Livrable de l'exercice d'implémentation 1 : Heuristiques de construction et d'amélioration gloutonnes

Formulation du SPP

Présenter la formulation du SPP.
La formulation mathématique est la suivante :

Ensembles et Indices :

- I = L'ensemble des lignes (contraintes).
- J = L'ensemble des colonnes (variables).

Paramètres (Données) :

- c_j : Le coût de la colonne $j \in J$.
- $A = (a_{ij})$: Une matrice binaire où $a_{ij} = 1$ si la ligne i est couverte par la colonne j, et 0 sinon.

Variables de Décision :

- x_j : Une variable binaire qui vaut 1 si la colonne j est sélectionnée dans la solution, et 0 sinon.

L'objectif est de maximiser le coût total des colonnes sélectionnées
Sous condition qu'on respecte la contrainte que chaque ligne est couverte au maximum une fois

Exemple pratique

Emploi du temps — éviter les conflits : chaque colonne représente une proposition (cours) et chaque ligne un créneau horaire. La contrainte empêche qu'un même créneau soit attribué à deux cours, profs ou groupes d'élèves simultanément. On sélectionne ainsi un sous-ensemble de propositions non conflictuelles afin de maximiser la fréquentation ou l'utilité associée.

Modélisation JuMP (ou GMP) du SPP

Notre modélisation JuMP traduit directement la formulation mathématique. La fonction `setSPP` prend la matrice des coûts `C` et la matrice des contraintes `A` en entrée et crée un modèle JuMP.

Explication des lignes clés de la fonction :

- `@variable(spp, x[1:n], Bin)` : Déclare les n variables de décision x_j et indique qu'elles sont binaires ($x_j \in \{0, 1\}$).
- `@objective(spp, Max, dot(C, x))` : Définit la fonction objectif. On cherche à `Max`imiser le produit scalaire entre les coûts `C` et les variables `x` (ce qui équivaut à $\max \sum c_j x_j$).
- `@constraint(spp , ...)` : Crée une contrainte pour chaque ligne i (de 1 à m), garantissant que la somme des $a_{ij}x_j$ sur cette ligne ne dépasse pas 1 ($\sum a_{ij}x_j \leq 1, \forall i$).

Instances numériques de SPP

Instance
pb_100rnd0100.dat
pb_200rnd0100.dat
pb_500rnd0300.dat
pb_500rnd1500.dat
pb_500rnd1700.dat
pb_200rnd0400.dat
pb_200rnd0700.dat
pb_1000rnd0100.dat
pb_1000rnd0300.dat
didactic.dat

Heuristique de construction appliquée au SPP

Présenter l'algorithme mis en œuvre. Illustrer sur un exemple didactique.

Mon heuristique de construction gloutonne (construction_gloutonne)

L'heuristique procède comme suit :

1. Elle calcule d'abord l'utilité de chaque élément (colonne) avec sa fonction d'utilité :

$$u = \frac{\text{coût}}{\text{nombre de contraintes couvertes}}$$

2. Ensuite, elle trie les éléments par ordre décroissant d'utilité.
3. Puis, elle parcourt la liste des éléments triés et ajoute chaque élément à la solution si cela ne crée pas de conflit avec les éléments déjà sélectionnés (c'est-à-dire, si aucune des contraintes couvertes par cet élément n'est déjà couverte par la solution partielle).

Illustration sur `didactic.dat`

Étape 1 & 2 : Calcul et Tri des Utilités

- Col 6 (Coût=13, Lignes=2) → Utilité = 6.5
- Col 7 (Coût=11, Lignes=2) → Utilité = 5.5
- Col 1 (Coût=10, Lignes=3) → Utilité = 3.33
- Col 4 (Coût=6, Lignes=2) → Utilité = 3.0
- Col 5 (Coût=9, Lignes=4) → Utilité = 2.25
- Col 3 (Coût=8, Lignes=4) → Utilité = 2.0
- Col 9 (Coût=6, Lignes=4) → Utilité = 1.5
- Col 2 (Coût=5, Lignes=4) → Utilité = 1.25
- Col 8 (Coût=4, Lignes=4) → Utilité = 1.0

Étape 3 : Construction
Solution initiale = [], Lignes couvertes = []

- **Évaluation Col 6** (utilité 6.5)...
  - Action: **ACCEPTÉE**. (Pas de conflit)
  - Solution = [6]
  - Lignes couvertes par Col 6: [3, 5]

  - Lignes couvertes (Total) = [3, 5]
- **Évaluation Col 7** (utilité 5.5)...
  - Action: **ACCEPTÉE**. (Pas de conflit)
  - Solution = [6, 7]
  - Lignes couvertes par Col 7: [1, 6]
  - Lignes couvertes (Total) = [1, 3, 5, 6]
- **Évaluation Col 1** (utilité 3.33)...
  - Action: **REJETÉE**. (Conflit sur Ligne 1)
- **Évaluation Col 4** (utilité 3.0)...
  - Action: **ACCEPTÉE**. (Pas de conflit)
  - Solution = [6, 7, 4]
  - Lignes couvertes par Col 4: [4, 7]
  - Lignes couvertes (Total) = [1, 3, 4, 5, 6, 7]
- **Évaluation Col 5** (utilité 2.25)...
  - Action: **REJETÉE**. (Conflit sur Ligne 1)
- **Évaluation Col 3** (utilité 2.0)...
  - Action: **REJETÉE**. (Conflit sur Ligne 1)
- **Évaluation Col 9** (utilité 1.5)...
  - Action: **REJETÉE**. (Conflit sur Ligne 3)
- **Évaluation Col 2** (utilité 1.25)...
  - Action: **REJETÉE**. (Conflit sur Ligne 1)
- **Évaluation Col 8** (utilité 1.0)...
  - Action: **REJETÉE**. (Conflit sur Ligne 1)

Solution gloutonne finale: [6, 7, 4]
Valeur (z_{glouton}): 30

Heuristique d'amélioration appliquée au SPP

Présenter l'algorithme mis en oeuvre. Illustrer sur un exemple didactique.

Mon heuristique d'amélioration (descente_local)

Mon heuristique d'amélioration est une descente locale, `descente_local`, qui fonctionne comme ceci : L'algorithme commence à partir d'une solution initiale (ici, la solution gloutonne).

1. **Génération de voisinage :** À chaque itération, la fonction appelle la fonction de voisinage (`generer_voisinage_1_1`). Celle-ci génère l'ensemble de tous les voisins valides pouvant être atteints en effectuant un échange k-p de type 1-1 (retirer un élément de la solution et en ajouter un qui n'y est pas, tout en respectant les contraintes).
2. **Évaluation et sélection :** On évalue le coût de chaque voisin généré pour trouver le meilleur voisin (celui avec le coût le plus élevé).
   - **Si** le coût de ce meilleur voisin est supérieur au coût de la solution courante, on met à jour la solution courante avec ce meilleur voisin et on répète le processus (retour à l'étape 1).
   - **Sinon**, si aucun voisin n'améliore la solution courante, l'algorithme s'arrête et retourne la solution courante comme optimum local.

Illustration sur `didactic.dat`

Itération de Descente n°1

- Solution courante: [6, 7, 4] (Coût = 30)
- Génération de 0 voisins (1-1) valides...
- **Résultat :** PAS D'AMÉLIORATION. Optimum local atteint.

Analyse de l'illustration Ici, pendant la génération des voisins, aucun voisin valide n'a pu être généré à partir de la solution initiale gloutonne [6, 7, 4]. L'algorithme s'est donc arrêté immédiatement et a retourné la solution initiale comme solution finale. Notre solution était déjà un optimum local pour ce voisinage.

Expérimentation numérique

Présenter l'environnement machine sur lequel les algorithmes vont tourner (référence). Présenter sous forme synthétique (tableau, graphique...) les résultats obtenus pour les 10 instances sélectionnées.
Environnement machine : Les algorithmes ont tourné sur la machine suivante :

- **Système d'exploitation :** Windows 11
- **Processeur :** AMD RYZEN 7 5700X 8-Core
- **Mémoire :** 16Go

Table 1: Résultats des heuristiques gloutonne et locale sur 10 instances SPP.

Instance	$z_{\textbf{glouton}}$	$t_{\textbf{glouton}}$ **(s)**	$z_{\textbf{local}}$	$t_{\textbf{local}}$ **(s)**
`pb_100rnd0100.dat`	342.0	0.000	343.0	0.001
`pb_200rnd0100.dat`	351.0	0.001	365.0	0.008
`pb_500rnd0300.dat`	674.0	0.004	693.0	0.046
`pb_500rnd1500.dat`	1059.0	0.009	1115.0	0.136
`pb_500rnd1700.dat`	150.0	0.004	154.0	0.003
`pb_200rnd0400.dat`	55.0	0.000	55.0	0.004
`pb_200rnd0700.dat`	945.0	0.000	955.0	0.004
`pb_1000rnd0100.dat`	49.0	0.192	49.0	0.001
`pb_1000rnd0300.dat`	507.0	0.023	528.0	0.119
`didactic.dat`	30.0	0.000	30.0	0.000

Discussion

Questions type pour mener votre discussion :

- au regard des temps de résolution requis par le solveur MIP (GLPK) pour obtenir une solution optimale à l'instance considérée, l'usage d'une heuristique se justifie-t-il?

  Oui, l'usage d'une heuristique est totalement justifié. Nos tests montrent que si le solveur GLPK fonctionne parfaitement pour les petites instances comme `didactic.dat`, son temps de résolution augmente énormément quand on augmente le nombre de variables et de contraintes. Le temps requis pour trouver la solution devient beaucoup trop long.L'heuristique, en revanche, fournit une solution super rapidement

- avec pour référence la solution optimale, quelle est la qualité des solutions obtenues avec l'heuristique de construction et l'heuristique d'amélioration?

Sur le plan des temps de résolution, quel est le rapport entre le temps consommé par le solveur MIP et vos heuristiques?

Sur le plan des temps de résolution, le rapport est en faveur de notre heuristiques. Pour l'instance `pb_1000rnd0300.dat`, notre heuristique complète ($t_{\text{glouton}} + t_{\text{local}}$) prend 0.142 secondes. Le solveur lui, prendrait plusieurs minutes. Le rapport est donc très largement en faveur de l'heuristique.

- Le recours aux (méta)heuristiques apparaît-il prometteur ?
  Entrevoyez-vous des pistes d'amélioration à apporter à vos heuristiques?

  Oui, le recours aux métaheuristiques paraît prometteur. En partant déjà d'une solution gloutonne qui est très rapide à obtenir, on devrait pouvoir améliorer nos résultats grâce à des algorithmes comme GRASP et ses amélioration.

Livrable de l'exercice d'implémentation 2 : Métaheuristique GRASP, ReactiveGRASP et extensions

Présentation succincte de GRASP appliqué sur le SPP

Présenter l'algorithme mis en oeuvre. Illustrer sur un exemple didactique (poursuivre avec l'exemple pris en DM1). Présenter vos choix de mise en oeuvre.

Le grasp que j'ai réalisé fonctionne de la manière suivante :

1. **Construction gloutonne randomisée :** On construit une solution initiale en sélectionnant itérativement des éléments à ajouter à la solution. À chaque étape, on crée une RCL. Un élément qui respecte les contraintes et qui a une valeur d'utilité supérieur au Ulimit calculé comme ceci : $U_{limit} = u_{min} + \alpha \cdot (u_{max} - u_{min})$ Ensuite, on séléctionne aléatoirement un élément dans la RCL à ajouter à la solution.
2. **Amélioration locale :** Une fois la solution initiale construite, on applique notre descente locale avec un voisinage 1-1 pour améliorer la solution.
3. **Itérations :** Les étapes de construction et d'amélioration sont répétées un certain nombre de fois (ou jusqu'à une condition d'arrêt), et la meilleure solution trouvée au cours de ces itérations est retenue comme solution finale.

Présentation succincte de ReactiveGRASP appliqué sur le SPP

Présenter l'algorithme mis en oeuvre. Illustrer sur un exemple didactique (poursuivre avec l'exemple pris en DM1). Présenter vos choix de mise en oeuvre.

Mon ReactiveGRASP est une extension de GRASP classique qui ajuste dinamiquement les α au cours de l'exécution.

1. **Initialisation :** On définit un ensemble de valeurs de α à tester (dans notre cas : $\{0.1, 0.3, 0.5, 0.7, 0.9\}$). Chaque α se voit attribuer une probabilité initiale uniforme ($p_i = 0.2$ pour 5 valeurs).
2. **Sélection adaptative :** À chaque itération, on sélectionne un α selon une distribution de probabilités. Plus un α a produit de bonnes solutions, plus sa probabilité d'être sélectionné augmente.
3. **L'amélioration locale et la construction GRASP :** Garde le même fonctionnement que GRASP classique.
4. **Mise à jour des probabilités :** Tous les N itérations (dans notre cas, $N = 20$), on recalcule les probabilités selon la formule :

   $$q_i = \left(\frac{\bar{z}_i}{z^*}\right)^k$$

   où $\bar{z}_i$ est la qualité moyenne des solutions obtenues avec α_i, z^* est la meilleure solution globale trouvée, et k est un paramètre d'intensification (ici $k = 5$).

   Les probabilités sont ensuite normalisées : $p_i = \frac{q_i}{\sum_j q_j}$.

Expérimentation numérique de GRASP

Présenter le protocole d'expérimentation (environnement matériel; budget de calcul; condition(s) d'arrêt; réglage des paramètres).

1 Protocole d'expérimentation

1.1 Environnement matériel

Les algorithmes ont tourné sur la machine suivante :

- **Système d'exploitation :** Windows 11
- **Processeur :** AMD RYZEN 7 5700X 8-Core
- **Mémoire :** 16Go

1.2 Budget de calcul et conditions d'arrêt

L'expérimentation s'est déroulée en deux phases :

Premier temps Un test a été réalisé avec une condition d'arrêt basée sur le nombre d'itérations.

- **Condition d'arrêt :** 200 itérations.
- **Répétitions :** 3 fois par instance avec un alpha différent à chaque fois.
- **Instances :** 10 instances différentes.

Second temps Un test a été réalisé avec une condition d'arrêt basée sur le temps d'exécution.

- **Condition d'arrêt :** 60 secondes.
- **Répétitions :** 3 fois par instance avec un alpha différent à chaque fois.
- **Instances :** 10 instances différentes.

Rapporter graphiquement vos résultats selon $\hat{z}_{min}$, $\hat{z}_{max}$, $\hat{z}_{moy}$ mesurés à intervalles réguliers (exemple de pas de 10 secondes).

Les résultats sont obtenus avec 3 runs avec chacun 3 alpha différents et 200 itérations par instances ($\alpha = \{0.1, 0.5, 0.9\}$).

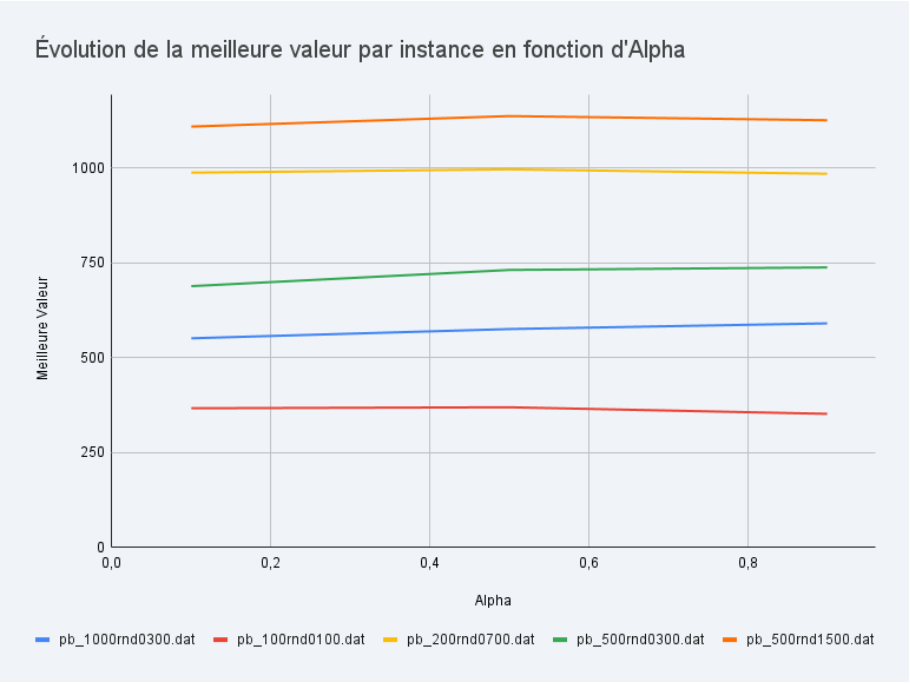

Figure 1: Z_mean.

Rapporter l'étude de l'influence du paramètre α.

Dans notre expérimentation, nous avons testé trois valeurs de α : 0.1, 0.5 et 0.9. Voici les observations principales :
Présenter sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées.

Table 1: Résultats finaux pour les 10 instances (200 itérations)

Instance	α	**Val. best**	**CPU (s)**
	0.1	368	0.66
`pb_100rnd0100.dat`	0.5	370	0.40
	0.9	352	0.40
	0.1	403	3.45
`pb_200rnd0100.dat`	0.5	408	2.38
	0.9	386	2.20
	0.1	718	53.13
`pb_500rnd0300.dat`	0.5	715	22.86
	0.9	736	15.19
	0.1	1093	78.01
`pb_500rnd1500.dat`	0.5	1135	41.52
	0.9	1122	30.15
	0.1	166	1.92
`pb_500rnd1700.dat`	0.5	170	1.47
	0.9	165	1.14
	0.1	59	1.81
`pb_200rnd0400.dat`	0.5	59	1.98
	0.9	57	2.04
	0.1	992	7.13
`pb_200rnd0700.dat`	0.5	992	2.17
	0.9	981	1.71
	0.1	57	3.01
`pb_1000rnd0100.dat`	0.5	67	3.14
	0.9	57	2.59
	0.1	554	128.72
`pb_1000rnd0300.dat`	0.5	590	64.70
	0.9	594	45.91
	0.1	30	0.00
`dat/didactic.dat`	0.5	30	0.00
	0.9	30	0.00

Table 2: Résultats finaux pour les 10 instances (60 secondes)

Instance	α	**Val. best**	**CPU (s)**	**Itérations**
	0.1	370	60.0	18640
`pb_100rnd0100.dat`	0.5	372	60.0	32237
	0.9	352	60.0	28761
	0.1	414	60.01	3724
`pb_200rnd0100.dat`	0.5	415	60.01	7362
	0.9	386	60.0	7483
	0.1	697	60.06	224
`pb_500rnd0300.dat`	0.5	724	60.08	481
	0.9	739	60.02	786
	0.1	1080	60.0	163
`pb_500rnd1500.dat`	0.5	1145	60.11	285
	0.9	1134	60.06	414
	0.1	180	60.0	8089
`pb_500rnd1700.dat`	0.5	192	60.0	11527
	0.9	165	60.0	11727
	0.1	61	60.0	8472
`pb_200rnd0400.dat`	0.5	60	60.0	6516
	0.9	57	60.0	6754
	0.1	991	60.03	2168
`pb_200rnd0700.dat`	0.5	1003	60.0	6888
	0.9	992	60.0	8420
	0.1	67	60.0	5409
`pb_1000rnd0100.dat`	0.5	67	60.01	5217
	0.9	57	60.01	5563
	0.1	546	60.04	102
`pb_1000rnd0300.dat`	0.5	573	60.37	228
	0.9	587	60.13	363
	0.1	30	60.0	18693182
`dat/didactic.dat`	0.5	30	60.0	29640773
	0.9	30	60.0	29826432

Expérimentation numérique de ReactiveGRASP

Présenter le protocole d'expérimentation (env. matériel; budget de calcul; condition(s) d'arrêt).

2 Protocole d'expérimentation

2.1 Environnement matériel

Les algorithmes ont tourné sur la machine suivante :

- **Système d'exploitation :** Windows 11
- **Processeur :** AMD RYZEN 7 5700X 8-Core
- **Mémoire :** 16Go

2.2 Budget de calcul et conditions d'arrêt

L'expérimentation s'est déroulée en deux phases :

Premier temps Un test a été réalisé avec une condition d'arrêt basée sur le nombre d'itérations.

- **Condition d'arrêt :** 200 itérations.
- **Répétitions :** 3 fois par instance.
- **Instances :** 10 instances différentes.

Second temps Un test a été réalisé avec une condition d'arrêt basée sur le temps d'exécution.

- **Condition d'arrêt :** 60 secondes.
- **Répétitions :** 3 fois par instance.
- **Instances :** 10 instances différentes.

Rapporter graphiquement vos résultats selon $\hat{z}_{min}$, $\hat{z}_{max}$, $\hat{z}_{moy}$ mesurés à intervalles réguliers (exemple de pas de 10 secondes).

Les résultats sont obtenus avec 3 runs et les paramètre "classique" défini précédemment avec 200 itérations par instances.

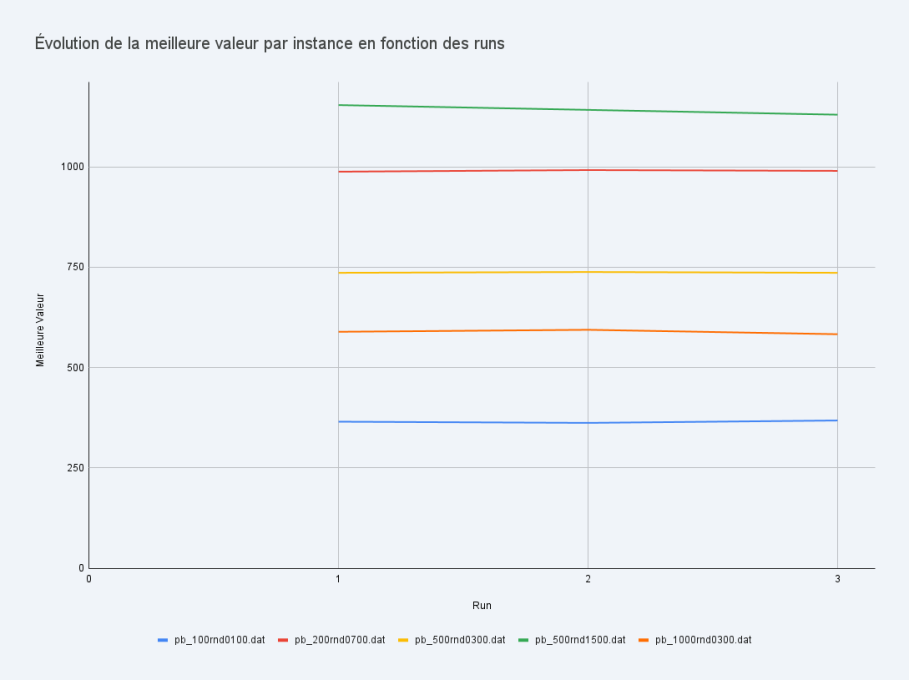

Figure 2: Z_mean.

Rapporter l'apprentissage du paramètre α réalisé par ReactiveGRASP, les valeurs saillantes établies.

Les valeurs de α ont été ajustées dynamiquement par ReactiveGRASP en fonction des performances observées. Voici un résumé des tendances observées :

- Les valeurs de α entre 0.1 et 0.3 ne donnent dans mon expérimentation jamais la meilleur solution.
- Les valeurs de α entre 0.5 et 0.7 peuvent être efficaces dans certains cas, dans nos données elles le sont surtout sur des instances de petite taille.
- Les α de 0.9 sont dans notre expérimentation les plus performantes sur la majorité des instances et sont sans concurrence sur les grandes instances.

Présenter sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées.

Table 3: Résultats finaux pour les 10 instances (200 itérations)

Instance	**Run**	**Val. Best**	**CPU (s)**	**α-Best**	**Val. Best Known**
	1	368	0.58	0.7	372*
pb_100rnd0100.dat	2	366	0.44	0.9	372*
	3	368	0.40	0.7	372*
	1	399	1.84	0.7	416*
pb_200rnd0100.dat	2	403	1.98	0.5	416*
	3	406	2.04	0.5	416*
	1	739	22.41	0.9	776
pb_500rnd0300.dat	2	736	23.58	0.9	776
	3	736	21.47	0.9	776
	1	1129	35.66	0.9	1196
pb_500rnd1500.dat	2	1136	32.75	0.9	1196
	3	1130	34.60	0.9	1196
	1	167	1.23	0.9	192
pb_500rnd1700.dat	2	172	1.01	0.9	192
	3	173	1.03	0.9	192
	1	59	1.43	0.7	64*
pb_200rnd0400.dat	2	58	1.46	0.7	64*
	3	60	1.44	0.7	64*
	1	994	2.36	0.5	1004*
pb_200rnd0700.dat	2	988	2.27	0.5	1004*
	3	992	2.44	0.5	1004*
	1	56	1.94	0.9	67*
pb_1000rnd0100.dat	2	59	1.93	0.9	67*
	3	57	1.97	0.9	67*
	1	584	39.12	0.9	661
pb_1000rnd0300.dat	2	583	52.87	0.9	661
	3	603	52.80	0.9	661
	1	30	0.00	0.5	30
didactic.dat	2	30	0.00	0.5	30
	3	30	0.00	0.5	30

Table 4: Résultats de l'expérimentation avec une condition d'arrêt de 60 secondes.

Instance	Run	Val. Best	CPU (s)	Itér.	α-Best	Val. Best Known
	1	372	60.00	27385	0.7	372*
pb_100rnd0100.dat	2	372	60.00	25738	0.7	372*
	3	372	60.00	25736	0.7	372*
	1	414	60.00	5368	0.5	416*
pb_200rnd0100.dat	2	415	60.01	5720	0.5	416*
	3	414	60.02	5425	0.5	416*
	1	739	60.15	499	0.9	776
pb_500rnd0300.dat	2	739	60.15	456	0.9	776
	3	738	60.03	483	0.9	776
	1	1138	60.18	305	0.9	1196
pb_500rnd1500.dat	2	1139	60.26	308	0.9	1196
	3	1136	60.18	293	0.9	1196
	1	188	60.00	9333	0.9	192
pb_500rnd1700.dat	2	189	60.00	10278	0.9	192
	3	186	60.00	9451	0.9	192
	1	61	60.00	5973	0.7	64*
pb_200rnd0400.dat	2	61	60.01	6998	0.7	64*
	3	60	60.01	6284	0.9	64*
	1	1001	60.02	4634	0.5	1004*
pb_200rnd0700.dat	2	997	60.03	5102	0.5	1004*
	3	1001	60.02	4541	0.5	1004*
	1	67	60.00	5429	0.9	67*
pb_1000rnd0100.dat	2	67	60.00	5397	0.9	67*
	3	67	60.01	5359	0.9	67*
	1	590	60.02	224	0.9	661
pb_1000rnd0300.dat	2	594	60.14	244	0.9	661
	3	592	60.07	233	0.9	661
	1	30	60.00	31001331	0.5	30
didactic.dat	2	30	60.00	31299926	0.5	30
	3	30	60.00	30727593	0.5	30

Eléments de contribution au bonus

Présenter vos contributions aux aspects proposés en bonus.

Discussion

Tirer des conclusions en comparant les résultats collectés avec vos deux variantes de métaheuristiques.

En comparant les résultats de GRASP et ReactiveGRASP j'ai pu constater que ReactiveGRASP offre globalement de meilleurs résultats. J'ai aussi remarqué sur le paramétrage que les résultats obtenus avec une limite de temps sont meilleurs que ceux obtenus par un nombre d'itération fixe, ce qui n'est pas réellement surprenant car il réalise souvent plus d'itérations que celle fixée dans ma première expérimentation surtout quand les instances sont petites. Pour finir j'ai pu constater que les valeurs de α les plus élevées sont celles qui donnent les meilleurs résultats dans la majorité des cas.

Je recommande l'utilisation de ReactiveGRASP. Son implémentation est facile a mettre en place a partir du GRASP classique et produit des résultats meilleurs ou au moins équivalents dans tous les cas. L'auto-ajustement du paramètre α est particulièrement intéressant, car il offre un diagnostic sur l'efficacité de notre heuristique de construction. En effet, le fait que les meilleurs résultats soient obtenus avec un α petit ou grand indique si notre fonction gloutonne est performante ou si elle est souvent piégée dans des optima locaux, ce qui a été notre cas.

Livrable de l'exercice d'implémentation 3 : Battle of metaheuristics

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

Présenter l'algorithme mis en oeuvre. Illustrer sur un exemple didactique (poursuivre avec l'exemple pris en EI1). Présenter vos choix de mise en oeuvre.

Algorithme Génétique

Pour cet exercice d'implémentation, j'ai décidé de mettre en place l'algorithme Génétique ; mon algorithme fonctionne avec 4 fonctions principales :

1. **Sélection des parents :** Cette fonction choisit le meilleur individu entre deux individus créés grâce à la fonction greedy_randomized_construction() ; le choix est fait en comparant leur coût total.
2. **Croisement :** Cette fonction permet de créer deux nouveaux individus à partir de deux parents sélectionnés avec la fonction de sélection des parents. Pour réaliser le croisement on attribue aléatoirement des éléments des parents aux enfants en veillant à ce que l'enfant respecte la contrainte de couverture des éléments.
3. **Mutation :** On garde les éléments communs, puis on complète aléatoirement avec des éléments non conflictuels de l'union des parents pour obtenir un enfant faisable.
4. **Sélection des enfants :** On sélectionne les meilleurs enfants parmi ceux créés, le choix est fait en comparant leur utilité totale.

Expérimentation numérique comparative GRASP vs métaheuristique concurrente

Présenter le protocole d'expérimentation (environnement matériel ; budget de calcul ; condition(s) d'arrêt ; réglage des paramètres).

Les algorithmes ont tourné sur la machine suivante :

— **Système d'exploitation :** Windows 11
— **Processeur :** AMD RYZEN 7 5700X 8-Core
— **Mémoire :** 16Go

0.1 Budget de calcul et conditions d'arrêt

Pour réalisé cette expérimentation, j'ai décidé de procéder comme ceci :

— J'ai selectionné 10 instances plus ou moins grandes du SPP
— J'ai choisi en paramètre de base pour l'algorithme génétique :
  — Taille de la population : 200
  — Nombre de génération : 500
  — Taux de mutation : 0.5
— J'ai choisi de répéter chaque expérimentation 3x pour chaque instances (3x 3 itération de population différente, 3x 3 itération de génération différente, 3x 3 itération de taux de mutation différent)
  — 5 itération ou on fais varier la taille de la population (100, 150, 200)
  — 5 itération ou on fais varier le nombre de génération (100, 250, 500)

— 5 itération ou on fais varier le taux de mutation (0.1, 0.5, 0.7)

Rapporter graphiquement vos résultats selon $\hat{z}_{min}$, $\hat{z}_{max}$, $\hat{z}_{moy}$ mesurés à intervalles réguliers (exemple de pas de 10 secondes).

1 Graphique des résultats

Les résultats sont obtenu avec 3 runs de chaque instances avec les paramètres "classique" défini plus haut.

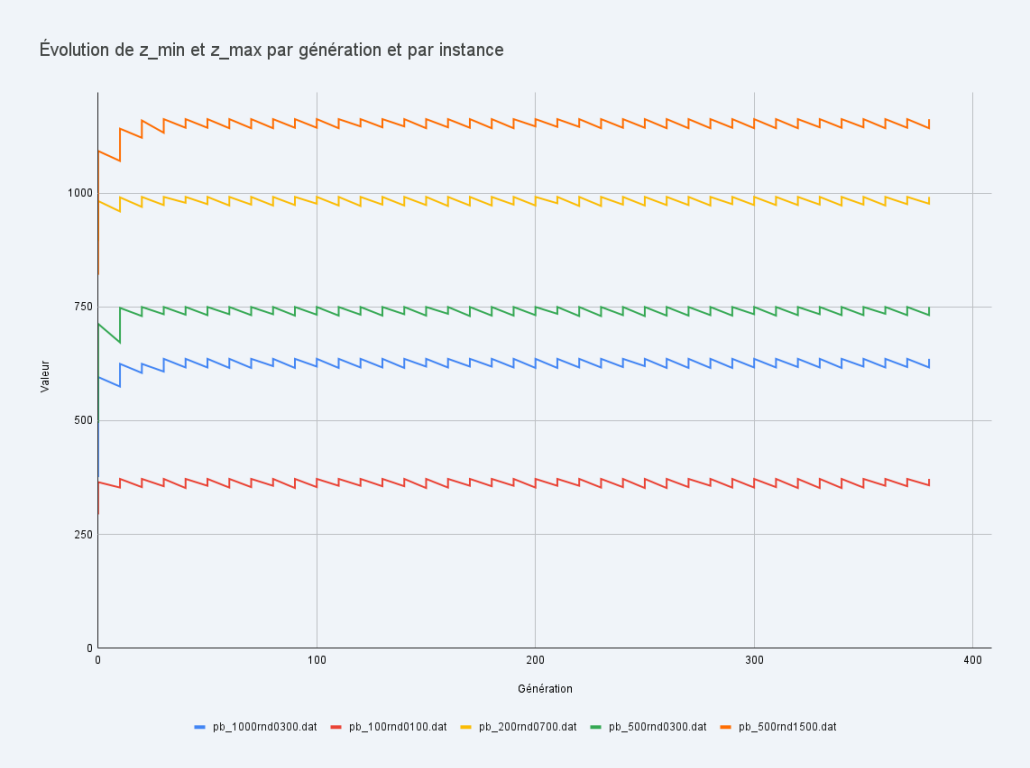

FIGURE 1 – Z_mean.

Rapporter l'étude de l'influence du paramètre α.

Concernant l'Algorithme Génétique (AG), l'influence de ses propres paramètres (étudiés dans le tableau 1) peut être résumée comme suit :

- **Taille de la population :** Augmenter la population (ex : 100 à 200) a l'impact le plus direct sur le **temps de calcul** (parfois doublé, cf. `pb_200rnd0100.dat`), mais son influence sur la qualité de la solution semble plafonner rapidement (rendements décroissants).
- **Nombre de générations :** Un nombre plus élevé permet à l'algorithme de converger. L'analyse montre que l'essentiel de l'amélioration est souvent obtenu avant 500 générations, suggérant une convergence ou stagnation par la suite pour un coût en temps linéaire.
- **Taux de mutation :** Ce paramètre est apparu comme **critique pour la qualité**. Un taux trop faible (ex : 0.1) résulte en des solutions de moins bonne qualité (ex : moyenne de 412 pour `pb_200rnd0100.dat`) comparé à des taux plus élevés (0.5 ou 0.7, moyenne 404), car il ne permet pas d'échapper aux optima locaux.

Présenter sous forme de tableau les résultats finaux obtenus pour les 10 instances sélectionnées.

TABLE 1: Résultats détaillés de l'algorithme génétique par groupe de paramètres.

Instance	Run Group	Run ID	Repeat	Pop.	Gens.	Mut.	Best	Time (s)
dat/pb_100rnd0100.dat	pop_size	1	1	100	500	0.5	372	2.085
			2				372	1.87
			3				372	1.872
		2	1	150	500	0.5	372	2.825
			2				372	2.789
			3				372	3.18
		3	1	200	500	0.5	372	4.425
			2				372	4.427
			3				372	4.472
	generations	4	1	200	100	0.5	372	1.089
			2				372	1.031
			3				372	1.018
		5	1	200	250	0.5	372	2.313
			2				372	2.299
			3				372	2.259
		6	1	200	500	0.5	372	4.36
			2				372	4.383
			3				372	4.394
	mutation	7	1	200	500	0.1	372	3.518
			2				372	3.587
			3				372	3.604
		8	1	200	500	0.5	372	4.35
			2				372	4.335
			3				372	4.296
		9	1	200	500	0.7	372	4.805
			2				372	4.706
			3				372	4.798
dat/pb_200rnd0100.dat	pop_size	1	1	100	500	0.5	409	4.075
			2				408	3.916
			3				408	3.635
		2	1	150	500	0.5	404	5.569
			2				416	6.716
			3				409	6.217
		3	1	200	500	0.5	412	7.481
			2				403	7.677
			3				410	8.065
	generations	4	1	200	100	0.5	400	1.862
			2				414	2.08
			3				403	1.895
		5	1	200	250	0.5	399	4.175
			2				403	3.912
			3				410	4.292
		6	1	200	500	0.5	403	7.618
			2				408	8.045
			3				412	7.476
	mutation	7	1	200	500	0.1	414	5.702
			2				414	5.727
			3				408	6.336
		8	1	200	500	0.5	399	7.634
			2				407	7.588
			3				408	7.179

Table 1: Résultats détaillés de l'algorithme génétique par groupe de paramètres (suite)

Instance	Run Group	Run ID	Repeat	Pop.	Gens.	Mut.	Best	Time (s)
		9	1	200	500	0.7	416	8.948
			2				399	8.201
			3				399	8.533
dat/pb_500rnd0300.dat	pop_size	1	1	100	500	0.5	744	16.545
			2				754	16.537
			3				730	15.946
		2	1	150	500	0.5	737	23.973
			2				740	24.598
			3				754	24.311
		3	1	200	500	0.5	744	32.789
			2				750	32.248
			3				754	33.185
	generations	4	1	200	100	0.5	750	9.565
			2				748	10.161
			3				750	10.005
		5	1	200	250	0.5	744	18.715
			2				740	18.231
			3				755	18.412
		6	1	200	500	0.5	752	32.737
			2				750	32.339
			3				754	33.822
	mutation	7	1	200	500	0.1	752	25.956
			2				751	26.721
			3				740	27.121
		8	1	200	500	0.5	754	36.276
			2				742	34.981
			3				754	35.492
		9	1	200	500	0.7	763	39.277
			2				732	39.005
			3				754	39.036
dat/pb_500rnd1500.dat	pop_size	1	1	100	500	0.5	1155	16.316
			2				1144	16.623
			3				1151	16.741
		2	1	150	500	0.5	1160	25.857
			2				1131	25.39
			3				1165	26.802
		3	1	200	500	0.5	1169	39.94
			2				1140	35.552
			3				1148	33.423
	generations	4	1	200	100	0.5	1162	10.513
			2				1162	10.668
			3				1165	10.434
		5	1	200	250	0.5	1152	19.124
			2				1146	19.045
			3				1151	18.949
		6	1	200	500	0.5	1137	33.501
			2				1151	33.343
			3				1162	32.292
	mutation	7	1	200	500	0.1	1164	24.817
			2				1141	24.396
			3				1148	25.201

TABLE 1: Résultats détaillés de l'algorithme génétique par groupe de paramètres (suite)

Instance	Run Group	Run ID	Repeat	Pop.	Gens.	Mut.	Best	Time (s)
		8	1	200	500	0.5	1162	31.844
			2				1155	31.26
			3				1162	31.533
		9	1	200	500	0.7	1172	34.537
			2				1162	35.406
			3				1160	38.371
dat/pb_500rnd1700.dat	pop_size	1	1	100	500	0.5	168	2.572
			2				168	2.292
			3				168	2.509
		2	1	150	500	0.5	185	3.74
			2				175	3.467
			3				173	3.698
		3	1	200	500	0.5	185	5.241
			2				185	5.15
			3				192	5.063
	generations	4	1	200	100	0.5	184	1.394
			2				179	1.669
			3				163	1.354
		5	1	200	250	0.5	174	2.829
			2				188	2.931
			3				170	3.302
		6	1	200	500	0.5	176	4.761
			2				167	4.565
			3				182	5.183
	mutation	7	1	200	500	0.1	174	3.881
			2				187	4.388
			3				171	3.828
		8	1	200	500	0.5	188	5.251
			2				185	5.318
			3				187	5.568
		9	1	200	500	0.7	176	5.205
			2				182	5.651
			3				185	6.057
dat/pb_200rnd0400.dat	pop_size	1	1	100	500	0.5	62	8.658
			2				62	8.951
			3				62	8.792
		2	1	150	500	0.5	62	13.227
			2				62	13.149
			3				62	12.844
		3	1	200	500	0.5	63	17.865
			2				62	17.898
			3				62	17.552
	generations	4	1	200	100	0.5	62	4.525
			2				62	4.367
			3				62	4.594
		5	1	200	250	0.5	62	9.399
			2				62	9.809
			3				62	9.571
		6	1	200	500	0.5	62	17.818
			2				62	18.936
			3				62	18.904

Table 1: Résultats détaillés de l'algorithme génétique par groupe de paramètres (suite)

Instance	Run Group	Run ID	Repeat	Pop.	Gens.	Mut.	Best	Time (s)
			1				62	14.524
		7	2	200	500	0.1	62	14.912
			3				62	15.199
			1				62	18.753
	mutation	8	2	200	500	0.5	62	19.039
			3				62	19.061
			1				62	21.029
		9	2	200	500	0.7	62	19.966
			3				62	20.336
			1				990	3.417
		1	2	100	500	0.5	997	3.451
			3				991	3.512
			1				991	5.452
	pop_size	2	2	150	500	0.5	989	5.152
			3				991	5.096
			1				992	6.857
		3	2	200	500	0.5	992	6.802
			3				992	7.186
			1				993	2.027
		4	2	200	100	0.5	994	1.604
dat/pb_200rnd0700.dat			3				991	1.61
			1				995	3.446
	generations	5	2	200	250	0.5	989	3.386
			3				989	3.441
			1				992	6.527
		6	2	200	500	0.5	986	6.435
			3				992	6.287
			1				991	5.655
		7	2	200	500	0.1	991	5.573
			3				991	5.682
			1				989	6.831
	mutation	8	2	200	500	0.5	991	6.603
			3				989	6.166
			1				989	6.514
		9	2	200	500	0.7	992	6.998
			3				990	7.028
			1				55	3.239
		1	2	100	500	0.5	67	3.577
			3				67	3.578
			1				67	5.155
	pop_size	2	2	150	500	0.5	59	4.366
			3				67	5.143
			1				57	5.661
		3	2	200	500	0.5	67	6.674
			3				67	6.724
			1				59	2.298
		4	2	200	100	0.5	59	2.533
dat/pb_1000rnd0100.dat			3				57	2.385
			1				56	3.557
	generations	5	2	200	250	0.5	67	4.089
			3				59	3.578

TABLE 1: Résultats détaillés de l'algorithme génétique par groupe de paramètres (suite)

Instance	Run Group	Run ID	Repeat	Pop.	Gens.	Mut.	Best	Time (s)
			1				67	6.717
		6	2	200	500	0.5	67	6.715
			3				67	6.804
			1				57	4.606
		7	2	200	500	0.1	67	5.417
			3				59	4.444
	mutation		1				67	6.782
		8	2	200	500	0.5	67	6.668
			3				67	6.661
			1				59	6.144
		9	2	200	500	0.7	57	6.345
			3				59	6.175
			1				589	24.005
		1	2	100	500	0.5	581	24.138
			3				593	24.226
			1				599	34.225
	pop_size	2	2	150	500	0.5	604	35.242
			3				593	31.314
			1				579	41.95
		3	2	200	500	0.5	592	42.788
			3				588	42.55
			1				598	16.965
		4	2	200	100	0.5	608	16.777
dat/pb_1000rnd0300.dat			3				596	16.58
	generations		1				616	27.418
		5	2	200	250	0.5	625	27.736
			3				595	26.014
			1				597	48.604
		6	2	200	500	0.5	592	49.824
			3				608	52.782
			1				633	41.84
		7	2	200	500	0.1	604	38.161
			3				612	40.256
	mutation		1				609	45.487
		8	2	200	500	0.5	617	42.64
			3				590	50.089
			1				599	54.801
		9	2	200	500	0.7	594	56.423
			3				598	53.568
			1				30	0.149
		1	2	100	500	0.5	30	0.122
			3				30	0.132
			1				30	0.2
	pop_size	2	2	150	500	0.5	30	0.178
			3				30	0.182
			1				30	0.251
		3	2	200	500	0.5	30	0.242
			3				30	0.269
			1				30	0.068
		4	2	200	100	0.5	30	0.07
dat/didactic.dat			3				30	0.059
	generations							

TABLE 1: Résultats détaillés de l'algorithme génétique par groupe de paramètres (suite)

Instance	Run Group	Run ID	Repeat	Pop.	Gens.	Mut.	Best	Time (s)
		5	1	200	250	0.5	30	0.137
			2				30	0.13
			3				30	0.127
		6	1	200	500	0.5	30	0.236
			2				30	0.264
			3				30	0.244
	mutation	7	1	200	500	0.1	30	0.24
			2				30	0.245
			3				30	0.244
		8	1	200	500	0.5	30	0.257
			2				30	0.276
			3				30	0.249
		9	1	200	500	0.7	30	0.258
			2				30	0.269
			3				30	0.26

Discussion

Tirer des conclusions en comparant les résultats collectés avec vos deux métaheuristiques.
En comparant les résultats obtenus avec GRASP et l'algorithme génétique, on observe un point commun : les deux méthodes tendent à produire de meilleures solutions lorsque que l'aléatoire est important (alpha élevé pour GRASP, taux de mutation élevé pour l'algorithme génétique). Toutefois, l'algorithme génétique présente, dans notre étude, de meilleures performances sur la majorité des instances testées. Cet avantage est particulièrement visible pour les instances de grande taille (par exemple `pb_1000rnd0300.dat`).
Quelles sont les recommandations que vous émettez à l'issue de l'étude ?
Sur la base des expériences menées, je recommande l'utilisation de l'algorithme génétique : il a fourni les meilleurs résultats dans la plupart des cas étudiés. Il reste néanmoins possible d'améliorer encore ses performances en affinant la sélection des parents et les opérateurs de croisement. Il faut aussi garder à l'esprit un inconvénient pratique : l'algorithme génétique comporte plusieurs hyperparamètres (taille de la population, nombre de générations, taux de mutation), ce qui rend le réglage plus exigeant que pour GRASP, dont le paramètre principal est α (ou la liste d'alpha en Reactive GRASP). En conséquence, si le temps et les ressources de calibration sont limités, GRASP reste une option intéressante pour sa simplicité.