$ then
            $neighborhoodSuitcase \leftarrow$ Repack($neighborhoodSuitcase$)
        end if
        $N \leftarrow N \cup$ CandidateSet($products, neighborhoodSuitcase$)
    end for
    return $N$
end function
```

## 3.5 Meta-heuristic Tuning

In this section, we will elaborate on the methodology of choosing a greedy function, $q$, and on parameter tuning of GRASP, specifically tuning of $\alpha$.

### 3.5.1 Greedy Function Selection

For each type of problem, the optimal greedy function was chosen wrt. the objective value of the returned solution with greedy search - as mentioned earlier, the greedy function is independent of the position of the product, as it is placed using the `MaxRect` algorithm. The

**Algorithm 4** Pseudocode for the GRASP Algorithm

```
function GRASP($products$, $q$, $alpha$)
    $suitcase \leftarrow$ EmptySuitcase($x, y$)
    while true do
        $candidates \leftarrow$ CandidateSet($products, suitcase$)
        if Empty($candidates$) then
            break
        end if
        $q_{min} \leftarrow \min\{q(p, suitcase) \mid p \in candidates\}$
        $q_{max} \leftarrow \max\{q(p, suitcase) \mid p \in candidates\}$
        $RCL_{max} \leftarrow \{p \in candidates \mid q(p, suitcase) \geq q_{max} - \alpha(q_{max} - q_{min})\}$
        $product \leftarrow$ ChooseRandom($RCL_{max}$)
        $suitcase \leftarrow$ AddSquare($suitcase, square$)
    end while
    $suitcase \leftarrow$ LocalSearch($products, suitcase$)
    return $suitcase$
end function
```

best-performing greedy function on the mix instance is chosen to benchmark the solver on the reference instances.

### 3.5.2 Parameter Tuning of GRASP

For tuning the $\alpha$ parameter, we considered $\alpha \in \{0, 0.1, \ldots, 1.0\}$ across the same problem types of problem instances as above, with a 300-second maximum runtime and a 60-second maximum incumbent update interval. For each type of problem, the optimal $\alpha$ was chosen wrt. the mean objective value of the returned solution with GRASP across five runs. The optimal $\alpha$ value on the mixed problem instance, was used for benchmarking the solver against reference instances.

## 3.6 Benchmarking

To evaluate the performance of the algorithms, we assessed both the computational time and the objective function values for solutions derived from the reference and generated instances across three problem types: bin packing, knapsack, and mix, with instance sizes $n = 50, 100, 200$. We utilize a memory limit of 8 GB.

The MILP models were developed in `OPL` and solved using `CPLEX 22.11` with a time limit of 1800 seconds. For the GRASP algorithm, the execution was also limited to 1800 seconds, with a maximum gap of 300 seconds allowed for incumbent updates.

# 4 Results

## 4.1 Greedy Function Selection

The objective values for the solutions obtained from the greedy search for each considered greedy function are provided in Table 1.

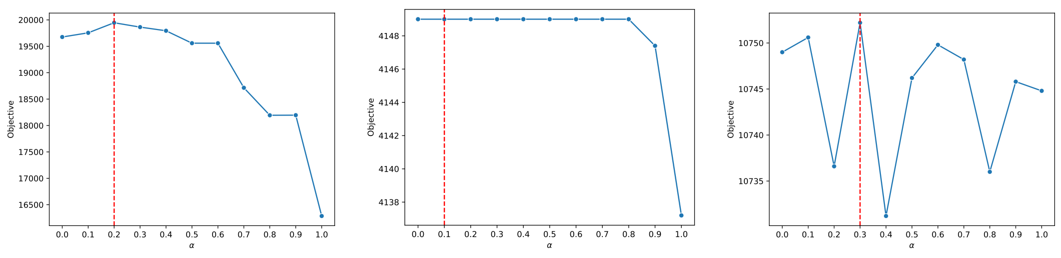
**Table 1:** *Performance of various greedy functions, $q(\cdot,\cdot)$, using greedy Search across generated instances.*

| Instance | Greedy function | Objective |
|---|---|---|
| **Bin packing** ($n = 50$) | $p_i$ | **4973** |
| | $s_i$ | **4973** |
| | $w_i$ | **4973** |
| | $\frac{p_i}{s_i}$ | **4973** |
| | $\frac{p_i}{w_i}$ | 3556 |
| | $\frac{p_i}{w_i \cdot s_i}$ | 2656 |
| | $\frac{p_i}{w_i \cdot s_i^2}$ | 2656 |
| **Knapsack** ($n = 50$) | $p_i$ | 1927 |
| | $s_i$ | 1472 |
| | $w_i$ | 987 |
| | $\frac{p_i}{s_i}$ | 1927 |
| | $\frac{p_i}{w_i}$ | **2011** |
| | $\frac{p_i}{w_i \cdot s_i}$ | **2011** |
| | $\frac{p_i}{w_i \cdot s_i^2}$ | **2011** |
| **Mix** ($n = 50$) | $p_i$ | **5137** |
| | $s_i$ | 4945 |
| | $w_i$ | 4141 |
| | $\frac{p_i}{s_i}$ | 5083 |
| | $\frac{p_i}{w_i}$ | 5136 |
| | $\frac{p_i}{w_i \cdot s_i}$ | 5102 |
| | $\frac{p_i}{w_i \cdot s_i^2}$ | 5090 |

In Table 1, we can observe that for the bin packing instance, the optimal greedy function is $q(product_i, \cdot) = s_i$ (where $s_i = w_i = p_i$ for this instance). For the knapsack instance, the optimal greedy function is $q(product_i, \cdot) = \frac{p_i}{w_i}$. In the case of the mix instance, the optimal greedy function is $q(product_i, \cdot) = p_i$.

## 4.2 Parameter Tuning of GRASP

The objective values for the solutions obtained from GRASP *without repacking in local search* for each considered $\alpha$ are provided in Figure 4.



**(a)** *Bin packing ($n = 50$)* **(b)** *Knapsack ($n = 50$)* **(c)** *Mix ($n = 50$)*

**Figure 4:** *Mean performance of various $\alpha$ values, using GRASP across generated instances.*

In the results shown in Fig. 4a, we observed that the optimal $\alpha$ for the bin packing instance is $\alpha = 0.2$. In Fig. 4b, we noticed multiple optimal $\alpha$ values for the knapsack instance. We arbitrarily selected $\alpha = 0.1$ from the optimal $\alpha$ values. In Fig. 4c, we found that the optimal $\alpha$ for the mixed instance is $\alpha = 0.1$. It is important to note that for the knapsack and bin-packing instances, we observed only minor variations in performance across different $\alpha$ values, as all solutions were optimal or near-optimal (as confirmed by MILP solvers). The tuning was repeated with repacking, resulting in fewer iterations within the time limit and, consequently, significantly worse performance.

## 4.3 Comparisons of Algorithms

The performance of various solvers across generated instances is given in Table 2. The performance of various solvers across reference instances is given in Table 3 in the appendix.

**Table 2:** *Performance of various solvers across generated instances.*

| **Instance** | **Solver** | **Objective** | **Time (s)** |
|---|---|---:|---:|
| **Bin packing** ($n = 50$) | Greedy | 4952 | 0.08 |
| | Greedy w. local search | 4952 | 0.17 |
| | GRASP | **4968** | 301.49 |
| | MILP I | 4911[a] | 1800.10[a] |
| | MILP II | 1739[a] | 1811.23[a] |
| **Bin packing** ($n = 100$) | Greedy | 19568 | 0.09 |
| | Greedy w. local search | 19568 | 0.53 |
| | GRASP | **19880** | 547.41 |
| | MILP I | 17269[a] | 1804.89[a] |
| | MILP II | -[b] | -[b] |
| **Bin packing** ($n = 200$) | Greedy | 79100 | 0.81 |
| | Greedy w. local search | 79100 | 1.83 |
| | GRASP | **79678** | 747.56 |
| | MILP I | 36460[a] | 1801.10[a] |
| | MILP II | -[b] | -[b] |
| **Knapsack** ($n = 50$) | Greedy | 2105 | 0.01 |
| | Greedy w. local search | 2105 | 0.01 |
| | GRASP | 2105 | 300.00 |
| | MILP I | **2106** | 0.22 |
| | MILP II | **2106** | 0.19 |
| **Knapsack** ($n = 100$) | Greedy | 4606 | 0.01 |
| | Greedy w. local search | 4606 | 0.03 |
| | GRASP | **4611** | 300.07 |
| | MILP I | **4611** | 3.42 |
| | MILP II | **4611** | 0.14 |
| **Knapsack** ($n = 200$) | Greedy | 9905 | 0.01 |
| | Greedy w. local search | 9905 | 0.05 |
| | GRASP | **9906** | 300.83 |
| | MILP I | **9906** | 27.64 |
| | MILP II | **9906** | 1.42 |
| **Mix** ($n = 50$) | Greedy | 5982 | 0.01 |
| | Greedy w. local search | 5982 | 0.11 |
| | GRASP | 5994 | 602.20 |
| | MILP I | **5998**[a] | 1800.51[a] |
| | MILP II | -[b] | -[b] |

**Table 2:** *Performance of various solvers across generated instances. (continued)*

| Instance | Solver | Objective | Time (s) |
|---|---|---|---|
| **Mix** ($n = 100$) | Greedy | **7230** | 0.03 |
| | Greedy w. local search | **7230** | 0.74 |
| | GRASP | **7230** | 300.25 |
| | MILP I | **7230**[a] | 1809.09[a] |
| | MILP II | -[b] | -[b] |
| **Mix** ($n = 200$) | Greedy | **16907** | 0.39 |
| | Greedy w. local search | **16907** | 16.99 |
| | GRASP | **16907** | 302.90 |
| | MILP I | 11007[a] | 1800.16[a] |
| | MILP II | -[b] | -[b] |

[a] Stopped due to time limit
[b] Stopped due to memory limit

Based on the data presented in Table 2, the second MILP formulation encountered memory limit issues, causing it to stop prematurely on all mixed instances and all but the smallest bin-packing instance. However, the second MILP formulation outperformed the first MILP formulation across all knapsack instances. The GRASP algorithm performed best in all instances except the smallest mixed instance. The bin packing problem was designed so that the objective value of the optimal solution was $2n^2$. The objective value of the GRASP solutions is within 99.36%, 99.4%, and 99.6% of the optimal solution for bin packing with $n = 50$, $n = 100$, and $n = 200$ respectively.

## 5 Discussion & Conclusion

We analyzed the results of our work and summarized the key findings as follows:

**The MILP formulations exhibited different performance**. The model that used the geometric interpretation of product overlap performed better on geometric problems such as bin packing and mixing, while the binary formulation performed better on knapsack problems. However, it exceeded the memory limit on almost all remaining instances, likely due to the larger search tree in the branch and cut procedure caused by the many more binary variables.

**Metaheuristic techniques proved efficient**, with the GRASP implementation being the best-performing solver for all but one generated instance and for many problems reaching the global optimum.

**It is essential to tune metaheuristic techniques to the specific problem**. Our tuning of the heuristics and GRASP parameters showed that the heuristics and parameters that performed well for one type of instance (e.g., bin packing) did not necessarily perform well in others (e.g., knapsack or mixing).

**Faster GRASP implementation had better performance than clever heuristics**. Faster GRASP implementations outperformed more complex heuristics. We experimented with repacking strategies and larger candidate sets using a greedy function that considered the remaining space after placing a product. These implementations were significantly slower, leading to fewer iterations of the GRASP constructive and local search phases and, consequently, worse performance.

## References

[1] Thomas A Feo and Mauricio G.C Resende. "A probabilistic heuristic for a computationally difficult set covering problem". en. In: *Operations Research Letters* 8.2 (Apr. 1989), pp. 67–71. ISSN: 01676377. DOI: `10.1016/0167-6377(89)90002-3`. URL: `https://linkinghub.elsevier.com/retrieve/pii/0167637789900023` (visited on 05/14/2024).

[2] Jukka Jylänki. "A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing." In: (Feb. 2010).

# Appendix

**Table 3:** *Performance of various solvers across reference instances.*

| Instance | Algorithm | Best solution | Time (s) |
|---|---|---:|---:|
| **project.0** | Greedy | 7 | 0.00 |
| | Greedy w. local search | 7 | 0.00 |
| | GRASP | 7 | 300.00 |
| | MILP I | **8** | 0.00 |
| | MILP II | **8** | 0.01 |
| **project.1** | Greedy | 11 | 0.00 |
| | Greedy w. local search | 11 | 0.00 |
| | GRASP | 11 | 300.00 |
| | MILP I | 11 | 0.01 |
| | MILP II | 11 | 0.00 |
| **project.2** | Greedy | 11 | 0.00 |
| | Greedy w. local search | 11 | 0.00 |
| | GRASP | 11 | 300.00 |
| | MILP I | 11 | 0.01 |
| | MILP II | 11 | 0.00 |
| **project.3** | Greedy | 10 | 0.00 |
| | Greedy w. local search | 10 | 0.00 |
| | GRASP | 10 | 300.00 |
| | MILP I | 10 | 0.00 |
| | MILP II | 10 | 0.00 |
| **project.4** | Greedy | 16 | 0.00 |
| | Greedy w. local search | 16 | 0.00 |
| | GRASP | 16 | 300.00 |
| | MILP I | 16 | 0.01 |
| | MILP II | 16 | 0.00 |
| **project.5** | Greedy | 15 | 0.00 |
| | Greedy w. local search | 15 | 0.00 |
| | GRASP | 15 | 300.00 |
| | MILP I | 15 | 0.01 |
| | MILP II | 15 | 0.00 |
| **project.6** | Greedy | 16 | 0.00 |
| | Greedy w. local search | 16 | 0.00 |
| | GRASP | **17** | 300.00 |
| | MILP I | **17** | 0.01 |
| | MILP II | **17** | 0.01 |
| **project.7** | Greedy | 12 | 0.00 |
| | Greedy w. local search | 12 | 0.00 |
| | GRASP | 12 | 300.00 |
| | MILP I | **13** | 0.04 |
| | MILP II | **13** | 0.01 |
| **project.8** | Greedy | 34 | 0.00 |
| | Greedy w. local search | 34 | 0.00 |
| | GRASP | **35** | 304.56 |
| | MILP I | **35** | 0.76 |

**Table 3:** *Performance of various solvers across reference instances. (continued)*

| Instance | Algorithm | Best solution | Time (s) |
|---|---|---|---|
| | MILP II | **35** | 0.47 |
| **project.9** | Greedy | 29 | 0.00 |
| | Greedy w. local search | 29 | 0.01 |
| | GRASP | 29 | 300.06 |
| | MILP I | **30** | 0.18 |
| | MILP II | 26[a,b] | 4393.87[a,b] |

[a] Stopped due to time limit
[b] Stopped due to memory limit