

Application des méthodes génétiques au problème du BIN-PACKING

Jules BESSON

19 mars 2020

Résumé

Mon TIPE a pour objectif primaire de trouver des algorithmes de résolution (exact ou non) pour le problème du BIN-PACKING. Ce problème fait partie des problèmes de packing et de recouvrement comme le célèbre problème du Sac à dos.

Son objectif est de minimiser le nombre de boîtes de taille fixée (appelées bins) nécessaires au rangement d'une instance d'objets. Il s'applique notamment dans l'optimisation des transports de containers : si l'on arrive à minimiser le nombre de containers pour transporter les objets, on fait des économies financières et environnementales sur le transport de ces marchandises. Il se place donc au cœur des problématiques actuelles de la mondialisation.

Table des matières

1	Introduction au problème	2
1.1	Définition pour la première dimension	2
1.2	Passage aux dimensions supérieures	2
2	\mathcal{NP}-Complétude	3
2.1	Implémentation des algorithmes de résolution exacts	3
2.2	Preuve de la \mathcal{NP} -Complétude	3
3	Heuristiques	4
3.1	Heuristiques classiques	4
3.2	Méthodes de programmation linéaire appliquées au BIN-PACKING par classes	4
4	Métaheuristiques	5
4.1	Méthode Aléatoire	5
4.2	Méthode génétique	6
5	Résultats	7
A	Algorithmes	9
A.1	\mathcal{NP} -complétude	9
A.2	Algorithmes gloutons	9
A.3	Un algorithme plus efficace en pratique	10
A.4	Heuristiques	10
B	Complexité des algorithmes gloutons	12
B.1	Une fonction classique pour les sommes à coefficients binomiaux	12
B.2	Une conjecture	12
B.3	Preuve de la complexité auxiliaire	12
B.4	Généralisation à l'aide des polynômes de Newton	12
B.5	Première transition	13
B.6	Deuxième transition	13
B.7	Conclusion	14
C	Code Ocaml	14
D	Code python	15
D.1	Descriptif	15
D.2	Importations et variables globales	15
D.3	Fonctions élémentaires	16
D.4	Heuristiques classiques	21
D.5	Métaheuristiques	24

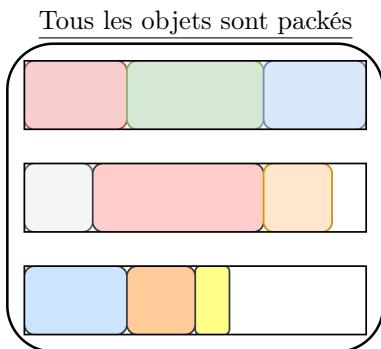
1 Introduction au problème

Le problème énoncé dans le résumé n'est pas assez rigoureux pour se prêter à une étude scientifique, il faut alors travailler la forme et l'expression du problème du BIN-PACKING.

1.1 Définition pour la première dimension

On va commencer par la définition rigoureuse en dimension 1. Intuitivement cela revient à ajouter des objets dont la taille est un réel positif, dans des bins, tel que la somme de leurs tailles ne dépasse pas celle des bins.

Elle doit formaliser les deux aspects suivants qui régissent le packing des objets :



Tous les objets sont packés

Les objets ne doivent pas dépasser la capacité des bins

Il ne nous reste plus qu'à exprimer cela mathématiquement [1, 2] :

Soit a_1, \dots, a_n des réels positifs non tous nuls.

Trouver $k \in \mathbb{N}^*$ et une fonction d'assignation $f : \llbracket 1, n \rrbracket \mapsto \llbracket 1, k \rrbracket$ tq :
 $\forall j \in \llbracket 1, k \rrbracket, \sum_{i \in f^{-1}(\{j\})} a_i \leq 1$ de façon à ce que k soit minimal¹.

On a ici opéré une normalisation des objets : si l'on veut packer les objets $a_1, \dots, a_n \in \mathbb{R}_+^*$ dans des bins de taille $c > 0$, on va packer les objets $\frac{a_1}{c}, \dots, \frac{a_n}{c}$ dans des bins de taille 1. Pour formaliser les choses, la fonction d'assignation donne le numéro du bin dans lequel on packe l'objet a_i .

Si les objets ont une taille inférieure à 1, k est fini car la fonction d'assignation $f : i \in \llbracket 1, n \rrbracket \mapsto i$, convient pour $k = n$; on a alors l'ensemble $\{(k, f)\}$, admissible pour \mathcal{BP} qui est non vide, minoré pour les k car inclus dans \mathbb{N} , il admet donc un élément (k, f) tel que k soit minimal.

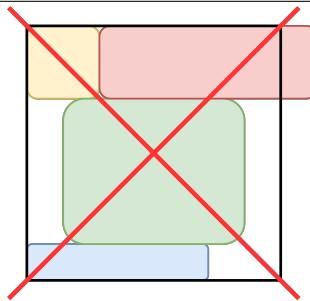
Le couple (k, f) n'est pas unique, sauf si $k = 1$, on peut d'ailleurs introduire une relation d'équivalence entre deux solutions : $(k_1, f_1) \sim (k_2, f_2) \Leftrightarrow k_1 = k_2 = k$ et $\exists \sigma \in \mathfrak{S}_k | f_1 = \sigma \circ f_2$, on notera $\overline{(k, f)}$ la classe d'équivalence de (k, f) pour \sim .

On a un petit résultat évident : $k \geq \left\lceil \sum_{i=1}^m a_i \right\rceil$

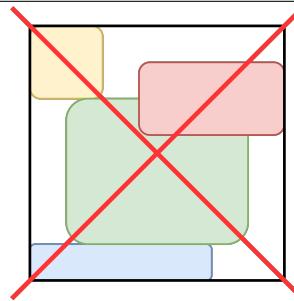
1.2 Passage aux dimensions supérieures

Lorsque l'on passe en dimension supérieure, on ne bénéficie plus de la simplification des contraintes qui se ramène à une somme, il faut donc modifier la définition en fonction de deux contraintes ici illustrées en dimension 2 [3, 4] :

Les objets d'un bin ne dépassent pas sa capacité (1)



Il n'y a pas de chevauchement (2)



Soit $l_2, \dots, l_m \in \mathbb{R}_+^*$, on pose $C := [0, 1] \times [0, l_2] \times \dots \times [0, l_m]$. On considère l'instance d'objets o_1, \dots, o_m , des pavés de \mathbb{R}_+^m . Trouver (k, f) avec $f : i \in \llbracket 1, m \rrbracket \rightarrow (j, p) \in \llbracket 1, k \rrbracket \times \mathbb{R}_+^m$ tels que :

- $\forall i \in \llbracket 1, n \rrbracket, f_2(i)^2 + o_i \subset C$ (1)
- $\forall i \neq i' \in \llbracket 1, n \rrbracket, f_1(i) = f_1(i') \Rightarrow \text{int}(f_2(i) + o_i) \cap \text{int}(f_2(i') + o_{i'}) = \emptyset$ (2)
- k soit minimal

On a alors réussi à généraliser le problème du BIN-PACKING en toute dimension. On notera cependant que l'étude ensembliste de ce problème est généralisable avec des objets ne se réduisant plus aux pavés, mais aux polytopes, ou même aux compacts quelconques. On restera cependant pour plus de simplicité et pour une étude informatique plus légère avec des pavés, de plus cela reflète l'aspect concret du problème : lorsque l'on transporte des objets, le paquetage est un pavé.

2. On a $f(i) = (j, p) = (f_1(i), f_2(i))$

Pour la finitude de k , on peut observer le même raisonnement que dans la dimension 1, il suffit d'avoir les dimensions de chaque objet inférieures à celles de C .

Enfin, pour faciliter d'avantage l'étude en dimension supérieure, il conviendra de discréteriser le problème : tous les ensembles seront des pavés de \mathbb{Z}^m .

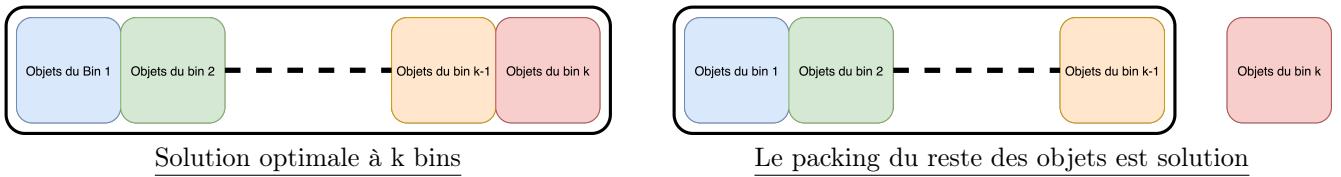
2 \mathcal{NP} -Complétude

2.1 Implémentation des algorithmes de résolution exacts

L'objectif premier est de résoudre le problème du BIN-PACKING, pour cela, il existe deux approches : soit on calcule tous les packings possibles, et on en prend un tel que le nombre de bins soit minimal, soit on exploite le lemme que j'ai élaboré :

Lemme 1. Soit $\mathcal{I} := a_1, \dots, a_n$ une instance de \mathcal{BP} , (k, f) une solution au problème pour cette instance, alors $\exists J \subset \llbracket 1, n \rrbracket$, $g : J \rightarrow \llbracket 1, k - 1 \rrbracket$ tels que $(k - 1, g)$ solution de \mathcal{BP} pour l'instance $(a_j)_{j \in J}$ et $\sum_{j \in J^c} a_j \leq 1$.

Démonstration. Soit (k, f) une solution à \mathcal{BP} pour \mathcal{I} , On note $J := \bigcup_{j=1}^{k-1} f^{-1}(\{j\})$. Déjà $\sum_{j \in J^c} a_j = \sum_{j \in f^{-1}(k)} a_j \leq 1$.



On a alors $(k - 1, f|_J)$ solution de \mathcal{BP} pour $(a_j)_{j \in J}$, sinon il existerait un packing avec $l < k - 1$ bins, ce qui implique que l'on peut packer tous les objets en $l + 1 < k$ bins, ce qui est exclu. \square

On peut généraliser ce résultat.

Lemme 2. Soit $\mathcal{I} := a_1, \dots, a_n$ une instance de \mathcal{BP} , (k, f) la solution au problème pour cette instance, alors $\exists J \subset \llbracket 1, n \rrbracket$, $f_1 : J \rightarrow \llbracket 1, k - 1 \rrbracket$, $f_2 : J^c \rightarrow \llbracket 1, k - 1 \rrbracket$ tels que (k_1, f_1) , (k_2, f_2) solutions de \mathcal{BP} pour les instances $(a_j)_{j \in J}$ et $(a_j)_{j \in J^c}$, et $k_1 + k_2 = k$.

Démonstration. C'est le même schéma que précédemment, seulement on isole les k_1 premiers bins et les k_2 derniers. \square

À partir de ces lemmes, il est aisément d'élaborer des algorithmes récursifs. (Voir dans l'annexe les algorithmes 2 à 6).

On remarque que les complexités de ces algorithmes ne sont pas polynômiales, le problème semble \mathcal{NP} -complet (partie B de l'annexe).

2.2 Preuve de la \mathcal{NP} -Complétude [1, 5]

Appartenance à la Classe \mathcal{NP}

Pour montrer que $\mathcal{BP} \in \mathcal{NP}$, il faut trouver un vérificateur polynomial, qui est plutôt trivial ici : l'algorithme 1.

Cet algorithme finit car il est constitué d'une boucle itérative. Comme il termine, on a vérifié l'assertion $\forall j \in \llbracket 1, k \rrbracket$, $\sum_{i \in f^{-1}(\{j\})} a_i \leq 1$ (les v_j sont les sommes). Il est évident qu'il est en complexité polynomiale par rapport à n (dans ce modèle, la complexité est linéaire).

Ainsi, \mathcal{BP} admet un vérificateur polynomial donc il est dans la classe \mathcal{NP} .

Preuve de la \mathcal{NP} -difficulté

Nous allons pour cela utiliser le problème de PARTITION certifié \mathcal{NP} -complet par Richard KARP en 1972 :

Soit c_1, \dots, c_n des entiers naturels non tous nuls.

Existe-t-il une partition (S, S^c) de $\llbracket 1, n \rrbracket$ tq : $\sum_{i \in S} c_i = \sum_{i \in S^c} c_i$?

Considérons la fonction f qui à une instance du problème de PARTITION c_1, \dots, c_n associe une instance du \mathcal{BP} a_1, \dots, a_n , telle que $\forall i \in \llbracket 1, n \rrbracket$, $a_i = \frac{2c_i}{\sum_{j=1}^n c_j}$.

Si \mathcal{BP} était résolu pour cette instance avec la sortie (k, g) , alors :

- $k \geq 2$ car $\sum_{i=1}^n a_i > 1$

- Si $k = 2$, alors on note $S = g^{-1}(\{1\})$, (et $S^c = g^{-1}(\{2\})$).

Comme $\sum_{i \in S} a_i \leq 1$, $\sum_{i \in S^c} a_i \leq 1$ et $\sum_{i \in S \cup S^c} a_i = 2$, donc par soustraction $\sum_{i \in S} a_i \geq 1$, $\sum_{i \in S^c} a_i \geq 1$, i.e. $\sum_{i \in S} a_i = 1$, $\sum_{i \in S^c} a_i = 1$ et donc bien évidemment $\sum_{i \in S} c_i = \sum_{i \in S^c} c_i$, le problème de PARTITION est donc résolu.

Si le problème de PARTITION est résolu avec la partition (S, S^c) , alors $\sum_{i \in S} c_i = \sum_{i \in S^c} c_i$, donc $\sum_{i \in S} a_i = \sum_{i \in S^c} a_i$.

$$g : \llbracket 1, n \rrbracket \rightarrow \{1, 2\}$$

Considérons $i \mapsto \begin{cases} 1 & \text{si } i \in S \\ 2 & \text{sinon} \end{cases}$, alors $(2, g)$ est une solution du \mathcal{BP} , et comme pour toute solution (l, h) , $l \geq 2$, elle est minimale.

On sait donc que toute instance w du problème de PARTITION est satisfiable si et seulement si $f(w)$ est satisfiable pour $k = 2$ (On peut résoudre le problème de PARTITION si on sait résoudre \mathcal{BP}).

Donc \mathcal{BP} est plus difficile que le problème de PARTITION, ainsi il est \mathcal{NP} -difficile.

Conclusion

Étant donné que \mathcal{BP} est dans la classe \mathcal{NP} et qu'il est \mathcal{NP} -difficile, alors le Problème du BIN-PACKING est \mathcal{NP} -Complexe.

On peut également prouver la \mathcal{NP} -complétude du problème en toute dimension, car si l'on sait résoudre \mathcal{BP} en dimension $l \in \mathbb{N}$, grâce à une instance d'objets bien choisie pour la dimension l , on a la solution optimale pour le problème du BIN-PACKING en une dimension.

3 Heuristiques

3.1 Heuristiques classiques [1, 3, 4, 6]

Next-Fit

La première heuristique est la plus élémentaire, la méthode consiste à considérer les objets un à un, et de les ajouter dans le premier bin, puis dès que ce n'est plus possible, on passe au deuxième bin, puis au troisième et ainsi de suite (Cf algorithme 7).

Sa complexité est en $\mathcal{O}(n)$, et on a un nombre de bins inférieur au nombre d'objets.

First-Fit

On affine ensuite le raisonnement : on considère toujours les objets un à un, et on les ajoute dans le premier bin dans lequel c'est possible, si on ne peut pas l'ajouter, on rajoute un bin. (Cf algorithmes 8 et 9).

Sa complexité est en $\mathcal{O}(n^2)$. On peut facilement montrer que, dans la solution de \mathcal{FF} , il y a au maximum un bin qui n'est rempli qu'à moins de la moitié, donc $\mathcal{NF}(\mathcal{I}) \leq \left\lceil 2 \sum_{i=1}^m \right\rceil$, donc $\mathcal{NF}(\mathcal{I}) \leq 2\mathcal{BP}(\mathcal{I}) + 1$

Il a une variante appelée FIRST-FIT DÉCROISSANT qui consiste à d'abord trier l'instance par décroissance, et de lui appliquer le FIRST-FIT (Cf algorithme 10). De manière générale, les algorithmes se comportent mieux lorsque l'instance est triée dans l'ordre décroissant.

3.2 Méthodes de programmation linéaire appliquées au BIN-PACKING par classes [1]

On considère l'instance \mathcal{I} constituée d'objets du type : $s_1, \dots, s_m \geq 0$, avec $\forall i \in \llbracket 1, m \rrbracket$, b_i exemplaires de s_i .

On commence par calculer toutes les façons de remplir un bin avec des objets du type s_1, \dots, s_m , elles sont caractérisées par les N m -uplets : $\{T_1, \dots, T_N\} := \left\{ (k_1, \dots, k_m) \in \mathbb{N}^{*m} \mid \sum_{i=1}^m k_i s_i \leq 1 \right\}$. On note $\forall j \in \llbracket 1, N \rrbracket$, $(t_{1,j}, \dots, t_{m,j}) := T_j$.

Dans la solution optimale pour l'instance \mathcal{I} , on retrouvera ces façons de packer les objets, On note alors $\forall j \in \llbracket 1, N \rrbracket$, x_j le nombre de fois que la façon T_j est utilisée. L'objectif devient alors de minimiser $\sum_{j=1}^N x_j$, soit le nombre de bins nécessaires. Il reste cependant des contraintes : il faut au moins utiliser b_i fois l'objet s_i , or $\forall i, j \in \llbracket 1, m \rrbracket \times \llbracket 1, N \rrbracket$, $t_{i,j}$ représente le nombre de fois où l'on utilise s_i dans la façon d'assigner les objets T_j .

Les contraintes deviennent donc $\forall i \in \llbracket 1, m \rrbracket$, $\sum_{j=1}^N t_{i,j} x_j \geq b_i$, l'égalité est vraie, mais l'inégalité lui est préférable, car la solution sera tronquée dans la suite de l'algorithme.

On pose $T := (t_{i,j})_{\substack{0 \leq i \leq m \\ 0 \leq j \leq N}} = (T_1^T \cdots T_N^T)^T$, $X := (x_1 \cdots x_N)^T$, $C := (1 \cdots 1)^T$, $B := (b_1 \cdots b_m)^T$

\mathcal{BP} est alors équivalent à :

minimiser	$C^T X$
tel que :	$TX \geq B$
et	$X \in \mathcal{M}_{N,1}(\mathbb{N})$

Le problème est que ce genre d'entrée ne correspond pas à la programmation linéaire, il faut donc le résoudre pour X réel, soit une relaxation continue, puis revenir au cas entier.

Ce qui donne :

minimiser	$C^T X^*$
tel que :	$TX^* \geq B$
et	$X^* \in \mathcal{M}_{N,1}(\mathbb{R}_+)$

Or ceci est une instance de programmation linéaire, on peut alors trouver le vecteur X^* tel que $C^T X^*$ soit minimal (à l'aide de la méthode du simplexe par exemple). On note $X := \lfloor X^* \rfloor$. À partir de ce vecteur, on obtient un packing partiel en $C^T X$ bins, et on ajoute les éventuels objets non assignés avec la méthode du FIRST-FIT DÉCROISSANT. La complexité de ce type de résolution se fait donc en $\mathcal{O}(mN)$.

Fernandez de la Vega et Lücker [1]³

Cet algorithme permet, avec une complexité contrôlable, d'obtenir un résultat asymptotiquement proche de l'optimum. En effet il a une complexité en $\mathcal{O}(\frac{n \log(n)}{\epsilon^2})$ et si k est le nombre de bins de la solution qu'il renvoie pour l'instance \mathcal{I} , on a $k \leq (1 + \epsilon)opt(\mathcal{I})$.

Démonstration. D'abord on a $\sum_{i=1}^n a_i \geq \gamma(n - l)$ et donc $m \leq \frac{n-l}{h} \leq \frac{n-l}{\epsilon \sum_{i=1}^n a_i} \leq \frac{1}{\gamma \epsilon} = \frac{\epsilon+1}{\epsilon^2}$.

Le découpage de l'instance est en $\mathcal{O}(n \log(n))$, l'assignation des bins de R se fait en complexité linéaire. Il reste la complexité de la programmation linéaire. On sait que le simplexe est en $\mathcal{O}(nN)$, or on peut majorer N facilement : les objets de l'instance sont supérieurs à γ , on a donc au maximum $\frac{1}{\gamma}$ objets pour un bin au maximum, soit $N < m^{\frac{1}{\gamma}} = \mathcal{O}(m)$. La complexité de la résolution via le simplexe est en $\mathcal{O}(nm) = \mathcal{O}(\frac{n}{\epsilon^2})$ d'où la conclusion.

Pour la deuxième partie de la preuve, se référer à [1]. □

4 Métaheuristiques

Dans cette section nous introduirons des heuristiques non classiques appelées processus stochastiques ou métahéuristiques.

4.1 Méthode Aléatoire

Une première méthode connue et élémentaire consiste à ajouter les objets dans les bins de manière aléatoire, on peut distinguer deux cas d'ajout aléatoires.

Aléatoire classique

Ici on part d'un constat : le nombre de bins nécessaire est facilement majoré par le nombre d'objets n (à supposer que les objets n'excèdent pas la capacité des bins). $\forall i \in \llbracket 1, n \rrbracket$ on donne alors à $f(i)$ le premier entier j choisi aléatoirement dans $\llbracket 1, n \rrbracket$ tel que l'objet a_i est ajoutable dans le bin j . On a ainsi une fonction d'assignation, le seul problème est que le nombre de bins est toujours de n , alors on évite la fonction d'assignation, de façon à "supprimer" les bins éventuellement vides. Cf : Code python "Aléatoire".

Aléatoire décroissant

On sait que le packing des objets se comporte mieux lorsque l'instance d'objets est triée dans l'ordre décroissant [1], alors d'abord on trie la liste d'objets, ensuite on ajoute les objets un à un dans la liste des bins de la manière suivante :

- on a r la longueur de la liste des bins
- on choisit aléatoirement un indice $1 \leq j \leq r + 1$
- si $j = r + 1$, on rajoute un bin dans la liste auquel on ajoute a_i
- sinon on vérifie que l'on peut ajouter a_i dans le bin j
 - si oui on l'ajoute
 - sinon on re-choisit un indice aléatoirement
- Enfin, $f(i) := j$

Cf : Code python "Aléatoire décroissant".

Cette méthode donne de bien meilleurs résultats, car ses façons de packer sont semblables à celle du FIRST-FIT DÉCROISSANT, avec quelques modifications dues à l'aléatoire.

On constate également qu'il existe un unique représentant de chaque classe d'équivalence (définie au début), qui peut être généré par cet algorithme.

Convergence

Dans l'aléatoire classique et l'aléatoire décroissant, on constate que l'ensemble des assignations engendrables par chacun de ces algorithmes est fini, et qu'il contient une solution à \mathcal{BP} . Si on note A_n l'événement "durant n itérations l'algorithme a construit une solution de \mathcal{BP} ". On sait que la probabilité à une itération de tirer une bonne solution est constante et vaut $p > 0$, et que les itérations sont indépendantes, donc $\mathbb{P}(A_n^c) = (1-p)^n \rightarrow 0$ lorsque $n \rightarrow \infty$, donc $\mathbb{P}(A_n) \rightarrow 1$ lorsque $n \rightarrow \infty$, donc l'algorithme converge vers la solution au problème.

3. Voir algorithme 11

4.2 Méthode génétique

Principe général [7, 8, 9]

La méthode de résolution génétique est une méthode moderne s'inspirant de la théorie de l'évolution introduite par CHARLES DARWIN. Elle s'applique aux problèmes d'optimisation en particulier. Le principe est simple : on cherche à trouver une solution optimale à un ensemble de contraintes, pour ce faire, on engendre un population de solution initiale, on les fait évoluer, selon un mécanisme bien défini, puis on effectue une sélection sur les individus de cette population, de manière à garder les plus proches de l'optimum, et on réitère ce schéma jusqu'à avoir une population finale proche de l'optimum.

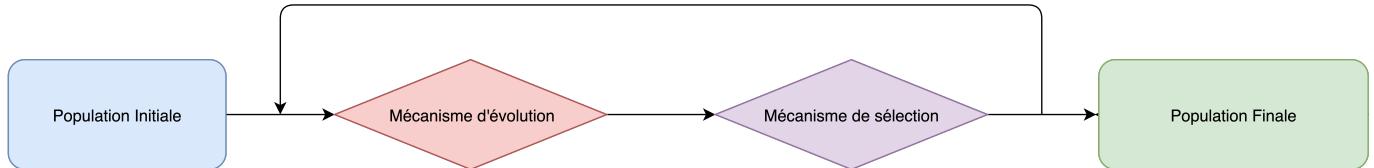
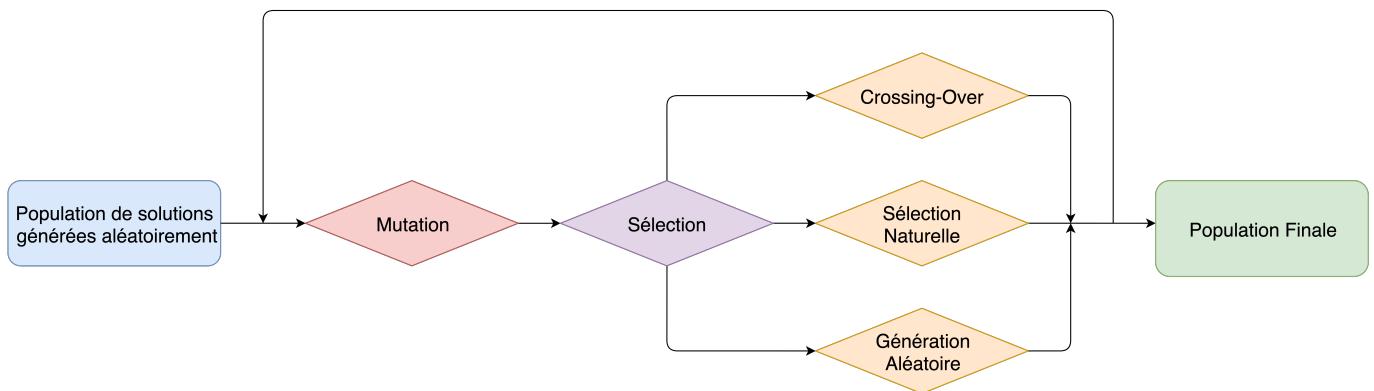


Schéma de mon algorithme génétique

Dans mon algorithme génétique, mon but est de minimiser le nombre de bins nécessaires, le problème est que la fonction qui à une solution associe son nombre de bins est à valeurs entières, et ne différencie pas deux solutions de même nombre de bins. Je définis donc une fonction *note* que l'on cherchera à minimiser.

À chaque génération on suivra alors les étapes suivantes :

- On effectue une mutation sur la population
- On la trie selon la fonction note et on la sépare en 3 : les meilleures, les intermédiaires, et les mauvaises
- On sélectionne une partie des solution intermédiaires avec une roulette
- On génère des solutions avec un crossing-over
- On génère des solutions aléatoirement (méthode 2)
- la génération suivante correspond à la concaténation des listes :
 - des meilleures solutions
 - des solutions de la roulette
 - du crossing-over
 - de l'aléatoire.

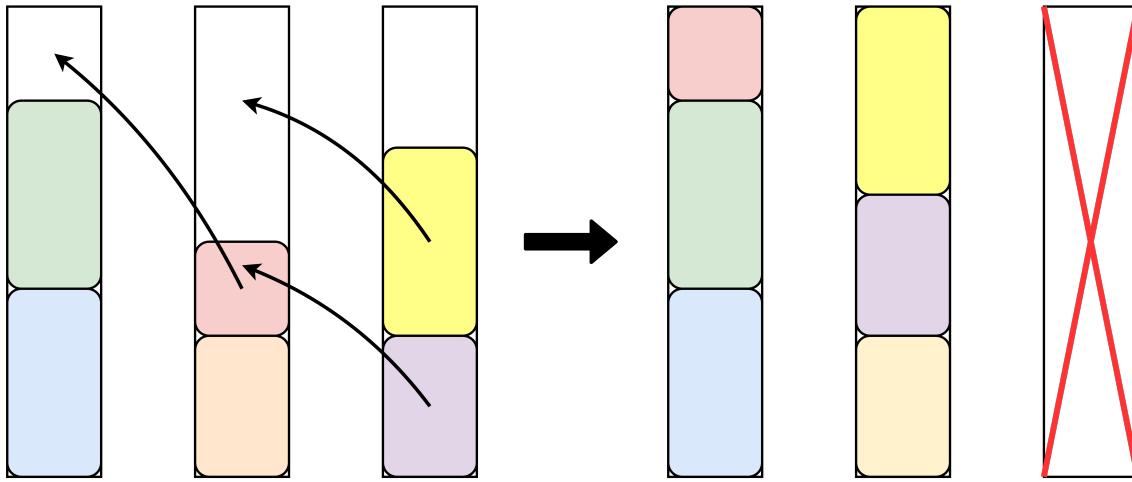


À chaque étape, la meilleure solution est conservée, donc on a une différence par rapport à l'optimum décroissante au cours des générations.

Mutations

Pour opérer la mutation, on procède ainsi pour chaque individu :

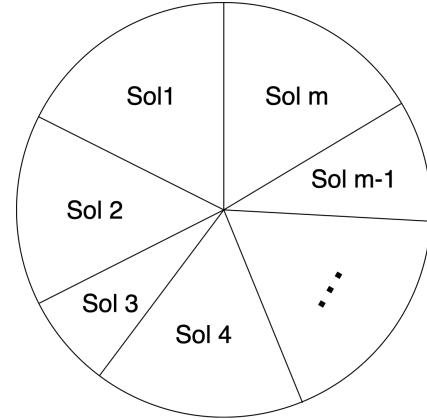
- on a $0 < p < 1$ fixé
- on choisit nb le nombre de changements à opérer selon une loi géométrique de paramètre p (collé aux lois naturelles)
- on choisit nb indices d'objets (o_1, \dots, o_{nb}) et nb indices de bins (b_1, \dots, b_{nb}) tous indépendants avec une loi uniforme sur $\llbracket 1, n \rrbracket$ et $\llbracket 1, k \rrbracket$
- $\forall i \in \llbracket 1, nb \rrbracket$, si on peut déplacer a_{o_i} dans le bin b_i , on le fait, sinon on ne modifie rien
- si la nouvelle solution a une meilleure note, on la garde, sinon on conserve l'ancienne

Exemple de mutation pour $nb = 3$ conduisant à la diminution du nombre de bins

Roulette

Pour opérer la roulette, on fonctionne ainsi :

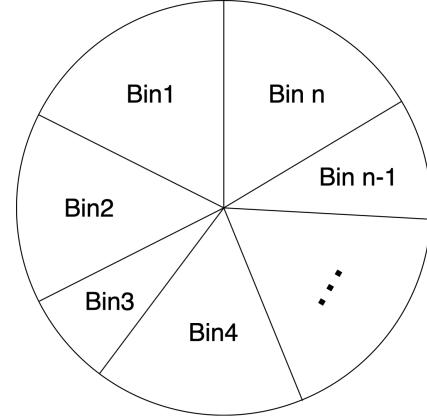
- La génération est triée grâce à la fonction note
- à chaque solution, on associe $\mathcal{P}(sol)$ son poids grâce à une fonction de poids, qui valorise les notes basses (meilleures solutions), à la manière du poids de BOLTZMANN
- dans la liste des solutions intermédiaires, chaque solution a un poids, qu'on normalise, de façon à avoir un germe de probabilité
- à l'aide de ce germe, on tire un certain nombre (fixe au cours des générations) de ces solutions, sans remise.



Crossing-Over

On initialise $M = (p_{i,j})_{i,j \in \llbracket 1, n \rrbracket}$ à $(0)_{i,j \in \llbracket 1, n \rrbracket}$ car on sait que le nombre de bins est inférieur à n , ensuite on procède par étapes :

- pour chaque solution dans les meilleures et les intermédiaires, $\forall i \in \llbracket 1, n \rrbracket$, i est assigné dans le bin j , alors $p_{i,j} \leftarrow p_{i,j} + \mathcal{P}(sol)$
- ensuite on normalise chaque ligne de la matrice de manière à avoir $\forall i \in \llbracket 1, n \rrbracket$, $(p_{i,j})_{j \in \llbracket 1, n \rrbracket}$ germe de probabilité où $p_{i,j}$ correspond à la probabilité pour l'objet i d'aller dans le bin j
- On construit ensuite de nouvelles solutions : pour chaque objet, on tire j sans remise avec le germe calculé précédemment et
 - si a_i ajoutable dans le bin j , on l'ajoute
 - sinon on continue le tirage
 - si on ne peut pas l'ajouter, on l'ajoute selon la méthode de l'aléatoire décroissant



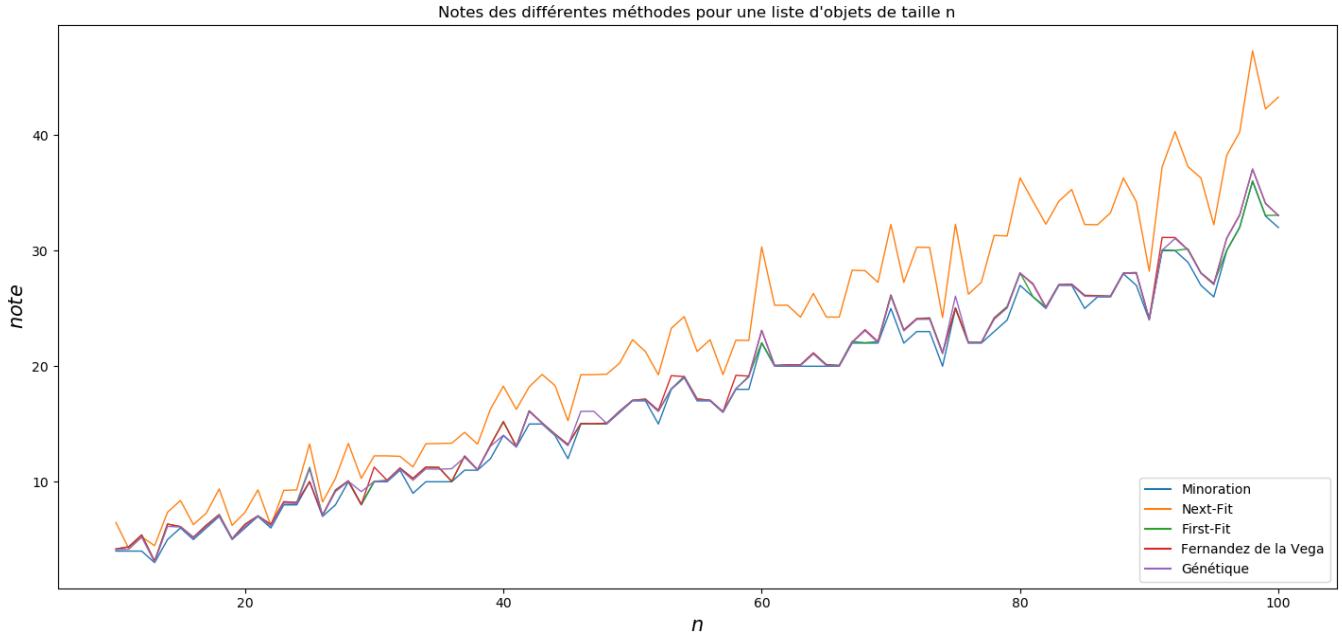
Convergence

La convergence de l'algorithme est assurée par l'aléatoire, et d'avantage encore par la présence de la mutation.

5 Résultats

J'ai ensuite comparé les différents algorithmes avec une répartition d'objets dite naturelle. En effet, pour générer le tableau d'objets en entrée, j'utilise une loi gaussienne, à laquelle j'applique un changement de variable avec \arctan pour me ramener à $[-\frac{\pi}{2}, \frac{\pi}{2}]$, puis je me ramène à $[0, 1]$, on a alors une pseudo loi gaussienne, que j'ai centrée en $\frac{1}{4}$ car cela traduit une réalité physique : les objets ont une taille moyenne de l'ordre de $\frac{1}{4}$ de la taille des boîtes.

Ensuite, avec ces tableaux d'objets de taille n variant de 10 à 100, j'ai comparé le NEXT-FIT DÉCROISSANT, FIRST-FIT DÉCROISSANT, la méthode de FERNANDEZ DE LA VEGA ET LÜCKER, ainsi que la méthode génétique.



On constate que les trois derniers algorithmes ont une précision de même ordre par rapport à l'optimum, on pourrait alors penser que faire un algorithme génétique était fortuit, cependant, le BIN-PACKING en une dimension a peu d'intérêt dans la pratique, on va plutôt chercher à l'implémenter en trois dimensions. Dans ce cas le FIRST-FIT nécessite une modification : des règles de priorité (on place l'objet en bas à gauche en priorité par exemple), et il est bien moins efficace, alors que l'algorithme génétique présente des résultats plus acceptables et est facilement modifiable.

Références

- [1] Jens VYGEN Bernhard KORTE. *Combinatorial Optimisation* : Theory and Algorithms, chapter 18. Springer, 4 edition, 2007.
- [2] Mathworld. Bin-Pack Problem. <https://www.wolframalpha.com/input/?i=bin-packing+problem>, <http://mathworld.wolfram.com/Bin-PackProblem.html>, <https://www.wolframalpha.com/input/?i=51E23&lk=1&assumption=%22ClashPrefs%22+-%3E+%7B%22MathWorldClass%22,+%2251E23%22%7D>.
- [3] Ali KHANAFER. *Algorithmes pour des problèmes de Bin Packing mono- et multi-objectif*. PhD thesis, Université des Sciences et Technologies de Lille, 2010.
- [4] Joseph EL-HAYEK. *Le problème de bin-packing en deux-dimensions, le cas non-orienté : résolution approchée et bornes inférieures*. PhD thesis, Université de Technologie de Compiègne, 2006.
- [5] Olivier BOURNEZ. Fondements de l'informatique : Logique, modèles, calculs ; cours 8 : NP-complétude. <https://www.lri.fr/~jcohen/documents/enseignement/Cours-NP-completude.pdf>, 2014.
- [6] James HAMBLIN. Math for liberal studies : *Bin-Pack Algorithms*. <https://www.youtube.com/watch?v=vUxhAmfXs2o>, 8 juillet 2011.
- [7] Cyril BERTELLE. Cours de Master 1 : *Intelligence Artificielle*, chapitre 2 : *Algorithmes Génétiques*. <http://litis.univ-le havre.fr/~bertelle/iaweb/ia-algo-evol-comp-2p.pdf>.
- [8] Francois-Gérard RADET Amédée SOUQUET. TE de fin d'année. http://souqueta.free.fr/Project/files/TE_AG.pdf, 2004.
- [9] Philip KIELY. Introduction to genetic algorithms. <https://blog.floydhub.com/introduction-to-genetic-algorithms/>, 2019.

A Algorithmes

A.1 \mathcal{NP} -complétude

Algorithme 1 Vérificateur polynômial pour \mathcal{BP}

Entrée: Une instance a_1, \dots, a_n d'objets pour le BIN-PACKING, un entier naturel k avec une fonction d'assignation associée f

Sortie: vérifier que (k, f) est une solution (non minimale)

```

 $v_1, \dots, v_k \leftarrow 0, \dots, 0$ 
pour  $i = 1$  à  $n$  faire
     $j \leftarrow f(i)$ 
     $v_j \leftarrow v_j + a_i$ 
    si  $v_j > 1$  alors
        renvoyer faux
    fin si
fin pour
renvoyer vrai

```

A.2 Algorithmes gloutons

Algorithmes issus de la définition récursive

Algorithme 2 Récursif glouton pour BIN-PACKING

Entrée: Une instance a_1, \dots, a_m d'objets pour le BIN-PACKING.

Sortie: Une solution (k, f) du BIN-PACKING.

```

si  $\sum_{i=1}^m a_i \leq 1$  alors
    renvoyer  $(1, i \mapsto 1)$ 
sinon
    renvoyer  $(\infty, i \mapsto 0)$ 
fin si
 $((k, f), J) \leftarrow \min_k \left\{ \mathcal{BP}((a_i)_{i \in I}), I \middle| I \subsetneq \llbracket 1, m \rrbracket, I \neq \emptyset, \sum_{i \in I^c} a_i \leq 1 \right\}$ 4
    renvoyer  $(k+1, i \mapsto \mathbb{1}_J(i)f(i) + \mathbb{1}_{J^c}(k+1))$ 

```

Algorithme 3 Récursif glouton pour BIN-PACKING autorisant un nombre nul d'objets

Entrée: Une instance a_1, \dots, a_m d'objets pour le BIN-PACKING tous inférieurs à 1.

Sortie: Une solution (k, f) du BIN-PACKING.

```

si  $m = 0$  alors
    renvoyer  $(0, i \mapsto 0)$ 
fin si
 $((k, f), J) \leftarrow \min_k \left\{ \mathcal{BP}((a_i)_{i \in I}), I \middle| I \subsetneq \llbracket 1, m \rrbracket, \sum_{i \in I^c} a_i \leq 1 \right\}$ 
    renvoyer  $(k+1, i \mapsto \mathbb{1}_J(i)f(i) + \mathbb{1}_{J^c}(k+1))$ 

```

Algorithme 4 Récursif glouton élégant pour BIN-PACKING

Entrée: Une instance a_1, \dots, a_m d'objets pour le BIN-PACKING.

Sortie: Une solution (k, f) du BIN-PACKING.

```

si  $\sum_{i=1}^m a_i \leq 1$  alors
    renvoyer  $(1, i \mapsto 1)$ 
sinon
    renvoyer  $(\infty, i \mapsto 0)$ 
fin si
 $((k_1, f_1), (k_2, f_2), J) \leftarrow \min_{k_1+k_2} \{ (\mathcal{BP}((a_i)_{i \in I}), \mathcal{BP}((a_i)_{i \in I^c}), I) \mid I \subsetneq \llbracket 1, m \rrbracket, I \neq \emptyset \}$ 
    renvoyer  $(k_1 + k_2, i \mapsto \mathbb{1}_J(i)f_1(i) + \mathbb{1}_{J^c}(f_2(i) + k_1))$ 

```

4. $\min_k \{x, x \in A\}$ désigne un élément de A tq $k(x)$ soit minimal, ici par abus k est un paramètre.

A.3 Un algorithme plus efficace en pratique

Algorithme 5 Récursif glouton pour BIN-PACKING au rang k

Entrée: Une instance a_1, \dots, a_m d'objets pour le BIN-PACKING, un entier naturel k .

Sortie: vérifier que l'instance d'objets est satisfiable pour k bins.

```

si  $k = 1$  alors
    renvoyer  $\sum_{i=1}^m a_i \leq 1$ 
fin si
pour  $i = 0$  to  $m - 1$  faire
    pour tout  $J \in \mathcal{P}_i(\llbracket 1, m \rrbracket)$  faire
        si  $\sum_{j \in J^c} a_j \leq 1$  and l'instance  $(a_j)_{j \in J}$  satisfiable pour  $k - 1$  bins alors
            renvoyer vrai
        fin si
    fin pour
fin pour
renvoyer vrai
```

Algorithme 6 Récursif glouton pour BIN-PACKING au rang k moins gourmand

Entrée: Une instance a_1, \dots, a_m d'objets pour le BIN-PACKING, un entier naturel k .

Sortie: vérifier que l'instance d'objets est satisfiable pour k bins.

```

si  $k = 1$  alors
    renvoyer  $\sum_{i=1}^m a_i \leq 1$ 
fin si
pour  $i = k - 1$  to  $m - 1$  faire
    pour tout  $J \in \mathcal{P}_i(\llbracket 1, m \rrbracket)$  faire
        si  $\sum_{j \in J^c} a_j \leq 1$  and l'instance  $(a_j)_{j \in J}$  satisfiable pour  $k - 1$  bins alors
            renvoyer vrai
        fin si
    fin pour
fin pour
renvoyer vrai
```

A.4 Heuristiques

Heuristiques classiques

Algorithme 7 NEXT-FIT.

Entrée: Une instance a_1, \dots, a_n , d'objets pour le BIN-PACKING.

Sortie: Une solution (k, f) non optimale.

```

 $k \leftarrow 1$ 
 $S \leftarrow 0$ 
pour  $i = 1$  à  $n$  faire
    si  $S + a_i > 1$  alors
         $k \leftarrow k + 1$ 
         $S \leftarrow 0$ 
    fin si
     $f(i) \leftarrow k$ 
     $S \leftarrow S + a_i$ 
fin pour
```

Algorithme 8 Algorithme du FIRST-FIT, version théorique.**Entrée:** Une instance a_1, \dots, a_n , d'objets pour le BIN-PACKING**Sortie:** Une solution (k, f)

```

pour  $i = 1$  à  $n$  faire
     $f(i) \leftarrow \min \left\{ j \in \mathbb{N} \mid \sum_{\substack{h \in f^{-1}\{j\} \\ h < i}} a_h + a_i \leq 1 \right\}$ 
fin pour
 $k \leftarrow \max_{i \in [1, n]} f(i)$ 

```

Algorithme 9 FIRST-FIT, version pratique.**Entrée:** Une instance a_1, \dots, a_n , d'objets pour le BIN-PACKING**Sortie:** Une solution (k, f)

```

 $listeBins \leftarrow []$ 
pour  $i = 1$  à  $n$  faire
     $m \leftarrow \text{longueur}(listeBins)$ 
     $j \leftarrow 0$ 
    tant que  $j < m$  et  $a_i > listeBins[j]$  faire
         $j \leftarrow j + 1$ 
    fin tant que
    si  $j < m$  alors
         $listeBins[j] \leftarrow listeBins[j] - a_i$ 
    sinon
         $listeBins \leftarrow listeBins + [1 - a_i]$ 
    fin si
     $f(i) \leftarrow j + 1$ 
fin pour
 $k \leftarrow \text{longueur}(listeBins)$ 

```

Algorithme 10 FIRST-FIT DÉCROISSANT.**Entrée:** Une instance a_1, \dots, a_n , d'objets pour le BIN-PACKING**Sortie:** Une solution (k, f)

trier l'instance en permutant de telle sorte que $a_{\sigma(1)} \geq a_{\sigma(2)} \geq \dots \geq a_{\sigma(n)}$
appliquer l'algorithme du FIRST-FIT à l'instance $a_{\sigma(1)}, \dots, a_{\sigma(n)}$

Le tri par catégories**Algorithme 11** FERNANDEZ DE LA VEGA ET LÜCKER [1]**Entrée:** Une instance a_1, \dots, a_n , d'objets pour le BIN-PACKING, un nombre $\epsilon > 0$ **Sortie:** Une solution (k, f) pour \mathcal{I}

$$\gamma \leftarrow \frac{\epsilon}{\epsilon+1}$$

$$h \leftarrow \left\lceil \epsilon \sum_{i=1}^n a_i \right\rceil$$

On trie l'instance d'objets dans l'ordre décroissant cela donne b_1, \dots, b_n $R, K_1, \dots, K_m, L \leftarrow b_1, \dots, b_n$ où— $\forall x \in L, x < \gamma$ et on pose $l := |L|$ — $|R| = h - 1$ — $\forall i < m, |K_i| = h$ et $|K_m| = r + 1$ — $\forall i \in [1, m]$, on pose $y_i := \max(K_i) = K_i[1]$ Ajouter les objets de R dans $|R|$ bins, en assignant les indices avec $f : i \mapsto i$ Trouver une solution avec au plus $opt(Q) + \frac{m+1}{2}$ bins grâce à la méthode du BIN-PACKING par classes où Q est l'instance des $(y_i)_{1 \leq i \leq m}$ avec h copies de chaque y_i . Puis réattribuer cette assignation aux éléments de K_1, \dots, K_m Assigner les éléments non assignés des K_i avec le FIRST-FIT.Assigner les éléments de L avec le NEXT-FIT

B Complexité des algorithmes gloutons

Afin de calculer la complexité de l'algorithme glouton pour le problème du BIN-PACKING, nous allons utiliser la relation de récurrence déduite de l'algorithme qui packe m objets dans k bins, avec k optimal :

$$\text{on note } d \text{ la fonction donnée par } \begin{cases} \forall(m, k) \in \mathbb{N} \times \mathbb{N}^*, d(m, k+1) = \sum_{i=k}^{m-1} \binom{m}{i} d(i, k) + P(i) \\ \forall m \in \mathbb{N}, d(m, 1) = Q(m), \text{ avec } P \text{ et } Q \text{ des polynômes à choisir et } Q(0) = 0. \end{cases} .$$

B.1 Une fonction classique pour les sommes à coefficients binomiaux

On introduit $\forall m \in \mathbb{N}$, $f_m : \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \mapsto & (1+x)^m \end{array}$, on a alors $\forall x \in \mathbb{R}$, $f_m(x) = \sum_{i=0}^m \binom{m}{i} x^i$, et f_m est de classe C^∞ , et $\forall x \in \mathbb{R}$, $f'_m(x) = m(1+x)^{m-1} = \sum_{i=1}^m \binom{m}{i} i x^{i-1}$.

On a donc $\sum_{i=1}^{m-1} \binom{m}{i} i x^{i-1} = f'_m(x) - \binom{m}{m} m x^{m-1} = m((x+1)^{m-1} - x^{m-1})$.

B.2 Une conjecture

Calculons pour $m \in \mathbb{N}$ les premiers termes de c , définie par $\begin{cases} \forall(m, k) \in \mathbb{N} \times \mathbb{N}^*, c(m, k+1) = \sum_{i=k}^{m-1} \binom{m}{i} c(i, k) \\ \forall m \in \mathbb{N}, c(m, 1) = m \end{cases}$

- $c(m, 2) = \sum_{i=1}^{m-1} \binom{m}{i} c(i, 1) = \sum_{i=1}^{m-1} \binom{m}{i} i = m(2^{m-1} - 1)$
- $c(m, 3) = \sum_{i=1}^{m-1} \binom{m}{i} c(i, 2) = \sum_{i=1}^{m-1} \binom{m}{i} i(2^{i-1} - 1) = m((3^{m-1} - 2^{m-1}) - (2^{m-1} - 1)) = m(3^{m-1} - 2 \cdot 2^{m-1} + 1)$
- $c(m, 4) = m((4^{m-1} - 3^{m-1}) - 2(3^{m-1} - 2^{m-1}) + (2^{m-1} - 1)) = m(4^{m-1} - 3 \cdot 3^{m-1} + 3 \cdot 2^{m-1} - 1)$

On retrouve ici le mécanisme de construction usuel des coefficients binomiaux.

On conjecture donc que $\forall m, k \in \mathbb{N} \times \mathbb{N}^*$, $c(m, k) = m \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-1}$.

B.3 Preuve de la complexité auxiliaire

Montrons la conjecture par récurrence sur k : $\forall k \in \mathbb{N}^*$, on pose :

$$P_k : \text{"}\forall m \in \mathbb{N}, c(m, k) = m \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-1}\text{"}.$$

- La propriété a déjà été vérifiée pour $k = 1$.
- Soit $k \geq 1$ tel que P_k soit vérifiée, soit $m \in \mathbb{N}$,

$$\begin{aligned} c(m, k+1) &= \sum_{i=1}^{m-1} \binom{m}{i} c(i, k) = \sum_{i=1}^{m-1} \binom{m}{i} i \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{i-1} = \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} \sum_{i=1}^{m-1} \binom{m}{i} i j^{i-1} \\ &= \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} m((j+1)^{m-1} - j^{m-1}) = m \left(\sum_{j=2}^{k+1} (-1)^{k+1-j} \binom{k-1}{j-2} j^{m-1} - \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-1} \right) \\ &= m \left(\binom{k-1}{k-1} (k+1)^{m-1} + \sum_{j=2}^k (-1)^{k+1-j} \left(\binom{k-1}{j-2} + \binom{k-1}{j-1} \right) j^{m-1} - (-1)^{k-1} \binom{k-1}{0} \right) \\ &= m \left(\binom{k}{k} (k+1)^{m-1} + \sum_{j=2}^k (-1)^{k+1-j} \binom{k}{j-1} j^{m-1} + (-1)^{k+1-1} \binom{k-1}{0} \right) = m \sum_{j=1}^{k+1} (-1)^{k+1-j} \binom{k}{j-1} j^{m-1} \end{aligned}$$

Donc P_{k+1} est vérifiée.

- Par récurrence, $\forall k \in \mathbb{N}^*$, $\forall m \in \mathbb{N}$, $c(m, k) = m \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-1}$.

B.4 Généralisation à l'aide des polynômes de Newton

Pour le calcul de la complexité en dimension supérieure du problème du BIN-PACKING, Il est possible d'avoir une complexité polynomiale en m pour $k = 1$, il convient donc de calculer cette complexité pour $c(m, 1) = Q(m)$, avec $Q \in \mathbb{R}[X]$ et $c(m, k+1) = \sum_{i=0}^{m-1} \binom{m}{i} c(i, k)$.

Il est connu que la base des polynômes de Newton $(L_n)_{n \in \mathbb{N}}$ telle que $\forall n \in \mathbb{N}$, $L_n(X) = \prod_{l=0}^{n-1} X - l$ est utile dans les calculs des coefficients binomiaux. On obtient un résultat utile : Soit $m \in \mathbb{N}$, $p \in \mathbb{N}$,

$$\sum_{i=0}^m \binom{m}{i} L_p(i) x^{i-p} = f_m^{(p)}(x) = m(m-1) \cdots (m-p+1)(1+x)^{m-p} = L_p(m)(1+x)^{m-p},$$

$$\text{donc } \sum_{i=0}^{m-1} \binom{m}{i} L_p(i) x^{i-p} = f_m^{(p)}(x) = L_p(m)(1+x)^{m-p} - \binom{m}{m} L_p(m) x^{m-p} = L_p(m)((1+x)^{m-p} - x^{m-p})$$

On pose donc $r := \deg(Q)$ et $Q = \sum_{p=0}^r a_p L_p$, la décomposition unique en polynômes de Newton de P .

$$\text{On va montrer par récurrence que } \forall m, k \in \mathbb{N} \times \mathbb{N}^*, c(m, k) = \sum_{p=0}^r a_p L_p(m) \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-p}$$

- La propriété est vérifiée pour $k = 1$.

- Soit $k \geq 1$ tel que P_k soit vérifiée, soit $m \in \mathbb{N}$,

$$\begin{aligned} c(m, k+1) &= \sum_{i=1}^{m-1} \binom{m}{i} c(i, k) = \sum_{i=1}^{m-1} \binom{m}{i} \sum_{p=0}^r a_p L_p(i) \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{i-p} \\ &= \sum_{p=0}^r a_p \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} \sum_{i=1}^{m-1} \binom{m}{i} L_p(i) j^{i-p} = \sum_{p=0}^r a_p \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} L_p(m)((j+1)^{m-p} - j^{m-p}) \\ &= \sum_{p=0}^r a_p L_p(m) \left(\sum_{j=2}^{k+1} (-1)^{k+1-j} \binom{k-1}{j-2} j^{m-p} - \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-p} \right) \\ &= \sum_{p=0}^r a_p L_p(m) \left(\binom{k-1}{k-1} (k+1)^{m-p} + \sum_{j=2}^k (-1)^{k+1-j} \left(\binom{k-1}{j-2} + \binom{k-1}{j-1} \right) j^{m-p} - (-1)^{k-1} \binom{k-1}{0} \right) \\ &= \sum_{p=0}^r a_p L_p(m) \left(\binom{k}{k} (k+1)^{m-p} + \sum_{j=2}^k (-1)^{k+1-j} \binom{k}{j-1} j^{m-p} + (-1)^{k+1-1} \binom{k-1}{0} \right) \\ &= \sum_{p=0}^r a_p L_p(m) \sum_{j=1}^{k+1} (-1)^{k+1-j} \binom{k}{j-1} j^{m-p} \end{aligned}$$

Donc P_{k+1} est vérifiée.

- Par récurrence, $\forall k \in \mathbb{N}^*, \forall m \in \mathbb{N}, c(m, k) = \sum_{p=0}^r a_p L_p(m) \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-p}$.

B.5 Première transition

Calculons maintenant \tilde{d} définie par $\tilde{d}(m, 1) = Q(m)$, $Q(0) = 0$ et $\tilde{d}(m, k+1) = \sum_{i=k}^{m-1} \binom{m}{i} \tilde{d}(i, k)$. Un simple calcul des premiers termes dans le cas $Q(X) = X$ nous montre que $\tilde{d} = c$ définie précédemment, car pour $m < k$, $c(m, k) = 0$.

Ce résultat se montre assez facilement par récurrence, car on a au rang 1, $c(0, 1) = Q(0) = 0$, et pour tout $k > 0$, en supposant la propriété vraie à ce rang, pour $m < k+1$, $\tilde{d}(m, k+1) = \sum_{i=k}^{m-1} \binom{m}{i} c(i, k)$, or si $i < m-1$, alors $i < k$, donc $c(i, k) = 0$, donc $c(m, k+1) = 0$.

On peut donc maintenant montrer que $\tilde{d} = c$, encore une fois par récurrence : au rang 1, pour $m \in \mathbb{N}$, $\tilde{d}(m, 1) = Q(m) = c(m, 1)$, et pour $k > 0$ tel que la propriété soit vérifiée, $\tilde{d}(m, k+1) = \sum_{i=k}^{m-1} \binom{m}{i} \tilde{d}(i, k) = \tilde{d}(m, k+1) = \sum_{i=k}^{m-1} \binom{m}{i} c(i, k) = \tilde{d}(m, k+1) = \sum_{i=0}^{m-1} \binom{m}{i} c(i, k) - \tilde{d}(m, k+1) = \sum_{i=0}^{k-1} \binom{m}{i} c(i, k) = \sum_{i=0}^{m-1} \binom{m}{i} c(i, k) = c(i, k+1)$. Donc $\tilde{d} = c$.

B.6 Deuxième transition

Pour se rapprocher encore de la complexité désirée, il nous reste un calcul préliminaire. Posons pour P et Q , des polynômes, c telle que $c(m, 1) = Q(m)$ et $c(m, k+1) = \sum_{i=0}^{m-1} \binom{m}{i} c(i, k) + P(i)$.

Introduisons \tilde{c} et \hat{c} définies comme c dans la partie généralisation avec les polynômes de Newton respectivement avec P et Q . Nous allons montrer que $c(m, k) = \hat{c}(m, k) + \sum_{l=2}^k \tilde{c}(m, l)$. La récurrence est encore une fois reine car les suites et les algorithmes sont définis de cette manière.

Pour $k = 1$, on a $c(m, 1) = Q(m) = \hat{c}(m, 1) + 0$, pour $k > 1$ tel que cette propriété soit vérifiée, on a $c(m, k+1) = \sum_{i=0}^{m-1} \binom{m}{i} c(i, k) + P(i) = \sum_{i=0}^{m-1} \binom{m}{i} \hat{c}(i, k) + \sum_{l=2}^k \tilde{c}(i, l) + P(i) = \sum_{i=0}^{m-1} \binom{m}{i} \hat{c}(i, k) + \sum_{l=1}^k \sum_{i=0}^{m-1} \tilde{c}(i, l) = \hat{c}(m, k+1) + \sum_{l=1}^k \tilde{c}(m, l+1) = \hat{c}(m, k+1) + \sum_{l=2}^{k+1} \tilde{c}(m, l)$. Donc la propriété est vérifiée à tout rang. Notons également que le même raisonnement est valable pour les complexités similaires à d

B.7 Conclusion

Pour la complexité de l'algorithme 5, cela revient à poser c de la manière suivante :

$$c(m, 1) = m$$

$$c(m, k + 1) = \sum_{i=0}^{m-1} \binom{m}{i} c(i, k) + i.$$

Cela donne donc au vu de la partie précédente $P(X) = Q(X) = X$, soit $c(m, k) = m \sum_{j=1}^k (-1)^{k-j} \binom{k-1}{j-1} j^{m-1} + \sum_{l=2}^k m \sum_{j=1}^l (-1)^{l-j} \binom{l-1}{j-1} j^{m-1}$. Soit une complexité de $m \sum_{l=1}^k (1 + \delta_l^k) \sum_{j=1}^{l-1} (-1)^{l-j} \binom{l-1}{j-1} j^{m-1}$

C Code Ocaml

```

let parties k m =
    let tab = Array.make_matrix (k+1) (m+1) [] in
    for i = 1 to k do
        for j = i to m do
            let lis = ref [] in
            for l = 1 to (j-i+1) do
                List.iter (function li -> lis := (l::(List.map (function x -> x+l) li))::(!lis)) (tab.(i-1).(j-1))
            done;
            tab.(i).(j) <- !lis
        done;
    done;
    tab;;
;

let separer liste listind =
    let rec sepaux liste listind i =
        match liste, listind with
            | _, [] -> [], liste
            | [], _ -> failwith "des indices en trop"
            | t::q,r::s when i=r -> let listtext,listcomp = sepaux q s (i+1) in
                (t::listtext),listcomp
            | t::q,_ -> let listtext,listcomp = sepaux q listind (i+1) in
                listtext,(t::listcomp)
        in
    sepaux liste listind 1;;
;

separer [1;2;3;4;5;6;6;7] [1;3;5;6;7];;
let rec longueur liste =
    match liste with
        | [] -> 0
        | t::q -> 1 + (longueur q);;
;

let rec somme liste =
    match liste with
        | [] -> 0
        | t::q -> t + somme q;;
;

let bin_packing_montant lisobj c=
    let m = longueur lisobj in
    let partie = parties m m in
    let rec bin_packing_k lisobj m k =
        if k=1 then ((somme lisobj) <= c)
        else(
            let i = ref (k-1) in
            while (!i < m) && not (List.exists (function liste -> let
                lisobjext,lisobjcomp = separer lisobj liste in ((somme lisobjcomp)
                <= c)&&(bin_packing_k lisobjext !i (k-1))) partie.(!i).(m)) do
                i := !i + 1
            done;
            (!i < m)) in
;
```

```

let rec bp_aux lisobj m k=
    if (bin_packing_k lisobj m k) then k else (bp_aux lisobj m (k+1)) in
    bp_aux lisobj m 1;;

```

D Code python

D.1 Descriptif

''Ce programme implémente un groupe de fonctions qui ont pour but de résoudre avec une bonne convergence le problème du Bin-Packing en une dimension. Elles utilise la méthode de l'algorithme génétique qui consiste à faire une selection naturelle sur plusieurs générations de solutions au problème du Bin-Packing.

- *les objets ici sont des entiers, le tableau des objets est trié dans le sens décroissant les Bins sont des entiers, les solutions sont un couple d'une liste de bins remplis et d'un tableau de m positions (les entiers correspondant à l'indice du Bin dans lequel est placé l'objet) et une génération est une liste de k solutions.*

- *En paramètres, nous avons:*
 - m le nombre d'objets à placer
 - k le nombre de solutions dans une génération
 - l le nombre de sélections naturelles

Le programme génétique fonctionne de la manière suivante:

- On génère une première population de solutions aléatoirement*
- Puis on réitère les étapes qui suivent:*
 - On effectue avec une probabilité d'exécution des mutations sur les éléments de la génération précédente si elles sont viables*
 - On trie la génération en fonction d'une fonction de notation (une note plus basse signifie que la solution est meilleure)*
 - Les meilleures solutions sont gardées*
 - Les pires sont supprimées*
 - On conserve momentanément la liste des solutions intermédiaires*
 - À l'aide d'une fonction de poids (à la manière de Boltzmann en physique statistique, on attribue une probabilité relative à chaque solution intermédiaire, on parlera quelques fois de pseudo-probabilité si la loi n'est pas normalisée)*
 - On effectue une roulette de sélection de solutions à partir de cette loi et de la liste de solutions intermédiaires, on garde les solutions sélectionnées dans la génération suivante*
 - On ajoute des solutions du type crossing-over, qui se constituent à l'aide de la liste de probabilité: pour chaque objet, on a une liste qui correspond à la probabilité de s'ajouter dans le bin numero i*
 - On ajoute des solutions générées aléatoirement. '''*

D.2 Importations et variables globales

```

import numpy as np
import random
from copy import deepcopy
from time import time
from math import ceil, floor, tan, atan as arctan
from scipy.optimize import linprog

## Variables globales

m = 100
alpha = 3
k = ceil(alpha*m)
beta = 1.
l = ceil(beta*m)

```

D.3 Fonctions élémentaires

Fonctions générales

```
def somme (X):
    """Pour X liste ou tableau de type int ou float, renvoie la somme de ses
éléments"""
    r = len(X)
    s = 0
    for i in range(r):
        s += X[i]
    return s

def chgt_de_var_aller_int (f,a,b):
    """
    Renvoie la fonction g qui est la fonction f à laquelle on a appliqué un
    changement de variable de [0,1] vers [a,b] en chemin convexe.
    •
    chgt_de_var_aller:
    fun(float, 'a') * float * float -> fun(float, 'a')
    •
    sert à:
    """
    return lambda x : f((1-x)*a + x*b)

def chgt_de_var_retour_int (f,a,b):
    """
    Renvoie la fonction g qui est la fonction f à laquelle on a appliqué un
    changement de variable de [a,b] vers [0,1] en chemin convexe.
    •
    chgt_de_var_retour:
    fun(float, 'a') * float * float -> fun(float, 'a')
    •
    sert à:
    """
    return lambda x : f((x-a)/(b-a))

def chgt_de_var_aller_ext (f,a,b):
    """
    Renvoie la fonction g qui est la fonction f à laquelle on a appliqué un
    changement de variable extérieur de [0,1] vers [a,b] en chemin convexe.
    •
    chgt_de_var_aller:
    fun(float, 'a') * float * float -> fun(float, 'a')
    •
    sert à:
    """
    return lambda x : (1-f(x))*a + f(x)*b

def chgt_de_var_retour_ext (f,a,b):
    """
    Renvoie la fonction g qui est la fonction f à laquelle on a appliqué un
    changement de variable extérieur de [a,b] vers [0,1] en chemin convexe.
    •
    chgt_de_var_retour:
    fun(float, 'a') * float * float -> fun(float, 'a')
    •
    sert à:
    """
    return lambda x : (f(x)-a)/(b-a)

def composition (f,g):
    """
    Renvoie la fonction fog.
```

```

•
composition:
fun('a', 'b') * fun('c', 'a') -> fun('c', 'b')
•
sert à:
"""
return lambda x : f(g(x))

```

Fonctions pour le tableau d'objet

```

def genereTabObjUnif ():

    """
    Génère avec une loi uniforme sur ]0,1[ un tableau de m objet de taille inférieure à n, et
    renvoie un couple de ce tableau trié en décroissant, et non trié.
    """
    ajout: unit -> (m int array) * (m int array)
    """
    sert aux variables globales
    """

    liste = [random.random() for i in range(m)]
    return np.array(sorted(liste, key = lambda x : -x)), np.array(liste)

# Fonction qui va de R dans ]0,1[ grâce à arctan et à un chemin convexe
de_R_vers_0_1 = chgt_de_var_retour_ext(arctan,-pi/2, pi/2)

# Fonction qui va de ]0,1[ dans R grâce à un chemin convexe et à tan
de_0_1_vers_R = composition(tan,chgt_de_var_aller_int(lambda x : x,-pi/2,pi/2))

def loi_Gauss_sur_0_1(centre, sigma = 1):
    """
    Génère un nombre aléatoire entre 0 et 1 de la manière suivante: avec une loi
    gaussienne centrée en centre (à la fin) et d'écart type sigma (au début), on
    génère aléatoirement un réel, que l'on envoie ensuite sur ]0,1[.
    """
    loi_Gauss_sur_0_1:
        float * float -> float
    """
    sert à:
    genereTabObjGauss
    """

    return de_R_vers_0_1(random.gauss(de_0_1_vers_R(centre), sigma))

def genereTabObjGauss (centre = 1/2):
    """
    Génère avec une loi gaussienne sur [0,1] un tableau de m objet de taille inférieure à n, et
    renvoie un couple de ce tableau trié en décroissant, et non trié.
    """
    ajout: unit -> (m int array) * (m int array)
    """
    sert aux variables globales
    """

    liste = [loi_Gauss_sur_0_1(centre) for i in range(m)]
    return np.array(sorted(liste, key = lambda x : -x)), np.array(liste)

def minoration():
    """
    Renvoie une minoration du nombre de bins nécessaire pour le packing de tabObj.
    """
    minoration:
        unit -> int
    """
    ne sert pas à une fonction en particulier
    """

```

```
    return ceil(somme(tabObj))
```

Fonctions pour les bins

```
def creerBin ():

    """
    Renvoie un bin vide.
    •
    ajout: unit -> int
    •
    sert à:
    -un peu partout
    """
    return 1.0

def estVide (bine):

    """
    Renvoie True si le bin est vide, False sinon.
    •
    ajout: int -> bool
    •
    sert à:
    -un peu partout
    """
    return bine == 1.0

def ajoutable (objet, bine):

    """
    Renvoie True si l'objet est ajoutable dans le bin, False sinon.
    •
    ajout: int * int -> bool
    •
    sert à:
    -un peu partout
    """
    return objet <= bine

def ajout (objet, bine):

    """
    Renvoie le bin auquel on a ajouté l'objet.
    •
    ajout: int * int -> int
    •
    sert à:
    -ajoutAlea
    """
    return bine - objet

def retire (objet, bine):

    """
    Renvoie le bin auquel on a enlevé l'objet.
    •
    ajout: int * int -> int
    •
    sert à:
    -mutationIndices
    """
    return bine + objet

def viableBin (bine):

    """
    Renvoie False si le bin est viable, True sinon.
    •
    """
    return False
```

```

viableBin: int -> bool
•
sert à:
-viableSol
"""

return bine >= 0

def viableSol (lisBin):
    """
    Renvoie True si la liste de bins est viable, False sinon.
    •
ajout: int -> bool
•
sert à:
-mutationIndices
"""

for bine in lisBin:
    if not(viableBin(bine)):
        return False
return True

def numero_vide (lisBin):
    """
    Renvoie le premier indice tel que à sa droite tous les bins soient vides.
    •
numero_vide:
int list -> int
•
sert à:
-generesolProba
"""

i = len(lisBin)
while i > 0 and estVide(lisBin[i-1]):
    i -= 1
return i

def evide (Liste):
    """
    Tronque une liste qui n'a que des bins vides à sa fin.
    •
evide:
int list -> int list
•
sert à:
-generesolProba
"""

i = len(Liste)
while i > 0 and estVide(Liste[i-1]):
    i -= 1
return Liste[0:i]

def evideSol (sol):
    """
    Retire les bins vides d'une solution.
    •
evideSol:
(int list * m int array) -> unit
•
sert à:
-mutationSol
"""

```

```

lisBin, tabPos = sol
r = len(lisBin)
i = 0
while i < r:
    # le variant de boucle est r-i
    if estVide(lisBin[i]):
        lisBin.pop(i)
        r -= 1
    for j in range(m):
        if tabPos[j] > i:
            tabPos[j] -= 1
    else:
        i += 1

```

Fonctions pour les probabilités

```

def extrait (tabProb):
    """
    Enlève les zéros du tableau de probabilité.
    •
    extract:
    m int array -> (int list) * (int list)
    •
    sert à:
    -genereSolProba
    """
    lisPos, lisProb = [], []
    for i in range(m):
        if tabProb[i] != 0:
            lisPos.append(i)
            lisProb.append(tabProb[i])
    return lisPos, lisProb

def normalisation (X):
    """
    Pour X liste ou tableau de type int ou float, renvoie un élément du même
    type normalisé (dont la somme des éléments vaut 1)
    """
    s = somme(X)
    r = len(X)
    for i in range(r):
        X[i] /= s

```

Fonctions pour le tri

```

def permute(Liste, permutation):
    """
    Applique une permutation à une liste.
    •
    permute:
    float list * int list -> float list
    •
    sert à:
    -genetik
    """
    return [Liste[permutation[i]] for i in range(len(Liste))]

def inverse (permutation):
    """
    Inverse une permutation.
    •
    inverse:
    int list -> int list
    •
    sert à:
    -genetik
    """

```

```

"""
r = len(permuation)
Liste2 = [-1]*r
for i in range(r):
    Liste2[permuation[i]] = i
return Liste2

def triAssocie (N):
    """
    À partir d'une liste de note, associe la permutation associée au tri de celle ci.
    •
    triAssocie:
    float list -> int list
    •
    sert à:
    -genetik
    """
    r = len(N)
    return (sorted([i for i in range(r)], key = lambda j : N[j]))

```

D.4 Heuristiques classiques

NEXT-FIT

```

def next_fit():
    """
    Renvoie une solution au problème du Bin-Packing pour l'instance tabObj, avec la méthode du Next-Fit.
    •
    next_fit:
    unit -> int list * m int array
    •
    ne sert pas à une fonction en particulier
    """
    lisBin = [creerBin()]
    tabPos = np.zeros(m)
    j = 0
    for i in range(m):
        if ajoutable(tabObj[i],lisBin[j]):
            lisBin[j] = ajout(tabObj[i],lisBin[j])
        else:
            lisBin.append(ajout(tabObj[i],creerBin()))
            j += 1
        tabPos[i] = j
    return lisBin, tabPos

```

FIRST-FIT

```

def first_fit():
    """
    Renvoie une solution au problème du Bin-Packing pour l'instance tabObj, avec la méthode du First-Fit.
    •
    first_fit:
    unit -> int list * m int array
    •
    ne sert pas à une fonction en particulier
    """
    lisBin = [creerBin()]
    tabPos = np.zeros(m)
    for i in range(m):

```

```

j = 0
while not(ajoutable(tabObj[i],lisBin[j])):
    if j == len(lisBin)-1:
        lisBin.append(creerBin())
    j += 1
lisBin[j] = ajout(tabObj[i],lisBin[j])
tabPos[i] = j
return lisBin, tabPos

```

FERNANDEZ DE LA VEGA ET LÜCKER

Paramètres

```

epsilon = np.sqrt(2/m)
# On trouve une précision plus élevée pour cette valeur

```

Fonctions

```
def ensembleP(tabS, tabB):
    """
    Calcule récursivement  $P := \{(k_1, \dots, k_n) \text{ dans } N^m \text{ tq } k_1.s_1 + \dots + k_n.s_n \leq 1 \text{ et pour tout } i \ k_i \leq b_i\}$ .
    """
    ensembleP: m float array * m int array -> (N,m) int array
    """
    Sert à:
    -bp_par_classes_simplexe_continu
    """

```

```
n = len(tabS)
```

```
def aux(c,i):
    """À partir de la capacité restante, on calcule les indices à droite"""
    # possibles
    if i == n:
        # Si on est à la fin du tableau, plus d'indices à droite, on renvoie
        # donc que des zéros
        return [[0]*n]
    else:
        # Sinon, on prend les valeurs de  $k_i$  possibles
        L = []
        kmax = min(floor(c/tabS[i]), tabB[i])
        for k in range(kmax + 1):
            #  $k_i$  parcourt les entiers de 0 à  $\min(E(c/s_i), b_i)$ 
            l = aux(c-k*tabS[i],i+1)
            # On calcule tous les indices possibles à droite avec une capacité
            # restante de  $c - k_i.s_i$ 
            for j in range(len(l)):
                # On leur attribue la valeur  $k_i$  pour l'indice  $i$ 
                l[j][i] = k
            L += l
        # On ajoute ces indices à la liste des indices
        return L
    return np.array(aux(1.0,0))
```

```
def bp_par_classes_simplexe_continu (tabS, tabB):
    """
    Renvoie les solutions pour la relaxation linéaire du BP par classes, méthode de programmation linéaire.
    """
    bp_par_classes_simplexe_continu:
    m float array * m int array -> N,m float array * (m,N) int array
    """
    Sert à:
    -bp_par_classes_simplexe
    
```

```

    """
P = ensembleP(tabS,tabB)
# On calcule les T_1,...,T_N
A = -P.T
M,N = np.shape(A)
b = -tabB
c = np.array([1]*N)
# Puis A, b, c les matrices de la programmation linéaire
s = linprog(c, A_ub = A, b_ub = b, method='simplex')
return s.x, -A

def bp_par_classes_simplexe (tabS, tabB):
    """
Renvoie le tableau des objets de tabs partiellement assignés, de manière à
respecter les contraintes du BP.
•
bp_par_classes_simplexe: m float array * m int array -> m (int list) array
•
Sert à:
-vega
"""

n = len(tabS)
X, A = bp_par_classes_simplexe_continu(tabS, tabB)
Y = np.vectorize(floor)(X)
S = np.dot(A,Y)
for i in range(n):
    S[i] = min(tabB[i], S[i]) #nombre d'objets à traiter a priori
tabPos = [[-1]*tabB[i] for i in range(n)]
M,N = np.shape(A)
c = np.array([1]*N)
k = np.dot(c,Y)
avancement = [0]*n
l = 0
for i in range(N): #pour tous les Ti
    for j in range(Y[i]): # pour le nombre de fois ou on utilise la package Ti
        for p in range(M): #pour tous les objets
            for q in range(A[p][i]): #nombre de fois où l'on packe bp dans Ti
                if avancement[p] < S[p]:
                    tabPos[p][avancement[p]] = 1
                    avancement[p] += 1
    l += 1
return tabPos

def vega():
    """
Renvoie la solution approchée à BP selon la méthode de Fernandez de la Vega
et Lücker.
•
bp_par_classes_simplexe: m float array * m int array -> m (int list) array
•
Ne sert à aucune fonction en particulier
"""

## Tri
perm_tri_tab = triAssocie(-tabObj)
inv_perm = inverse(perm_tri_tab)
I = permute(tabObj, perm_tri_tab)

## Variables
gamma = epsilon/(1. + epsilon)
h = ceil(epsilon * somme(tabObj))

```

```

## Découpage
i = premier_inf(I, gamma)
R, M, L = I[:h-1], I[h-1:i], I[i:]
s = len(M)
q, r = s//h, s%h
K = [M[h*j:h*(j+1)] for j in range(q-1)] + [M[-r:]]*int(r != 0)
n = len(K)

## Simplexe
B = np.array([len(K[j]) for j in range(n)])
S = np.array([K[j][0] for j in range(n)])
tabPos = bp_par_classes_simplexe(S,B)

## Réalisation
lisBin = [creerBin()]*m
tabPos2 = [-1]*m

# On paque R
for j in range(h-1):
    tabPos2[j] = j
    lisBin[j] = ajout(I[j], lisBin[j])
l = h-1

# Ajout de ceux du simplexe
for j in range(n):
    for p in range(len(tabPos[j])):
        q = tabPos[j][p]
        if q != -1:
            if ajoutable(I[l], lisBin[q + h-1]):
                tabPos2[l] = q + h-1
                lisBin[q + h-1] = ajout(I[l], lisBin[q + h-1])
            else:
                tabPos2[l]=-1
        l += 1
lisBin2 = deepcopy(lisBin)

# Ajout du reste en First-Fit
for j in range(m):
    if tabPos2[j] == -1:
        l = 0
        while not(ajoutable(I[j], lisBin[l])):
            l += 1
        tabPos2[j] = l
        lisBin[l] = ajout(I[j], lisBin[l])

return evide(lisBin), np.array(permute(tabPos2, inv_perm))

```

D.5 Métaheuristiques

Aléatoire

```

def alea_classique ():

    lisBin = [creerBin()]*m
    tabPos = np.array([-1]*m)
    for i in range(m):
        liste = [i for i in range(m)]
        random.shuffle(liste)
        j = 0
        while not(ajoutable(tabObj[i],lisBin[liste[j]])):
            j += 1
        lisBin[liste[j]] = ajout(tabObj[i],lisBin[liste[j]])
        tabPos[i] = liste[j]
    sol = (lisBin,tabPos)

```

```
evideSol(sol)
return sol
```

Aléatoire décroissant

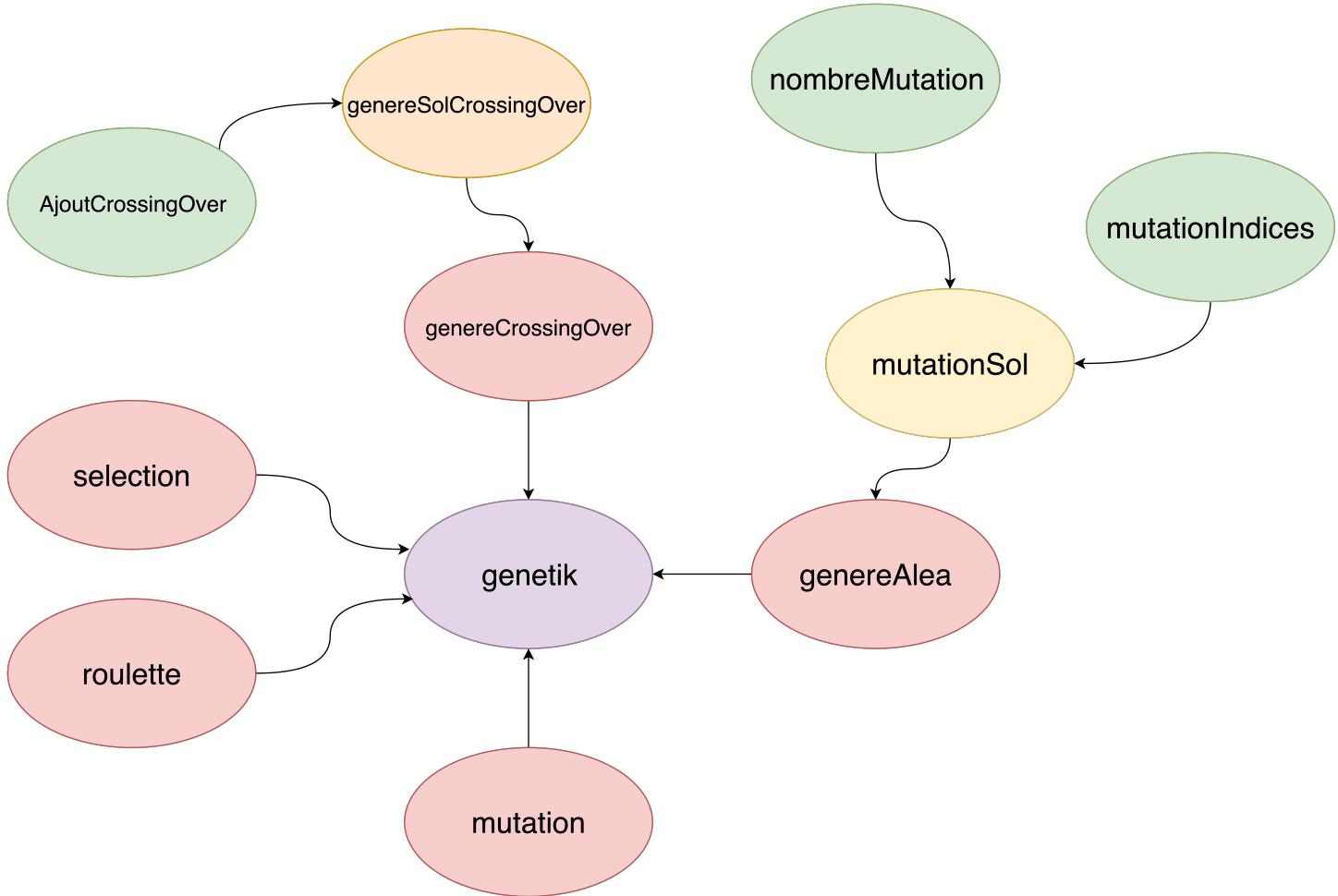
```
def ajoutAlea (objet, i, lisBin, r, tabPos):
    """
    Modifie la liste des Bins et la liste des Positions en ajoutant aléatoirement
    l'objet indexé par i.
    """
    ajoutAlea: int * int * int list * int * m int array -> unit
    """
    Sert à:
    -generesSolAlea"""
    liste = [j for j in range(r+1)]
    random.shuffle(liste)
    # permute les indices de 0 à r, pour savoir l'ordre dans lequel on va
    # considérer les bins
    j = 0
    cd = True
    while cd:
        # puis on considère les indices un à un et si l'indice vaut r on crée un
        # nouveau bin, sinon dès que l'on peut ajouter un objet au bin i, on
        # l'ajoute.
        s = liste[j]
        if s == r:
            lisBin.append(ajout(objet, creerBin()))
            tabPos[i] = r
            cd = False
        elif ajoutable(objet, lisBin[s]):
            lisBin[s] = ajout(objet,lisBin[s])
            tabPos[i] = s
            cd = False
        else:
            j += 1

def genereSolAlea():
    """
    À partir de tabObj, renvoie une solution aléatoire au problème du Bin-Pack.
    """
    genereSolAlea: unit -> int list * m int array
    peut servir à:
    genere"""
    lisBin = []
    # on crée la liste de Bins
    tabPos = np.array([-1]*m)
    # on crée le tableau des positions
    for i in range(m):
        # On ajoute un à un les objets dans les bins
        r = len(lisBin)
        ajoutAlea(tabObj[i], i, lisBin, r, tabPos)
    return lisBin,tabPos

def genereAlea (q):
    """
    Renvoie une liste de q solutions générées aléatoirement.
    """
    genere:
    m int array * int * fun(m int array, int list * m int array) -> (int list *
    m int array) list
    """
    sert à:
    -genetik"""
```

```
return [genereSolAlea() for j in range(q)]
```

Algorithme génétique



Paramètres

```

pourcentageMeilleurs = 1/8
# pourcentage de meilleures solutions conservées
pourcentagePires = 1/8
# pourcentage de pires solutions supprimées
pourcentageCrossingOver = 1/8
# pourcentage de solutions générées par la méthode de Crossing-Over parmi le
# nombre de solutions intermédiaires
probaMut = 1/2
# raison de la loi géométrique relative aux mutations

nM = round(k*pourcentageMeilleurs)
# nombre de meilleures solutions conservées
nP = round(k*pourcentagePires)
# nombre de pires solutions supprimées
nI = k - (nM + nP)
# nombre de solutions jugées intermédiaires
nCO = round(nI*pourcentageCrossingOver)
# pourcentage de solutions générées par Crossing-Over
nR = nI - nCO
# nombre de solutions sélectionnées par la roulette
nA = nP
# nombre de solutions générées aléatoirement

## Fonction de poids
```

```

proba = lambda x : np.exp(-x/(m/2))

## Mutation

def nombreMutation () :
    """
    Résultat d'une variable aléatoire suivant la loi géométrique de raison p.
    •
    nombreMutation: unit -> int
    •
    sert à:
    -mutation...
    """

    if random.random() < probaMut:
        return 1 + nombreMutation()
    else:
        return 0

def mutationIndices (sol, listeChgt):
    """
    À partir d'une solution et d'une liste de changements à opérer, renvoie la
    solution où l'on a opéré ces changements, si elle est viable False sinon.
    •
    mutationIndices:
    int list * m int array * (int * int) list * m int array * m float array ->
    unit
    •
    sert à:
    -mutation"""
    lisBin, tabPos = deepcopy(sol)
    for x in listeChgt:
        # x est un couple d'un indice d'objet, et d'un indice dans la liste de
        # bins
        iObj, jBin = x
        lisBin[tabPos[iObj]], lisBin[jBin], tabPos[iObj] = retire(tabObj[iObj],lisBin[tabPos[iObj]]), ajout
        # On déplace l'objet de son bin initial au nouveau bin, et on change
        # l'assignation dans le tableau
    if viableSol(lisBin):
        return lisBin, tabPos
    else:
        return False

def mutationSol (g, i):
    """
    À partir d'une génération et d'un indice i, effectue une mutation sur l'indice
    i.
    •
    mutationSol:
    (int list * m int array) list * int -> unit
    •
    sert à:
    -mutation"""

    sol = g[i]
    r = len(sol[0])
    nb = nombreMutation()
    # on calcule avec une loi de probabilité discrète le nombre de mutations que

```

```

# va subir la solution
if nb > 0:
    listeMut = [(random.randint(0,m-1), random.randint(0,r-1)) for i in range(nb)]
    sol2 = mutationIndices(sol, listeMut)
    if sol2 != False:
        evideSol(sol2)
        # on calcule la solution mutée sans modifier la première
        if note(sol2[0]) < note(sol[0]):
            # si elle est mieux, on remplace la solution initiale
            g[i] = sol2

def mutation (g):
    """
    Fait muter une génération.
    •
    mutationSol:
    (int list * m int array) list * int -> unit
    sert à:
    -genetik"""

    for i in range(k):
        mutationSol(g,i)

## Roulette

def selection (g):
    """
    Opère la sélection selon les paramètres, en renvoyant à partir d'une
    génération, la liste des meilleures, et la liste des intermédiaires.
    •
    selection:
    (int list * m int array) list -> (int list * m int array) list * (int list *
    m int array) list
    •
    sert à:
    -genetik"""

    return g[0:nM], g[nM:nM+nI], g[nM+nI:k]

def roulette (lisInter, P, q):
    """
    Opère la sélection selon les paramètres, en renvoyant à partir d'une
    génération, la liste des meilleures, et la liste des intermédiaires.
    •
    roulette:
    (int list * m int array) list -> (int list * m int array) list
    •
    sert à:
    -genetik"""

    tabR = np.random.choice([i for i in range(nI)], q, replace = False, p = P)
    return [lisInter[tabR[i]] for i in range(q)]

## Génération par Crossing-Over

def ajoutCrossingOver(i, objet, lisBin, tabPos, tabProb):
    """
    •

```

À partir d'un objet, de son indice et de son tableau de probabilité associé et de la solution provisoire, ajoute selon ce schéma l'objet.

- ```
ajoutCrossingOver: int * int * int list * m int array * m float array -> unit
sert à:
-genereSolCrossingOver"""
```

```
lisPos, lisProb = extrait(tabProb)
r = len(lisPos)
liste = np.random.choice(lisPos, r, replace = False, p = lisProb)
j = 0
cd = True
while j < r and cd:
 # on ajoute les objets selon le tirage sans remise avec la loi de
 # probabilité pour les position des bins choisies
 s = liste[j]
 if ajoutable(objet, lisBin[s]):
 lisBin[s] = ajout(objet, lisBin[s])
 tabPos[i] = s
 cd = False
 else:
 j += 1
if cd:
 # si cela n'a pas été possible, on ajoute aléatoirement l'objet, selon
 # le même schéma
 r2 = numero_vide(lisBin)
 liste2 = [j for j in range(r2+1)]
 random.shuffle(liste2)
 j2 = 0
 cdt = True
 while cdt:
 s2 = liste2[j2]
 if s2 == r2 and r2 < m:
 lisBin[r2] = ajout(objet, creerBin())
 tabPos[i] = r
 cdt = False
 elif ajoutable(objet, lisBin[s2]):
 lisBin[s2] = ajout(objet, lisBin[s2])
 tabPos[i] = s
 cdt = False
 else:
 j2 += 1

def genereSolCrossingOver(M):
 """
 À partir de tabObj et d'une matrice de probabilités, renvoie une solution
 du type crossing-over au problème du Bin-Packing
 •
 genereSolCrossingOver: m*m int matrix -> int list * m int array
 sert à:
 -genereCrossingOver"""
 lisBin = [creerBin()]*m
 tabPos = np.array([-1]*m)
 for i in range(m):
 ajoutCrossingOver(i,tabObj[i],lisBin,tabPos,M[i])
 return (evide(lisBin),tabPos)

def genereCrossingOver(q, M):
 """
 À partir de tabObj et d'une matrice de probabilités, renvoie une génération
```

```

de q solutions du type crossing-over au problème du Bin-Packning
•
genereCrossingOver: m*m int matrix -> (int list * m int array) list
sert à:
-genetik"""

return [genereSolCrossingOver(M) for j in range(q)]

Fonctions de notation

def vide (lisBin):
 """
 Renvoie un chiffre proportionnel à la somme du vides dans chaque bin au carré.
 •
 vide:
 int list -> float
 •
 sert à:
 note"""

r = len(lisBin)
s = 0
for i in range(r):
 s += lisBin[i]**2
return np.sqrt(s/r)

def note(lisBin):
 """
 Avec une liste de bins, renvoie sa note définie par taille(lisBin) +
 vide(lisBin).
 •
 note:
 int list -> float
 •
 Sert à:
 -noteSolution"""

r = len(lisBin)
return r + vide(lisBin)

def noteGeneration (g):
 """
 Avec une génération, renvoie le tableau des notes associées)
 •
 note:
 k (int list * m int array) array -> k float array
 •
 Sert à:
 -noteSolution"""

return np.array([note(g[i][0]) for i in range(k)])

def probaGeneration(N):
 """
 Avec le tableau des notes associées, renvoie le tableau des
 pseudo-probabilités associées
 •
 note:

```

```

k (int list * m int array) array -> k float array
•
Sert à:
-noteSolution"""

return np.array([proba(N[i]) for i in range(k)])

def matriceNotePositions (g,P):
 """
 À partir d'une génération ou d'un extrait de génération et de la liste des
 pseudo-probabilités associées, renvoie le tableau des tableaux des notes
 cumulées normalisées.
 •
 genereProba
 (int list * m int array) list * m float array -> m*m int array
 •
 sert à:
 -la roulette
 """
 s = len(g)
 T = np.zeros((m,m))
 for i in range(m):
 for r in range(s):
 lisBin, lisPos = g[r]
 T[i][lisPos[i]] += P[r]
 normalisation(T[i])
 return T

Fonction génétique

def genetik():
 """
 Fonction attribuant une sélection naturelle sur l générations de k
 individus avec comme critère de selection la valuation d'un bin:
 valuationBin et la notation note et tabObj le tableau des m
 objets. Renvoie dernière génération.
 •
 genetik:
 unit -> (int list * m int array) list
 """

 ## première génération

 g = genereAlea(k)
 # On génère la première génération de solution aléatoirement.

 ## sélection naturelle

 for i in range(l):
 print('génération:', i+1, '/', l)

 # Mutation
 mutation(g)

 # Notation
 N = noteGeneration(g)
 P = probaGeneration(N)

 # Tri selon Note

```

```
perm = triAssocie(N)
g = permute(g,perm)
N = permute(N,perm)
P = permute(P,perm)

Écrémage
lisM, lisI, lisP = selection(g)
PI = P[nM:nM+nI]
lisCO = lisM + lisI
PCO = P[0:nM+nI]

Roulette
normalisation(PI)
lisR = roulette(lisI,PI,nR)

Crossing-Over
M = matriceNotePositions(lisCO,PCO)
lisCO = genereCrossingOver(nCO,M)

Aléatoire
lisA = genereAlea(nA)

Nouvelle Génération
g = lisM + lisR + lisCO + lisA

résultat

g.sort(key = lambda x : note(x[0]))

return g
```