LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

Nantes Université

GRASP 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 DM2

Juanfer MERCIER – Adrien PICHON

Mardi 25 Octobre 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.

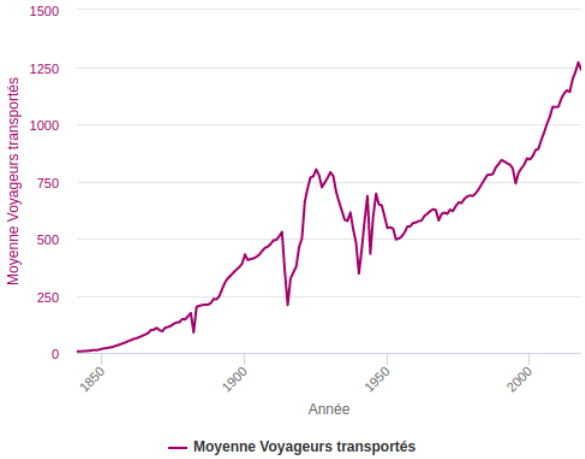

FIGURE 1 – Moyenne des voyageurs transportés par la SNCF depuis 1937 et par les compagnies privées qu'elle a absorbées, avant 1937. [5] [6]

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).

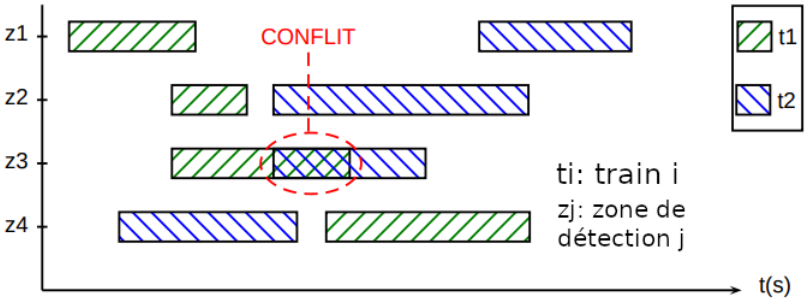

FIGURE 2 – Délai (en secondes) nécessaire pour que les trains t_i transitent par les zones z_j ($i \in \mathbb{N}, j \in \mathbb{N}$). Avec t_1 les trains initiaux et t_2 les trains saturants. — `https://www.emse.fr/~delorme/Papiers/These/These_slides.pdf` (Adaptation de l'illustration p.15)

Chapitre 2

Les instances numériques de SPP

Le tableau 1 indique les noms des 10 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.

Toutes les instances numériques utilisées pour les expérimentations que nous présenterons ont été sélectionnées arbitrairement et suivent le format de la OR-Library.

Instances	**Nb. de variables**	**Nb. de contraintes**	**Meilleure valeur connue**
`didactic`	9	7	30
`pb_100rnd0100`	100	500	372
`pb_200rnd0100`	200	1000	416
`pb_200rnd0300`	200	1000	731
`pb_200rnd0900`	200	200	1324
`pb_200rnd1500`	200	600	926
`pb_500rnd0100`	500	2500	323*
`pb_500rnd0700`	500	500	1141
`pb_1000rnd0100`	1000	5000	67
`pb_1000rnd0500`	1000	1000	222*

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
$a_{i,j} =$	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
	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 deux 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 et le 1-1 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. Notez aussi que nous avons présenté notre heuristique d'amélioration comme utilisant deux voisinages $\mathcal{N}_{2-1}(x_0)$ et $\mathcal{N}_{1-1}(x_0)$ obtenus par 2-1 et 1-1 échanges. En pratique, nous utilisons un troisième voisinage $\mathcal{N}_{0-1}(x_0)$ obtenu par 0-1 échange. Nous avons choisie d'omettre l'inclusion de $\mathcal{N}_{0-1}(x_0)$ dans la présentation de notre heuristique 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
  $initialSol \leftarrow greedyRandomized(problem, \alpha)$
  $improvedSol \leftarrow localSearch(initialSol)$
  $Solutions \leftarrow Solutions \cup \{improvedSol\}$
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
  $\alpha^* \leftarrow RandomSelect(alphaSet, proba)$
  $initialSol \leftarrow greedyRandomized(problem, \alpha^*)$
  $improvedSol \leftarrow localSearch(initialSol)$
  $Solutions \leftarrow Solutions \cup \{improvedSol\}$
  $Pool_{\alpha^*} \leftarrow Pool_{\alpha^*} \cup \{improvedSol\}$
  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

Expérimentation numérique

5.1 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™ i7-10750H de 10ème génération (6 coeurs, 12 threads, 2.6 GHz de base / 5.0 GHz maximum). Chaque solveur est lancé 10 fois sur les instances présentées au Chapitre 2 avec 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. Pour GLPK, la valeur de la meilleure solution obtenu $\hat{z}$ sera donnée.
- les valeurs saillantes de α pour le Reactive GRASP

5.1.1 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.

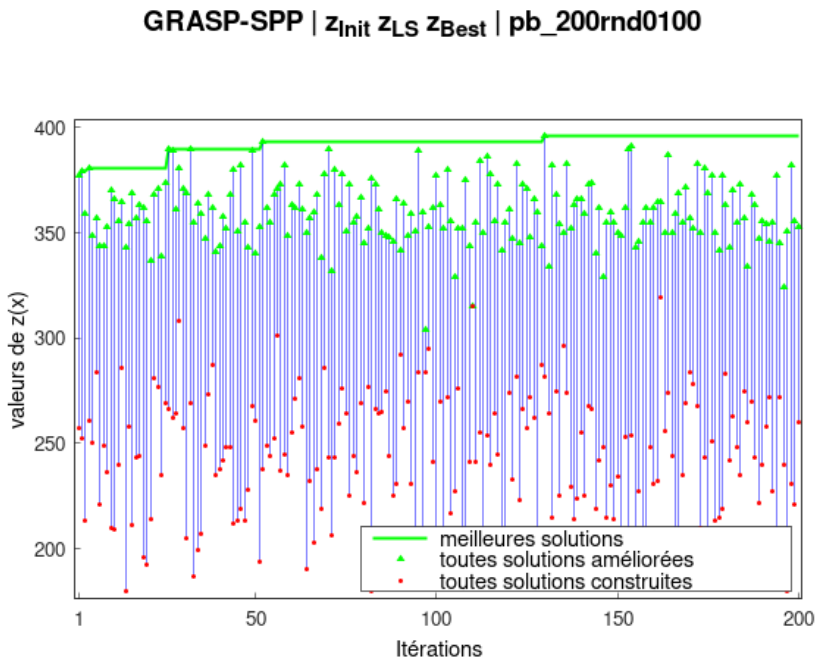

FIGURE 3 – Exemple d'une exécution de (Reactive) GRASP

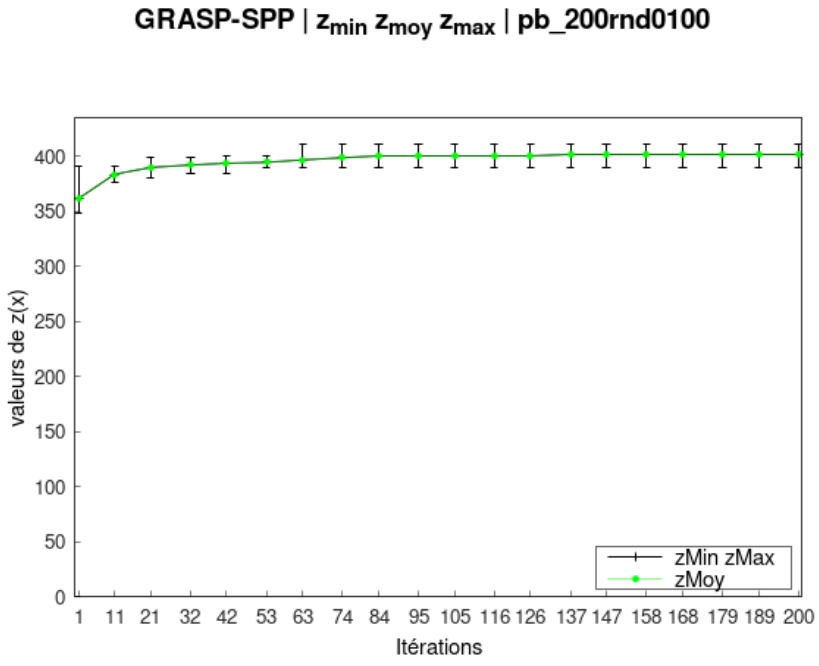

FIGURE 4 – Exemple d'expérimentation numérique de (Reactive) GRASP

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.

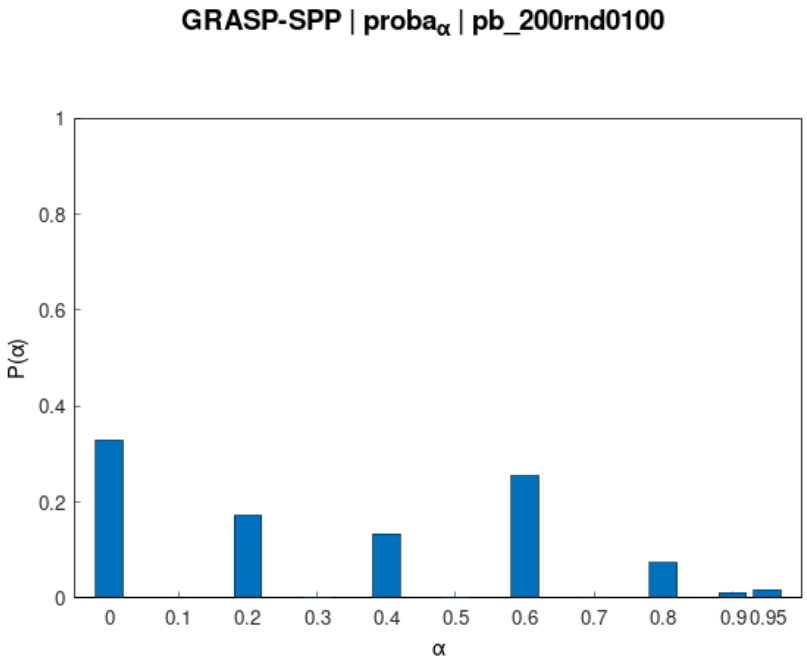

FIGURE 5 – Exemple de probabilités des valeurs de α obtenus à la fin d'une exécution Reactive GRASP

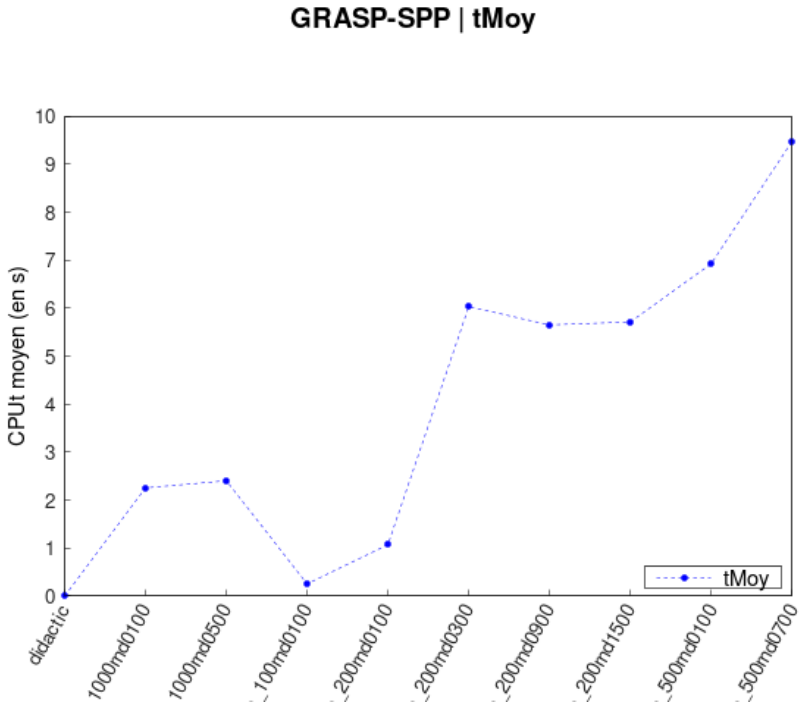

FIGURE 6 – Exemple d'affichage des temps CPUt moyen

5.2 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 $\delta = 4$. De plus, l'ensemble des résultats des expérimentations sont présentées Annexe A et B.

5.2.1 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. Dans l'annexe A, 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 étant donné que ces recherches locales sont démarrées à partir de solution construite plus variées). Pour les valeurs α plus proche de 1, nous remarquons des différences de résultats entre les instances. Pour l'instance `pb_200rnd0100` on remarque que plus la valeur de α est proche de 1, moins les valeurs $\hat{z}_{max}$ et $\hat{z}_{min}$ obtenues sont bonnes (figures 15 pour $\alpha = 0$ et 23 pour $\alpha = 0.8$). À l'opposé, pour l'instance `pb_1000rnd0100`, les solutions sont meilleures pour des valeurs de α proches de 1 (figures 16 pour $\alpha = 0$ et 24 pour $\alpha = 0.8$). Enfin, il y a des instances telles que `pb_1000rnd0500` qui ne sont pas très affectées par les valeurs de α (figures 17 pour $\alpha = 0$ et 25 pour $\alpha = 0.8$). Les valeurs 0.4, 0.6 et 0.7 ont aussi été testées et les résultats confirment le choix d'implémenter le Reactive GRASP car aucune valeur de α ne semble idéale pour l'ensemble des instances.

5.2.2 Réglage de N_α pour Reactive GRASP

En comparant les figures obtenus avec $N_\alpha = 50$ (figures 27 à 33) aux figures obtenus avec $N_\alpha = 20$ (figures 34 à 40) on remarque qu'avec $N_\alpha = 20$ les probabilités des valeurs de α pour lesquelles ont a de moins bonnes solutions (*e.g.* FIGURE 35) 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.

En effet, on peut observer en comparant les figures 27, 29 et 31 aux figures 34, 36 et 38 que les marges d'erreurs (c'est-à-dire les valeurs $\hat{z}_min$ et $\hat{z}_max$) sont moins amples pour $N_\alpha = 20$. 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 (figures 33, 40 et 41).

5.2.3 Réglage de la parallélisation

Nous pouvons observé en comparant la FIGURE 10 et la FIGURE 18 qu'en descente profonde sans parallélisation, GRASP résout les instances jusqu'à 5 fois plus lentement que la version parallélisée. Naturellement, la comparaison des figures 7 à 9 aux figures 15 à 17 montre que le comportement de GRASP est le même pour une valeur α donnée que les itérations soit parallélisées ou non. Le même raisonnement peut être fait lorsque GRASP est lancé en descente rapide : l'algorithme est jusqu'à 6 fois plus lent sans parallélisation (FIGURE 14 et 22) pour des résultats très similaires (figures 11 à 13 et 19 à 21). On remarque aussi que l'algorithme est plus rapide en descente rapide qu'en descente profonde avec des résultats très similaires. Bien sûr, ces comparaisons entre les versions non-parallélisées et celles parallélisées auraient été plus flagrantes en choisissant une même graine aléatoire pour comparer les différentes versions mais nous espérons tout de même convaincre le lecteur de l'efficacité de la version parallélisée du GRASP.

Pour le Reactive GRASP, le même raisonnement peut être fait.

5.3 Résultats de l'expérimentation

Au vu des résultats discutés section 5.2, nous avons décidé de comparer GLPK à notre 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

Chaque solveur est lancé 10 fois. Les résultats sont rapportés dans le tableau 2. Nous pouvons observer sur le tableau 2 que Reactive GRASP est plus rapide que GLPK pour toutes les instances testées. Il est 34 fois plus rapide que GLPK sur l'instance `pb_500rnd_0100` et 191 fois plus rapide que GLPK sur l'instance `pb_1000rnd_0100`. De plus, Reactive GRASP trouve des solutions très approchées de celles de GLPK (`pb_200rnd_0100`, `pb_200rnd_0900`, `pb_200rnd_1500`, `pb_500rnd_0700` et `pb_1000rnd_0100`), égales à celles de GLPK (`didactic` et `pb_100rnd_0100`) et parfois même supérieur à celles de GLPK (`pb_200rnd_0300`, `pb_500rnd_0100` et `pb_1000rnd_0500`). Enfin, GLPK trouve la solution optimale (ou la meilleure solution connue) de trois instances : `pb_100rnd_0100`, `pb_200rnd_0900` et `pb_500rnd_0700`). Le Reactive GRASP trouve la solution optimale (ou la meilleure solution connue) de l'instance `pb_100rnd_0100`.

	Reactive GRASP		GLPK		
Instances	$\hat{z}_{max}$	CPUt moyen (en s)	$\hat{z}_{max}$	CPUt moyen (en s)	**Instances**
didactic	30	0.0009	30	0.001	didactic
pb_100rnd_0100	372*	0.107	372*	1.323	pb_100rnd_0100
pb_200rnd_0100	404	0.83	408	180.04	pb_200rnd_0100
pb_200rnd_0300	714	2.496	712	180.02	pb_200rnd_0300
pb_200rnd_0900	1323	0.37	1324*	0.001	pb_200rnd_0900
pb_200rnd_1500	925	1.14	926*	6.889	pb_200rnd_1500
pb_500rnd_0100	312	5.2	266	180.353	pb_500rnd_0100
pb_500rnd_0700	1140	5.095	1141*	6.864	pb_500rnd_0700
pb_1000rnd_0100	53	2.002	59	183.01	pb_1000rnd_0100
pb_1000rnd_0500	221	2.004	194	180.3	pb_1000rnd_0500
Instances	$\hat{z}_{max}$	CPUt moyen (en s)	$\hat{z}_{max}$	CPUt moyen (en s)	**Instances**
	Reactive GRASP		**GLPK**		

Tableau 2 – Meilleurs valeurs obtenues pour Reactive GRASP et GLPK. Les valeurs marquée d'un * sont les valeurs optimales (ou les meilleurs valeurs connues) pour l'instance donnée.

5.4 Conclusion de l'expérimentation

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 Reactive GRASP qui permet d'automatiser le réglage de α. Les résultats trouvés en utilisant notre version du Reactive GRASP sont très proches de ceux obtenus par GLPK et parfois même supérieur. Cette métaheuristique est beaucoup plus rapide que GLPK sur les instances testées. Pour finir, il serait intéressant de chercher à munir notre version parallélisée de Reactive GRASP d'un mécanisme de mémoire (*e.g.* path relinking).

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.

Annexe A

Réglage du paramètre α du GRASP

Nous présentons ici les résultats obtenus pour chaque valeur de α testée et sur les instances suivantes :

- `pb_200rnd0100`
- `pb_1000rnd0100`
- `pb_1000rnd0500`

Résultats pour $\alpha = 0.0$

Sans parallélisation

Descente profonde

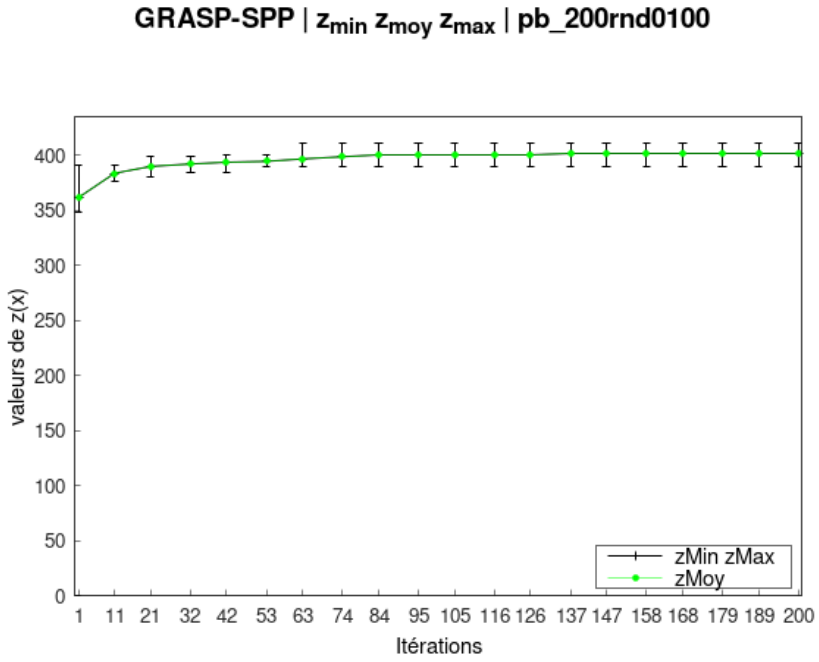

FIGURE 7 – Expérimentation numérique GRASP sur l'instance `pb_200rnd0100`

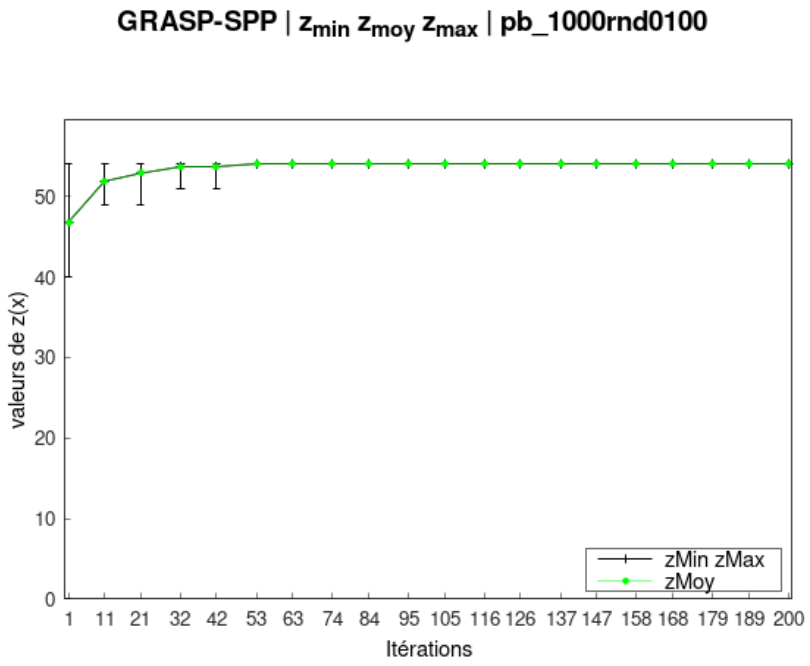

FIGURE 8 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0100`

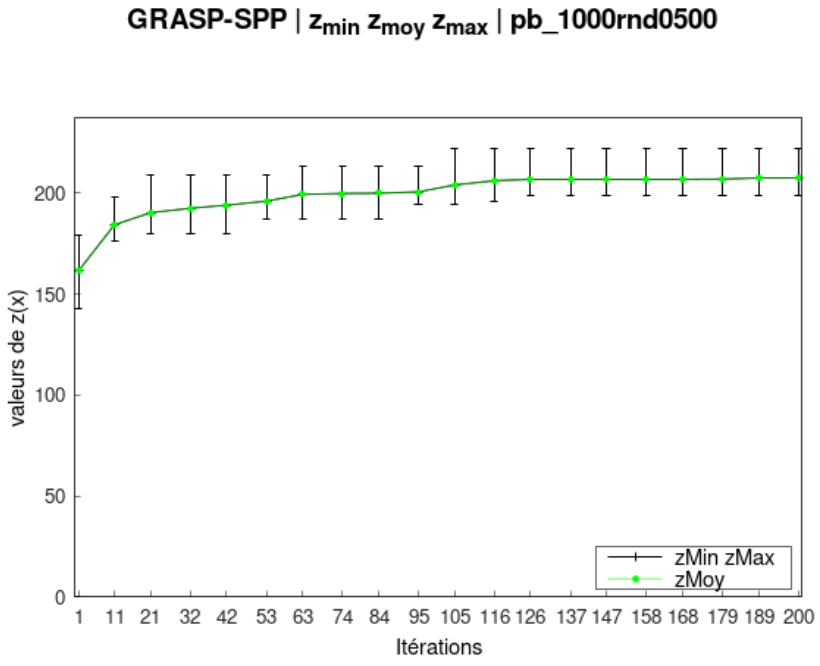

FIGURE 9 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0500`

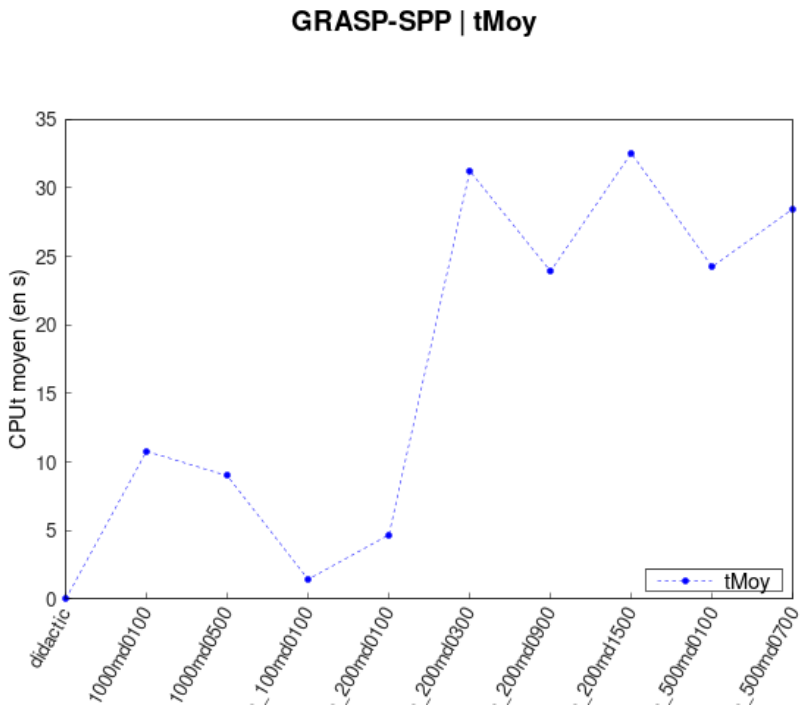

FIGURE 10 – Temps CPUt moyen pour chaque instance

Descente rapide

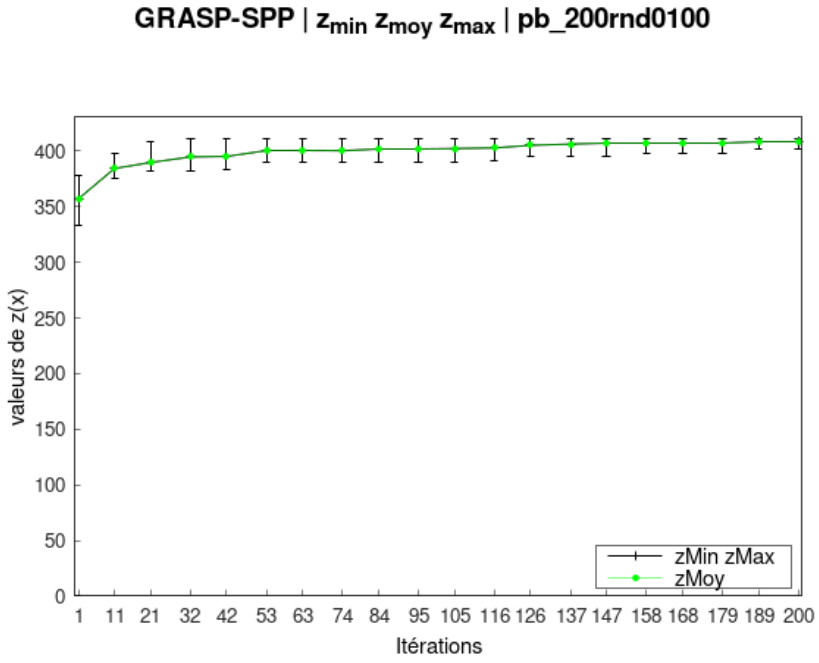

FIGURE 11 – Expérimentation numérique GRASP sur l'instance `pb_200rnd0100`

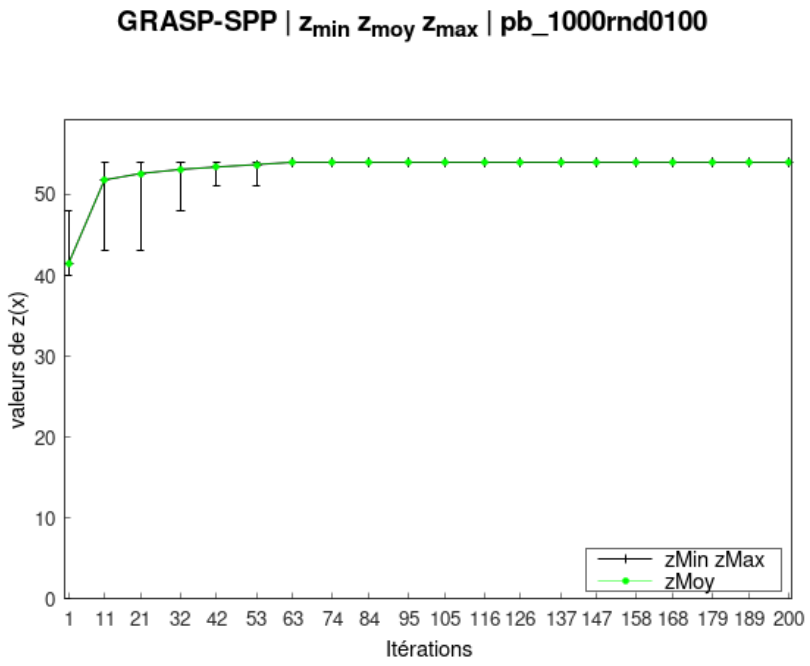

FIGURE 12 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0100`

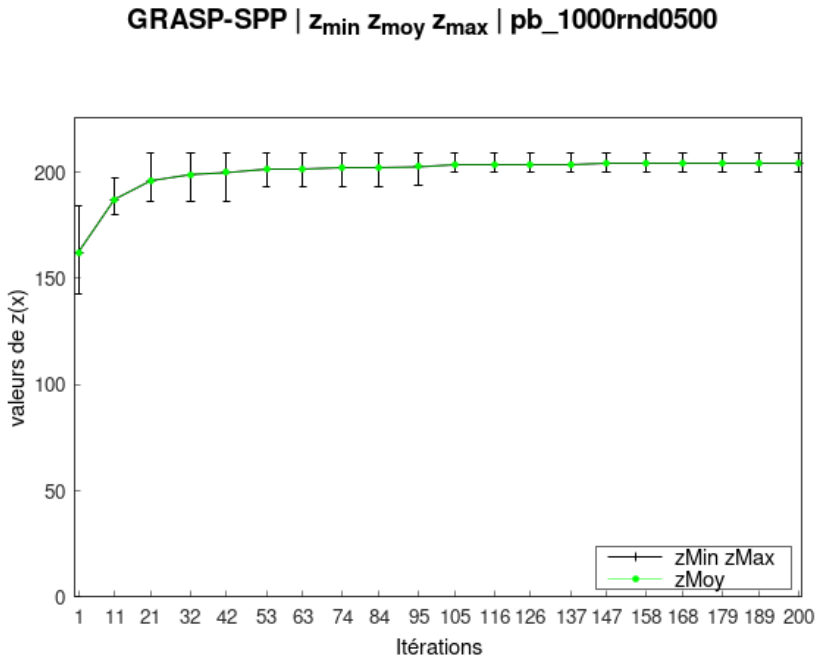

FIGURE 13 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0500`

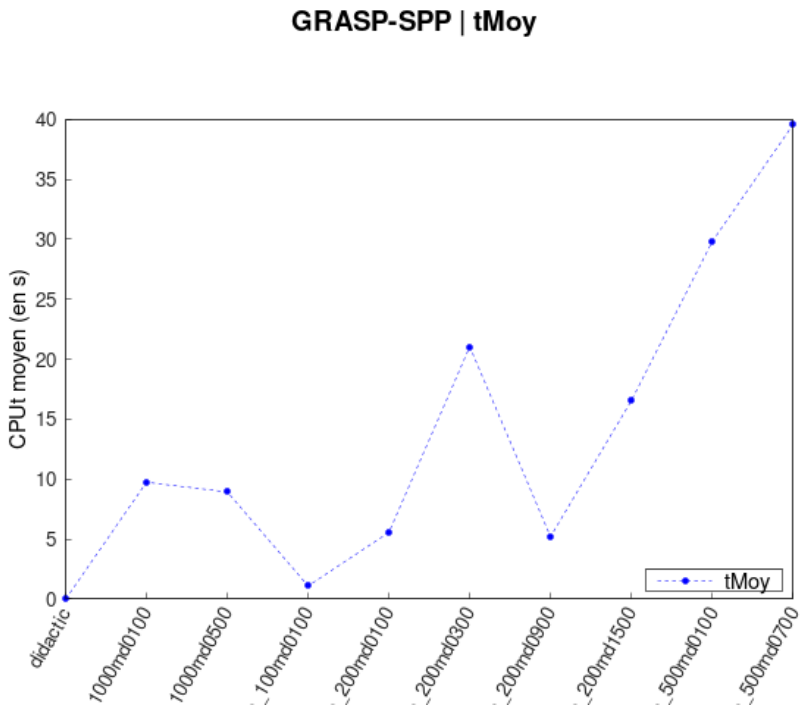

FIGURE 14 – Temps CPUt moyen pour chaque instance

Avec parallélisation

Descente profonde

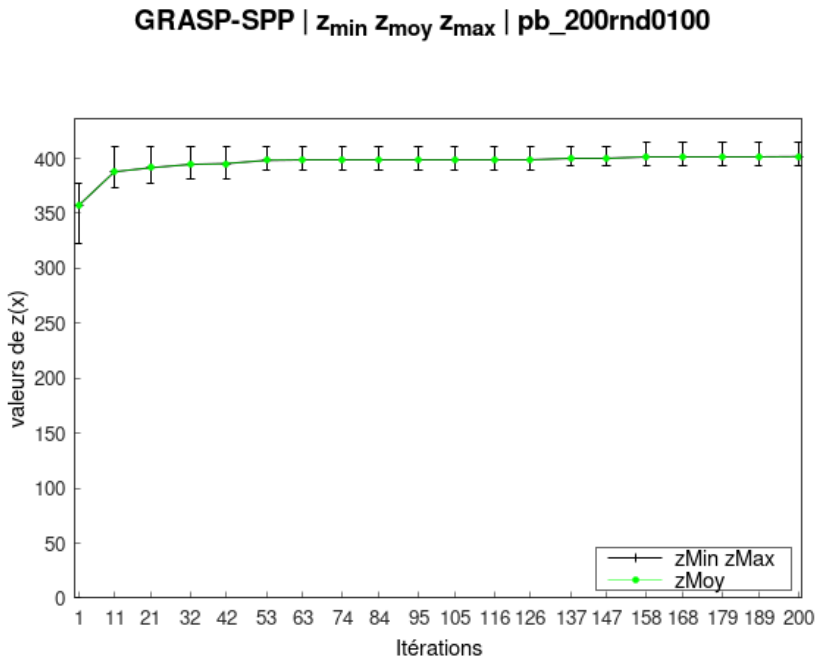

FIGURE 15 – Expérimentation numérique GRASP sur l'instance `pb_200rnd0100`

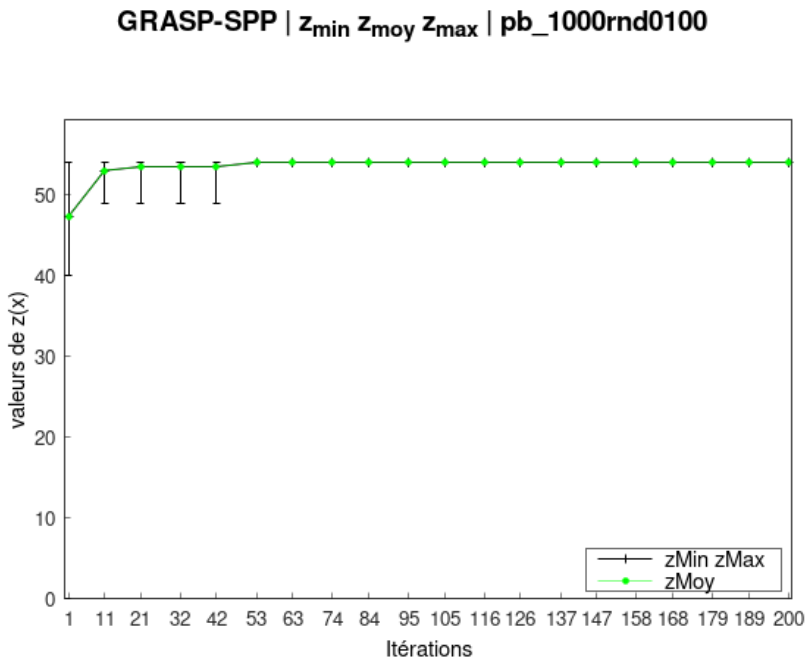

FIGURE 16 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0100`

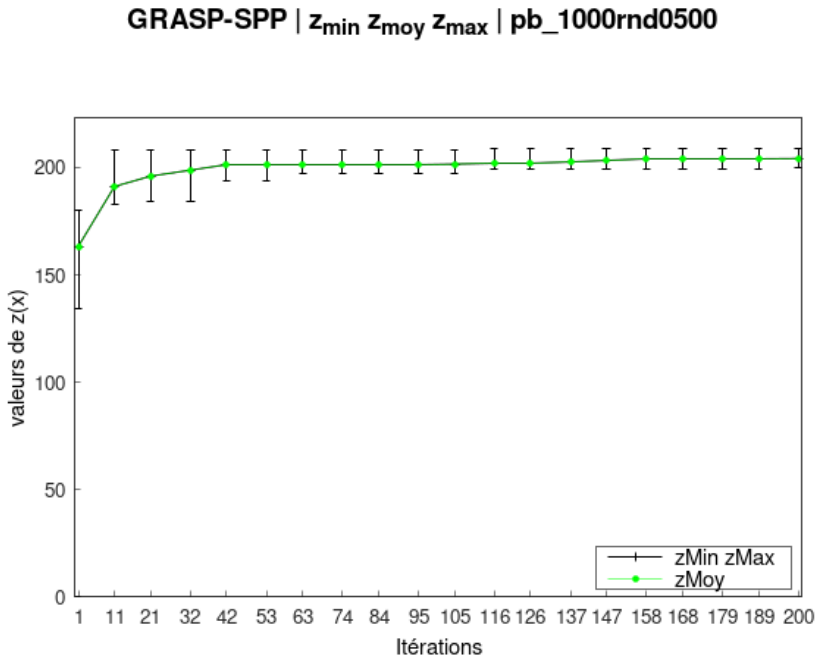

FIGURE 17 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0500`

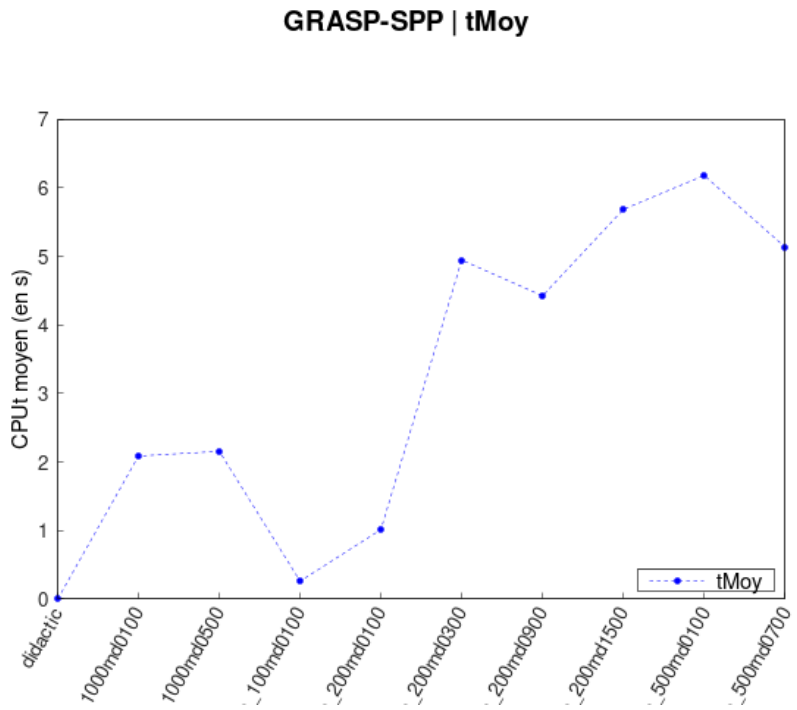

FIGURE 18 – Temps CPUt moyen pour chaque instance

Descente rapide

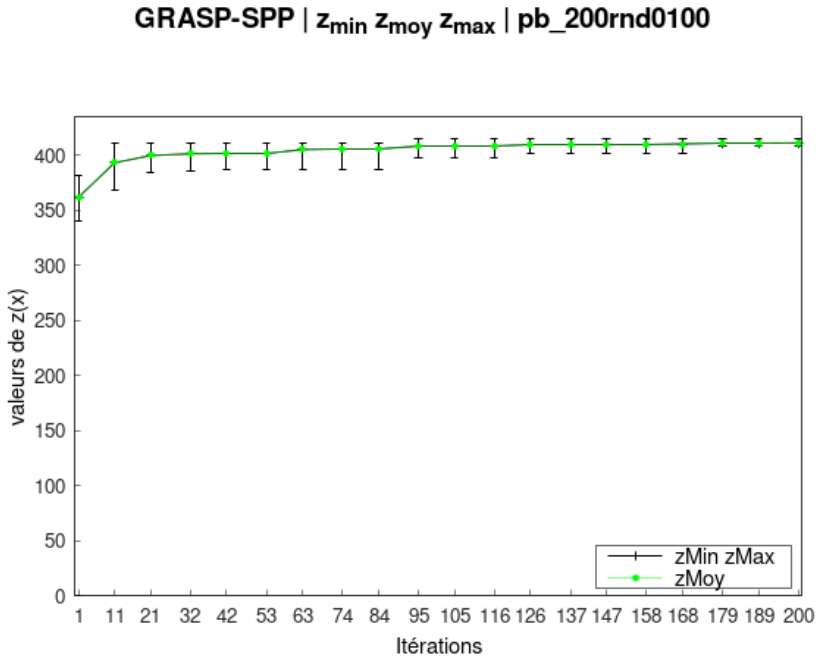

FIGURE 19 – Expérimentation numérique GRASP sur l'instance `pb_200rnd0100`

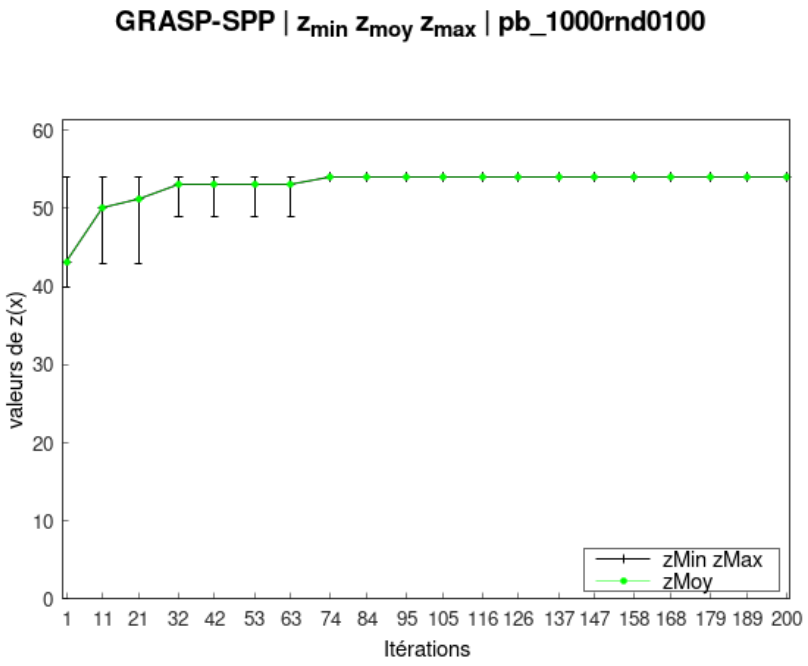

FIGURE 20 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0100`

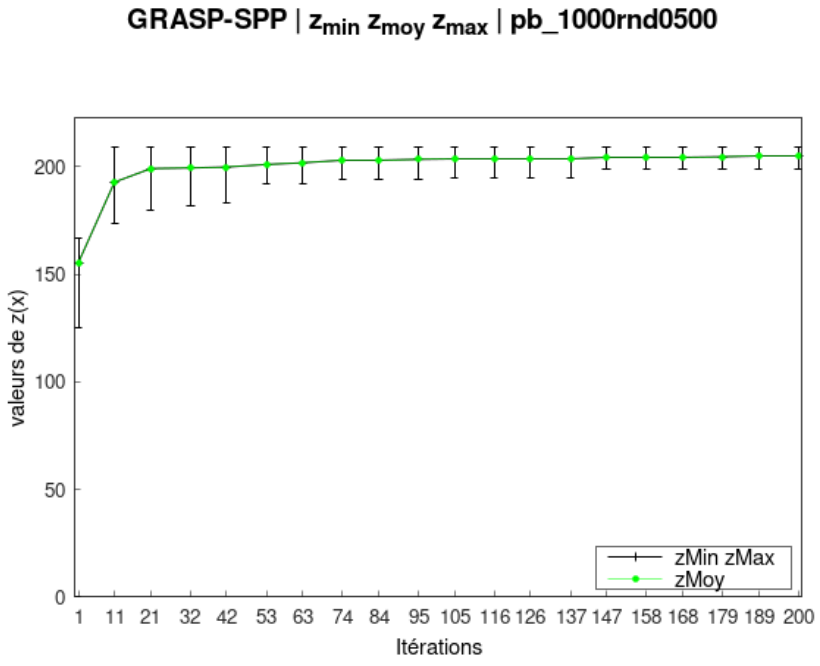

FIGURE 21 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0500`

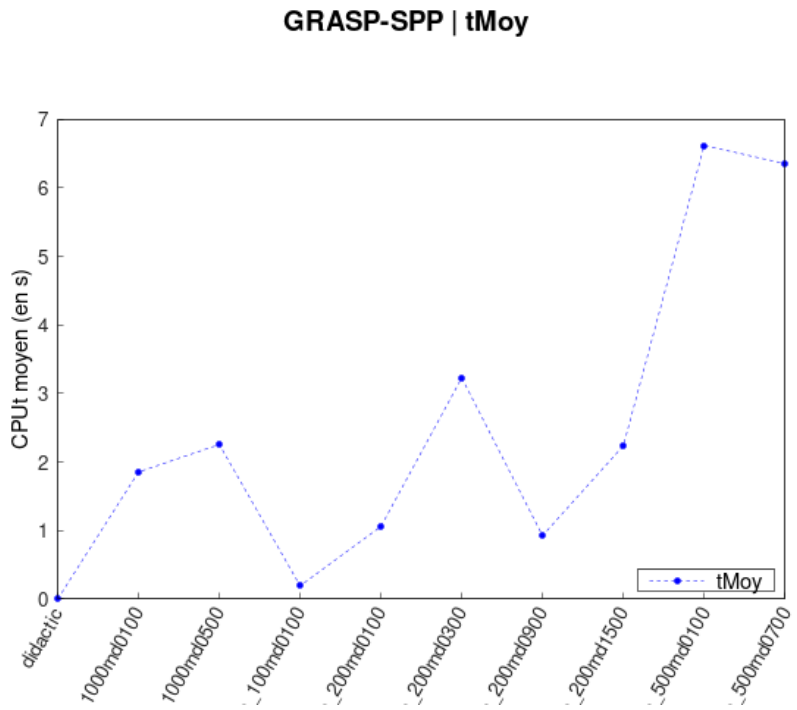

FIGURE 22 – Temps CPUt moyen pour chaque instance

Résultats pour $\alpha = 0.8$ avec parallélisation et descente profonde

Descente profonde

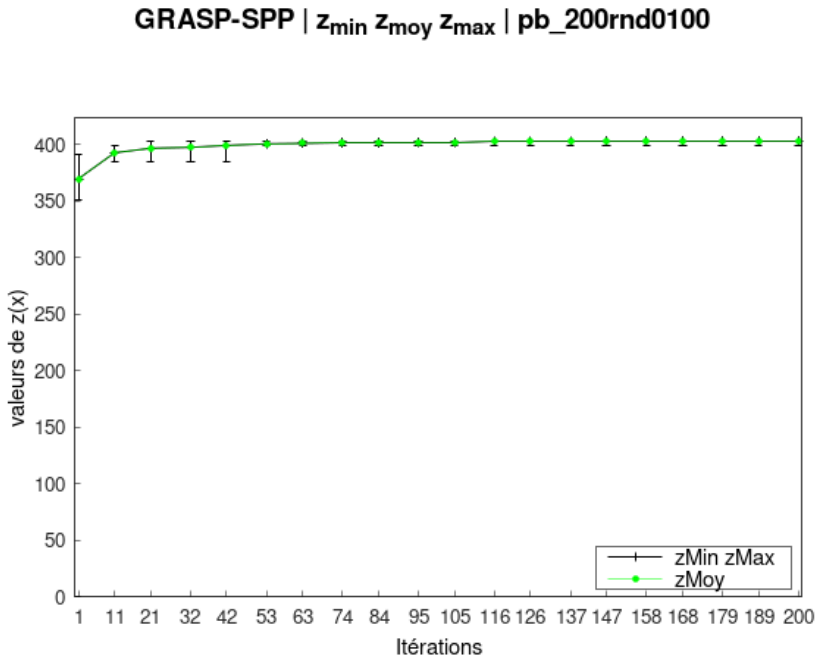

FIGURE 23 – Expérimentation numérique GRASP sur l'instance `pb_200rnd0100`

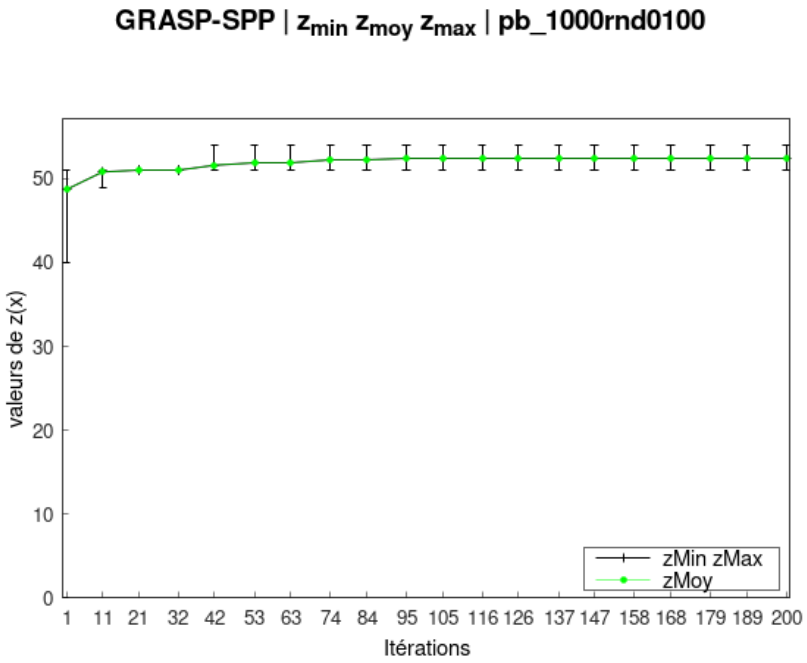

FIGURE 24 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0100`

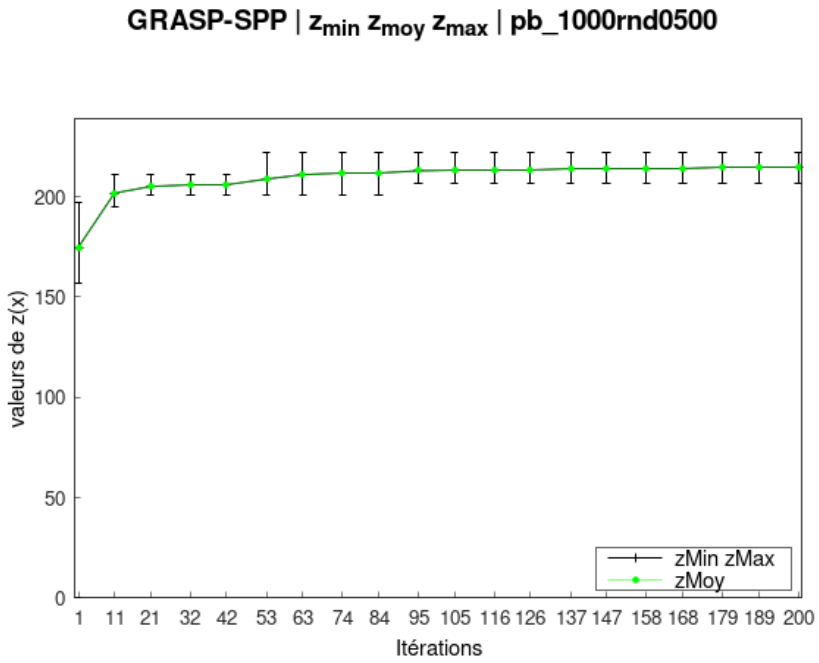

FIGURE 25 – Expérimentation numérique GRASP sur l'instance `pb_1000rnd0500`

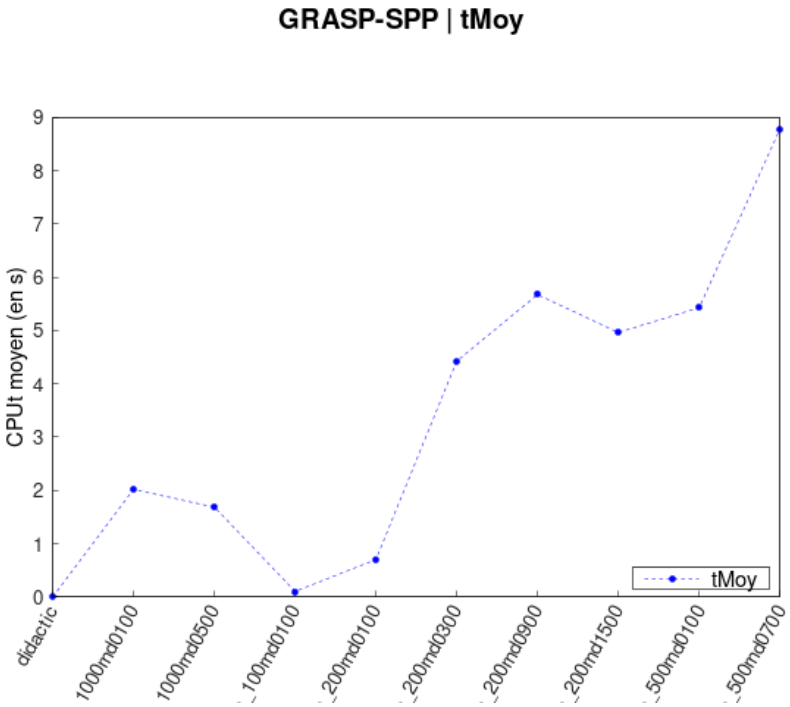

FIGURE 26 – Temps CPUt moyen pour chaque instance

Annexe B

Réglage du paramètre N_α du Reactive GRASP

Nous présentons ici les résultats obtenus pour chaque valeur de N_α testée avec $alphaSet = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95\}$ et $\delta = 4$ sur les instances :

- `pb_200rnd0100`
- `pb_200rnd0300`
- `pb_500rnd0100`

Résultats pour $N_\alpha = 50$ avec parallélisation et descente profonde

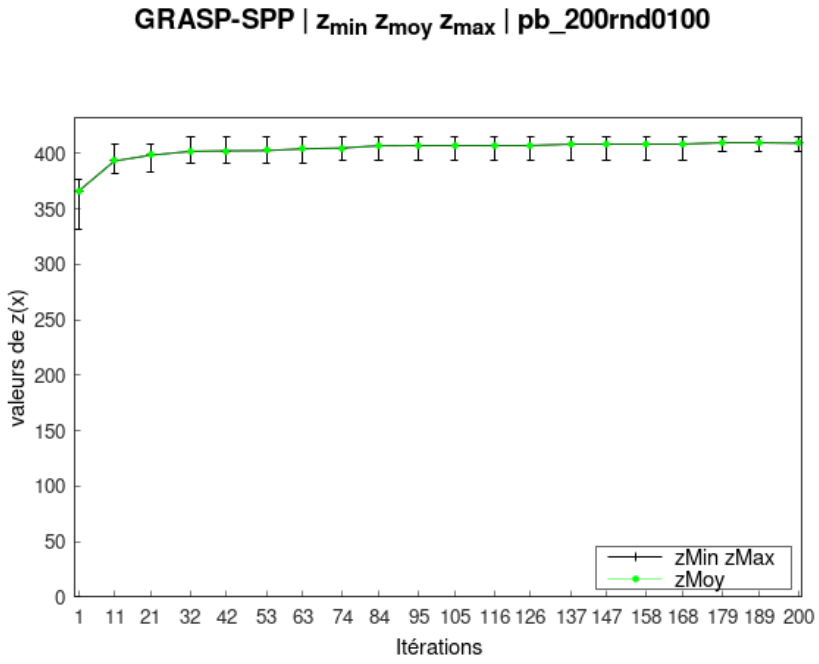

FIGURE 27 – Expérimentation numérique Reactive GRASP sur l'instance `pb_200rnd0100`

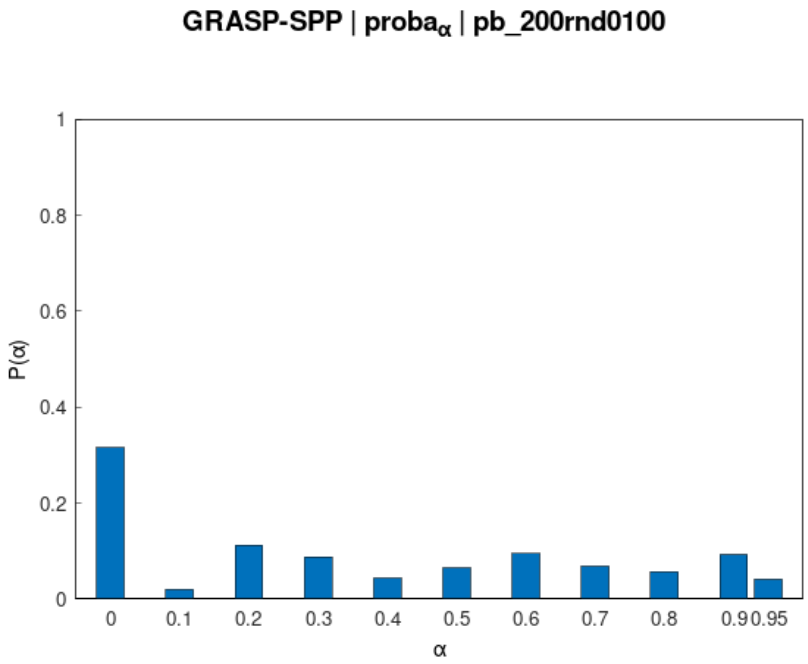

FIGURE 28 – Probabilités des valeurs de α obtenus à la fin de la dernière exécution de Reactive GRASP sur l'instance `pb_200rnd0100`

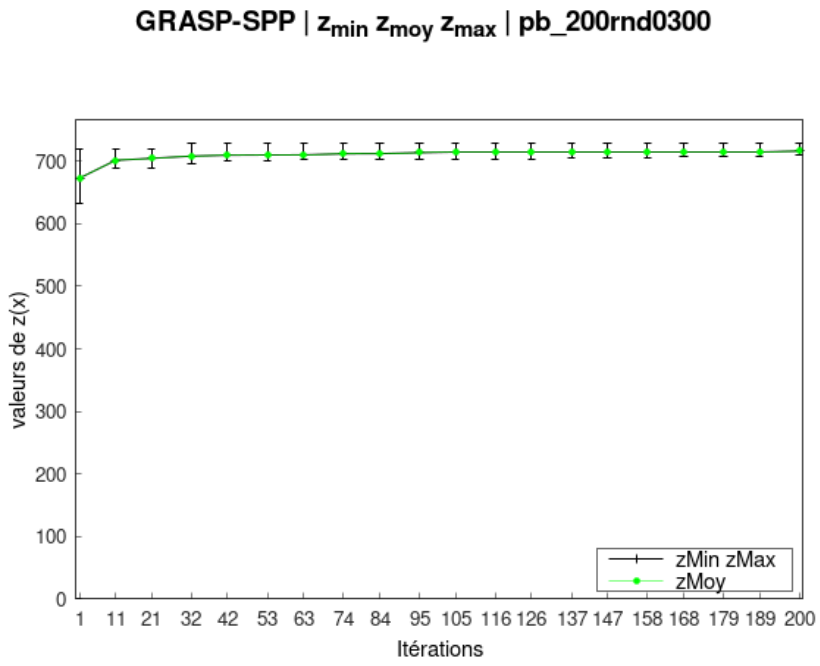

FIGURE 29 – Expérimentation numérique Reactive GRASP sur l'instance `pb_200rnd0300`

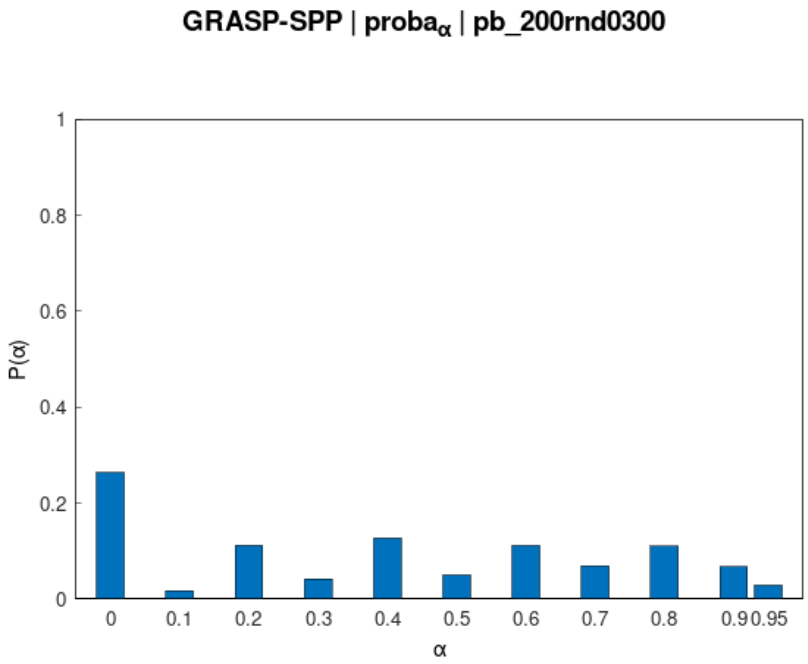

FIGURE 30 – Probabilités des valeurs de α obtenus à la fin de la dernière exécution de Reactive GRASP sur l'instance `pb_200rnd0300`

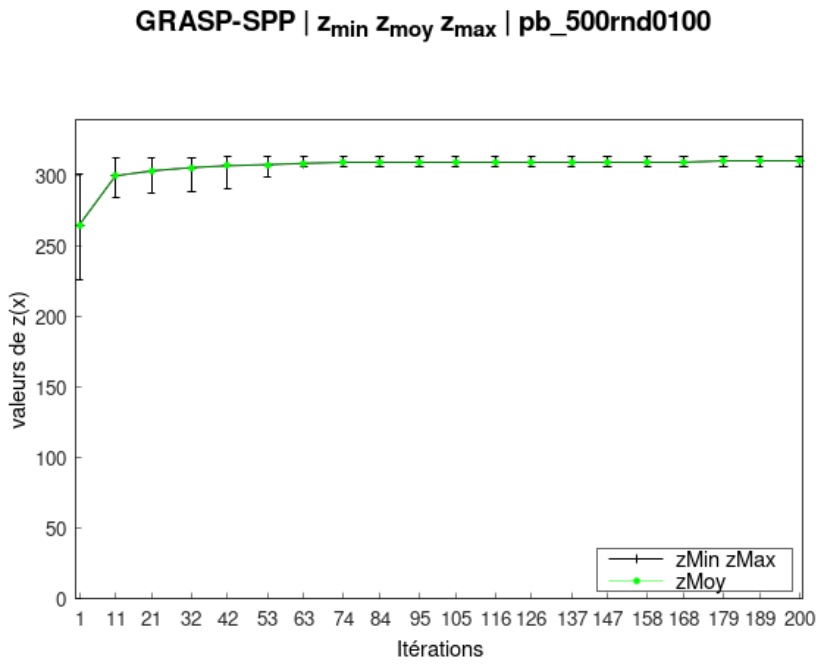

FIGURE 31 – Expérimentation numérique Reactive GRASP sur l'instance `pb_500rnd0100`

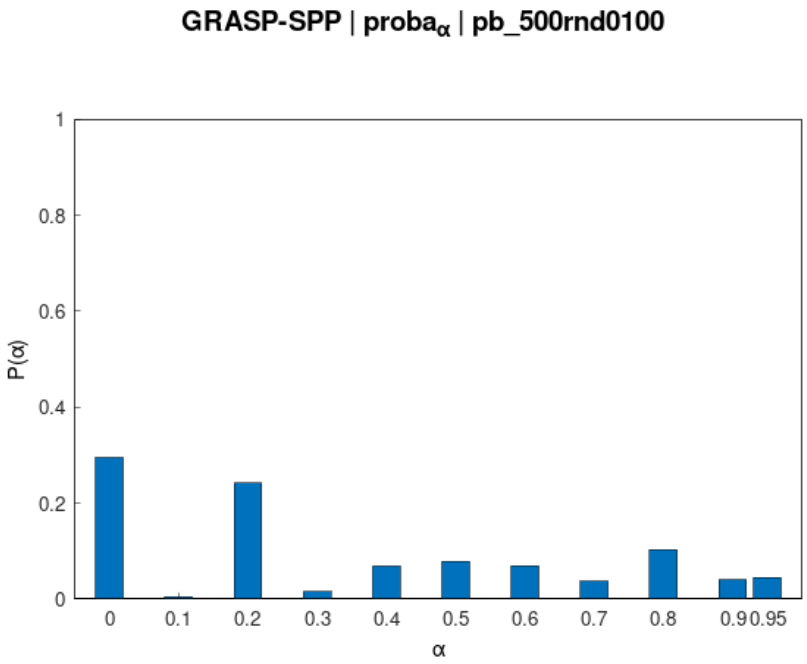

FIGURE 32 – Probabilités des valeurs de α obtenus à la fin de la dernière exécution de Reactive GRASP sur l'instance `pb_500rnd0100`

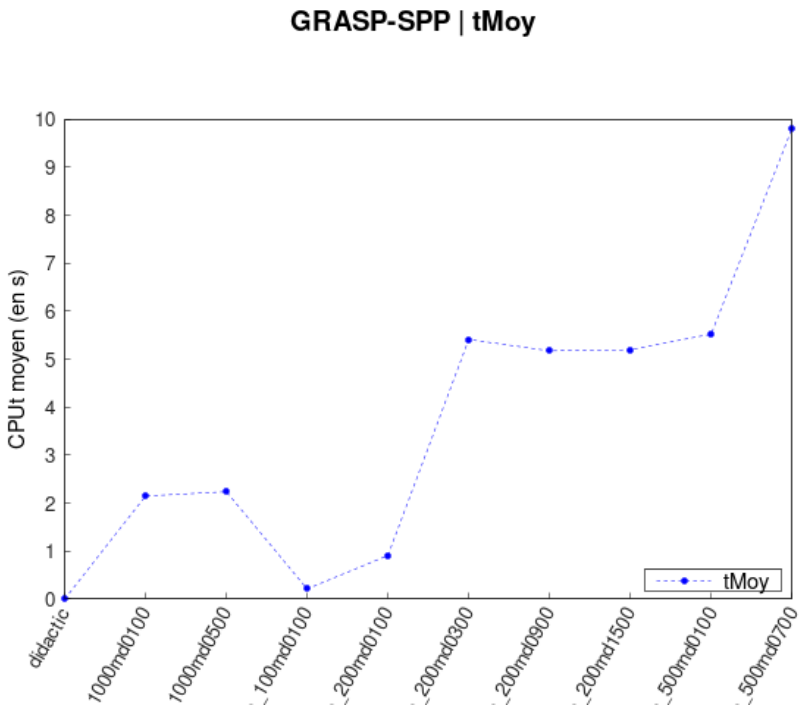

FIGURE 33 – Temps CPUt moyen pour chaque instance

Résultats pour $N_\alpha = 20$ avec parallélisation et descente profonde

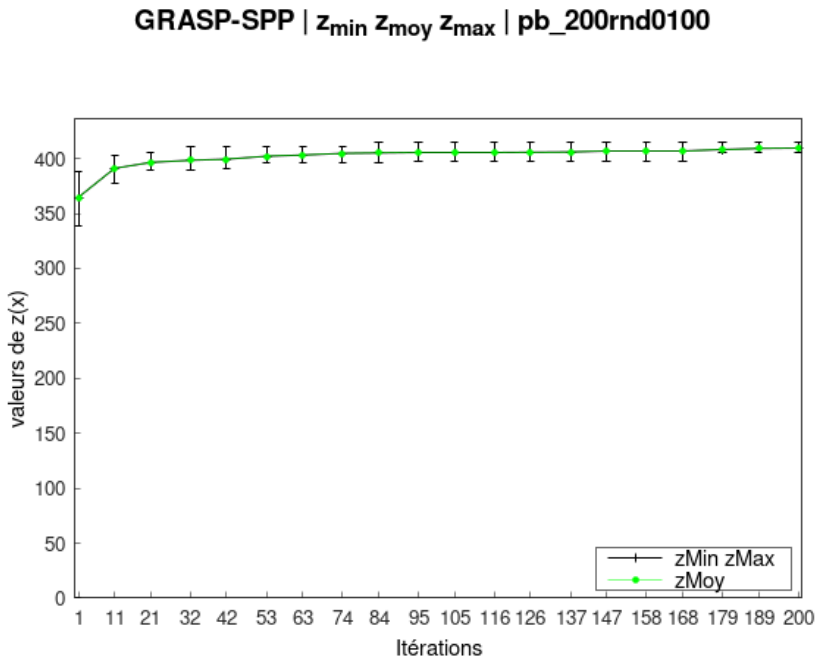

FIGURE 34 – Expérimentation numérique Reactive GRASP sur l'instance `pb_200rnd0100`

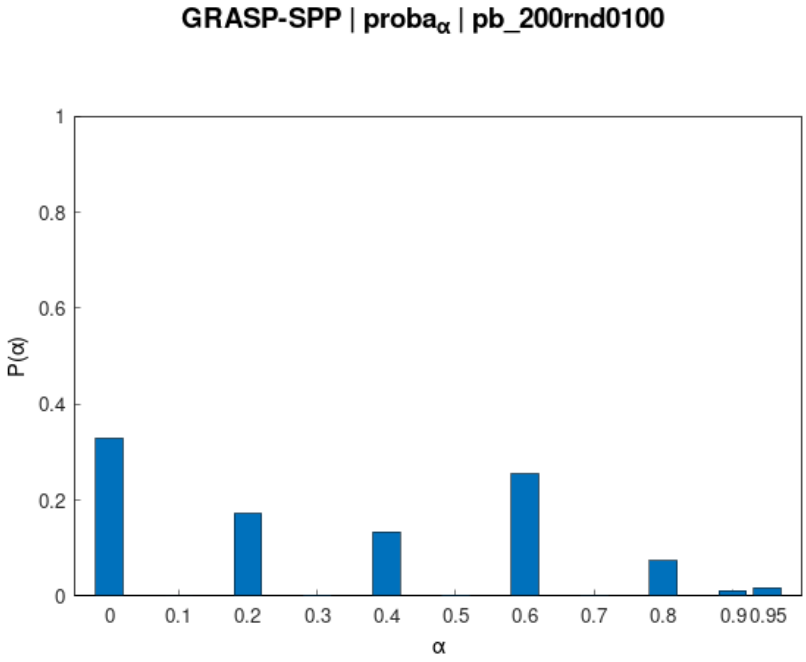

FIGURE 35 – Probabilités des valeurs de α obtenus à la fin de la dernière exécution de Reactive GRASP sur l'instance `pb_200rnd0100`

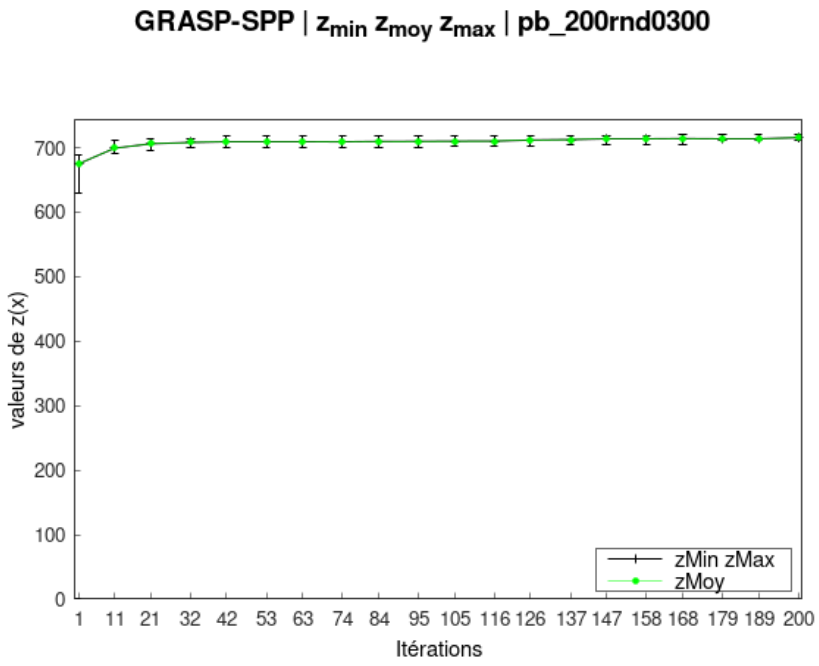

FIGURE 36 – Expérimentation numérique Reactive GRASP sur l'instance `pb_200rnd0300`

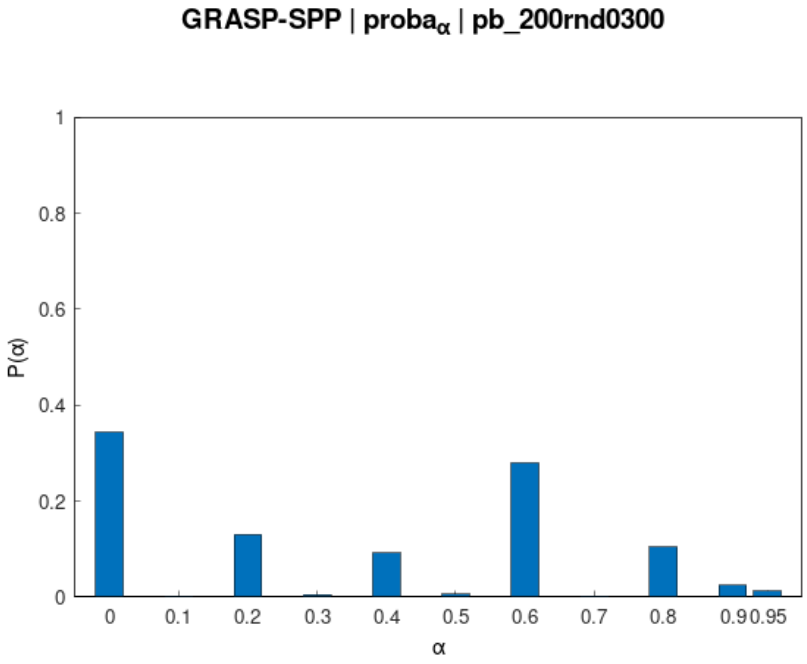

FIGURE 37 – Probabilités des valeurs de α obtenus à la fin de la dernière exécution de Reactive GRASP sur l'instance `pb_200rnd0300`

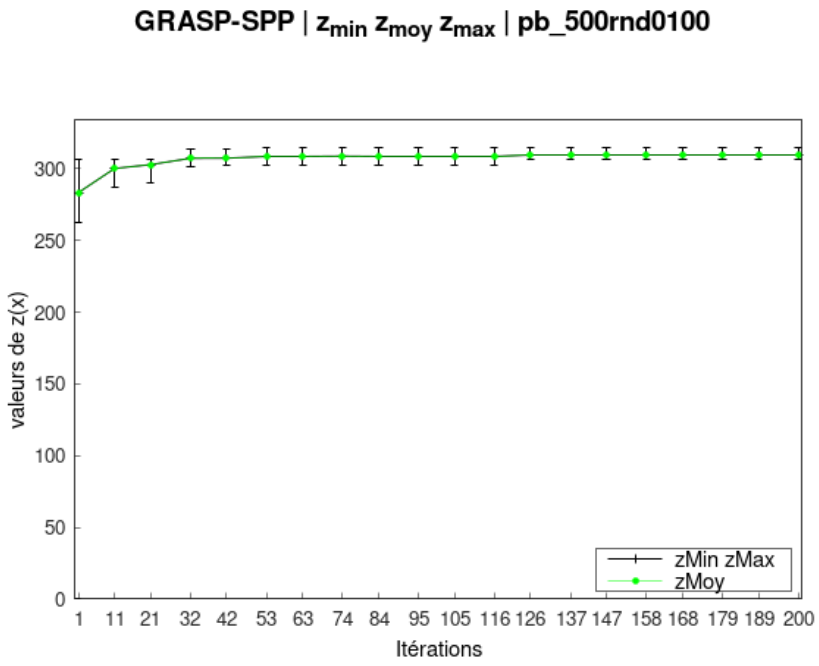

FIGURE 38 – Expérimentation numérique Reactive GRASP sur l'instance `pb_500rnd0100`

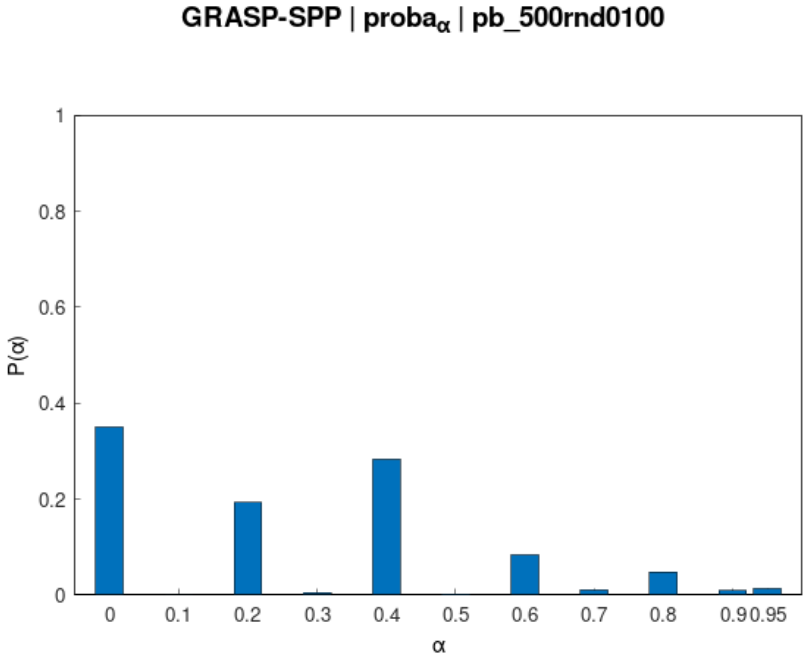

FIGURE 39 – Probabilités des valeurs de α obtenus à la fin de la dernière exécution de Reactive GRASP sur l'instance `pb_500rnd0100`

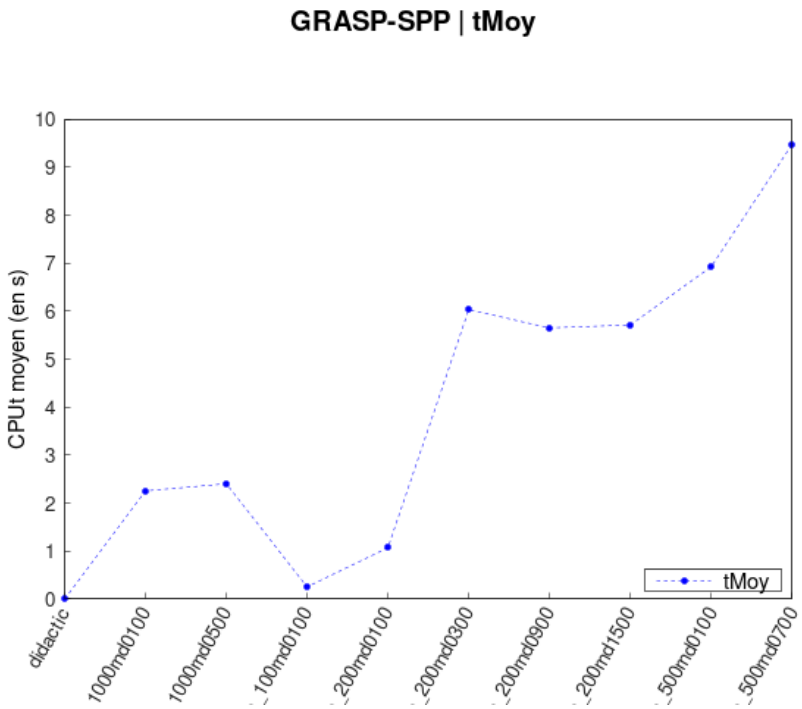

FIGURE 40 – Temps CPUt moyen pour chaque instance

Résultats pour $N_\alpha = 10$ avec parallélisation et descente profonde

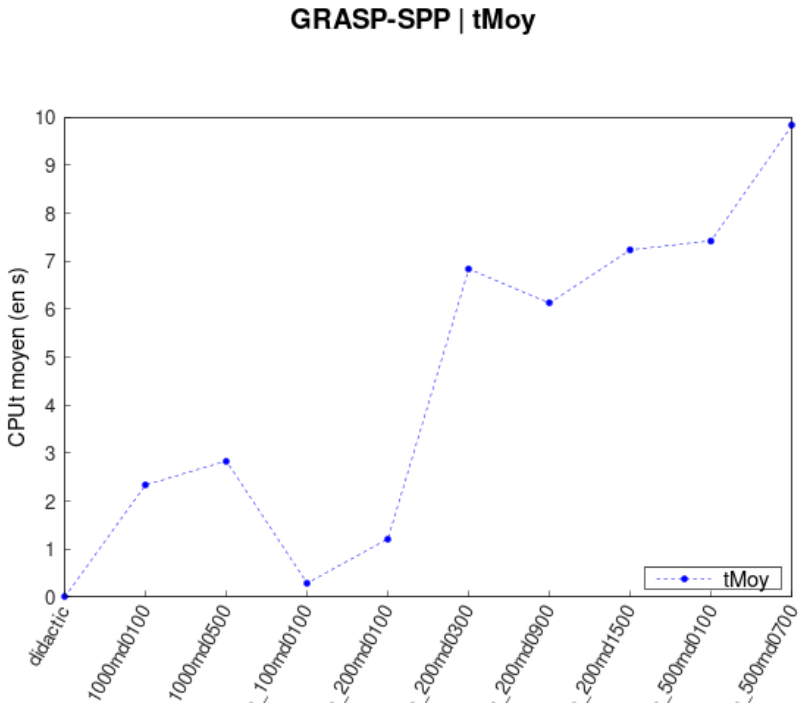

FIGURE 41 – Temps CPUt moyen pour chaque instance