Nantes Université — UFR Sciences et Techniques
Master informatique
Année académique 2025-2026

Dossier Exercice d'implémentation

Métaheuristiques

Riccardo VENTURINI

23 novembre 2025

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

Formulation du SPP

Soit $U = \{1, 2, \ldots, n\}$ l'ensemble universel et une collection d'ensembles $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, où $S_j \subseteq U$, chacun ayant un poids associé $c_j \in R^+$.

L'objectif est de sélectionner un sous-ensemble d'ensembles de $\mathcal{S}$ qui sont mutuellement disjoints et dont la somme des poids est maximale.

Variables de Décision

$x_j \in \{0, 1\}, \quad \forall j \in \{1, 2, \ldots, m\}$

$$x_j = \begin{cases} 1 & \text{si l'ensemble } S_j \text{ est sélectionné} \\ 0 & \text{sinon} \end{cases}$$

Formulation du Problème

$$\textbf{Maximiser} \quad \sum_{j=1}^{m} c_j \cdot x_j$$

Sous les contraintes :

$$\sum_{j:i \in S_j} x_j \leq 1 \qquad \forall i \in U \qquad \text{(Contrainte de Disjonction)}$$

$$x_j \in \{0, 1\} \qquad \forall j \in \{1, 2, \ldots, m\} \qquad \text{(Variables Binaires)}$$

Cas practique : Le Problème de Planification Ferroviaire (RPP)

Le RPP se concentre sur l'optimisation de la construction ou de la reconstruction d'infrastructures ferroviaires. Il s'agit spécifiquement de planifier l'utilisation et la capacité des composants critiques d'un système ferroviaire.

Étant donné :

— Un ensemble fini d'éléments $I = \{1, \ldots, n\}$

— $\{T_j\}$, $j \in J = \{1, \ldots, m\}$, une collection de m sous-ensembles de I

Une sélection (ou recouvrement) est un sous-ensemble $P \subseteq I$ tel que $|T_j \cap P| \leq 1, \forall j \in J$, ce qui mène à la formulation :

$$\begin{aligned} \textbf{Maximiser} \quad & z(\mathbf{x}) = \sum_{i \in I} c_i x_i \\ \textbf{Sous les contraintes :} \quad & \sum_{i \in I} t_{i,j} x_i \leq 1 && \forall j \in J \\ & x_i \in \{0, 1\} && \forall i \in I \\ & t_{i,j} \in \{0, 1\} && \forall i \in I, \forall j \in J \end{aligned}$$

— Fortement NP-Difficile (Garey and Johnson 1979)

— Problème du Recouvrement de Nœuds : $\sum_{i \in I} t_{i,j} = 2, \forall j \in J$

Modélisation JuMP (ou GMP) du SPP

La librairie **JuMP** est un langage de modélisation mathématique open-source conçu pour le langage Julia. Elle permet aux utilisateurs de formuler des problèmes d'optimisation complexes. Dans cet exemple, nous avons utilisé l'optimiseur GLPK.

```
C, A = loadSPP(fullpath)

    solverSelected = GLPK.Optimizer
    spp = setSPP(C, A)

    set_optimizer(spp, solverSelected)
     t_start = time()
    optimize!(spp)
    time_jump = time() - t_start
```

Vous trouverez ci-dessous les résultats de l'expérience.

Result GLPK

File	GLPK (JuMP)	Time (s)
pb_1000rnd0100.dat	67	212.380985
pb_1000rnd0800.dat	NA	+10 min.
pb_100rnd0500.dat	639	0.000267
pb_100rnd1200.dat	23	0.231301
pb_2000rnd0400.dat	NA	+10 min.
pb_2000rnd0500.dat	NA	+10 min.
pb_200rnd0300.dat	NA	+10 min.
pb_200rnd1800.dat	19	174.513032
pb_500rnd0700.dat	NA	+10 min.
pb_500rnd1700.dat	NA	+10 min.

Instances numériques de SPP

Dans le (Tableau 2) les 10 instances sélectionnées.

Instance	n° Variables	n° contrantes	Densité (%)	Max-Uns	Meillere valer connue
pb_1000rnd0100.dat	1000	5000	2,60	50	67*
pb_1000rnd0800.dat	1000	1000	0,60	10	175*
pb_100rnd0500.dat	100	500	2	2	639*
pb_100rnd1200.dat	100	300	2,97	4	23*
pb_2000rnd0400.dat	2000	10000	0,55	20	32
pb_2000rnd0500.dat	2000	2000	2,55	100	140
pb_200rnd0300.dat	200	1000	1	2	731*
pb_200rnd1800.dat	200	200	1,50	4	83*
pb_500rnd0700.dat	500	500	1,20	10	1141*
pb_500rnd1700.dat	500	1500	2,17	20	192*

Table 2 – instances sélectionnées

Heuristique de construction appliquée au SPP

L'algorithme glouton a été utilisé pour la construction heuristique. L'algorithme glouton sélectionne la première colonne admissible, l'ajoute à la solution, puis avance en ne choisissant que des colonnes qui restent compatibles avec celles déjà sélectionnées. Il construit une solution initiale de manière séquentielle, en ajoutant progressivement des éléments non conflictuels.

Algorithm 1 La construction gloutonne

$S \leftarrow \emptyset$ ▷ Initialise la solution comme un ensemble vide
Initialiser l'ensemble des candidats C, et évaluer l'utilité $u(e), \forall e \in C$
while $(C \neq \emptyset)$ **do**
    Sélectionner le meilleur élément courant e de C :
        $e \leftarrow \underset{e \in C}{\text{next}}\, u(e)$ ▷ Choisir l'élément suivante
    Incorporer e dans la solution :
        $S \leftarrow S \cup \{e\}$
    Mettre à jour l'ensemble des candidats C et réévaluer l'utilité $u(e), \forall e \in C$
        $C \leftarrow C \setminus \text{conflict}(\{e\})$ ▷ Retirer e et tous les éléments en conflit avec e
retourner S

— S : ensemble des solutions
— C : solutions candidates
— e : variable

Exemple Didactique de la Construction Gloutonne (Sélection Forcée)

Considérons un problème de sélection d'éléments pour maximiser l'utilité totale, soumis à des contraintes de conflit.

Données :
— **Ensemble initial des candidats** C **:** $C = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$.
— **Utilités** $u(e)$ **:**

Élément	e_1	e_2	e_3	e_4	e_5	e_6	e_7
Utilité $u(e)$	10	5	8	6	9	13	11

Déroulement de l'algorithme :

1. **Initialisation** : $S = \emptyset$. $C = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$.
2. **Itération 1 (Sélection de** e_1**)** :
   — **Sélection** : On choisit $e = e_1$ (Utilité 10).
   — **Incorporation** : $S \leftarrow \{e_1\}$. Utilité totale = 10.
   — **Mise à jour** C : conflict$(\{e_1\}) = \{e_2, e_3, e_5, e_6, e7\}$.
   — $C \leftarrow C \setminus \{e_2, e_3, e_5, e_6, e7\}$.
   — Nouveau $C = \{e_4\}$.
3. **Itération 2 (Gloutonne)** :
   — **Sélection** : Dans $C = \{e_4(6)\}$, $\max_{e \in C} u(e) = u(e_4) = 6$. L'élément choisi est $e = e_4$.
   — **Incorporation** : $S \leftarrow \{e_1, e_6\}$. Utilité totale = $10 + 6 = 16$.
   — **Mise à jour** C : conflict$(\{e_4\}) = \{e_4\}$.
   — $C \leftarrow C \setminus \{e_4\}$.
   — Nouveau $C = \emptyset$.
4. **Arrêt** : La boucle **While** se termine car $C = \emptyset$.
5. **Résultat final** : La solution gloutonne retournée est $S = \{e_1, e_4\}$, avec une utilité totale de 16.

Heuristique d'amélioration appliquée au SPP

— X : l'espace de toutes les solutions réalisables (l'espace de recherche)

Algorithm 2 Procédure de Descente Simple

```
1: procedure DescenteSimple(solution initiale x ∈ X)
2:     repeat
3:         choisir x' ∈ N(x)                          ▷ Sélectionner un voisin x' de x
4:         if z(x') < z(x) then                       ▷ Si le voisin x' est meilleur (minimisation)
5:             x ← x'                                 ▷ Déplacer la solution courante vers ce meilleur voisin
6:     until f(x') ≥ f(x), ∀x' ∈ N(x)                 ▷ Jusqu'à ce que x soit un minimum local
7:     Retourner x
```

— N(x) : le voisinage de la solution courante x
— x' : une solution voisine candidate, appartenant au voisinage $N(x)$
— z(x) : la fonction objectif de la solution x
— f(x) : la condition d'arrêt qui signifie que la procédure s'arrête lorsque la solution courante x est un optimum local, c'est-à-dire qu'aucune solution voisine x' n'est meilleure que x

Exemple Didactique de la Descente Simple

Déroulement de l'algorithme (avec $x_{initial} = \{1, 4\}$) :

1. **Étape 1 : Initialisation.**
   — Solution courante : $x = \{1, 4\}$.
   — Coût courant : $z(x) = c_1 + c_4 = 10 + 6 = 16$.
2. **Étape 2 : Exploration (Boucle Repeat 1).**
   — Voisinage $N(x)$ (Ajout ou Retrait d'un élément) :
      — Retrait : $x'_a = \{1\}$, $z(x'_a) = 10$. NON ($10 < 16$).
      — Retrait : $x'_b = \{4\}$, $z(x'_b) = 6$. NON ($6 < 16$).
      — Ajout : $x'_c = \{1, 4, 5\}$, $z(x'_c) = 10 + 6 + 9 = 25$. OUI ($25 > 16$).
   — Voisin choisi (le meilleur trouvé ou le premier trouvé meilleur) : $x' = \{1, 4, 5\}$.
   — Mise à jour : $x \leftarrow \{1, 4, 5\}$.
3. **Étape 3 : Exploration (Boucle Repeat 2).**
   — Solution courante : $x = \{1, 4, 5\}$, $z(x) = 25$.
   — Voisinage $N(x)$ (Ajout ou Retrait) :
      — Retrait : $x'_d = \{4, 5\}$, $z(x'_d) = 6 + 9 = 15$. NON.
      — Ajout : $x'_e = \{1, 4, 5, 6\}$, $z(x'_e) = 10 + 6 + 9 + 13 = 38$. OUI ($38 > 25$).
   — Voisin choisi : $x' = \{1, 4, 5, 6\}$.
   — Mise à jour : $x \leftarrow \{1, 4, 5, 6\}$.
4. **Étape 4 : Exploration (Boucle Repeat 3).**
   — Solution courante : $x = \{1, 4, 5, 6\}$, $z(x) = 38$.
   — Voisinage $N(x)$ (Ajout ou Retrait) :
      — Retrait : $x'_f = \{1, 5, 6\}$, $z(x'_f) = 10 + 9 + 13 = 32$. NON.
      — Ajout : $x'_g = \{1, 4, 5, 6, 7\}$, $z(x'_g) = 10 + 6 + 9 + 13 + 11 = 49$. OUI ($49 > 38$).
   — Voisin choisi : $x' = \{1, 4, 5, 6, 7\}$.
   — Mise à jour : $x \leftarrow \{1, 4, 5, 6, 7\}$.
5. **Étape 5 : Exploration (Boucle Repeat 4).**
   — Solution courante : $x = \{1, 4, 5, 6, 7\}$, $z(x) = 49$.
   — Voisinage $N(x)$ (Retrait d'un élément) :
      — Retrait de 1 : $z = 39$. NON.
      — Retrait de 4 : $z = 43$. NON.
      — Retrait de 7 : $z = 38$. NON.
   — **Condition d'arrêt atteinte :** Aucune solution voisine n'a un coût supérieur à $z(x) = 49$.
6. **Résultat :** L'algorithme s'arrête en $x = \{1, 4, 5, 6, 7\}$ avec un coût $z(x) = 49$. Cette solution est un **maximum local**.

Expérimentation numérique

Machine sur laquelle les résultats ont été enregistrés :

MacMini M4 (ARM) - CPU 10 cœurs

Result E1

File	Heuristic	Time (s)	Local Search	Time (s)
pb_1000rnd0100.dat	22	0.156478	40	5.488210
pb_1000rnd0800.dat	108	0.001138	108	25.506977
pb_100rnd0500.dat	533	0.000038	620	0.025510
pb_100rnd1200.dat	17	0.000039	17	0.012132
pb_2000rnd0400.dat	20	0.051748	20	222.783113
pb_2000rnd0500.dat	36	0.005408	91	30.030356
pb_200rnd0300.dat	424	0.000159	662	4.381937
pb_200rnd1800.dat	12	0.000074	12	0.073140
pb_500rnd0700.dat	667	0.000266	975	38.437973
pb_500rnd1700.dat	98	0.000663	137	3.493589

Discussion

Dans ces exemples, l'utilisation de GLPK n'est pas justifiée car, dans la plupart des cas, sa résolution prend plus de 10 minutes, et obtenir de bonnes solutions n'est pas seulement une question de résultats, mais aussi de temps. En revanche, mon heuristique trouve une solution beaucoup plus rapidement, mais nous sommes encore un peu loin de l'optimalité. Avec la recherche locale par échange 1-1, nous nous approchons dans un délai acceptable. J'ai choisi cette implémentation pour sa simplicité et, surtout, pour sa priorité donnée au gain de temps. Ainsi, comparée à GLPK, elle est préférable car au moins, nous avons toujours une solution.

Je suppose que l'utilisation de métaheuristiques pourrait être prometteuse , il existe certainement des solutions plus éloignées, difficiles à explorer avec les techniques de voisinage classiques.

Dans mon exemple, je n'utilise pas l'algorithme de Glouton traditionnel, qui ne sélectionne pas la valeur maximale dans l'espace des candidats, mais la suivante admissible. La solution initiale importe peu, c'est le chemin d'exploration qui compte pour trouver la solution optimale.

Dans ce cas, nous n'avons pas d'aléatoire, une piste d'amélioration pourrait consister à réduire le déterminisme, comme c'était le cas dans GRASP.

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.

La méthode GRASP (Greedy Randomized Adaptive Search Procedure) est une métaheuristique que combine les méthodes gloutonnes et aléatoires. La construction d'une solution se déroule par étapes et à chacune de celles-ci, l'ensemble des morceaux de solution qu'il est possible d'ajouter est placé dans une liste appelée RCL (Restricted Candidate List). Dans la partie gloutonne cette liste est triée, mais ce n'est pas nécessairement le meilleur morceau qui est ajouté à la solution

courante. Pour la partie aleatoire on tire aléatoirement parmi les meilleurs possibilités le morceau à ajouter, ça permet donc de varier la forme des solutions générées mais celles-ci sont quand même de bonne qualité, puisque le choix aléatoire se fait parmi un ensemble de bons candidats. La recherche locale s'applique sur la solution réalisable résultante de la phase de construction afin de voir s'il est encore possible d'améliorer cette solution.

Je n'analyserai ci-dessous que la partie construction, car pour l'amélioration, on utilise la recherche d'échanges locaux 1-1 vue en EI1.

Algorithm 3 La construction gloutonne randomisée

$S \leftarrow \emptyset$ ▷ Solution courante
Initialiser l'ensemble des candidats C, et évaluer $u(e), \forall e \in C$
while $(C \neq \emptyset)$ **do**
    Construire la Liste des Candidats Restreints (RCL) :
        $u_{min} \leftarrow \min_{e \in C} u(e)$
        $u_{max} \leftarrow \max_{e \in C} u(e)$
        $u_{Limit} \leftarrow u_{min} + \alpha \times (u_{max} - u_{min})$ ▷ $\alpha \in [0, 1]$ est le paramètre de gloutonnerie
        $RCL \leftarrow \{e \in C \mid u(e) \geq u_{Limit}\}$ ▷ RCL contient les éléments "suffisamment bons"
    Sélectionner un élément e du RCL au hasard :
        $e \leftarrow \text{RandomSelect}(RCL)$
    Incorporer e dans la solution :
        $S \leftarrow S \cup \{e\}$
    Mettre à jour l'ensemble des candidats C :
        $C \leftarrow C \setminus \text{conflict}(\{e\})$
retourner la solution construite S

Note sur le paramètre α :
— Si $\alpha = 0$, Il n'y a pas d'aleatoire.
— Si $\alpha = 1$, la sélection est totalement randomisée.

Exemple Didactique du GRASP

Données de l'Instance

Éléments I et Utilités $u(e)$:

$$I = \{e_1, \ldots, e_9\}$$

Élément	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
Utilité $u(e)$	10	5	8	6	9	**13**	11	4	6

Contraintes de Conflit (Sous-ensembles $\mathcal{T}$) :
— $T_1 = \{e_1, e_2, e_3, e_5, e_7, e_8\}$
— $T_2 = \{e_2, e_3, e_8\}$
— $T_3 = \{e_2, e_5, e_6, e_8, e_9\}$
— $T_4 = \{e_4\}$
— $T_5 = \{e_1, e_3, e_5, e_6, e_9\}$
— $T_6 = \{e_2, e_3, e_7, e_9\}$
— $T_7 = \{e_1, e_4, e_5, e_8, e_9\}$

1. Phase de Construction Gloutonne Randomisée ($\alpha = 0.5$)

L'objectif est de construire une solution initiale S en utilisant le critère de la Liste des Candidats Restreints (RCL).

$$u_{Limit} = u_{min} + \alpha \times (u_{max} - u_{min})$$

Itération 1

- $C = \{e_1, \ldots, e_9\}$. $u_{max} = 13$ (e_6), $u_{min} = 4$ (e_8).
- $u_{Limit} = 4 + 0.5 \times (13 - 4) = 4 + 4.5 = \mathbf{8.5}$.
- **RCL** $= \{e \in C \mid u(e) \geq 8.5\} = \{e_1(10), e_5(9), e_6(13), e_7(11)\}$.
- **Sélection Aléatoire (Hypothèse)** : On choisit $\mathbf{e_7}$ (Utilité 11).
- **Mise à jour** S : $S = \{e_7\}$. Utilité Totale : 11.
- **Candidats retirés** : e_7 est dans T_1 et T_6. Tous les éléments de T_1 et T_6 (sauf e_7) sont retirés, plus e_7.

$$\text{Retiré} = \{e_7\} \cup (T_1 \setminus \{e_7\}) \cup (T_6 \setminus \{e_7\}) = \{e_7, e_1, e_2, e_3, e_5, e_8, e_9\}$$

- **Nouveau** C : $\{e_4(6), e_6(13)\}$.

Itération 2

- $C = \{e_4(6), e_6(13)\}$. $u_{max} = 13$ (e_6), $u_{min} = 6$ (e_4).
- $u_{Limit} = 6 + 0.5 \times (13 - 6) = 6 + 3.5 = \mathbf{9.5}$.
- **RCL** $= \{e \in C \mid u(e) \geq 9.5\} = \{e_6(13)\}$.
- **Sélection Aléatoire** : On choisit $\mathbf{e_6}$ (Utilité 13).
- **Mise à jour** S : $S = \{e_7, e_6\}$. Utilité Totale : $11 + 13 = 24$.
- **Candidats retirés** : $\{e_6\}$. (Les conflits avec e_6 sont déjà retirés dans C).
- **Nouveau** C : $\{e_4(6)\}$.

Itération 3

- $C = \{e_4(6)\}$. $u_{max} = 6, u_{min} = 6$. $u_{Limit} = 6$.
- **RCL** $= \{e_4(6)\}$.
- **Sélection Aléatoire** : On choisit $\mathbf{e_4}$ (Utilité 6).
- **Mise à jour** S : $S = \{e_7, e_6, e_4\}$. Utilité Totale : $24 + 6 = 30$.
- **Candidats retirés** : $\{e_4\}$.
- **Nouveau** C : $\emptyset$.

Solution de Construction S : $S = \{e_4, e_6, e_7\}$, avec une utilité de **30**.

2. Phase d'Amélioration Locale (déjà analisé)

Présentation succincte de Path-Relinking appliqué sur le SPP

Données et Solutions Élite

L'instance SPP comporte 9 éléments (e_1 à e_9) avec les utilités suivantes :

Élément	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	**Total**
Utilité $u(e)$	10	5	8	6	9	13	11	4	6	

Nous sélectionnons les deux solutions élites suivantes (représentées par un vecteur binaire où 1 signifie sélectionné, 0 non sélectionné) :

Algorithm 4 Path Relinking

Ensure: Meilleure Solution $\mathbf{x}_{\text{best}}$ trouvée sur le chemin, et sa valeur z_{best}.

$\mathbf{x}_i \leftarrow \text{copy}(\mathbf{x}_A)$ ▷ Initialiser la solution courante
$\mathbf{x}_{\text{best}} \leftarrow \text{copy}(\mathbf{x}_i)$
$z_{\text{best}} \leftarrow \sum(C \cdot \mathbf{x}_i)$
Diff $\leftarrow \{i \mid \mathbf{x}_i[i] \neq \mathbf{x}_B[i]\}$ ▷ Identifier les indices de différence
while Diff $\neq \emptyset$ **do**
  Sélectionner aléatoirement un indice i dans Diff
  $\mathbf{x}_i[i] \leftarrow \mathbf{x}_B[i]$ ▷ Appliquer le mouvement de la solution guide $\mathbf{x}_B$
  $z_i \leftarrow \sum(C \cdot \mathbf{x}_i)$
  if $z_i > z_{\text{best}}$ **then**
    $\mathbf{x}_{\text{best}} \leftarrow \text{copy}(\mathbf{x}_i)$
    $z_{\text{best}} \leftarrow z_i$
  Optionnel : Appliquer une Recherche Locale sur $\mathbf{x}_i$
  $(\mathbf{x}_{\text{LS}}, z_{\text{LS}}) \leftarrow \text{localSearch\_1\_1}(C, A, \mathbf{x}_i)$
  if $z_{\text{LS}} > z_{\text{best}}$ **then**
    $\mathbf{x}_{\text{best}} \leftarrow \text{copy}(\mathbf{x}_{\text{LS}})$
    $z_{\text{best}} \leftarrow z_{\text{LS}}$
  Diff $\leftarrow \{i \mid \mathbf{x}_i[i] \neq \mathbf{x}_B[i]\}$ ▷ Mettre à jour les indices de différence
Ajouter $(\mathbf{x}_{\text{best}}, z_{\text{best}})$ à EliteSet ▷ Mettre à jour l'ensemble élite
return $\mathbf{x}_{\text{best}}, z_{\text{best}}$

— **Solution de Démarrage** $\mathbf{x}_A$: $\{e_4, e_6, e_7\}$.

$$\mathbf{x}_A = (0,0,0,1,0,1,1,0,0) \implies z_A = 6 + 13 + 11 = \mathbf{30}$$

— **Solution de Guidage** $\mathbf{x}_B$: $\{e_1, e_2, e_8, e_9\}$.

$$\mathbf{x}_B = (1,1,0,0,0,0,0,1,1) \implies z_B = 10 + 5 + 4 + 6 = \mathbf{25}$$

— **Meilleure Solution trouvée** : $\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_A$, $z_{\text{best}} \leftarrow 30$.

Déroulement du Path Relinking

Initialisation

L'ensemble des indices de différence (Diff) est :

$$\text{Diff} = \{i \mid \mathbf{x}_A[i] \neq \mathbf{x}_B[i]\} = \{1, 2, 4, 6, 7, 8, 9\}$$

(7 différences à résoudre pour transformer $\mathbf{x}_A$ en $\mathbf{x}_B$).

Itération 1 : Résolution de la différence e_7

— **Sélection Aléatoire** : $i = 7$ (élément e_7).
— **Mouvement** : e_7 est dans $\mathbf{x}_A$ (1) mais pas dans $\mathbf{x}_B$ (0). On retire e_7.
— **Nouvelle Solution** $\mathbf{x}_i$: $(0,0,0,1,0,1,\mathbf{0},0,0)$.
— **Utilité** z_i : $30 - 11 = 19$.
— **Mise à Jour** : $19 \not> z_{\text{best}}$. Aucune amélioration.
— **Diff** mis à jour : Diff $= \{1, 2, 4, 6, 8, 9\}$.

Itération 2 : Résolution de la différence e_1

— **Sélection Aléatoire** : $i = 1$ (élément e_1).
— **Mouvement** : e_1 n'est pas dans $\mathbf{x}_i$ (0) mais est dans $\mathbf{x}_B$ (1). On ajoute e_1.
— **Nouvelle Solution** $\mathbf{x}_i$: $(\mathbf{1},0,0,1,0,1,0,0,0)$ (soit $\{e_1, e_4, e_6\}$).

— **Utilité** z_i : $19 + 10 = 29$.
— **Mise à Jour** : $29 \not> z_{\text{best}}$. Aucune amélioration.
— **Diff** mis à jour : Diff $= \{2, 4, 6, 8, 9\}$.

Itération 3 : Résolution de la différence e_9

— **Sélection Aléatoire** : $i = 9$ (élément e_9).
— **Mouvement** : On ajoute e_9.
— **Nouvelle Solution** $\mathbf{x}_i$: $(1, 0, 0, 1, 0, 1, 0, 0, \mathbf{1})$ (soit $\{e_1, e_4, e_6, e_9\}$).
— **Utilité** z_i : $29 + 6 = \mathbf{35}$.
— **Mise à Jour** : $35 > z_{\text{best}} = 30$.

$$\mathbf{x}_{\text{best}} \leftarrow \{e_1, e_4, e_6, e_9\}$$

$$z_{\text{best}} \leftarrow 35$$

— **Diff** mis à jour : Diff $= \{2, 4, 6, 8\}$.

Itération 4 : Résolution de la différence e_6

— **Sélection Aléatoire** : $i = 6$ (élément e_6).
— **Mouvement** : e_6 est dans $\mathbf{x}_i$ (1) mais pas dans $\mathbf{x}_B$ (0). On retire e_6.
— **Nouvelle Solution** $\mathbf{x}_i$: $(1, 0, 0, 1, 0, \mathbf{0}, 0, 0, 1)$ (soit $\{e_1, e_4, e_9\}$).
— **Utilité** z_i : $35 - 13 = 22$.
— **Mise à Jour** : $22 \not> z_{\text{best}}$. Aucune amélioration.
— **Diff** mis à jour : Diff $= \{2, 4, 8\}$.

Itérations Finales

Le processus se poursuit jusqu'à ce que $\mathbf{x}_i$ soit identique à $\mathbf{x}_B$.

— **Itération 5** : Résoudre e_4 (Retrait). $\mathbf{x}_i \leftarrow \{e_1, e_9\}$. $z_i = 22 - 6 = 16$.
— **Itération 6** : Résoudre e_2 (Ajout). $\mathbf{x}_i \leftarrow \{e_1, e_2, e_9\}$. $z_i = 16 + 5 = 21$.
— **Itération 7** : Résoudre e_8 (Ajout). $\mathbf{x}_i \leftarrow \{e_1, e_2, e_8, e_9\}$. $z_i = 21 + 4 = 25$.

Conclusion du Chemin

La solution finale du chemin est $\mathbf{x}_i = \mathbf{x}_B$ avec une utilité de 25.

Expérimentation numérique de GRASP

Paramètres de test pour les graphiques :
— nbInstances = 3
— nbRunGrasp = 30
— nbIterationGrasp = 100
— nbDivisionRun = 10
— alpha = [0.0, 0.25, 0.5, 0.75, 1.0]

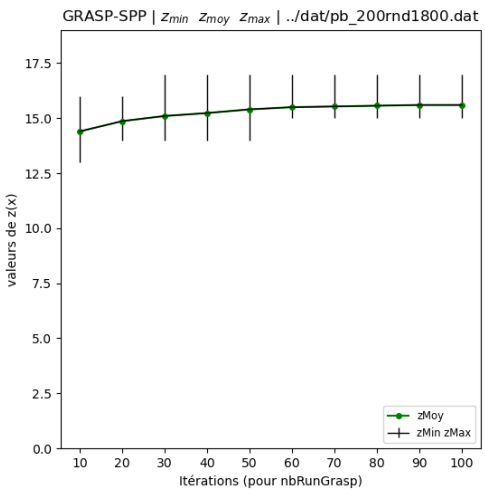

FIGURE 1 – Bilan sur tous les run

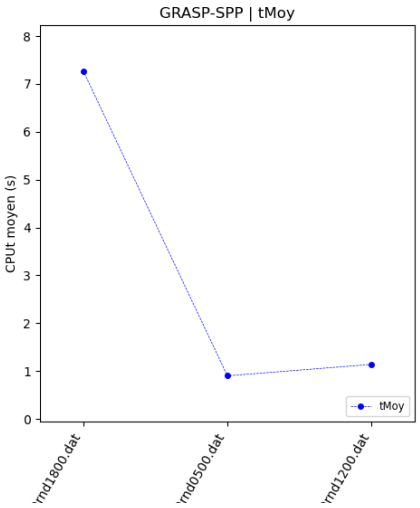

FIGURE 2 – CPU temps d'execution

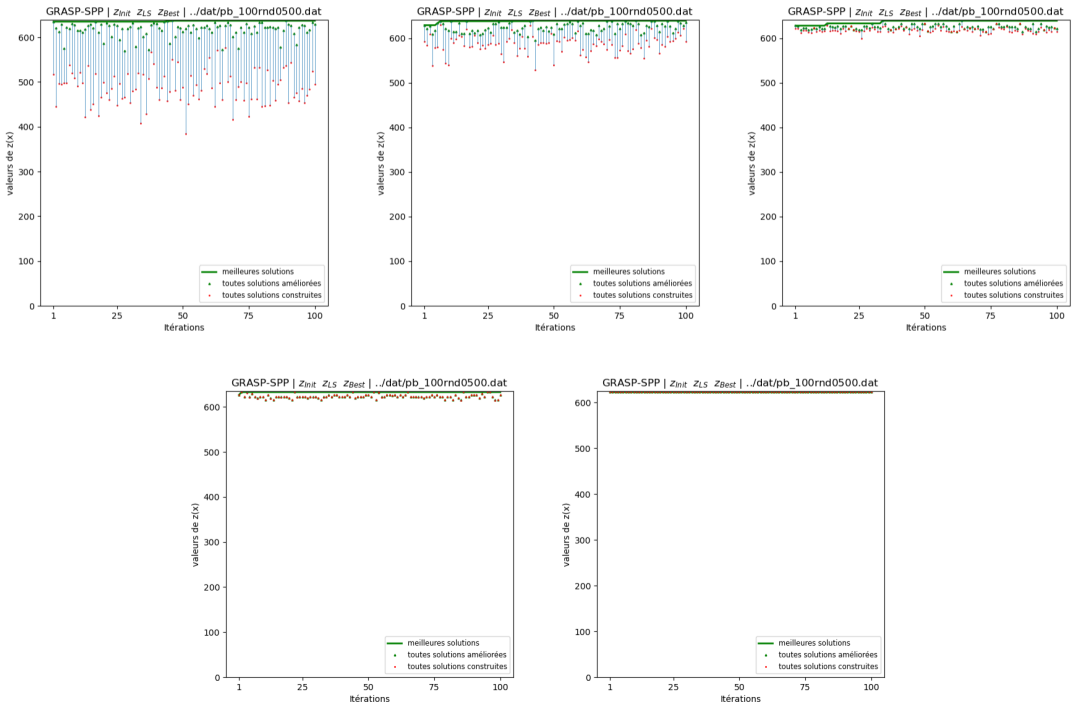

FIGURE 3 – Comparison changement d'alpha sur GRASP

Paramètres pour test sur best value :

- alpha = 0,8
- itérations = 5

Fichier	Solution trouvée	Temps (s)
pb_1000rnd0100.dat	56	17.461596
pb_1000rnd0800.dat	126	135.841186
pb_100rnd0500.dat	627	0.033512

pb_100rnd1200.dat	18	0.055464
pb_2000rnd0400.dat	20	1094.570787
pb_2000rnd0500.dat	119	68.136535
pb_200rnd0300.dat	682	5.037251
pb_200rnd1800.dat	14	0.377953
pb_500rnd0700.dat	987	42.822087
pb_500rnd1700.dat	164	6.648394

Si on change alpha à 0,3 par exemple :

Fichier	Solution trouvée	Temps (s)
pb_1000rnd0100.dat	40	32.844663
pb_1000rnd0800.dat	130	135.449871
pb_100rnd0500.dat	633	0.083169
pb_100rnd1200.dat	18	0.058411
pb_2000rnd0400.dat	21	1075.133779
pb_2000rnd0500.dat	108	137.884737
pb_200rnd0300.dat	672	12.449107
pb_200rnd1800.dat	13	0.356583
pb_500rnd0700.dat	1023	159.023518
pb_500rnd1700.dat	145	9.482767

Si l'on augmente le déterminisme, GRASP n'explorera pas grand-chose en dehors des résultats trouvés, au contraire, avec trop d'aléatoire, il explorera trop loin. Il est nécessaire de doser correctement alpha en fonction du type de problème.

Ex alpha sur pb_200rnd0300 :
— alpha 1 : 639 en 2 secondes
— alpha 0 : 678 en 24 secondes
En conclusion, comme nous pouvons le voir sur le graphique, plus nous augmentons la valeur alpha, moins il y aura d'exploration. C'est pourquoi avoir un alpha faible prend plus de temps à s'exécuter.

Expérimentation numérique de Path-Relinking

Paramètres de test pour les graphiques :
— nbInstances = 2
— nbRunGrasp = 3
— nbIterationGrasp = 100
— nbDivisionRun = 10
— alpha = [0.0, 0.25, 0.5, 0.75, 1.0]

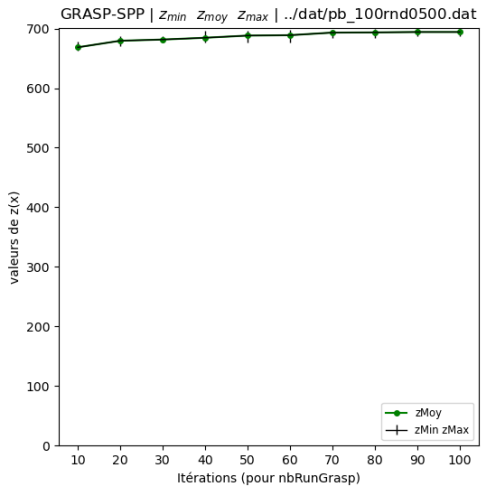

FIGURE 4 – Bilan sur tous les run GRASP+PR

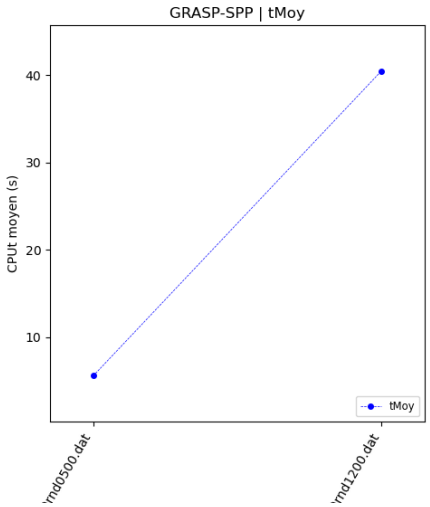

FIGURE 5 – CPU temps d'execution GRASP+PR

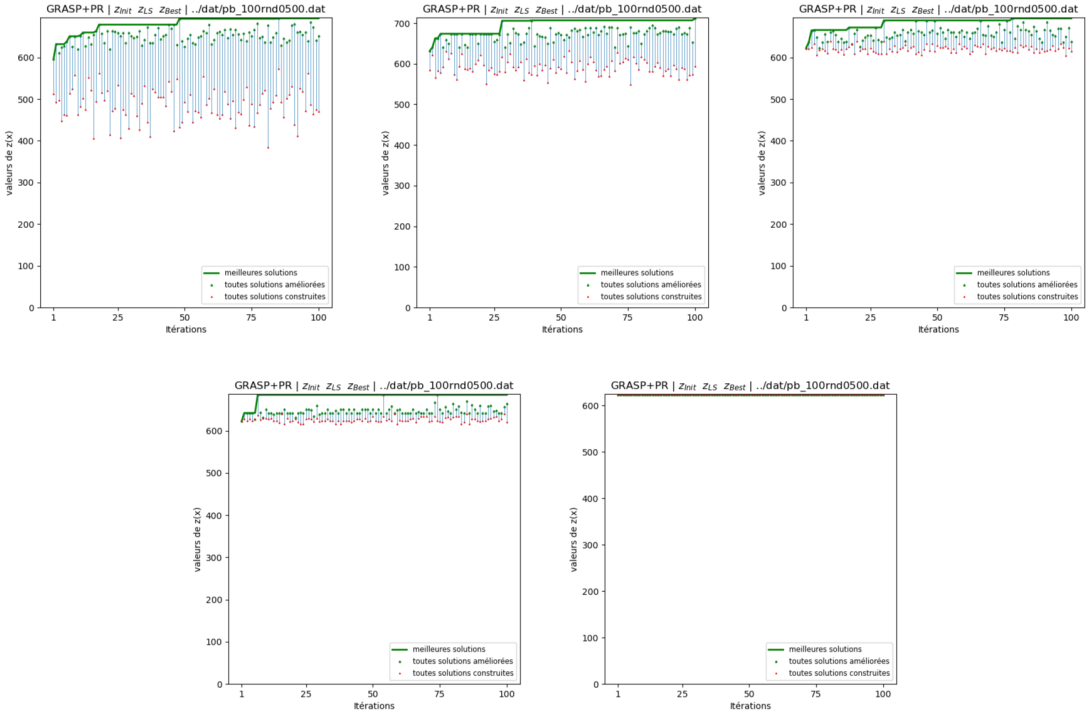

FIGURE 6 – Comparison changement d'alpha sur GRASP+PR

Paramètres pour les tests :

- alpha = 0,8
- itérations = 5

Fichier	zmin	zmoy	zmax	Temps (s)
pb_1000rnd0100.dat	37	52.0	60	55.17
pb_100rnd0500.dat	620	643.0	648	0.19
pb_100rnd1200.dat	13	19.2	22	1.41
pb_200rnd0300.dat	643	735.4	810	82.71
pb_200rnd1800.dat	11	16.2	18	8.87
pb_500rnd1700.dat	98	166.2	211	53.92

Le Path Relinking améliore considérablement les solutions trouvées par GRASP, mais prend beaucoup plus de temps, c'est pourquoi je n'ai pas les résultats pour tous les cas.

Discussion

Le GRASP seul approche déjà une solution optimale dans un temps relativement rapide, en veillant à bien paramétrer le nombre d'itérations et α. Avec le Path Relinking, nous observons une amélioration notable, au détriment du temps de calcul qui reste cependant acceptable si on le compare à Jump, par exemple. Le Path Relinking, avec un nombre d'itérations adéquat, peut vraiment donner d'excellents résultats, très proches des optima connus.

Comme nous pouvons le voir sur le graphique, le comportement de l'alpha variable pour le GRASP+PR est similaire au GRASP, mais avec une meilleure exploration et un amélioration de la valeur optimale.

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

Présentation succincte des choix de mise en œuvre de ACO à GRASP appliquée au SPP

Pour cette étape on a choisi d'implémenter **l'Algorithme de colonies de fourmis (ACO)**, un algorithme inspirés du comportement des fourmis.

Le paradigma :

1. une fourmi parcourt plus ou moins au hasard l'environnement autour de la colonie
2. si celle-ci découvre une source de nourriture, elle rentre plus ou moins directement au nid, en laissant sur son chemin une piste de phéromones
3. Les fourmis attirées par les phéromones auront tendance à suivre, plus ou moins directement, la piste ainsi tracée
4. en revenant au nid, ces mêmes fourmis vont renforcer la piste
5. si deux pistes sont possibles pour atteindre la même source de nourriture, celle étant la plus courte sera, dans le même temps, parcourue par plus de fourmis que la longue piste
6. la piste courte sera donc de plus en plus renforcée, et donc de plus en plus attractive
7. la longue piste, elle, finira par disparaître, les phéromones étant volatiles
8. à terme, l'ensemble des fourmis a donc déterminé et « choisi » la piste la plus courte

Les variables :

- `num_ants` : nombre de fourmis utilisées à chaque itération pour construire des solutions candidates.
- `num_iter` : nombre total d'itérations de l'algorithme ACO.
- `alpha` : paramètre qui contrôle l'influence des phéromones lors de la sélection des éléments par les fourmis. Plus α est grand, plus les phéromones guideront fortement le choix.
- `beta` : paramètre qui contrôle l'influence de la qualité heuristique (ici, le coût $C[i]$) sur le choix des éléments par les fourmis. Plus β est grand, plus la sélection favorise les éléments de coût élevé.
- `rho` : taux d'évaporation des phéromones à chaque itération. Détermine la vitesse à laquelle l'information historique est oubliée.
- `Q` : facteur de renforcement des phéromones, proportionnel à la qualité de la solution trouvée.

Algorithm 5 WeightedChoice

if A est vide **then**
    erreur : ensemble vide
Remplacer tout poids négatif ou non fini par 0
$W \leftarrow \sum w$
if $W = 0$ **then**
    retourner un élément choisi uniformément dans A
Calculer la somme cumulative : $cw_i = \sum_{j \leq i} \frac{w_j}{W}$
Tirer un nombre aléatoire $r \sim U(0, 1)$
Trouver le plus petit i tel que $cw_i \geq r$
retourner $A[i]$

Algorithm 6 ConstructSolutionAnt

$x \leftarrow$ vecteur de 0 (solution)
$U \leftarrow$ ensemble des lignes non encore couvertes
$Cnd \leftarrow$ ensemble des colonnes disponibles
while $Cnd \neq \emptyset$ **do**
    $F \leftarrow \emptyset$ ▷ colonnes faisables
    $W \leftarrow \emptyset$ ▷ poids
    for chaque colonne $i \in Cnd$ **do**
        if i ne couvre aucune ligne déjà utilisée **then**
            Ajouter i à F
            $\eta_i \leftarrow \max(10^{-9}, C[i])$ ▷ visibilité
            Ajouter $(\tau[i]^{\alpha} \cdot \eta_i^{\beta})$ à W
    if F est vide **then**
        break
    $c \leftarrow \text{WeightedChoice}(F, W)$
    Marquer toutes les lignes couvertes par c comme utilisées
    $x[c] \leftarrow 1$
    Retirer c de Cnd
retourner x

Algorithm 7 ACO_SPP

Require: Coûts C, matrice A, nombre de fourmis m, itérations T, paramètres α, β, ρ, Q
Initialiser les phéromones $\tau_i \leftarrow 1$ pour tout i
$x^{best} \leftarrow$ solution nulle
$z^{best} \leftarrow -\infty$
for $t = 1$ **à** T **do**
    $S \leftarrow \emptyset$ ▷ solutions des fourmis
    $Z \leftarrow \emptyset$ ▷ valeurs
    for chaque fourmi $k = 1..m$ **do**
        $x^k \leftarrow \textsc{ConstructSolutionAnt}(C, A, \tau, \alpha, \beta)$
        if local search activée **then**
            $x^k \leftarrow \textsc{LocalSearch}(x^k)$
        $z^k \leftarrow \sum_i C[i] \cdot x^k[i]$
        Ajouter x^k à S et z^k à Z
        if $z^k > z^{best}$ **then**
            $x^{best} \leftarrow x^k$
            $z^{best} \leftarrow z^k$
    ▷ Évaporation
    $\tau_i \leftarrow (1 - \rho) \cdot \tau_i$ pour tout i
    ▷ Dépôt de phéromones des fourmis
    for chaque solution x^k **do**
        if $z^k > 0$ **then**
            $\Delta\tau \leftarrow Q \cdot z^k / \sum C$
            for chaque i tel que $x^k[i] = 1$ **do**
                $\tau_i \leftarrow \tau_i + \Delta\tau$
    ▷ Dépôt élitiste
    $\Delta\tau^{elite} \leftarrow 0.5 \cdot Q \cdot z^{best} / \sum C$
    for chaque i tel que $x^{best}[i] = 1$ **do**
        $\tau_i \leftarrow \tau_i + \Delta\tau^{elite}$
retourner (x^{best}, z^{best})

Déroulement de la première itération

Initialisation

— Nombre de colonnes : $n = 9$, nombre de lignes : $m = 7$.

— Coûts : $C = [10, 5, 8, 6, 9, 13, 11, 4, 6]$.
— Matrice d'incidence A :
  — $A_1 = \{1, 2, 3, 5, 7, 8\}$, $A_2 = \{2, 3, 8\}$, $A_3 = \{2, 3, 8\}$,
  — $A_4 = \{2, 5, 6, 8, 9\}$, $A_5 = \{4\}$, $A_6 = \{1, 3, 5, 6, 9\}$,
  — $A_7 = \{2, 3, 7, 9\}$, $A_8 = \{1, 4, 5, 8, 9\}$, $A_9 = \{?\}$
— Phéromones initiales : $\tau_i = 1$ pour $i = 1..9$.
— Solution courante : $\mathbf{x} = [0, 0, 0, 0, 0, 0, 0, 0, 0]$.
— Lignes couvertes : $U = \emptyset$.
— Colonnes disponibles : $Cnd = \{1, 2, \ldots, 9\}$.

Choix de la première colonne par la fourmi

— **Colonnes faisables** : $F = \{i \in Cnd \mid i$ couvre au moins une ligne non couverte$\}$. Supposons que toutes les colonnes sont faisables au départ : $F = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
— **Calcul des probabilités** :

$$p_i \propto (\tau_i)^\alpha \cdot (C[i])^\beta$$

avec $\alpha = 1$, $\beta = 2$. Exemple de calcul (approximatif) :

$$p_1 \propto 1 \cdot 10^2 = 100, \quad p_2 \propto 1 \cdot 5^2 = 25, \ldots$$

— **Sélection aléatoire selon les probabilités** : supposons que la colonne choisie est $i = 1$.
— **Mouvement** : ajouter la colonne 1 à la solution.
— **Nouvelle solution x** :

$$\mathbf{x} = [1, 0, 0, 0, 0, 0, 0, 0, 0]$$

(soit $\{e_1\}$).
— **Mise à jour des lignes couvertes** : $U \leftarrow U \cup A_1 = \{1, 2, 3, 5, 7, 8\}$.
— **Colonnes disponibles** : retirer la colonne 1 et toutes celles qui ne sont plus faisables à cause de lignes déjà couvertes.

Calcul de l'objectif et mise à jour des phéromones

— **Valeur de la solution** : $z = \sum_i C[i]x[i] = 10$.
— **Mise à jour des phéromones** :
  — Évaporation : $\tau_i \leftarrow (1-\rho)\tau_i$, ici $\rho = 0.1$, donc $\tau_i \leftarrow 0.9 \cdot 1 = 0.9$ pour toutes les colonnes.
  — Dépôt par la solution : pour les colonnes sélectionnées ($i = 1$) :

$$\tau_1 \leftarrow \tau_1 + \Delta\tau_1, \quad \Delta\tau_1 = \frac{Q \cdot z}{\sum C} = \frac{1 \cdot 10}{\sum C}$$

  — Dépôt élitiste : idem pour la meilleure solution globale (ici même que la solution courante).
— **Nouvelle valeur des phéromones** :

$$\tau = [\tau_1, \tau_2, \ldots, \tau_9] \approx [0.9 + \Delta\tau_1, 0.9, \ldots, 0.9]$$

Fin de la première itération

— Solution courante : $\mathbf{x} = \{e_1\}$.
— Valeur : $z = 10$.
— Phéromones mises à jour : $\tau_1 > 0.9$, les autres $\tau_i = 0.9$.

Expérimentation numérique comparative GRASP vs ACO

Paramètres de test pour les graphiques :

— nbInstances = 10
— nbRunGrasp = 30
— nbIterationGrasp = 100
— nbDivisionRun = 10
— alpha = [0.0, 0.5, 1.0]
— beta=[0.0, 1.0, 2.0]
— num_ants=30
— rho=0.1
— Q=1.0

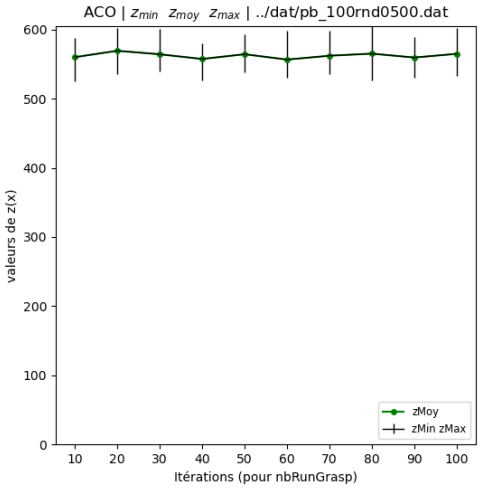

FIGURE 7 – Bilan sur tous les run d'ACO

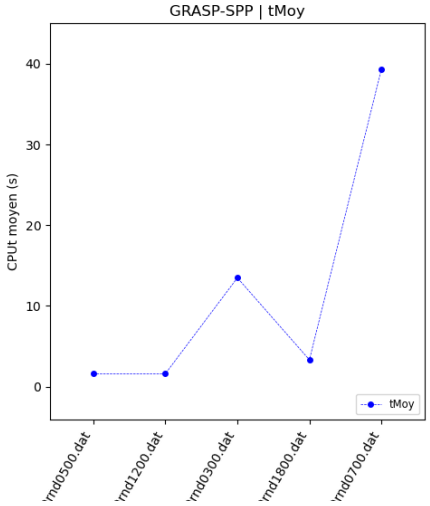

FIGURE 8 – CPU temps d'execution d'ACO

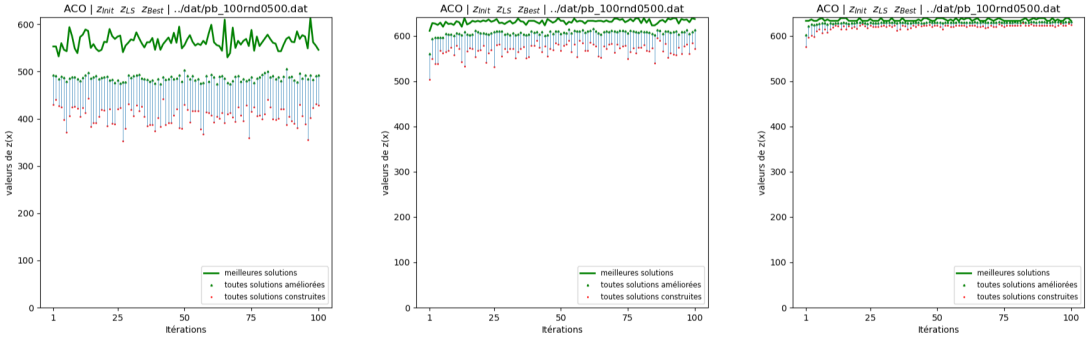

FIGURE 9 – Comparison changement d'alpha et Beta sur ACO

Les parametres :
— Alpha (α) :
  — Bas (0) : Recherche aléatoire
  — Haut (2+) : Stagnation rapide.

— Beta (β) :
  — Bas (0) : Convergence lente.
  — Haut (5+) : Comportement glouton.

— Évaporation (ρ) :
  — Bas (0.01) : Mémoire trop longue.
  — Haut (0.5+) : "Amnésie", l'algorithme n'apprend pas.

Fichier	Solution trouvée	Temps (s)
pb_1000rnd0100.dat	67.0	19.77

pb_1000rnd0800.dat	168.0	135.58
pb_100rnd0500.dat	639.0	1.00
pb_100rnd1200.dat	23.0	1.02
pb_2000rnd0400.dat	26.0	389.48
pb_2000rnd0500.dat	126.0	27.73
pb_200rnd0300.dat	722.0	8.37
pb_200rnd1800.dat	18.0	1.98
pb_500rnd0700.dat	1127.0	26.58
pb_500rnd1700.dat	182.0	6.64

Discussion

Après avoir exécuté les deux métaheuristiques sur plusieurs instances, on peut tirer les observations suivantes :

Qualité des solutions :

GRASP+PR produit de bonnes solutions dès les premières itérations, tandis qu'ACO les explore de manière plus progressive et nécessite donc un plus grand nombre d'itérations pour atteindre la solution optimale.

Variabilité et robustesse :

L'ACO présente une plus grande variabilité de solutions pour différentes exécutions, aussi en raison des paramétrages possibles.

Temps de calcul :

GRASP+PR est plus rapide sur les tailles moyennes, tandis que le temps d'exécution d'ACO augmente avec le nombre de fourmis, mais globalement, pour les meilleures solutions, ACO est l'algorithme qui prend le moins de temps.

On peut donc dire que GRASP + PR est généralement plus rapide et robuste pour obtenir des solutions de haute qualité sur des instances variées, tandis que l'ACO est plus exploratoire et paramétrable, ce qui peut permettre d'atteindre de meilleures solutions sur des instances spécifiques avec un temps de calcul suffisant.

Note : *Mon travail a été accompagné par des systèmes LLM pour les traductions, la mise en forme LaTeXde certaines étapes/formules, et l'écriture des algorithmes. Dans mon cas, cela m'a été absolument utile comme* ***support didactique****, m'aidant à traduire du pseudocode en Julia ou pour l'explication de fonctions / les illustrations étape-par-étape de passages qui m'étaient peu clairs. Même lorsque les réponses n'étaient pas exactes, elles ont tout de même amélioré une recherche de solution en ligne.*