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.