A 2D Bin-packing and Knapsack problem

Algorithmic Methods for Mathematical Models

Vincent Olesen   Joakim Svensson

Universitat Politècnica de Catalunya – BarcelonaTech

Table of contents

1. Introduction
2. Methodology
3. Mixed Integer Linear Programming Approaches
4. Metaheuristc Approaches
5. Results
6. Discussion and Conclusion

Intro

Problem description

Dimensions	$x \times y$ (in millimeters)	**Price**	p_i (in euros)
Capacity	c (in grams)	**Weight**	w_i (in grams)
Products	n	**Side**	s_i (in millimeters)

Methodology

Methodology

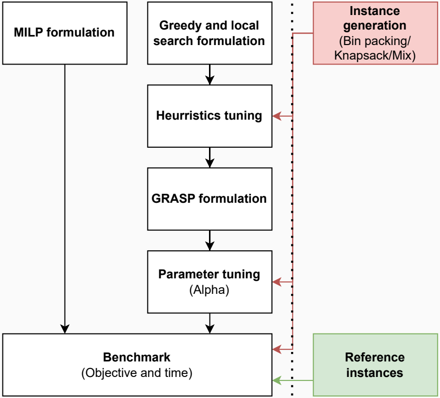

Figure 1: Proposed methdology

Instance Generation

We consider three types of generated instances

- Bin packing (designed with optimal objective $2n^2$)
- Knapsack
- Mix (mimics reference instances)

Mixed Integer Linear Programming Approaches

Plan-of-attack

Two ideas on how to express the non-overlapping constraint

Plan-of-attack

Two ideas on how to express the non-overlapping constraint

- If we select products $i, j \in P$ then, product i is to the left or below product j – or vice verse.

Plan-of-attack

Two ideas on how to express the non-overlapping constraint

- If we select products $i, j \in P$ then, product i is to the left or below product j – or vice verse.
- Or, each suitcase cell can at most be occupied by one product

MILP I - Plan-of-attack

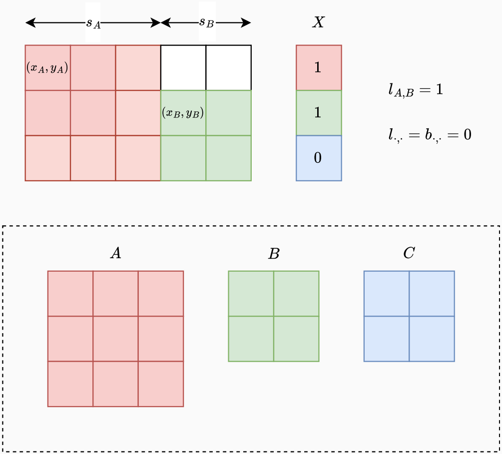

Figure 2: Non-overlapping constraint using a geometric interpretation

MILP I - Formulation

- X_i, binary, is one if product $i \in P$ is chosen, otherwise zero.
- (x_i, y_i), integers, coordinate of the bottom left corner of product $i \in P$

MILP I - Formulation (continued)

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

MILP I - Linearization

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

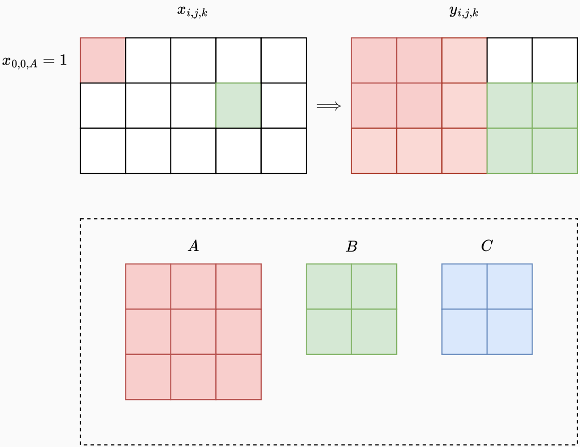

Figure 3: Non-overlapping constraint using an assignment interpretation

MILP II - Formulation

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

MILP II - Formulation (continued)

$$
\begin{aligned}
\max \quad & \sum_{(i,j)\in S, k\in P} p_k \cdot x_{i,j,k} && && (8)\\
\text{s.t.} \quad & \sum_{(i,j)\in S,\ k\in P} w_k \cdot x_{i,j,k} \leq c && && (9)\\
& \sum_{k\in P} x_{i,j,k} \leq 1 && \forall (i,j) \in S && (10)\\
& \sum_{k\in P} y_{i,j,k} \leq 1 && \forall (i,j) \in S && (11)\\
& \sum_{(i,j)\in S} x_{i,j,k} \leq 1 && \forall k \in P && (12)\\
& x_{i,j,k} = 0 && \forall k \in P, \forall (i,j) \in O_k && (13)\\
& x_{i,j,k} \leq y_{i',j',k} && \forall (i,j) \in S, \forall k \in P && (14)\\
& && \forall (i',j') \in S'_{i,j,k} &&
\end{aligned}
$$

MILP II - Formulation (continued)

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

Metaheuristics

Greedy search

Selecting the greedy function

Greedy search

Selecting the greedy function

- 🚫 Generate all product and placement pairs. Select the one that maximizes q (depends on position).

Greedy search

Selecting the greedy function

- 🚫 Generate all product and placement pairs. Select the one that maximizes q (depends on position).
- ✅ Select product that maximizes q (independent of position). Place it according to `MaxRect` algorithm[2].

Greedy Search

```
function CANDIDATESET(products, suitcase)
    candidates ← {}
    for product ∈ products do
        if product ∉ suitcase then
            if suitcase.weight + product.weight ≤ c then
                if CANFIT(product, suitcase) then          ▷ Try to add square w.
MaxRect algo.
                    candidate ← ADDPRODUCT(suitcase, product)
                    candidates ← candidates ∪ {candidate}
                end if
            end if
        end if
    end for
    return candidates
end function
```

Greedy Search

```
function GreedySearch(products, q)
    suitcase ← EmptySuitcase(x, y)
    for product in Sort(products, q) do                          ▷ Sort by q(·,·)
        if suitcase.weight + product.weight ≤ c then             ▷ Check weight limit
            if CanFit(product, suitcase) then                    ▷ Try MaxRect algo.
                suitcase ← AddProduct(suitcase, product)
            end if
        end if
    end for
    return suitcase
end function
```

Local Search - Plan of Attack

1. We remove a product from the suitcase
2. Optionally, we repack the suitcase
3. We consider new suitcase, with each product that is not in the suitcase added.

Local Search - Pseudocode

```
function LocalSearch(products, suitcase, repack)
    improved ← false
    while ¬improved do
        improved ← false
        for neighbour ∈ NeighborhoodSolutions(products, suitcase) do
            if neighbour.value > suitcase.value then
                suitcase ← neighbour
                improved ← true
                break
            end if
        end for
    end while
    return suitcase
end function
```

Local Search - Pseudocode (continued)

```
function NeighborhoodSolutions(products, suitcase, repack)
    N ← {}
    for productToRemove ∈ products do

neighborhoodSuitcase ← RemoveProduct(suitcase, productToRemove)
        if repack then
            neighborhoodSuitcase ← Repack(neighborhoodSuitcase)
        end if
        N ← N ∪ CandidateSet(products, neighborhoodSuitcase)
    end for
    return N
end function
```

GRASP

```
function GRASP(products, q, alpha)
    suitcase ← EmptySuitcase(x, y)
    while true do
        candidates ← CandidateSet(products, suitcase)
        if Empty(candidates) then
            break
        end if
        q_min ← min{q(p, suitcase) | p ∈ candidates}
        q_max ← max{q(p, suitcase) | p ∈ candidates}
        RCL_max ← {p ∈ candidates | q(p, suitcase) ≥ q_max − α(q_max − q_min)}
        product ← ChooseRandom(RCL_max)
        suitcase ← AddSquare(suitcase, square)
    end while
    suitcase ← LocalSearch(products, suitcase)
    return suitcase
end function
```

[1]

Results

Heuristics tuning

- For bin-packing, s_i was optimal
- For knapsack, $\frac{p_i}{w_i}$ was optimal
- For mix, p_i was optimal. (On the reference instances, $\frac{p_i}{s_i}$ was optimal)

GRASP tuning

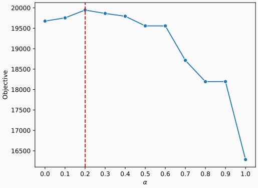

Figure 4: Bin packing

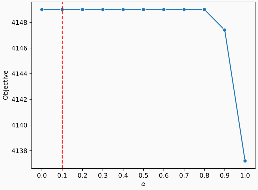

Figure 5: Knapsack

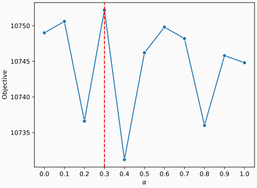

Figure 6: Mix

- Mean performance of various α values, using GRASP across generated instances ($n = 50$).

Comparisons of Algorithms - Bin Packing i

Table 1: Bin Packing: Success and Failure

	50	100	200
Greedy	✅	✅	✅
Greedy w. local search	✅	✅	✅
GRASP	✅	✅	✅
MILP I	⌛	⌛	⌛
MILP II	⌛	☠️	☠️

Comparisons of Algorithms - Bin Packing ii

Table 2: Bin Packing: Best Performance

	50	100	200
Greedy	🥈	🥈	🥈
Greedy w. local search	🥉	🥉	🥉
GRASP	🥇	🥇	🥇
MILP I	4️⃣	4️⃣	4️⃣
MILP II	5️⃣	5️⃣	5️⃣

- 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 %, 99.40% and 99.60% of the optimal solution for bin packing with $n = 50$, $n = 100$ and $n = 200$ respectively.

Comparisons of Algorithms - Knapsack i

Table 3: Knapsack: Success and Failure

	50	100	200
Greedy	✅	✅	✅
Greedy w. local search	✅	✅	✅
GRASP	✅	✅	✅
MILP I	✅	✅	✅
MILP II	✅	✅	✅

Comparisons of Algorithms - Knapsack ii

Table 4: Knapsack: Best Performance

	50	100	200
Greedy	4️⃣	4️⃣	4️⃣
Greedy w. local search	5️⃣	5️⃣	5️⃣
GRASP	🥉	🥉	🥉
MILP I	🥈	🥈	🥈
MILP II	🥇	🥇	🥇

Table 5: Mix: Success and Failure

	50	100	200
Greedy	✅	✅	✅
Greedy w. local search	✅	✅	✅
GRASP	✅	✅	✅
MILP I	⌛	⌛	⌛
MILP II	☠️	☠️	☠️

Comparisons of Algorithms - Mix ii

Table 6: Mix: Best Performance

	50	100	200
Greedy	🥉	🥇	🥇
Greedy w. local search	4️⃣	🥈	🥈
GRASP	🥈	🥉	🥉
MILP I	🥇	4️⃣	4️⃣
MILP II	5️⃣	5️⃣	5️⃣

Discussion and Conclusion

Discussion and Conclusion

Discussion and Conclusion

- The MILP formulations exhibited different performance.

Discussion and Conclusion

- The MILP formulations exhibited different performance.
- Metaheuristic techniques proved efficient.

Discussion and Conclusion

- The MILP formulations exhibited different performance.
- Metaheuristic techniques proved efficient.
- It is essential to tune metaheuristic techniques to the specific problem.

Discussion and Conclusion

- The MILP formulations exhibited different performance.
- Metaheuristic techniques proved efficient.
- It is essential to tune metaheuristic techniques to the specific problem.
- Faster GRASP implementation had better performance than clever heuristics.

T. A. Feo and M. G. Resende.
A probabilistic heuristic for a computationally difficult set covering problem.
Operations Research Letters, 8(2):67–71, Apr. 1989.

J. Jylänki.
A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing.
Feb. 2010.