LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

Nantes Université

ACO pour le Set Packing Problem (SPP)

$$\begin{bmatrix} \text{Max z} = & \Sigma_{i \in I}\, c_i x_i, & \\ & \Sigma_{i \in I}\, a_{i,j} x_i \leqslant 1, & \forall j \in J, \\ & x_i \in \{0,1\}, & \forall i \in I, \\ & a_{i,j} \in \{0,1\}, & \forall i \in I, \forall j \in J \end{bmatrix}$$

Métaheuristiques
Compte rendu DM3

Juanfer MERCIER – Adrien PICHON

Samedi 26 Novembre 2022

Université de Nantes — UFR Sciences et Techniques
Master informatique parcours Optimisation et Recherche Opérationnelle (ORO)
Année académique 2022-2023

Table des matières

Chapitre 1

Présentation du problème

Le set packing est un **problème NP-complet** en théorie de la complexité et en combinatoire [1]. Soit un ensemble fini I et une collection de sous-ensembles de I. Le problème du set packing (SPP) consiste à vérifier si quelques sous-ensembles de la collection sont deux à deux disjoints. Par ailleurs, il existe une version du problème comme problème d'optimisation dans laquelle le but du SPP est de maximiser le nombre d'ensembles deux à deux disjoints dans la collection d'ensembles [1]. Dans le cadre du devoir maison, c'est cette version du problème qui nous intéresse et que nous formulerons. Nous verrons ensuite comment la résolution du SPP contribue aux problèmes de capacité d'infrastructure ferroviaire.

1.1 Le SPP comme problème d'optimisation

En optimisation, le SPP peut être formulé comme un **programme linéaire en nombres entiers** [2] (Delorme, Gandibleux et Rodriguez, 2004). Soit un ensemble fini $I = \{1, ..., n\}$ d'éléments valués et $\{T_j\}, j \in J = \{1, ..., m\}$, une collection de sous-ensembles de I, un set packing est un sous-ensemble $P \subseteq I$ tel que $|T_j \cap P| \leqslant 1, \forall j \in J$. On dit alors que P est une solution admissible du problème [3]. L'ensemble J peut aussi être considéré comme un ensemble de contraintes entre les éléments de l'ensemble I. Chaque élément $i \in I$ a un poids positif noté c_i et l'objectif du SPP est alors d'obtenir le packing qui maximise le poids total [4] (Gandibleux et al., 2005). Le problème peut alors être formulé par un modèle mathématique (1) :

$$\begin{bmatrix} \text{Max z} = & \sum_{i \in I} c_i x_i, & \\ & \sum_{i \in I} a_{i,j} x_i \leqslant 1, & \forall j \in J, \\ & x_i \in \{0,1\}, & \forall i \in I, \\ & a_{i,j} \in \{0,1\}, & \forall i \in I, \forall j \in J \end{bmatrix} \tag{1}$$

avec

- un vecteur de variables $X = (x_i)$ où $x_i = \begin{cases} 1 & \text{si } i \in P \\ 0 & \text{sinon} \end{cases}$,
- un vecteur $C = (c_i)$ où $c_i =$ valeur de l'élément i,
- une matrice $A = (a_{i,j})$ où $a_{i,j} = \begin{cases} 1 & \text{si } i \in T_j \\ 0 & \text{sinon} \end{cases}$

En pratique, le SPP permet de modéliser des situations réelles. Sa résolution contribue à de nombreux domaines dont la logistique et le transport.

1.2 Application du SPP aux problèmes de capacité d'infrastructure ferrovaire

En France, le trafic de voyageurs a globalement augmenter depuis 1954. La FIGURE 1 montre que le transport ferroviaire de passagers n'a connu que deux périodes de récessions depuis 1954, entre 1967 et 1969 et entre 1989 et 1995.

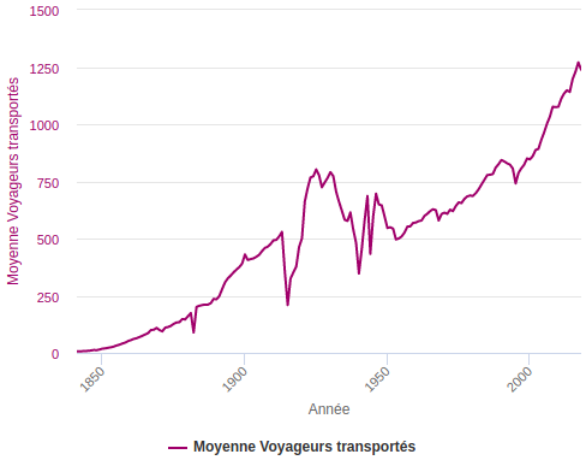

FIGURE 1 – Moyenne des voyageurs transportés par la SNCF depuis 1937 et par les compagnies privées qu'elle a absorbées, avant 1937. [5] [6]

Étant donné que de plus en plus de voyageurs voient le transport ferroviaire comme alternative à la route ou à d'autres moyens de locomotion, les opérateurs de lignes ferrées n'ont d'autres choix que d'augmenter la capacité des infrastructures ferroviaires. Les gestionnaires doivent alors gérer l'aménagement des infrastructures pour répondre à la demande des opérateurs. Cet aménagement est un processus délicat de par les investissements majeurs et les travaux de longue durée qu'il implique. Il faut alors évaluer la capacité d'une composante d'un système ferroviaire et, habituellement, cette évaluation se fait en mesurant le nombre maximum de trains pouvant circuler sur cette composante dans un certain délai de temps [4]. Mesurer la capacité d'une composante d'un système ferroviaire revient à résoudre un problème d'optimisation appelé **problème de saturation**, qui peut être formulé comme un SPP. Le but de ce problème est d'introduire le maximum de circulation sur une infrastructure, en plus d'une éventuelle circulation initiale déjà présente [7]. La situation est très bien illustrée dans la FIGURE 2 (remarquez la collision entre un train initial et un train saturant en z_3).

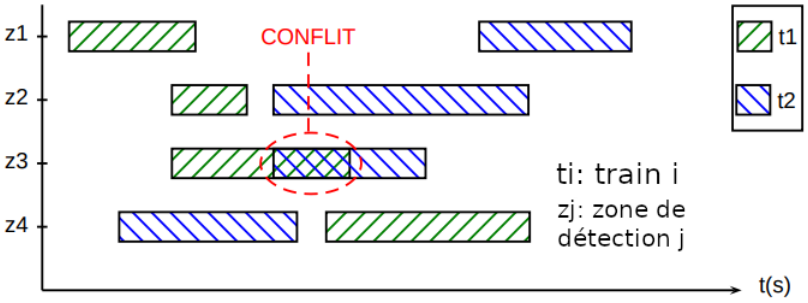

FIGURE 2 – Délai (en secondes) nécessaire pour que les trains t_i transitent par les zones z_j ($i \in \mathbb{N}, j \in \mathbb{N}$). Avec t_1 les trains initiaux et t_2 les trains saturants. — `https://www.emse.fr/~delorme/Papiers/These/These_slides.pdf` (Adaptation de l'illustration p.15)

Chapitre 2

Les instances numériques de SPP

Le Tableau 1 indique les noms des 15 instances de set packing que nous avons choisi pour mener notre expérimentation numérique et nos tests. Le nombre de variables, de contraintes et la meilleure valeur connue pour chaque instance sélectionnée sont spécifiés dans ce tableau. Ces données peuvent être retrouvées sur le site de l'École des Mines de Saint-Étienne [8] (sauf *didactic* qui est une instance didactique). Nous y trouvons d'ailleurs des données complémentaires sur ce jeu de données telles que la densité des données et le nombre maximum de colonnes à 1 dans une ligne de la matrice A. Veuillez notez aussi que lorsque la valeur de la meilleure solution connue **n'est pas optimale**, elle est suivie d'un astérisque. Les instances numériques sélectionnées présentent des caractéristiques variées (i.e. différentes densité, nombre maximum de colonnes à 1 dans une ligne de la matrice des contraintes, Max-Uns, différents) et ont été choisie pour ces mêmes raisons.

Instance	# Var.	# Cont.	Densité (%)	Max-Uns	Meilleure valeur connue
didactic	9	7	46	6	30*
100rnd0100	100	500	2	2	372*
200rnd0100	200	1000	1.49	4	416*
200rnd0300	200	1000	1	2	731*
200rnd0500	200	1000	2.48	8	184*
200rnd0900	200	200	1	2	1324*
200rnd1500	200	600	1	2	926*
500rnd0100	500	2500	1.23	10	323
500rnd0700	500	500	1.2	10	1141*
500rnd0900	500	500	0.7	5	2236*
500rnd1100	500	500	2.26	20	424*
1000rnd0100	1000	5000	2.6	50	67*
1000rnd0500	1000	1000	2.6	50	222
2000rnd0100	2000	10000	2.54	100	40*
2000rnd0500	2000	2000	2.55	100	140
Instance	**# Var.**	**# Cont.**	**Densité (%)**	**Max-Uns**	**Meilleure valeur connue**

Tableau 1 – Les instances de SPP sélectionnées pour nos tests

Chapitre 3

Présentation des heuristiques du DM1

Pour le DM1, nous avons cherché à mettre en place un solveur pour le Set Packing Problem (SPP). Pour ce faire, nous avons mis en place une **heuristique de construction d'une solution initiale réalisable**, que l'on notera x_0. Enfin, nous avons proposé une **heuristique de recherche locale** fondée sur trois voisinages pour chercher une meilleure solution. Nous noterons $\hat{x}$ notre meilleure solution ($\hat{x}$ éventuellement telle que $\hat{x} \neq x_0$) et $\hat{z}$ la valeur de la meilleure solution trouvée.

3.1 Heuristique de construction

Notre heuristique de construction consiste à choisir les variables qui ont le moins de contraintes et avec un grand poids. Pour ce faire, nous calculons d'abord les utilités u_i de chaque élément $i \in I$ telles que :

$$\forall j \in J, u_j = \frac{c_j}{\sum_{i \in I} a_{i,j}}$$

Nous ordonnons ensuite les utilités u_j par valeurs décroissantes. Enfin, nous sélectionnons dans l'ordre des utilités décroissantes toutes les variables pouvant augmenter le poids total tout en respectant les contraintes. La contrainte que nous devons respecter est $\sum_{i \in I} a_{i,j} x_i \leqslant 1, \forall j \in J$. Illustrant l'algorithme avec un exemple didactique. Prenons l'instance de SPP didactic :

$$\begin{bmatrix}
\text{Max z} = & 10x_1 & +5x_2 & +8x_3 & +6x_4 & +9x_5 & +13x_6 & +11x_7 & +4x_8 & +6x_9 & \\
s/c & x_1 & +x_2 & +x_3 & +x_5 & +x_7 & +x_8 & & & & \leqslant 1, \\
 & x_2 & +x_3 & +x_8 & & & & & & & \leqslant 1, \\
 & x_2 & +x_5 & +x_6 & +x_8 & +x_9 & & & & & \leqslant 1, \\
 & x_4 & & & & & & & & & \leqslant 1, \\
 & x_1 & +x_3 & +x_5 & +x_6 & +x_9 & & & & & \leqslant 1, \\
 & x_2 & +x_3 & +x_7 & +x_9 & & & & & & \leqslant 1, \\
 & x_1 & +x_4 & +x_5 & +x_8 & +x_9 & & & & & \leqslant 1, \\
 & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7, & x_8, & x_9 = (0,1) &
\end{bmatrix}$$

Simplifions les notations pour ne prendre en compte que les indices des variables, les poids des variables et les coefficients $a_{i,j}$ de la matrice A. Cette simplification permet de voir plus clairement comment sont calculés les utilités u_j mais aussi de mieux comprendre comment fonctionne l'heuristique gloutonne mise en place.

Nous avons donc :

$j =$	1	2	3	4	5	6	7	8	9
$c_j =$	10	5	8	6	9	13	11	4	6
	1	1	1	0	1	0	1	1	0
	0	1	1	0	0	0	0	1	0
	0	1	0	0	1	1	0	1	1
$a_{i,j} =$	0	0	0	1	0	0	0	0	0
	1	0	1	0	1	1	0	0	1
	0	1	1	0	0	0	1	0	1
	1	0	0	1	1	0	0	1	1
$\sum_{i \in I} a_{i,j} =$	3	4	4	2	4	2	2	4	4
$u_j =$	$3.\dot{3}$	1.25	2	3	2.25	6.5	5.5	1	1.5

Dans l'ordre décroissant des utilités, nous avons :

$u_j =$	6.5	5.5	$3.\dot{3}$	3	2.25	2	1.5	1.25	1
$j =$	6	7	1	4	5	3	9	2	8

Nous prenons la 6$^{\text{e}}$ colonne de la matrice A et nous essayons de la sommer avec la 7$^{\text{e}}$ colonne. La nouvelle colonne obtenue n'a pas de coefficient supérieur à 1 donc on valide l'opération. Nous retenons que la 6$^{\text{e}}$ et la 7$^{\text{e}}$ variable, x_6 et x_7, sont égales à 1 car l'inclusion de celles-ci est possible tout en respectant la contrainte.

6	7	6 + 7
0	1	1
0	0	0
1	0	1
0	0	0
1	0	1
0	1	1
0	0	0

Nous essayons alors de sommer la 1$^{\text{ère}}$ à la nouvelle colonne. Comme il y a des coefficients supérieurs à 1, nous ne validons pas l'opération. Ainsi, x_1 est égale à 0 car l'inclusion de la variable x_1 ne permet pas de respecter la contrainte quand x_6 et x_7 sont déjà incluses. Nous continuons ainsi de suite et nous obtenons la solution finale :

6	7	6 + 7	4	6 + 7 + 4
0	1	1	0	1
0	0	0	0	0
1	0	1	0	1
0	0	0	1	1
1	0	1	0	1
0	1	1	0	1
0	0	0	1	1

Ainsi, les variables retenues sont x_6, x_7 et x_4 donc $x_0 = [0,0,0,1,0,1,1,0,0]$ et $\hat{z} = z(x_0) = c_4 + c_6 + c_7 = 6 + 13 + 11 = 30$.

Cet algorithme s'arrête si on trouve $1^{|I|}$ (le vecteur colonne de dimension $|I|$ dont tous les coefficients sont à 1) ou si l'on a parcouru toutes les colonnes de A.

3.2 Heuristique d'amélioration

Notre heuristique d'amélioration consiste à trouver une meilleure solution à partir de trois voisinages d'une solution initiale admissible x_0 (que nous avons obtenu à partir de l'heuristique de construction). Ces voisinages sont basés sur des "k-p exchanges". Le voisinage par k-p échanges d'une solution x est l'ensemble des solutions obtenues à partir de x en remplaçant la valeur de k variables à 1 par 0 et en remplaçant la valeur de p variables à 0 par 1. Cependant, étant donné l'explosion combinatoire du nombre d'échanges possibles quand k et p augmentent, nous avons décidé d'utiliser le 2-1 échange, le 1-1 échange et le 0-1 échange. De plus, nous pouvons explorer ces voisinages en plus profonde descente (consiste à visiter tous les voisins d'une solution donnée en choisissant le meilleur voisin pour continuer l'exploration) ou en descente "rapide" (consiste à choisir le premier voisin améliorant pour continuer l'exploration). En d'autres termes, la descente profonde consiste à visiter tous le voisinage d'une solution x pour choisir le voisin pour lequel $\sum_{i \in I} c_i x_i$ est maximale, avec x_i les variables du problèmes et c_i les coûts associés à ces variables. D'un autre côté, la descente "rapide" consiste à choisir le premier voisin pour lequel on obtient une meilleure somme.

Voici le principe de notre heuristique en quelques étapes :

1. Soient $\mathcal{N}_{2-1}(x_0)$ et $\mathcal{N}_{1-1}(x_0)$ les voisinages par 2-1 et 1-1 échanges de la solution initiale x_0. Nous parcourons tous les voisins $n_{2-1} \in \mathcal{N}_{2-1}(x_0)$ et $n_{1-1} \in \mathcal{N}_{1-1}(x_0)$.
2. Si $z(n_{2-1}) \geqslant z(x_0)$ (respectivement $z(n_{1-1}) \geqslant z(x_0)$) alors n_{2-1} (respectivement n_{1-1}) est un voisin susceptible d'augmenter $\sum_{i \in I} c_i x_i$.
3. Or, il faut encore vérifier la contrainte $\sum_{i \in I} a_{i,j} x_i \leqslant 1, \forall j \in J$. Ainsi, il faut vérifier si n_{2-1} (respectivement n_{1-1}) est une solution admissible au problème. Nous définissons donc une fonction $feasible$ telle que

$$feasible(x) = \begin{cases} 1 & \text{si } x \text{ est admissible} \\ 0 & \text{sinon} \end{cases}$$

4. Si $z(n_{2-1}) \geqslant z(x_0)$ et $feasible(n_{2-1}) = 1$ alors $\hat{z} = z(n_{2-1})$ et n_{2-1} est une solution admissible au problème (même chose dans le cas de n_{1-1}).
5. Nous posons $x_0 = n_{2-1}$ (respectivement $x_0 = n_{1-1}$) et nous reprenons à l'étape 1. L'algorithme s'arrête si aucun voisin admissible améliorant $\hat{z}$ (avec $\hat{z}$ la somme $\sum_{i \in I} c_i x_i$ maximale) n'est trouvé.

Le nombre d'échanges de x_0 (pour $x_0 = [0,0,0,1,0,1,1,0,0]$) étant important, nous retenons que les étapes ci-dessus ont été appliquées à cette solution et qu'aucun voisin de x_0 n'améliore $\hat{z}$. Il s'avère que x_0 est la solution optimale de l'instance didactic figurant dans le tableau 1. Nous avons choisie d'omettre l'inclusion de $\mathcal{N}_{0-1}(x_0)$ dans la présentation de notre heuristique ci-dessus dans le but de simplifier la lisibilité et la compréhension de cette section.

Chapitre 4

Présentation des heuristiques du DM2

Les heuristiques du DM1 sont simples à mettre en place mais elles ont un défaut. En effet, pour une instance donnée, l'heuristique de construction donnera toujours la même solution initiale x_0 tandis que l'heuristique d'amélioration considère les voisins améliorants mais n'explore jamais les voisinages localement moins intéressant. En d'autres mots, l'heuristique résultante de la combinaison des heuristiques de construction et d'amélioration est susceptible de rester bloquée sur un optimum local. La méthode GRASP (pour "Greedy Randomized Adaptive Search Procedure") proposée par Feo et Resende [9] est une métaheuristique de type "multi-start descent" en deux phases.

4.1 L'heuristique GRASP

GRASP est une métaheuristique en deux phases. La première phase est une phase de construction qui génère une solution initiale (départ) en utilisant une procédure gloutonne randomisée. Cet aspect aléatoire permet d'obtenir des solutions de différentes régions de l'espace des solutions. La seconde phase est une phase de recherche locale (descente) qui améliore la solutions initiale. Ce processus en deux phases peut être réitéré (la meilleure solution est conservée à chaque itération) et l'avantage c'est que chaque itération est indépendante donc l'algorithme est parallélisable assez facilement. La méthode GRASP est décrite dans l'Algorithme 1.

Algorithme 1 L'algorithme GRASP

$Solutions \leftarrow \emptyset$
Répéter
  $initialSol \leftarrow$ `greedyRandomized(`$problem$, α`)`
  $improvedSol \leftarrow$ `localSearch(`$initialSol$`)`
  $Solutions \leftarrow Solutions \cup \{improvedSol\}$
Jusqu'à condition d'arrêt
$finalSol \leftarrow$ `best(`$Solutions$`)`

Dans notre cas la phase de construction est l'algorithme décrit dans l'Algorithme 2 tandis que la phase de recherche locale est identique à celle du DM1 (section 3.2). Nous calculons les utilités des variables puis nous construisons une liste de candidats restreints (Restricted Candidate List ou RCL) qui est constituée des variables prioritaires (par ordre décroissant des utilités) et dont la taille est définie en fonction d'un paramètre $\alpha \in [0, 1]$.

Lorsque le paramètre $\alpha = 0$, l'algorithme correspond à une construction aléatoire tandis qu'avec $\alpha = 1$, l'algorithme équivaut à un algorithme glouton. En effet, avec $\alpha = 0$ la RCL est constituée de tous les candidats et nous choisissons aléatoirement un candidat dans la RCL (donc un candidat au hasard parmis tous les candidats) tandis qu'avec $\alpha = 1$ la RCL n'est constituée que des éléments dont l'utilité est la plus grande (reprend le comportement de l'heuristique de construction du DM1 présentée section 3.1).

Algorithme 2 L'algorithme de construction greedyRandomized

$I_t \leftarrow I$
$x_i \leftarrow 0, \forall i \in I_t$
$Eval_i \leftarrow c_i / \sum_{j \in J} t_{i,j}, \forall i \in I_t$
Tant que $I_t \neq \emptyset$, **faire** :
  $Limit \leftarrow min_{i \in I_t}(Eval_i) + \alpha * (max_{i \in I_t}(Eval_i) - min_{i \in I_t}(Eval_i))$
  $RCL \leftarrow \{i \in I_t, Eval_i \geqslant Limit\}$
  $i^* \leftarrow$ `RandomSelect(`RCL`)`
  $x_{i^*} \leftarrow 1$
  $I_t \setminus \{i^*\}$
  $I_t \setminus \{i : \exists j \in J, t_{i,j} + ti^*, j > 1\}$
Fin tant que

Illustrons l'algorithme avec l'instance didactic vu section 3 :

$j =$	1	2	3	4	5	6	7	8	9
$c_j =$	10	5	8	6	9	13	11	4	6
	1	1	1	0	1	0	1	1	0
	0	1	1	0	0	0	0	1	0
	0	1	0	0	1	1	0	1	1
$a_{i,j} =$	0	0	0	1	0	0	0	0	0
	1	0	1	0	1	1	0	0	1
	0	1	1	0	0	0	1	0	1
	1	0	0	1	1	0	0	1	1
$\sum_{i \in I} a_{i,j} =$	3	4	4	2	4	2	2	4	4
$u_j =$	$3.\dot{3}$	1.25	2	3	2.25	6.5	5.5	1	1.5

Dans l'ordre décroissant des utilités, nous avons :

$u_j =$	6.5	5.5	$3.\dot{3}$	3	2.25	2	1.5	1.25	1
$j =$	6	7	1	4	5	3	9	2	8

Calculons la limite ainsi que la RCL :
$Limit = 1 + 0.7 * (6.5 - 1) = 4.85$
$RCL = \{x_6, x_7\}$

Nous choisissons x_7 aléatoirement (nous aurions très bien pu choisir x_6), nous obtenons :
$x_{Init} = [0, 0, 0, 0, 0, 0, 1, 0, 0]$

Maintenant, il faut faire en sorte que x_7 ne soit plus éligible aléatoirement (nous l'enlevons de la liste des candidats) et nous recalculons la limite :
$Limit = 1 + 0.7 * (6.5 - 1) = 4.85$
$RCL = \{x_6\}$

Nous choisissons x_6 aléatoirement (c'est le seul élément de la liste) : $x_{Init} = [0, 0, 0, 0, 0, 1, 1, 0, 0]$

Nous réitérons (x_6 n'est plus éligible) :
$Limit = 1 + 0.7 * (3.3 - 1) = 2.61$
$RCL = \{x_1, x_4\}$

Il est possible de choisir x_1 ou x_4 et nous poursuivons ainsi de suite jusqu'à ce qu'aucune variable ne soit éligible.

4.2 L'heuristique Reactive GRASP

Des expérimentations préliminaires ne nous ont pas permis d'identifier une valeur particulière du paramètre α pouvant être considérée comme une recommandation pour toutes les instances (ou au moins une grande partie). Prais et Ribeiro [10] ont proposé un composant appelé Reactive GRASP qui s'apparente à une procédure d'apprentissage statistique simple, fondée sur la qualité des solutions obtenues à chaque itération GRASP. Ce composant permet de régler automatiquement GRASP (valeur de α notamment). Nous avons repris l'algorithme présenté dans la thèse de Delorme [2] pour implémenter Reactive GRASP (Algorithme 3).

Algorithme 3 L'algorithme Reactive GRASP

$Solutions \leftarrow \emptyset$
$proba_\alpha \leftarrow 1/|alphaSet|,\ \forall\, \alpha \in alphaSet$
Répéter
  $\alpha^* \leftarrow$ `RandomSelect(`$alphaSet$, $proba$`)`
  $initialSol \leftarrow$ `greedyRandomized(`$proba$, α^*`)`
  $improvedSol \leftarrow$ `localSearch(`$initialSol$`)`
  $Solutions \leftarrow Solutions \cup \{improvedSol\}$
  $Pool_{\alpha^*} \leftarrow Pool_{\alpha^*} \cup \{improvedSol\}$
  Si condition $probaUpdate$ est vrai **alors**
    $valuation_\alpha \leftarrow \left(\frac{mean_{s \in Pool_\alpha}(z(s)) - z(worst(Solutions))}{z(best(Solutions)) - z(worst(Solutions))}\right)^\delta,\ \forall\, \alpha \in alphaSet$
    $proba_\alpha \leftarrow valuation_\alpha / \left(\sum_{\alpha' \in alphaSet} valuation_{\alpha'}\right),\ \forall\, \alpha \in alphaSet$
  Fin si
Jusqu'à condition d'arrêt
$finalSol \leftarrow best(Solutions)$

Notez que considérer toutes les valeurs continues entre 0 et 1 pour paramétrer α est invraisemblable. Nous considérerons plutôt un ensemble de valeurs discrètes pré-défini $alphaSet$. Soit $proba_\alpha$ un vecteur de probabilités assignées aux valeurs discrètes. En partant d'une distribution de probabilités uniforme pour les valeurs de $alphaSet$ (sélection équiprobable), $proba_\alpha$ est mis-à-jour selon une périodicité dépendant de la condition $probaUpdate$.

Dans notre cas, *probaUpdate* est un nombre d'itérations, dans la littérature il est souvent noté N_α (ici on utilisera les deux notations mais elles sont équivalentes). Toutes les N_α itération, nous recalculons les probabilités à partir de la valeur moyenne des meilleures solutions obtenues pour chaque valeur du paramètre α (ces meilleures solutions sont stockées dans l'ensemble $Pool_\alpha$ de taille N_α). Le paramètre δ permet d'atténuer les écarts entre les probabilités des différentes valeurs.

4.3 Phase d'intensification

Notez qu'après la phase de recherche locale il peut être intéressant de lancer une phase d'intensification des solutions améliorées par la recherche locale. Cette troisième phase peut permettre d'améliorer les solutions obtenus à la sortie de la phase de recherche locale. Delorme, Gandibleux et Rodriguez [2] ont proposé d'utiliser la reconstruction de chemin ("path relinking") présentée à l'origine pour la recherche tabou par Glover et Laguna [11]. Le path relinking confère au GRASP un mécanisme de collaboration et de mémoire entre les itérations. Malheureusement, l'inclusion du path relinking rend la parallélisation des algorithmes plus compliquée. Nous n'avons pas présenté l'algorithme GRASP avec cette phase d'intensification parce que nous ne l'avons pas implémentée dans le cadre de notre devoir.

Chapitre 5

Présentation des heuristiques du DM3

Reactive GRASP est une métaheuristique très agressive, rapide et efficace sur des instances de SPP. L'objectif du DM3 est de mettre en place une métaheuristique aussi compétitive avec GRASP que possible. Nous avons choisi d'implémenter l'algorithme de colonies de fourmis ("Ant Colony Optimization" ou ACO) car dans la littérature du SPP c'est l'une des heuristiques fournissant de bons résultats sur un ensemble varié d'instances. Notre algorithme se base en grande partie sur ceux proposés par Gandibleux *et al.* [4, 12, 13].

5.1 L'heuristique ACO

Comme son nom l'indique, l'algorithme de colonies de fourmis s'inspire du comportement collectif des fourmis. Une seule fourmi peut avoir du mal à trouver une source de nourriture pour toute une colonie mais si plusieurs fourmis travaillent ensemble elles augmentent leur chance de trouver des ressources. Le principe est simple :

- Soit un ensemble F de fourmis notées f_i pour $i \in \{1, \ldots, n\}$. Chaque fourmi f_i peut emprunter un chemin quelconque et à chaque chemin emprunté elles déposent une certaine quantité initiale ϕ_i de phéromones sur leur passage (**initialisation**)
- Lorsqu'une fourmi f_k trouve une ressource, phi_k augmente et les autres fourmis sont attirés vers le chemin que f_k a emprunté (**dépôt de phéromone**)
- Les phéromones phi_i déposés sur les chemins qui sont peu ou plus du tout empruntés par les fourmis s'évaporent petit à petit dans l'air (**phénomène d'évaporation**)
- Il existe des fourmis qui vont continuer de chercher des sources de nourritures même s'il existe déjà une piste de phéromone forte (**fourmi "curieuse"**) contrairement à celles qui poursuivent une piste tant qu'elle est bonne (**fourmi "fonceuse"**)

L'analogie avec les fourmis s'explique assez naturellement. Soit un problème d'optimisation, les fourmis cherchent à construire de bonnes solutions (bonne source de nourriture). À chaque itération de la recherche, chaque fourmi construit une solution en appliquant une procédure constructive qui utilise la mémoire commune de la colonie (les phéromones). Cette mémoire est appelée vecteur de phéromones, noté ϕ et elle contient des probabilités sur les variables du problème. Ainsi, la coopération entre les fourmis est réalisée en exploitant la matrice des phéromones. Par ailleurs, lors de la construction d'une solution, une fourmi est soit fonceuse (mode exploitation) soit curieuse (mode exploration).

La seule différence est qu'une fourmi curieuse ne va pas se contenter de construire une solution en prennant les meilleurs valeurs ϕ_i. Elle va aussi sélectionner, de temps en temps selon une probabilité $\mathcal{P}$, des variables i qu'elle va mettre dans sa solution. Dans le cadre d'une ACO de base, le choix du mode est fait selon une probabilité connue et fixe. Il existe dans la littérature une version plus élaborée du ACO, nommée SACO, qui utilise un principe tiré du recuit simulé : elle autorise une plus grande diversification au début de la recherche et intensifie vers la fin de la recherche. Par conséquent, dans le cadre du SACO, la probabilité $\mathcal{P}$ évolue tout au long du processus. En outre, après avoir construit une solution, une fourmi applique sur celle-ci une recherche locale. Si l'heuristique stagne ou qu'une piste a vu son niveau de phéromone baissé au-deça d'un seuil, une perturbation du terrain (les pistes de phéromones) est appliquée. La méthode ACO est décrite dans l'Algorithme 4.

Algorithme 4 L'algorithme ACO

```
        ▷ Genérer une solution initiale en utilisant un algorithme glouton et une recherche locale
sol ← elaborateSolGreedy()
sol ← localSearch(sol)
bestSolKnown ← copySol(sol)
                                                                          ▷ Algorithme ACO
phi ← initPheromones() ; iter ← 0 ; P ← 0
Tant que pas isFinished(iter, maxRestart), faire :
    bestSolIter ← emptySol()
    Pour ant dans 1...maxAnt, faire :
        Si isExploitation?(ant, P) alors
            sol ← elaborateSolutionGreedyPhi(phi)
        Sinon
            sol ← elaborateSolutionSelectionMethod(phi)
        Fin si
        sol ← localSearch(sol)
        Si z(sol) > z(bestSolIter) alors
            bestSolIter ← copySolution(sol)
            Si z(sol) > z(bestSolKnown) alors
                bestSolKnown ← copySolution(sol)
            Fin si
        Fin si
    Fin pour
    φ ← managePheromones(phi, bestSolKnown, bestSolIter)
    iter ← iter + 1
Fin tant que
return(bestSolKnown)                                    ▷ Meilleure solution trouvée
```

Soit ϕ le vecteur des phéromones et ϕ_i la probabilité d'avoir la variable i dans un bon packing. Les phéromones sont initialisées (procédure `initPheromones`) telles que $\phi_i \leftarrow$ `phiInit` pour toutes les variables $i \in I$ avec `phiInit` un certain paramètre. Chaque fourmi élabore une solution saturée réalisable en partant de la solution triviale réalisable, $x_i = 0, \forall i \in I$. Tant que la solution est admissible, certaines variables sont mises à 1. Les changements ne concernent qu'une seule variable à chaque étape et il n'y a plus de changement quand aucune variable ne peut être fixée à 1 sans perdre l'admissibilité de la solution.

La méthode utilisée pour choisir une variable x_i à fixer à 1 se fait selon le mode courant (exploitation ou exploration) avec les procédures présentées dans l'Algorithme 4 (`elaborateSolutionGreedyPhi` et `elaborateSolutionSelectionMethod` respectivement). Dans le mode exploration, selon la probabilité $\mathcal{P}$ soit une roulette est appliquée pour choisir la variable candidate soit on sélectionne la variable candidate ayant la plus grande valeur de phéromone (Algorithme 5). Dans le mode exploitation, on reprend les mêmes étapes qu'une construction gloutonne mais en ordonnant les variables dans l'ordre décroissant des ϕ_i plutôt que dans le sens décroissant des utilités des variables.

Algorithme 5 La procédure `elaborateSolutionSelectionMode`

$I_t \leftarrow I$; $x_i \leftarrow 0$, $\forall i \in I_t$
$\mathcal{P} \leftarrow \log_{10}(\texttt{iter}) / \log_{10}(\texttt{maxIter})$
Tant que $I_t \neq \emptyset$, **faire** :
  Si `randomValue(0,1)` $> \mathcal{P}$ **alors**
    $i^* \leftarrow$ `rouletteWheel(`ϕ_i, $i \in I_t$`)`
  Sinon
    $i^* \leftarrow$ `bestValue(`ϕ_i, $i \in I_t$`)`
  Fin si
  $x_{i^*} \leftarrow 1$; $I_t \leftarrow I_t \setminus \{i^*\}$; $I_t \leftarrow I_t \setminus \{i : \exists j \in J,\, t_{i,j} + t_{i^*,j} > 1\}$
Fin tant que

$\mathcal{P}$ est modifiée de sorte à suivre une courbe logarithme lorsqu'elle évolue ce qui permet aux fourmis de fortement diversifier leur recherche d'une bonne solution. Après que toutes les fourmis aient construit une solution, la meilleure pour l'itération en cours est retenue. Ensuite, l'évaporation et le dépôt de phéromones sont effectués. En d'autres termes, nous augmentons les phéromones ϕ_i correspondant aux éléments sélectionnés dans la solution de l'itération actuelle, tandis que nous diminuons les autres phéromones. Ce processus est décrit dans la procédure `managePheromones` (Algorithme 6). En outre, une stratégie de perturbation a été intégrée à cette procédure de gestion et ce déclenche lorsque deux conditions sont vraies :

1. la convergence de l'ACO est en stagnation (le prédicat `isStagnant?` est vrai)
2. au moins une phéromone a son niveau fixé à zéro (le prédicat `isExistsPhiNul?` est vrai)

L'algorithme s'arrête quand le nombre d'itérations maximum est atteint ou après un certain nombre de perturbations (selon un paramètre donné `maxRestart`). Les paramètres utilisés dans l'heuristique ACO sont indiqués dans le tableau 2.

`maxIter`	200	Nombre d'itérations maximum
`maxAnt`	15	Nombre de fourmis pour chaque itération
`phiInit`	1.0	Valeur initiale des phéromones
`rhoE`	0.8	Taux d'évaporation des phéromones
`rhoD`	`phiInit` * (1.0 - `rhoE`)	Taux de dépôt des phéromones
`phiNul`	0.001	Seuil de phéromone considéré comme nulle
`iterStagnant`	8	Déclare la méthode stagnante quand aucune amélioration n'est observée
`maxRestart`	2	Nombre maximum de perturbations avant l'arrêt de la méthode

Tableau 2 – Paramètres de la méthode ACO

Algorithme 6 Procédure `managePheromones`

Pour i **dans** $1 \dots ncol$, **faire** : ▷ Évaporation des phéromones
  $\phi_i \leftarrow \phi_i * rhoE$
Fin pour
Pour i **dans** $1 \dots ncol$, **faire** : ▷ Dépôt des phéromones
  Si $bestSolIter[i] = 1$ **alors**
    $\phi_i \leftarrow \phi_i + rhoD$
  Fin si
Fin pour
Si `isStagnant?(`$bestSolution$`,` $iterStagnant$`)`
  et `isExistsPhiNul?(`ϕ`)` **alors** ▷ Perturbation du terrain
  Pour i **dans** $1 \dots ncol$, **faire** : ▷ Perturber les phéromones
    $\phi_i \leftarrow \phi_i * 0.95 * \log_{10}(iter)/\log_{10}(maxIter)$
  Fin pour
  Pour i **dans** $random(0.0, 0.1 * ncol)$, **faire** :
    $\phi_{random(1,ncol)} \leftarrow random(0.05, (1.0 - iter/maxIter) * 0.5)$
  Fin pour
  Pour i **dans** $1 \dots ncol$, **faire** : ▷ Augmentation phéromones de faible valeurs
    $\phi_i \leftarrow \phi_i + random(0.05, (1.0 - iter/maxIter) * 0.5)$
  Fin pour
Fin si

Chapitre 6

Expérimentation numérique

6.1 Conditions de l'expérimentation

Nous avons mené une expérimentation numérique dans le but de démontrer l'utilité des heuristiques face à des solveurs de programmes linéaires comme GLPL. L'objectif pour nous est de montrer que l'on peut utiliser des heuristiques (simples) pour approcher une solution de bonne facture en un délai de temps souvent très inférieur aux délais de solveurs tels que GLPK. L'ensemble des tests que nous présenterons ont été réalisé sur une machine équipée d'un processeur Intel® Core™ i7-9750H de 9ème génération (6 coeurs, 12 threads, 2.6 GHz de base / 4.50 GHz maximum). Chaque solveur a été implémenté en C++20 et compiler avec gcc 11.3.0 avec -O3. Ils ont été lancés 10 fois sur les instances présentées au Chapitre 2 et peuvent utiliser jusqu'à 10 threads (si parallélisation), les critères pour la comparaison entre les différents solveurs sont :

- le temps moyen nécessaire CPUt (en secondes) pour trouver la solution donnée
- $\hat{z}_{min}$, $\hat{z}_{moy}$ et $\hat{z}_{max}$ le minimum, la moyenne et le maximum des valeurs des meilleures solutions trouvées avec le Reactive GRASP et ACO. Pour GLPK, la valeur de la meilleure solution obtenu $\hat{z}$ sera donnée.

6.1.1 Affichage des résultats de Reactive GRASP

Pour étudier les performances de notre solveur nous avons opté pour un affichage graphique des données importantes. Pour chaque instance l'affichage se découpe en quatre parties.

Résultats d'un run sur une instance

Pour chaque itération de l'algorithme Reactive GRASP les valeurs des solutions initiales fournit par l'heuristique de construction sont affichées par un point rouge, puis celles des solutions améliorées sont affichées par un triangle vert. Les deux sont reliées par un trait. Enfin la meilleur valeur connue à une itération donnée est représenter par le tracé vert. La Figure 3 illustre l'affichage obtenu.

Analyse des résultats pour l'ensemble des exécutions

Lorsque (Reactive) GRASP est lancé plusieurs fois, les valeurs maximales, minimales et moyennes des meilleurs solutions obtenus sur l'ensemble des exécutions sont retenues et affichés. La Figure 4 illustre l'affichage obtenu.

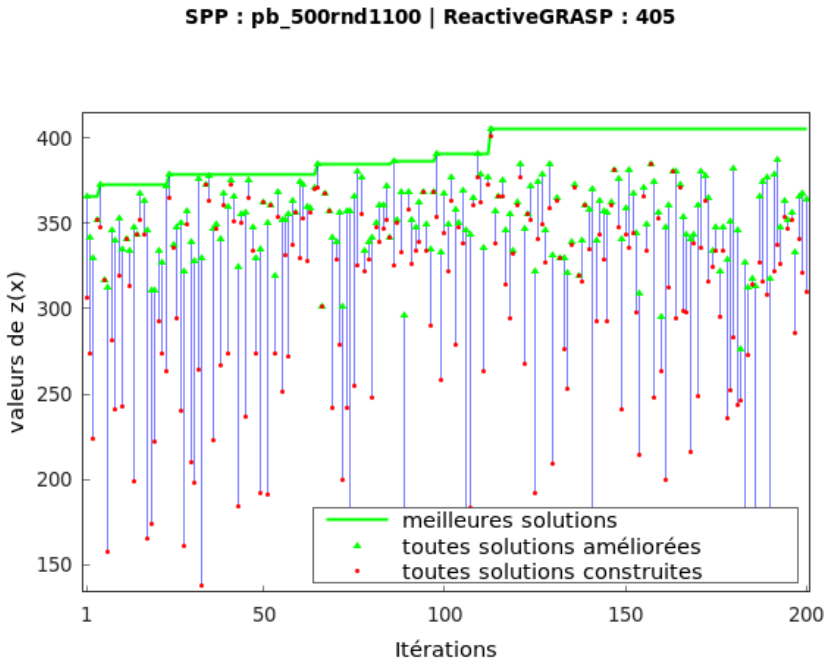

FIGURE 3 – Exemple d'une exécution de Reactive GRASP

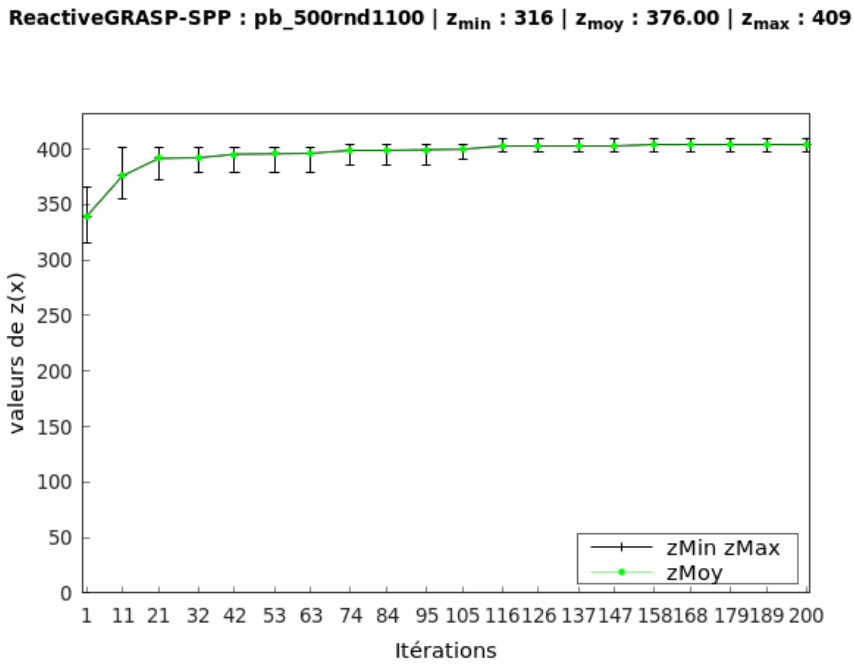

FIGURE 4 – Exemple d'expérimentation numérique de Reactive GRASP

Probabilités des valeurs saillantes de α pour Reactive GRASP

Les probabilités $proba_\alpha$ obtenus à la fin d'une exécution de Reactive GRASP sont affichés dans des plots comme sur la FIGURE 5.

Temps CPUt moyen de résolution d'une exécution de (Reactive) GRASP

Les temps CPUt de résolution moyen par exécution et pour chaque instance sont affichés comme sur la FIGURE 6.

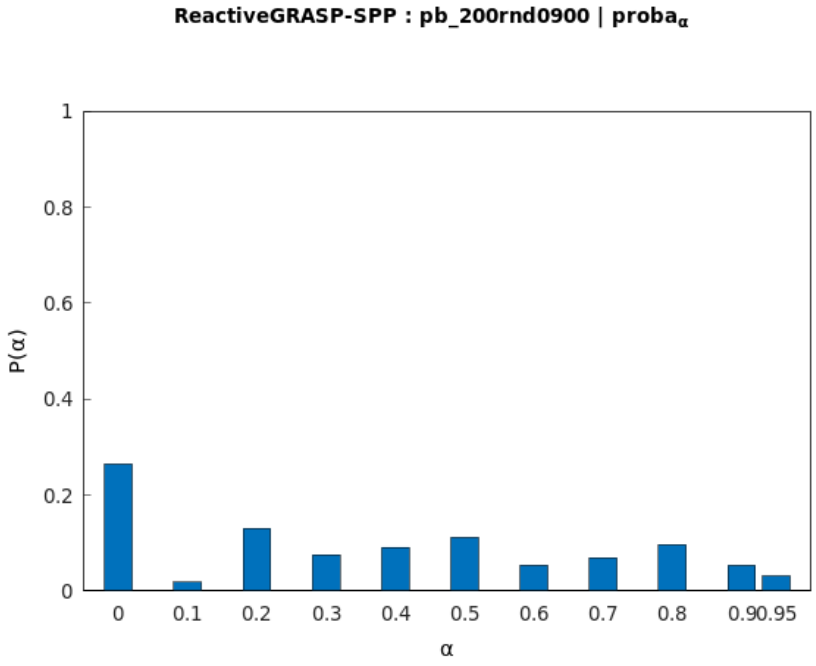

FIGURE 5 – Exemple de probabilités des valeurs de α obtenus à la fin d'une exécution Reactive GRASP

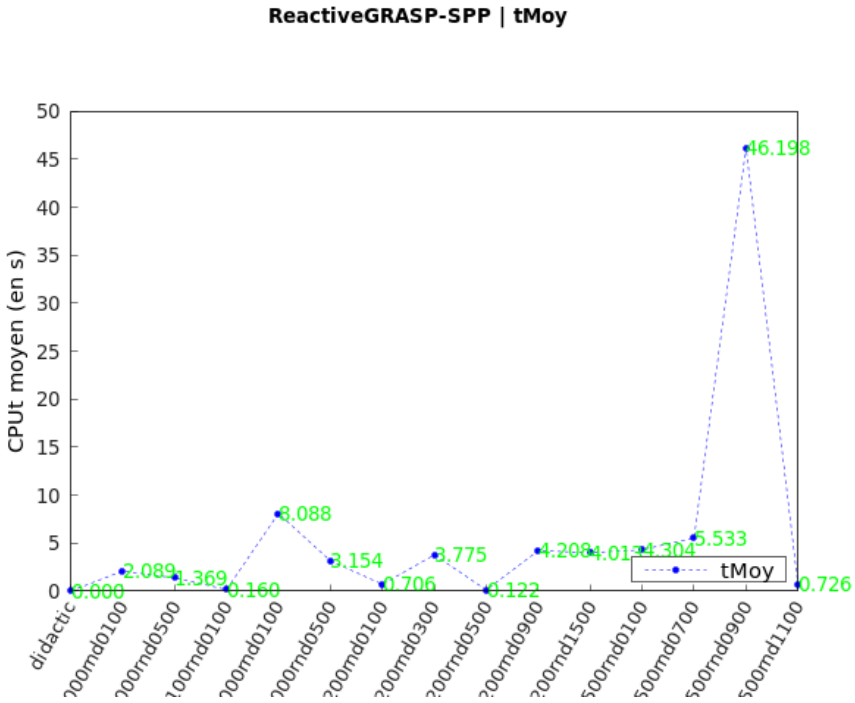

FIGURE 6 – Exemple d'affichage des temps CPUt moyen

6.1.2 Affichage des résultats de ACO

L'affichage des résultats du ACO sont très similaires à ceux de Reactive GRASP. Seuls les affichages suivants différent.

6.1.3 Niveaux de phéromones avant et après perturbation de ϕ

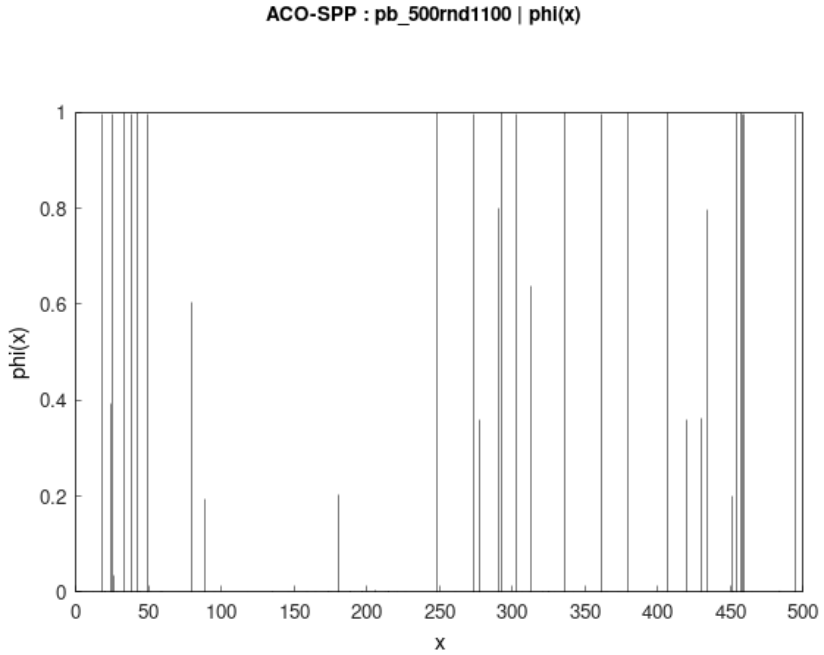

FIGURE 7 – État du vecteur de phéromone avant application de la procédure de perturbation

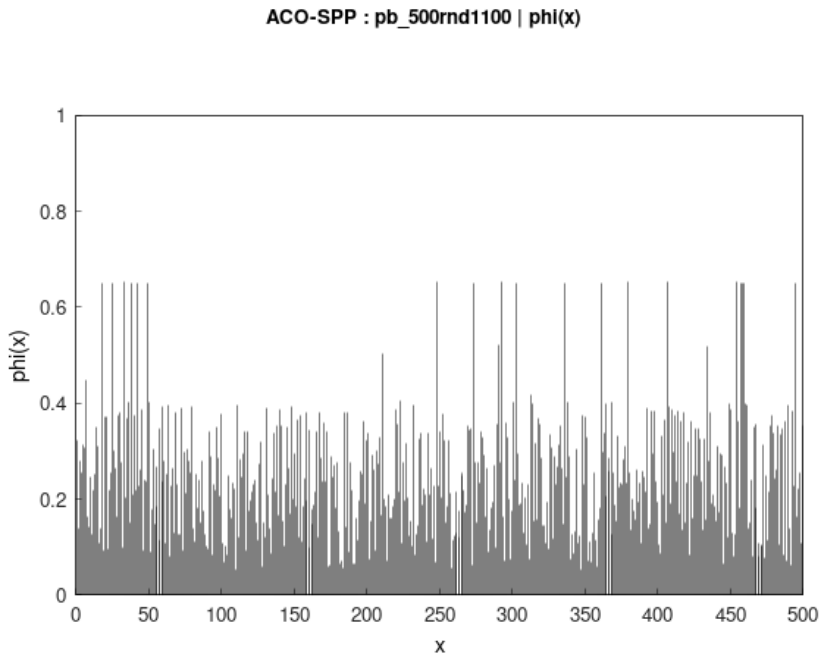

FIGURE 8 – État du vecteur de phéromone après application de la procédure de perturbation

Les figures 7 et 8 illustrent le mécanisme de perturbation de ACO.

Résultats d'un run sur une instance

Pour chaque itération de l'algorithme ACO les valeurs de toutes les solutions sont représentées par des lignes horizontales en pointillé allant de la valeur initiale de la solution à la valeur en sortie de la recherche locale. La meilleur valeur connue à une étape donnée est représentée par le tracé rouge. Enfin, la ligne bleue représente la valeur de $\mathcal{P}$ à chaque étape. La FIGURE 9 illustre l'affichage obtenu.

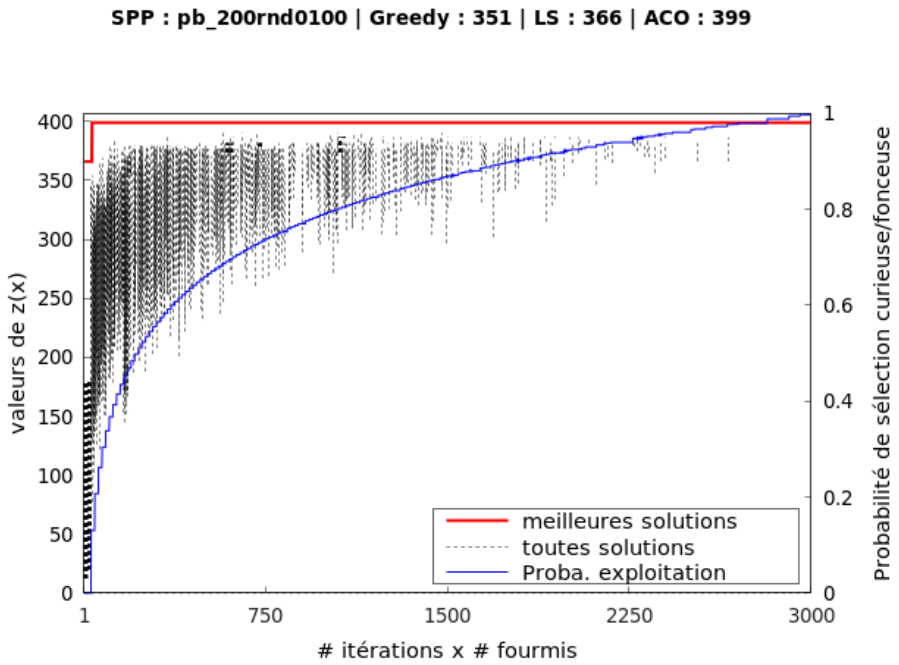

FIGURE 9 – Exemple d'une exécution de ACO

6.2 Paramétrage de (Reactive) GRASP

Nous avons réalisé une étude sur le réglage des paramètres de nos solveurs pour pouvoir donner une recommandation sur le réglage de ces-derniers. Typiquement, pour le GRASP nous avons mené une étude sur l'influence du réglage de α. Pour le Reactive GRASP, nous avons mené une étude sur la fréquence de mise-à-jour des probabilités des valeurs de α. Cette fréquence est paramétrée grâce à $probaUpdate$ (noté aussi N_α), le nombre d'itérations avant chaque mise-à-jour. Nous avons aussi étudier les performances du (Reactive) GRASP avec et sans parallélisation. Arbitrairement, nous présentons les résultats sur 200 itérations de (Reactive) GRASP. Pour Reactive GRASP, $alphaSet = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95\}$ et $\delta = 4$.

6.2.1 Réglage de α pour GRASP

Le réglage du paramètre α pour GRASP est effectué manuellement avant l'expérimentation (le but de Reactive GRASP est de remédier à faire ce choix).

En effet, les limites de GRASP reposent dans le fait qu'une seule valeur α n'est pas idéale pour l'ensemble des instances et ne va pas forcément fournir des valeurs de bonnes factures dans des instances spécifiques. Il a donc été nécessaire de faire plusieurs tests avec GRASP afin de déterminer si nous pouvions trouver une valeur α idéale dans notre cas. Les valeurs α proche de 0 donnent des solutions construites variées dont l'amélioration est assez bonne pour la plupart des instances (elles permettent d'explorer le plus de voisinages lors de la recherche locale). Pour les valeurs α plus proche de 1, nous remarquons des différences de résultats entre les instances. Les résultats empiriques confirment qu'aucune valeur de α ne semble idéale pour l'ensemble des instances.

6.2.2 Réglage de N_α pour Reactive GRASP

Nous avons remarqué empiriquement qu'avec $N_\alpha = 20$ les probabilités des valeurs d e α pour lesquelles ont a de moins bonnes solutions sont très proche de 0 ou nulles. Étant donné que les probabilités sont mise-à-jour plus souvent avec $N_\alpha = 20$ les probabilités des valeurs de α les moins prometteuses se rapprocheront rapidement de 0. Par la même occasion, la variabilité des solutions pour $N_\alpha = 20$ est inférieure à celle obtenu avec $N_\alpha = 50$ pour la plupart des instances. Le problème c'est qu'en perdant cette variabilité, l'algorithme peut éventuellement favoriser les valeurs α les plus prometteuses au début mais pour lesquels on peut très rapidement rester bloquer sur un optimum local car les solutions construites seront toutes construites à partir des mêmes valeurs α (les plus prometteuses). Ainsi, en choisissant une valeur N_α proche de 1 on risquera de perdre beaucoup en variabilité car les probabilités des valeurs α les moins prometteuses seront rapidement nulles. D'un autre côté, choisir une valeur N_α proche du nombre d'itérations à effectuer implique que les solutions construites seront très variées mais les probabilités des valeurs α resteront plus ou moins équiprobables et les valeurs de α les plus prometteuses ne seront pas favorisées (perte de qualité des solutions). Ainsi, en comparant les résultats obtenus avec $N_\alpha = 50$ à ceux obtenus avec $N_\alpha = 20$ et $N_\alpha = 10$, nous recommandons $N_\alpha = 50$. De plus, plus N_α est grand, moins Reactive GRASP fait de mise-à-jour des probabilités donc les temps CPUt sont meilleurs.

6.2.3 Réglage de la parallélisation

Nous avons parallélisé Reactive GRASP en utilisant la libraire OpenMP pour paralléliser la boucle for s'occupant des itérations. Ainsi, chaque itération de Reactive GRASP est lancé dans un thread indépendant. Le comportement de Reactive GRASP ne change pas et il est le même pour une valeur N_α donnée que les itérations soit parallélisées ou non. Le même raisonnement peut être fait lorsque Reactive GRASP est lancé en descente rapide : l'algorithme est jusqu'à 6 fois plus lent sans parallélisation. On remarque aussi que l'algorithme est plus rapide en descente rapide qu'en descente profonde avec des résultats très similaires.

6.3 Paramétrage de ACO

Nous avons parallélisé ACO en utilisant la librairie OpenMP avec une séparation du travail de chaque fourmis sur un thread. Lors du réglage des paramètres nous avons observé qu'une modification des paramètres `rhoE`, `rhoD` et `philInit` n'avait que peu d'effet. En revanche, changer `maxRestart` permet d'arrêter l'algorithme avant les 200 itérations maximum permettant ainsi au ACO d'être compétitive en temps avec Reactive GRASP.

6.4 Résultats de l'expérimentation

Au vu des résultats discutés section 6.2, nous avons décidé d'utiliser Reactive GRASP avec les paramètres suivants :

- $\delta = 4$
- $N_\alpha = 50$
- $alphaSet = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95\}$
- **recherche locale de type descente non profonde** ("rapide")
- **nombre de threads pour la parallélisation** : 10
- **condition d'arrêt** : 200 itérations

Pour ACO, nous avons utiliser les valeurs données dans le Tableau 2 pour les paramètres. Nous utilisons la même heuristique de recherche locale présentée section 3.2 pour ACO mais l'ensemble des tests, que nous présenterons ci-après, ont été effectués en mode descente normale. Nous présentons ici deux version d'ACO :

1. la première que l'on notera ACOv1 correspond au ACO avec le paramètre `maxRestart`$= +\infty$ (c'est-à-dire que l'on effectue systématiquement 200 itérations dans cette version et autant de perturbations qu'il le faudra)
2. la deuxième que l'on notera ACOv2 correspond au ACO avec le paramètre `maxRestart`$= 2$ (on effectue au plus 2 perturbations)

Chaque solveur est lancé 10 fois et les résultats sont rapportés dans le Tableau 3. Les cellules vertes indiquent les meilleures valeurs de la fonction objectif obtenue parmis tous les runs. Les cellules bleues indiquent les valeurs moyennes de la fonction objectif sur tous les runs. Les cellules rouges indiquent les temps CPUt des algorithmes. Toutes cellules foncées indiquent que la valeur du critère donné pour l'algorithme considéré est meilleure (ou égale) que pour les autres.

Nous constatons que ACOv1 n'est pas compétitive avec ReactiveGRASP en terme de temps de calcul (i.e. ACOv1 est 10 fois plus lent que ReactiveGRASP pour l'instance 2000rnd0100 et obtient les mêmes valeurs z_{max} et z_{moy}). De plus, ACOv1 n'obtient de meilleurs valeurs z_{max} que pour 3 instances par rapport à ReactiveGRASP, pour toutes les autres instances ACOv1 trouve les mêmes valeurs que ReactiveGRASP et en un temps nettement plus supérieur en général. ReactiveGRASP trouve un meilleur z_{max} pour l'instance 500rnd0700. ACOv2 est beaucoup plus rapide que ACOv1 et généralement plus rapide que ReactiveGRASP (qui est plus rapide que ACOv2 que sur quatre instances). De plus, ACOv2 donne de meilleurs solutions que ReactiveGRASP en moyenne (voir z_{moy}) bien que le ReactiveGRASP obtienne des valeurs maximums (z_{max}) plus grande. D'ailleurs, ACOv1 est la méthode qui donne les meilleurs solutions en moyenne. En comparaison avec GLPK, ACOv1 et ReactiveGRASP sont très rapides, pouvant fournir jusqu'à 300 fois plus vite des solutions très proches de celles donner par GLPK (i.e. 500rnd1100).

En conclusion, la version de base de ACO n'est pas compétitive avec le ReactiveGRASP. Elle fournie en moyenne des solutions plus intéressantes mais elle nécessite un temps inconsidérable de calcul. En contrepartie, il est utile d'introduire une condition d'arrêt plus ou moins dynamique pour que l'algorithme s'arrête avant d'effectuer toutes les itérations prévues. C'est ce que l'on fait avec ACOv2 qui s'arrête après deux perturbations et nous permet de réduire drastiquement le temps CPUt de ACO. Néanmoins, bien qu'ACOv2 fournisse (comme ACOv1) de meilleures solutions en moyenne par rapport au ReactiveGRASP, la méthode a du mal a trouver les mêmes valeurs z_{max}.

	ReactiveGRASP			ACOv1			ACOv2			GLPK		
	z_{max}	z_{moy}	CPUt(s)	z_{max}	z_{moy}	CPUt(s)	z_{max}	z_{moy}	CPUt(s)	z_{max}	CPUt(s)	
didactique	30*	30	0.001	30*	30	0.019	30*	30	0.0007	30*	0.001	didactique
100rnd0100	372*	361.4	0.160	372*	364.5	1.353	366	365.2	0.146	372*	1.245	100rnd0100
200rnd0100	416*	396.6	0.706	415	406.8	5.029	408	396.9	0.577	408	177.4	200rnd0100
200rnd0300	723	700	3.775	724	714.8	17.779	714	703.4	1.93	712	179.01	200rnd0300
200rnd0500	184*	159.8	0.122	184*	177.4	0.895	184*	175.2	0.174	184*	21.516	200rnd0500
200rnd0900	1324*	1322.1	4.208	1324*	1322.3	12.861	1314	1314	0.939	1324*	0.001	200rnd0900
200rnd1500	926*	919.6	4.013	926*	923.9	24.936	920	920	1.230	926*	6.9	200rnd1500
500rnd0100	308	292.4	4.304	319	308.2	19.184	309	303.2	2.458	266	176.04	500rnd0100
500rnd0700	1141*	1124.7	5.533	1131	1128.3	31.224	1128	1128	2.409	1141*	6.5	500rnd0700
500rnd0900	2223	2203.1	46.198	2218	2202.4	194.432	2190	2190	10.140	2236*	0.011	500rnd0900
500rnd1100	409	376	0.726	419	400.7	6.449	409	396.5	0.626	410	180.05	500rnd1100
1000rnd0100	67*	56.4	2.089	67*	63.7	30.104	57	53.9	2.923	59	183.01	1000rnd0100
1000rnd0500	215	191.2	1.369	213	208.5	7.143	208	202.7	0.747	194	180	1000rnd0500
2000rnd0100	40*	40	8.088	40*	40	83.651	40*	40	9.809	40*	206.694	2000rnd0100
2000rnd0500	140*	125.6	3.154	133	129.5	26.524	131	129.3	2.752	108	182.01	2000rnd0500
	z_{max}	z_{moy}	CPUt(s)	z_{max}	z_{moy}	CPUt(s)	z_{max}	z_{moy}	CPUt(s)	z_{max}	CPUt(s)	
	ReactiveGRASP			ACOv1			ACOv2			GLPK		

TABLEAU 3 – Meilleurs valeurs obtenues pour Reactive GRASP, ACO et GLPK. Les valeurs marquée d'un * sont les valeurs optimales (ou les meilleurs valeurs connues) pour l'instance donnée.

Chapitre 7

Conclusion

GRASP est une métaheuristique simple, très agressive et dispensée de toute forme d'apprentissage sur l'activité de recherche préliminaire à la production de solutions de qualité. Elle est particulièrement bien adaptée à des problèmes d'optimisation où le temps de calcul est limité. Son inconvénient principale est l'indépendance totale entre les itérations GRASP. Nous avons su en faire un avantage en parallélisant l'algorithme, ce qui nous a permis d'obtenir des temps de calcul bien meilleur que ceux de GLPK. Le choix de son unique paramètre (α) est un inconvénient mineur que nous avons résout en implémentant la variante ReactiveGRASP qui permet d'automatiser le réglage de α. Les résultats trouvés en utilisant notre version du ReactiveGRASP sont très proches de ceux obtenus par GLPK. Cette métaheuristique est beaucoup plus rapide que GLPK sur les instances testées. En comparaison, la version de base de ACO est loin d'être aussi compétitive que ReactiveGRASP en terme de temps de calcul. Parcontre, en arrêtant l'algorithme après un certain nombre de perturbations on obtient une métaheuristique dont les temps de calcul et la valeur moyenne des solutions sont légèrement meilleures que pour ReactiveGRASP sur les instances considérées. La méthode ReactiveGRASP reste cependant plus agressives et donne pour certaines instances des solutions optimales ou très proches de l'optimale connue. Une stratégie de restart revisitée et l'utilisation d'une recherche locale en descente profonde pourraient améliorer les résultats d'ACO.

Bibliographie

[1] Contributeurs Wikipedia. *Set packing — Wikipedia, The Free Encyclopedia*. [En ligne ; Page disponible le 22-septembre-2021]. 2021. URL : https://en.wikipedia.org/w/index.php?title=Set_packing&oldid=1020701606.

[2] Xavier Delorme, Xavier Gandibleux et Joaquin Rodriguez. « GRASP for set packing problems ». In : *European Journal of Operational Research* 153.3 (2004). EURO Young Scientists, p. 564-580. ISSN : 0377-2217. DOI : https://doi.org/10.1016/S0377-2217(03)00263-7. URL : https://www.sciencedirect.com/science/article/pii/S0377221703002637.

[3] Xavier Delorme, Xavier Gandibleux et Fabien Degoutin. « Une heuristique hybride pour le problème de set packing biobjectif ». In : *6eme congrès de la société française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'05)*. Tours, France, fév. 2005. URL : https://hal.archives-ouvertes.fr/hal-00387757.

[4] Xavier Gandibleux et al. « An ant colony optimization inspired algorithm for the set packing problem with application to railway infrastructure ». In : *Proceedings of the sixth metaheuristics international conference (MIC2005)*. Citeseer. 2005, p. 390-396.

[5] SNCF. *Trafic de voyageurs et marchandises depuis 1841*. [En ligne ; Page disponible le 23-septembre-2021]. 2018. URL : https://ressources.data.sncf.com/explore/dataset/trafic-de-voyageurs-et-marchandises-depuis-1841/analyze/.

[6] Contributeurs Wikipedia. *Transport ferroviaire en France — Wikipédia, l'encyclopédie libre*. [En ligne ; Page disponible le 23-septembre-2021]. 2021. URL : http://fr.wikipedia.org/w/index.php?title=Transport_ferroviaire_en_France&oldid=185270789.

[7] Aurelien Merel, Sophie Demassey et Xavier Gandibleux. « Un algorithme de génération de colonnes pour le problème de capacité d'infrastructure ferroviaire ». In : *ROADEF 2010*. Toulouse, France, fév. 2010, papier 104. URL : https://hal.archives-ouvertes.fr/hal-00466939.

[8] Delorme Xavier. *Instances tests pour le problème de Set Packing*. [En ligne ; Page disponible le 27-septembre-2021]. 2021. URL : https://www.emse.fr/~delorme/SetPackingFr.html.

[9] Thomas A Feo et Mauricio GC Resende. « Greedy randomized adaptive search procedures ». In : *Journal of global optimization* 6.2 (1995), p. 109-133.

[10] Marcelo Prais et Celso C Ribeiro. « Reactive GRASP : An application to a matrix decomposition problem in TDMA traffic assignment ». In : *INFORMS Journal on Computing* 12.3 (2000), p. 164-176.

[11] Fred Glover et Manuel Laguna. « Tabu search ». In : *Handbook of combinatorial optimization*. Springer, 1998, p. 2093-2229.

[12] Xavier Gandibleux, Xavier Delorme et Vincent T'Kindt. « An ant colony optimisation algorithm for the set packing problem ». In : *International Workshop on Ant Colony Optimization and Swarm Intelligence*. Springer. 2004, p. 49-60.

[13] J Jorge et al. *Algorithme de fourmis pour mesurer et optimiser la capacité d'un réseau ferroviaire*. 2009.