Optimisation de placement spacial d'objets

Problèmes de bin-packing : approches exactes et heuristiques, enjeux de passage à l'échelle, extensions géométriques

1 Introduction et contexte

Les problèmes de bin-packing (problèmes d'empaquetage) constituent une classe de problèmes classiques d'optimisation combinatoire, dont l'objectif est de répartir un ensemble d'objets dans un nombre minimal de boîtes ou de conteneurs, tout en respectant des contraintes de capacités imposées.

Ces problématiques trouvent de nombreuses applications en logistique, notamment pour optimiser le chargement de navires fluviaux, de flottes de camions ou de convois ferroviaires, ainsi que pour rationaliser l'organisation des espaces de stockage en entrepôt. Dans le secteur industriel, elles interviennent également dans les problèmes de découpe (cutting stock), où l'objectif est de minimiser les pertes de matière lors de la découpe de pièces à partir de matériaux bruts.

Les problèmes de bin-packing trouvent également des applications dans l'aménagement urbain. Ils peuvent notamment contribuer à optimiser l'implantation de micro-hubs logistiques urbains (plateformes de transit) ou l'aménagement d'aires de stationnement intégrant des ombrières photovoltaïques et des bornes de recharge pour véhicules électriques.



FIGURE 1 – Rangement optimal pour le fret.

L'objectif de ce projet est de modéliser par **programmation linéaire** en nombres entiers (PLNE) un problème de bin-packing en dimension 2 d'espace (2D-BPP) et d'implémenter (en Python) des méthodes exactes et heuristiques pour sa résolution. Il s'agit aussi d'évaluer le passage à l'échelle sur des instances de taille réaliste (nombre d'objets à placer $n \geq 40$) et d'explorer des extensions géométriques avancées.

2 Le problème de bin-packing 2D

Dans le problème de bin-packing bi-dimensionnel, on veut placer un ensemble d'objets rectangulaires de tailles différentes sur des plaques rectangulaires qui ont une taille identique, en utilisant un nombre minimal de plaques. Les contraintes suivantes doivent être respectées.

1. *Confinement.* Un objet rectangulaire est tout entier contenu dans la plaque sur laquelle il est placé.
2. *Orientations fixées.* L'orientation d'un objet rectangulaire est fixée à l'avance : il est toujours placé sur une plaque de sorte que ses bords restent parallèles à ceux de la plaque. Aucune rotation n'est permise.

3. *Non-chevauchement.* Il n'y a pas de chevauchement entre les objets rectangulaires d'une même plaque.

La figure 2 ci-dessous montre un exemple de placement admissible.

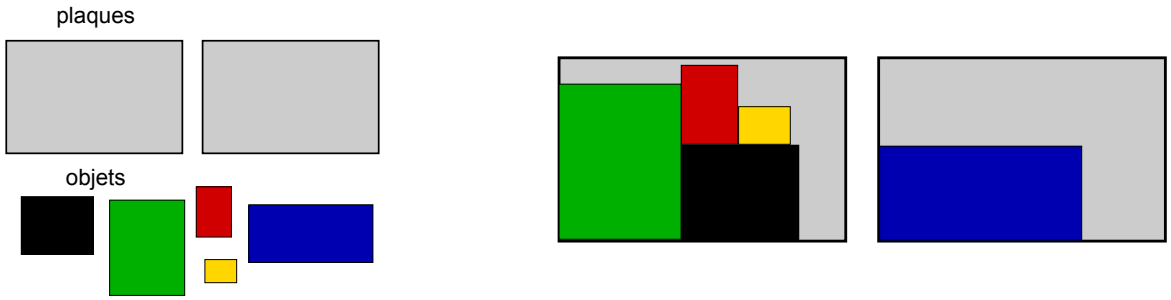


FIGURE 2 – Exemple de placement admissible. Plaques et objets rectangulaires (à gauche) et un placement admissible (à droite).

Plus formellement, on considère un ensemble de n objets rectangulaires $\mathcal{I} = \{1, \ldots, n\}$. Chaque objet $i \in \mathcal{I}$ possède une largeur $w_i \in \mathbb{N}^*$ et une hauteur $h_i \in \mathbb{N}^*$. On dispose d'une réserve de m plaques rectangulaires identiques $\mathcal{B} = \{1, \ldots, m\}$ de dimensions $W \times H$.

3 Travail Demandé

3.1 Partie 1 – Heuristiques

Une heuristique pour le bin-packing consiste à déterminer de façon algorithmique un placement admissible qui, sans être forcément optimal, correspond à une "bonne" solution généralement pas trop éloignée d'une solution optimale. Une heuristique fournit une (bonne) estimation du nombre m de plaques nécessaires [1].

Il vous est demandé de mettre en place des heuristiques qui consistent à trier les objets par "taille" décroissante et de placer successivement, l'un après l'autre, les objets dans cet ordre. Le placement se fait sur une plaque déjà utilisée. S'il n'y a pas de placement possible, il faut prendre une nouvelle plaque.

1. Proposer plusieurs heuristiques selon ce schéma, en précisant les éléments suivants.
   — *Choix de la taille.* La "taille" considérée d'un objet i pourra être égale à sa surface $w_i \times h_i$ ou bien à la dimension w_i ou h_i (la plus grande des 2 ?), ...
   Proposer d'autres choix.
   — *Stratégie de placement.* Proposer plusieurs stratégies pour le placement d'un objet. On pourra commencer par exemple à placer un objet dans le coin *en bas et à gauche* d'une plaque. Puis l'objet suivant à droite ou bien en haut. Et le suivant ... ?
2. Implémenter vos heuristiques proposées en Python.
   Visualisation graphique. Concevoir une fonction graphique avec `matplotlib` traçant chaque plaque utilisée sous la forme d'un rectangle distinct. Les objets doivent apparaître sur leur plaque sous forme de rectangles colorés, labellisés par leur indice i et leurs dimensions $(w_i \times h_i)$.
3. Tester avec les données fournies et vos propres données. Voir la Section 4 sur les données et la Section 5 sur les tests numériques.

1. On a toujours $m \leq n$ car au pire, si les contraintes sont respectées, on peut placer un objet par plaque.

3.2 Partie 2 – Modélisation mathématique en PLNE

1. **Modélisation.** Formuler le problème sous forme d'un programme linéaire en nombres entiers (PLNE) minimisant le nombre total de plaques utilisées. On pourra considérer les variables décisives suivantes (sans que ce soit une obligation).
   — *Variables binaires d'affectation.* Elles prennent en compte :
   - si une plaque est utilisée ou non
   - si un objet donné est placé sur une plaque donnée ou non

   — *Coordonnées des objets rectangles.* Variables (entières) représentant les coordonnées relatives du coin inférieur gauche des objets rectangulaires par rapport à la plaque où ils sont placés. L'origine $(0, 0)$ correspond au coin inférieur gauche de chacune des plaques
   — *Variables binaires de non-chevauchement.* Pour chaque paire d'objets rectangles, introduire des variables binaires qui indiquent si le premier objet de la paire est à gauche, à droite, au dessus, en dessous du second objet (4 types de variables binaires correspondant aux 4 directions cardinales).

   Pour les contraintes :
   (a) Chaque objet est placé sur une et une seule plaque.
   (b) Un objet ne peut être placé que sur une plaque utilisée.
   (c) *Confinement.* Chaque objet doit respecter les limites de la plaque.
   (d) *Non-chevauchement.* Deux objets placés sur la même plaque ne doivent pas se chevaucher selon les 4 directions cardinales.

   L'objectif initial est de minimiser le nombre total de plaques utilisées. Vous pourrez modifier cet objectif afin de forcer l'ordre croissant des numéros des plaques qui seront utilisées (plaque 1 d'abord, puis plaque 2, ...). De façon alternative, sans modifier l'objectif initial vous pourrez ajouter des contraintes sur les variables d'affectation pour forcer cet ordre.

   Remarque importante. Vous pourrez être amené à ajouter des contraintes supplémentaires afin de rendre plus efficace la résolution du PLNE. Cette contrainte est que la surface totale des objets d'une même plaque ne peut dépasser la surface totale $W \times H$ d'une plaque.

2. **Implémentation logicielle attendue.**
   (a) Développer un script Python avec l'API `gurobipy` pour définir (instancier) et résoudre le PLNE obtenu précédemment, afin de déterminer le placement optimal.
   (b) Mettre en place des mécanismes de contrôle de la résolution : limite de temps (`TimeLimit`), tolérance d'optimalité (`MIPGap`).
   (c) Utiliser la fonction graphique de la Partie 1. développée pour les heuristiques afin de visualiser le placement optimal.

3.3 Partie 3 – Extensions géométriques (traiter un axe)

Axe A – Rotation orthogonale à 90°. Les rectangles peuvent pivoter si leur manutention le permet. Introduire une variable binaire d'orientation $u_i \in \{0, 1\}$ et réécrire les contraintes (c) et (d). Quantifier le gain obtenu sur le nombre de bacs ouverts.

Axe B – Formes non rectangulaires. Proposer une formulation adaptée aux formes en « L » ou polygones simples (décomposition en rectangles solidaires contraints par translation rigide ou discrétisation par grille).

Axe C – Extension tridimensionnelle (3D-BPP). Généraliser le modèle à trois dimensions ($w_i \times h_i \times d_i$ dans des conteneurs $W \times H \times D$). Intégrer les contraintes physiques indispensables en exploitation logistique :

— *Gerbabilité / Capacité portante :* interdiction de superposer des charges lourdes sur des objets fragiles.

— *Stabilité statique :* surface d'appui minimale requise sous chaque objet.

4 Données

Les données sont :

— Nombre de plaques disponibles : m.

— Dimensions des plaques : W, H (des entiers).

— Nombres d'objets à placer : n.

— Dimensions des objets : $(w_i, h_i), i \in \{1, \ldots, n\}$ (des entiers).

Vous utiliserez d'une part des données fournies et d'autre part des données que vous aurez engendrées vous-même.

- **Données fournies**. Récupérer à l'adresse suivante les données sous forme de fonctions Python (contenues dans le fichier `databinpacking.py`) qui vous serviront à tester vos différents codes.

  http://scheid.perso.math.cnrs.fr/Enseignement/ENPC/ProjetRO2026/

- **Données engendrées.** Engendrer des données aléatoires [2] selon deux profils.

  — *Classe A (objets homogènes)* : objets de tailles comparables avec les w_i et h_i choisis aléatoirement (de façon uniforme et entière) avec [3]

  $$w_i \in [\![\lfloor 0.3W \rfloor,\ \lfloor 0.5W \rfloor]\!], \quad h_i \in [\![\lfloor 0.3H \rfloor,\ \lfloor 0.5H \rfloor]\!]$$

  — *Classe B (objets hétérogènes)* : forte disparité des dimensions avec

  $$w_i \in [\![1,\ \lfloor 0.8W \rfloor]\!], \quad h_i \in [\![1,\ \lfloor 0.8H \rfloor]\!]$$

  Remarque. Vous pourrez avoir besoin de reproduire toujours les mêmes données aléatoires. Pour cela, vous pouvez bloquer le germe du générateur de nombres aléatoires avec la fonction `seed()` du module `random` de `numpy`, par exemple avec l'instruction `np.random.seed(1)`. L'appel à la fonction `randint` engendrera alors toujours les mêmes nombres.

5 Campagne de tests numériques

Les données à utiliser sont précisées dans la Section 4. Utiliser les données fournies pour tester vos codes. Pour les différentes données et avec $W = H = 10$, vous devez obtenir :

— `data6()` : $m_{\text{opt}} = 2$ plaques.

— `data20ClasseA()` : $m_{\text{opt}} = 3$ plaques

— `data20ClasseB()` : $m_{\text{opt}} = 4$ plaques

Tester ensuite avec vos propres données engendrées (voir Section 5). Conduire une série de tests pour $n \in \{10, 20, 30, 40, 50\}$. Pour les données aléatoires, faire varier les bornes supérieures et inférieures des intervalles.

2. La fonction `randint` du sous-module `random` de `numpy` permet d'engendrer aléatoirement des nombres entiers de façon uniforme avec des valeurs comprises dans un intervalle spécifié.

3. $\lfloor x \rfloor$ désigne la partie entière de x.

Remarque importante. Pour les grandes valeurs de n, vous aurez besoin de la version "non-bridée" de GUROBI. Si vous avez installé GUROBI avec la commande `pip install gurobipy`, la version est alors bridée (nombre de contraintes et de variables limitées). Pour bénéficier de la version complète, vous devez d'abord demander une license sur le site de GUROBI (voir guide d'installation déjà fourni) puis installer le package d'outils de gestion de license à l'adresse https://support.gurobi.com/hc/en-us/articles/360059842732-How-do-I-set-up-a-license-without-installing-the-full-Gurobi-package

1. Pour chaque test, calculer le temps de calcul CPU et représenter l'évolution du temps CPU en fonction de n pour des objets de Classe A et de Classe B.
2. Pour chaque test, calculer le taux de recouvrement

$$\tau = \frac{\sum_{i=1}^{n} w_i h_i}{(W \cdot H) m_{\text{opt}}}$$

où m_{opt} est le nombre minimal de plaques obtenu. Etudier l'évolution de ce taux en fonction de n et du type de données (classe A/B).

6 Livrables

Le projet est à réaliser par groupes de 4 étudiants. Les livrables attendus sont :

1. **Un rapport de synthèse (format PDF, 10 à 15 pages) :** formalisation mathématique rigoureuse, plan d'expériences complet, analyse critique des résultats et des limites du modèle.
2. **Un dépôt de code source (Python)**
   - Modélisation sous `gurobipy` avec gestion fine des paramètres du solveur.
   - Scripts de visualisation graphique avec `matplotlib`.
   - Générateurs d'instances et scripts automatisant les campagnes de tests.
   - Fichier `README.md` exhaustif permettant la reproductibilité des benchmarks.

Jean-François Scheid
jean-francois.scheid@univ-lorraine.fr
http://scheid.perso.math.cnrs.fr/Enseignement/ENPC/