Université de Nantes

Département d'Informatique

Livrable de l'Exercice d'Implémentation 3

Battle des métaheuristiques

Étudiants : Sèdjro Amos GANDONOU
Florias TOKOTCHI

Enseignant : Pr. Xavier Gandibleux

Formation : Master 1 – ATAL (Apprentissage et Traitement Automatique de la Langue)

Année académique : 2025–2025

Nantes, le 23 novembre 2025

Présentation succincte des choix de mise en œuvre de la métaheuristique concurrente à GRASP appliquée au SPP

1 Choix de l'algorithme

Nous avons retenu l'Algorithme Génétique (AG) pour résoudre le Set Packing Problem (SPP). Ce choix est motivé par plusieurs raisons :

- il s'adapte naturellement aux problèmes combinatoires NP-difficiles ;
- il propose un bon compromis entre qualité de solution et temps de calcul ;
- il intègre une population de solutions, favorisant une exploration large de l'espace de recherche ;
- il se prête facilement à l'intégration de mécanismes de réparation, indispensables pour garantir la faisabilité des solutions.

Dans la suite, nous illustrons le fonctionnement de l'algorithme génétique sur un exemple didactique.

2 Instance didactique retenue

Nous considérons l'exemple suivant identique aux exercices d'implémentation précédents :

$$C = \begin{matrix} 6 & 5 & 4 & 7 & 3 \end{matrix}$$

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

où :

- C_j est le gain associé à la colonne j,
- A est la matrice définissant les contraintes du SPP,
- une solution est un vecteur binaire x avec $x_j = 1$ si la colonne j est sélectionnée.

3 Principe général de l'algorithme génétique

Algorithm 1: Algorithme génétique retenu

Data: Matrice A, gains C, taille de population P, nombre de générations $Ngen$
Result: Meilleure solution trouvée
Initialiser une population P de solutions aléatoires ; **for** $t = 1$ **to** $Ngen$ **do**
    Calculer la fitness de chaque individu ;
    Sélectionner les meilleurs individus comme parents ;
    Générer de nouveaux individus "enfants" par *croisement* ;
    Réparer les individus enfants éventuellement non faisables ;
    Appliquer une *mutation* aléatoire sur certains gènes ;
    Réparer les nouveaux individus enfants si non faisables
    Mettre à jour la population par remplacement élitiste ;
return Retourner le meilleur individu de la population ;

4 Choix de mise en œuvre

4.1 Fonction d'évaluation

Nous maximisons :

$$\text{fitness}(x) = \sum_{j|x_j=1} C_j$$

4.2 Vérification de faisabilité

Une solution x est faisable si :

$$Ax \leq 1$$

(colonne par colonne). Dès qu'une contrainte est violée, la solution est envoyée vers la procédure de réparation.

4.3 Méthode de réparation

La réparation utilisée est simple :

1. identifier les contraintes violées ;
2. pour chacune, localiser les colonnes responsables $(x_j = 1)$;
3. en retirer une au hasard $(x_j \leftarrow 0)$;
4. répéter jusqu'à faisabilité.

4.4 Mécanisme élitiste

À chaque génération :
— les nouveaux individus sont triés par fitness ;
— les meilleurs remplacent seulement les moins bons de la population courante.
Ce mécanisme garantit que la meilleure solution trouvée ne peut jamais être perdue.

5 Exécution sur l'exemple didactique

5.1 Population initiale

Une population initiale aléatoire peut être par exemple :

$$x^{(1)} = [1;0;1;0;0], \quad x^{(2)} = [0;1;0;1;0], \quad x^{(3)} = [1;1;0;0;1]$$

5.2 Évaluation initiale

$$\text{fitness}(x^{(1)}) = 6 + 4 = 10$$
$$\text{fitness}(x^{(2)}) = 5 + 7 = 12$$
$$\text{fitness}(x^{(3)}) = 6 + 5 = 14$$

On identifie les meilleurs individus pour la reproduction.

5.3 Croisement

À partir de $x^{(2)}$ et $x^{(3)}$, deux enfants grâce au point de coupure aléatoire en position 3 :

$$e^{(1)} = [0;1;0;0;1], \quad e^{(2)} = [1;1;0;1;0]$$

tels que :

$$\text{fitness}(e^{(1)}) = 6 + 4 = 8$$
$$\text{fitness}(e^{(2)}) = 5 + 7 = 18$$

Ces deux enfants sont dans le domaine des faisables et n'ont donc pas besoin de réparation c'est-à-dire $Ae^{(1)} \leq 1$ et $Ae^{(2)} \leq 1$

5.4 Mutation

Avec une probabilité p_m, un bit peut être inversé. Supposons que l'enfant $e^{(1)}$ subit une mutation en position 1. Les deux enfants sont maintenant :

$$e'^{(1)} = [1;1;0;0;1], \quad e'^{(2)} = [1;1;0;1;0]$$

tels que :

$$\text{fitness}(e'^{(1)}) = 14$$
$$\text{fitness}(e'^{(2)}) = 18$$

La contrainte $Ae'^{(1)} \leq 1$ étant violée, $e'^{(1)}$ doit être réparée.

5.5 Réparation

5.5.1 Méthode Swap

Une première étape de la réparation de changer deux bits de façon aléatoire.

$$Swap(e'^{(1)}, 1,\ 4) = [0; 1; 0; 1; 1]$$

d'où $e'^{(1)} = [1; 1; 0; 0; 1]$ mais reste toujours non faisable.

5.5.2 Approche aggressive

Dans ce cas, la réparation supprime une colonne active dans la contrainte concernée jusqu'à satisfaction de :

$$Ae'^{(1)} \leq 1$$

D'où nos enfants post-mutation :

$$e_f^{(1)} = [1; 0; 0; 1; 0], \quad e_f^{(2)} = [1; 1; 0; 1; 0]$$

5.6 Remplacement élitiste

Les nouveaux individu sont introduit dans la population en remplaçant les mauvais individus. D'où :

$$x^{(1)} = [1; 0; 0; 1; 0], \quad x^{(2)} = [1; 1; 0; 1; 0], \quad x^{(3)} = [1; 1; 0; 0; 1]$$

tels que :

$$\text{fitness}(x^{(1)}) = 13$$
$$\text{fitness}(x^{(2)}) = 18$$

$$\text{fitness}(x^{(2)}) = 14$$

5.7 Convergence vers la solution optimale

Après plusieurs générations, l'AG converge dans le meilleur des cas vers :

$$x^* = [1; 1; 0; 1; 0]$$

qui donne :

$$Z^* = 18$$

Ce résultat constitue la solution optimale pour l'instance test.

6 Conclusion

Cet exemple montre comment l'algorithme génétique, via sélection, croisement, mutation, réparation et remplacement élitiste, permet de converger progressivement vers une solution de haute qualité tout en respectant les contraintes du Set Packing Problem.

Expérimentation numérique comparative GRASP vs métaheuristique concurrente

7 Environnement d'implémentation

L'implémentation des heuristiques sur les dix (10) instances a été réalisé sur un ordinateur HP dont voici les caractéristiques :

— **Processeur :** Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.59 GHz
— **Mémoire RAM installée :** 8,00 Go (7,89 Go utilisable)
— **Carte graphique :** Intel(R) HD Graphics 520 (128 MB)
— **Type du système :** Système d'exploitation 64 bits, processeur x64
— **Édition :** Windows 10 Famille
— **Version :** 22H2

L'Environnement de développement intégré utilisé est Visual Studio Code version **1.105.1** pour l'exécution des différents programmes julia dans sa version **1.11.7**.

8 Réglage des paramètress

Les paramètres de l'expérimentation sont les mêmes pour les deux heuristiques à savoir :

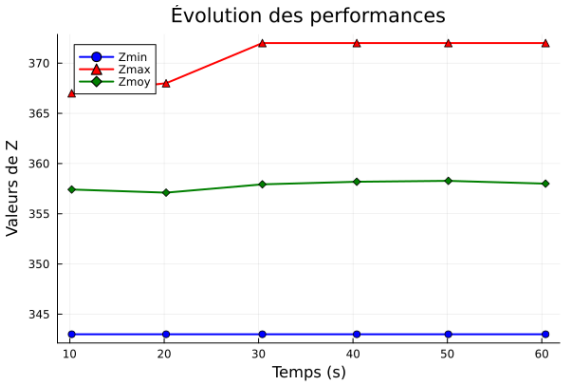
1. Une durée totale de recherche de solution de $\Delta T = 60s$
2. Un résumé de la recherche à chaque pas de $\delta t = 10s$

Pour l'AG, nous considérons :
— une population initiale constituée de façon aléatoire
— une taille 100 pour la population
— une probabilité de crossing-over **cop** = **0.8**
— une probabilité de mutation **mup** = **0.1**

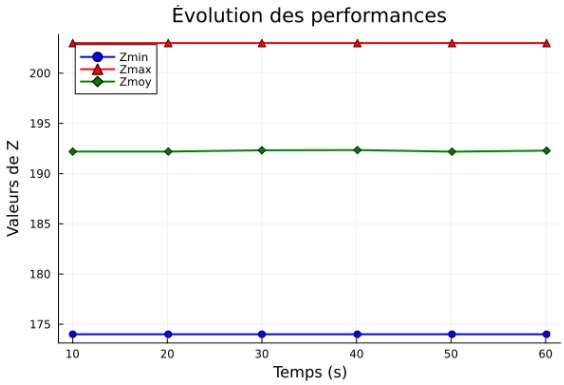
9 Résultats en graphiques

9.1 Résultats GRASP

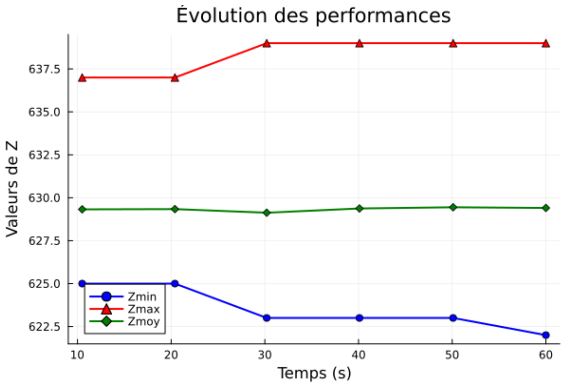
Les figures 1 et 2 résume les résultats de GRASP selon $\hat{z}_{min}$, $\hat{z}_{max}$, $\hat{z}_{moy}$ mésurés à intervalles réguliers $\delta t = 10s$



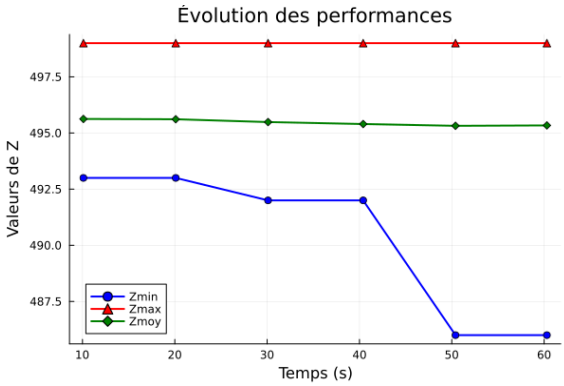
(a) pb_100rnd0100



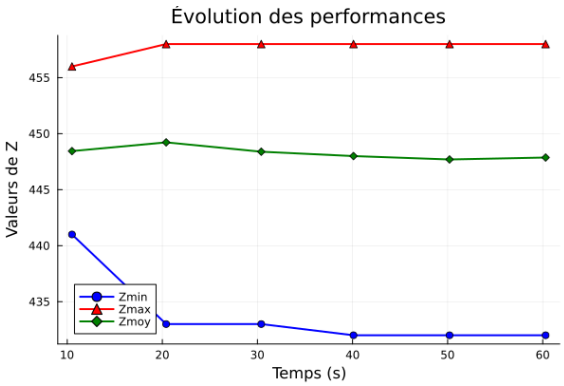
(b) pb_100rnd0300



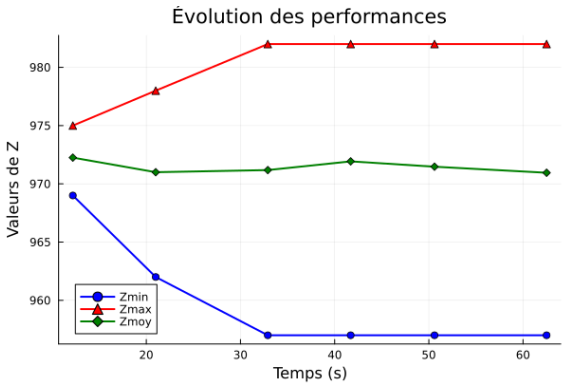
(c) pb_100rnd0500



(d) pb_100rnd0700

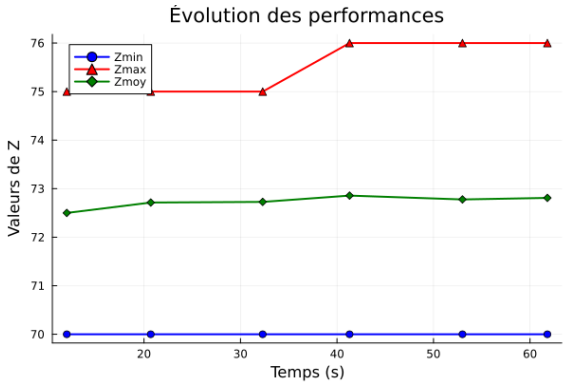


(e) pb_100rnd0900

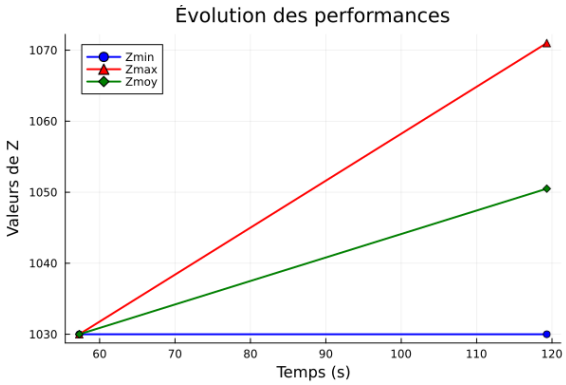


(f) pb_200rnd0700

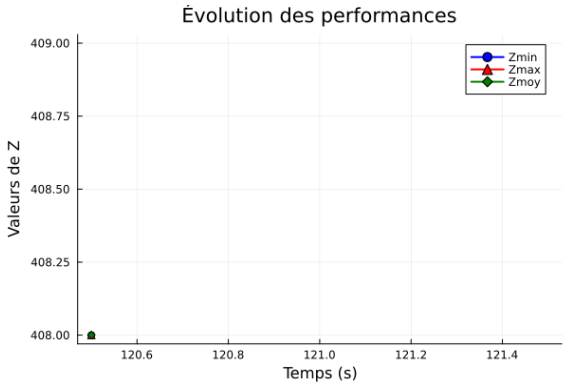
FIGURE 1 – Évolution de la qualité des solutions GRASP ($Z_{\min}$, $Z_{\max}$ et Z_{moy})



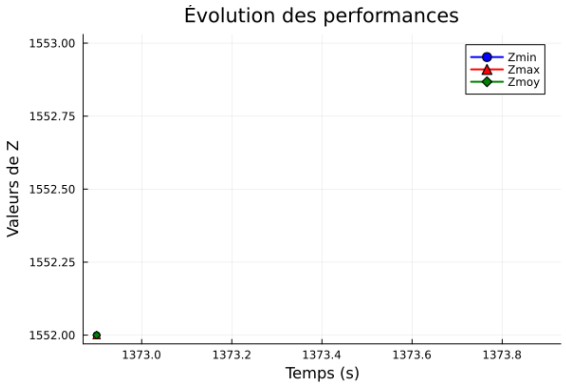
(a) pb_200rnd0800



(b) pb_500rnd1500



(c) pb_1000rnd0300

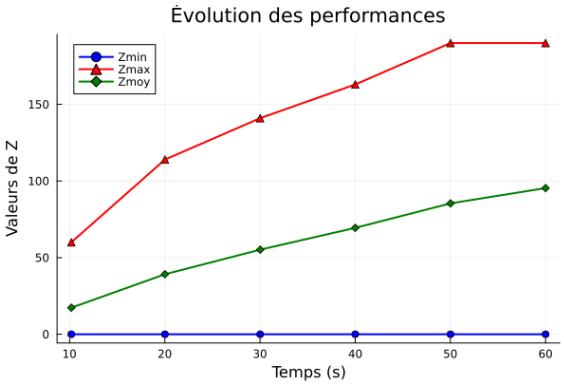


(d) pb_2000rnd0700

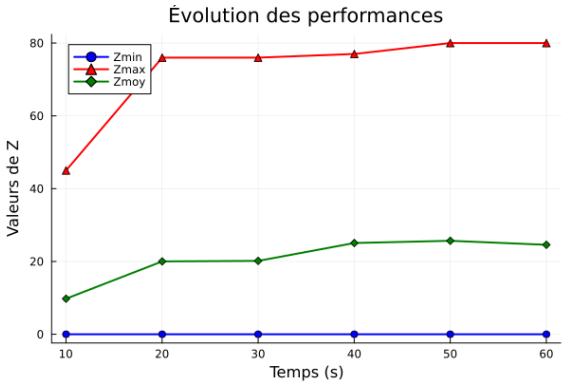
FIGURE 2 – Évolution de la qualité des solutions GRASP (suite)

9.2 Résultats AG

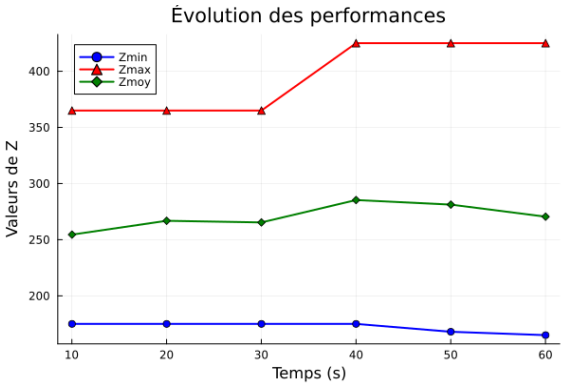
Les figures 3 et 4 résume les résultats de GRASP selon $\hat{z}_{min}$, $\hat{z}_{max}$, $\hat{z}_{moy}$ mésurés à intervalles réguliers $\delta t = 10s$



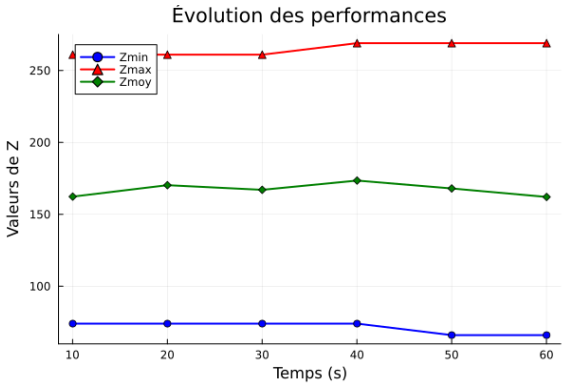
(a) pb_100rnd0100



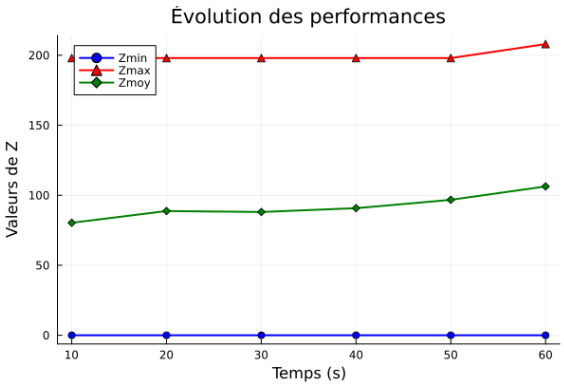
(b) pb_100rnd0300



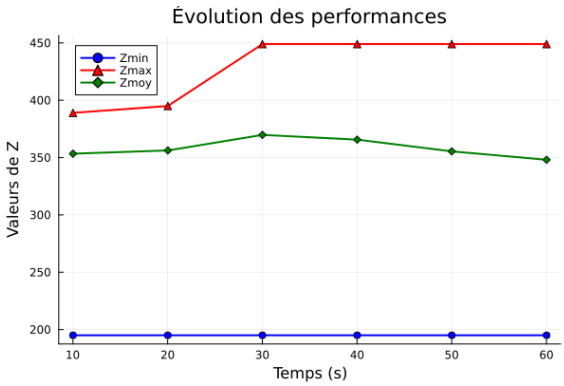
(c) pb_100rnd0500



(d) pb_100rnd0700

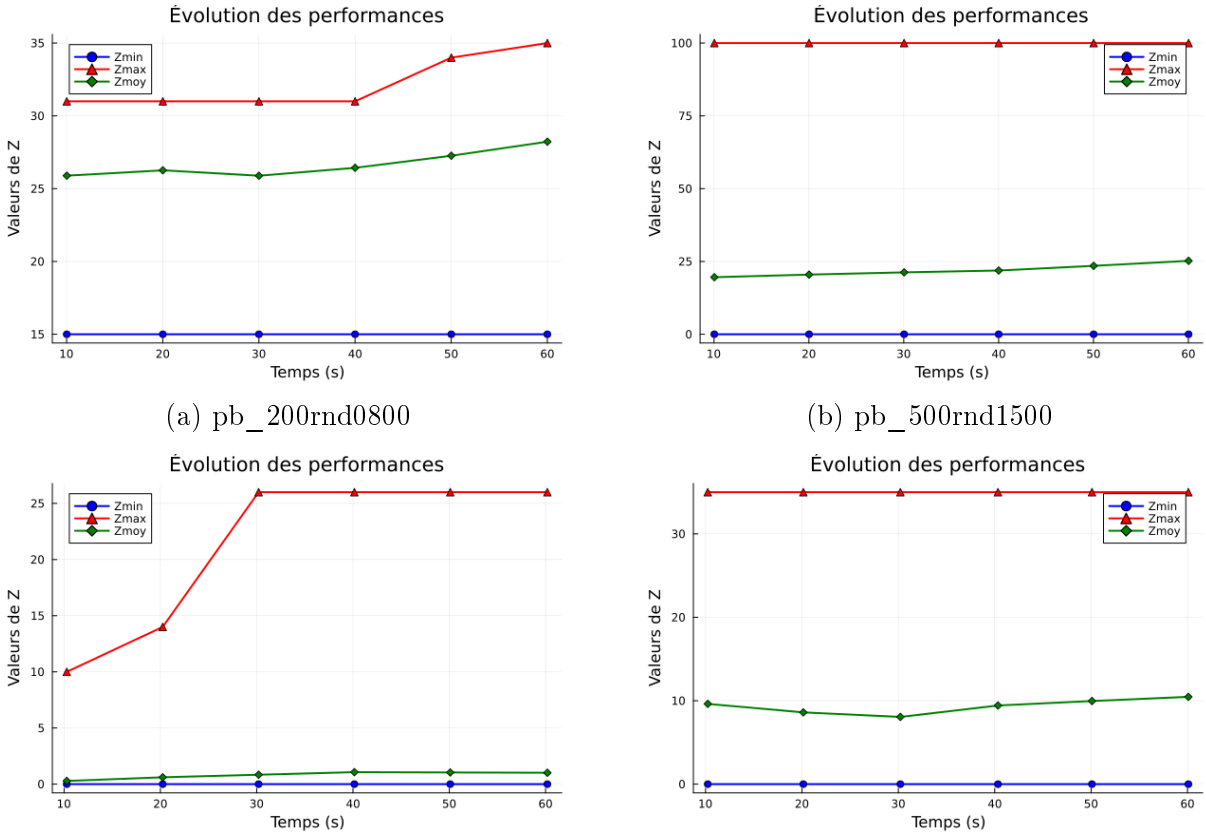


(e) pb_100rnd0900



(f) pb_200rnd0700

FIGURE 3 – Évolution de la qualité des solutions AG ($Z_{\min}$, $Z_{\max}$ et Z_{moy})



(a) pb_200rnd0800

(b) pb_500rnd1500

(c) pb_1000rnd0300

(d) pb_2000rnd0700

FIGURE 4 – Évolution de la qualité des solutions AG (suite)

9.3 Influence de α dans GRASP

Tel que rapporter dans l'exercice d'implémentation EI2, les solutions de GRASP deviennent meilleures à mesure que α tend vers 1 (voir tableau 1).

TABLEAU 1 – Comparaison des résultats de GRASP selon α pour 10 instances.

Instance	Z				t (s)			
	0.2	0.4	0.6	0.8	0.2	0.4	0.6	0.8
pb_100rnd0100	359	368	372	370	60.5	60.4	60.3	60.3
pb_100rnd0300	194	203	203	203	60.1	60.1	60.1	60.4
pb_100rnd0500	638	639	639	631	60.1	60.4	60.1	60.5
pb_100rnd0700	503	503	499	495	60.1	60.1	60.3	60.0
pb_100rnd0900	440	459	463	453	60.3	60.1	60.0	60.4
pb_200rnd0700	961	985	985	981	62.5	62.1	60.7	61.5
pb_200rnd0800	77	75	77	79	60.2	61.6	60.0	60.7
pb_500rnd1500	887	1023	1048	1075	100.1	61.2	62.1	65.3
pb_1000rnd0300	381	382	494	526	171.4	203.9	226.8	239.5
pb_2000rnd0700	1188	1526	1543	1589	1060.3	1640.9	2241.8	1723.8

10 Résumé expérimentation GRASP vs AG

Tel montré par le tableau 2 l'heuristique GRASP écrase l'Algorithme Génétique.

TABLEAU 2 – Comparaison des performances entre l'Algorithme Génétique et GRASP

Instance	**GA** Z	**GRASP** Z	ΔZ
`pb_100rnd0100`	190	372	182
`pb_100rnd0300`	80	203	123
`pb_100rnd0500`	425	639	214
`pb_100rnd0700`	269	499	230
`pb_100rnd0900`	208	458	250
`pb_200rnd0700`	449	982	533
`pb_200rnd0800`	35	76	41
`pb_500rnd1500`	100	1070	970
`pb_1000rnd0300`	26	408	382
`pb_2000rnd0700`	35	1552	1517

Discussion

11 Quelques conclusions

De nos résultats il ressort que la performance de l'AG est rapide en terme d'exécution, toutefois il dépend fortement des paramètres en entrée à savoir la qualité de la population initiale, les probabilités de crossing-over et de mutation et la méthode de constitution d'une nouvelle génération. Nous retenons que :

- une population initiale constituée de beaucoup d'individus non faisables, rend nécessaire de faire beaucoup de réparations ;
- une population de petit taille ne fait pas avancer rapidement les performances ;
- le crossing-over et la mutation ne garantissent pas d'avoir des enfants faisables ;
- l'AG n'explore pas assez l'espace des possibilités car uniquement deux parents sont choisis (avec la forte probabilité de choisir à chaque fois les mêmes parents si l'on choisit les deux meilleurs sur le critère de la fitness), ceux qui aboutit à peut d'amélioration de la qualité de la population de la génération suivante lorsqu'elle est de grande taille ;
- l'AG compte sur le hasard pour trouver des solutions optimales or l'espace explorable peut contenir un nombre exponentiel de possibilité non admissibles ;
- les générations successives on plus de probabilité d'être identiques avec une stratégie élitiste si les enfants obtenus non pas des fitness assez élevés ;
- etc.

L'heuristique GRASP quant à lui, est plus stable, évolue plus intélligemment d'une itération à une autre en améliorant ses résultats. En effet, GRASP construit des solutions admissibles qu'elle améliore par recherche locale simple ou profonde. On ne perd donc pas les bonnes solutions et on ne les détériore pas.

12 Quelques recommandations

Nous pouvons améliorer l'algorithme génétique (AG) de plusieurs manières :

1. Opter une population initiale de grande taille
2. Construire une population initiale de bonne qualité avec une méthode choisie
3. Sélectionner les parents par roulette aléatoire permettrait de mieux explorer et de diversifier
4. Faire des améliorations par recherche locale boosterait la recherche (**Memetic Algorithm**).

L'AG du fait de sa rapidité d'exécution peut être utile pour trouver des solutions qui serviraient de point d'entrée pour des explorations plus averties.