Laboratoire 4 - Métaheuristiques de dégradation

Exercice 1 - Explication de concepts

Expliquez brièvement les concepts suivants.

1. La connectivité d'un voisinage. Pourquoi est-ce une propriété intéressante à avoir ?
2. L'objectif principal d'une métaheuristique.
3. Les différences entre une métaheuristique de dégradation, de construction, et de voisinage.
4. Le fonctionnement d'un mécanisme de mémoire dans une métaheuristique. Donnez un exemple d'utilisation de ce mécanisme.
5. Le fonctionnement d'une population dans une métaheuristique. Donnez un exemple d'utilisation de ce mécanisme.

Exercice 2 - Problème du voyageur de commerce

On considère le problème du voyageur de commerce (TSP). Soit un ensemble de n lieux. Les distances $d_{i,j}$ entre les lieux sont toutes connues. On définit la variable $x_i \in \{1,\dots,n\}$ qui vaut k si le lieu k est le i^e lieu visité. Le modèle est définit comme suit.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n-1} d_{x_i,x_{i+1}} + d_{x_n,x_1} \\ \text{subject to} \quad & \texttt{allDifferent}(x) \\ & x_j \in \{1,\dots,n\} \quad \forall j \in \{1,\dots,n\} \end{aligned}$$

Ainsi, on minimise la distance totale du circuit (sans oublier le retour au point de départ), de sorte qu'aucun lieu ne soit visité deux fois. On souhaite résoudre ce problème avec une recherche locale simple. L'espace de recherche est constitué de l'ensemble des tours valides (la contrainte `allDifferent` est dure), et la fonction de voisinage considérée est le `2-opt`.
Soit un tour de la la forme $[u_1,\dots,u_i,a_1,\dots,a_k,u_j,\dots,u_n]$, le mouvement `2-opt` entre les arêtes $(u_i \to a_1)$ et $(a_k \to u_j)$ va générer le tour $[u_1,\dots,u_i,a_k,\dots,a_1,u_j,\dots,u_n]$. Prouvez à l'aide d'un algorithme que ce voisinage est connecté.

Exercice 3 - Variante du problème de localisation d'entrepôts

Soit un ensemble I de clients à livrer et soit un ensemble J de sites sur lesquels il est possible d'installer des entrepôts. Contrairement à la version vue dans le cours, on rajoute un paramètre binaire $a_{i,j}$ qui vaut 1 si le client $i \in I$ peut être livré depuis le site $j \in J$ et 0 sinon. Une formulation mathématique possible pour ce problème est la suivante. On définit des variables binaires $x_{i,j}$ valant 1 si le client $i \in I$ est affecté à l'entrepôt $j \in J$ et y_j valant 1 si l'entrepôt $j \in J$ est ouvert. Le modèle est définit comme suit.

$$\begin{aligned} \text{minimize} \quad & \sum_{j \in J} y_j \\ \text{subject to} \quad & \sum_{j \in J} x_{ij} = 1 \qquad \forall i \in I \\ & x_{ij} \le y_j \qquad \forall i \in I, j \in J \end{aligned}$$

1. Proposez un espace de recherche afin de résoudre ce problème via une recherche locale.
2. Soit une fonction de voisinage consistant à échanger les valeurs de deux entrepôts. Ce voisinage est-il connecté ? Prouvez-le ou donnez un contre-exemple.
3. Soit une fonction de voisinage consistant à changer la valeur d'un entrepôt. Ce voisinage est-il connecté ? Prouvez-le ou donnez un contre-exemple.

Exercice 4 : Problème de placement de machines - Implémentation

Dans cet exercice, on s'intéresse à un problème de placement de machines. Soit I un ensemble de jobs, soit J un ensemble de sites dans un atelier, et soit M un ensemble de machines. Chaque job doit passer sur un certain nombre connu de machines, dans un ordre également connu. A chaque job est donc associé un sous-ensemble ordonné de M. Notons également que chaque job démarre à la première machine qui doit la traiter, passe sur les machines qui lui sont assignées, puis termine son parcours à la dernière machine qu'elle visite. L'objectif est de placer les machines de M sur les sites de J de façon à minimiser la distance totale parcourue par les jobs. Les distances entre chaque paire de sites sont toutes connues. Un jeu de 4 instances vous est donné. Une instance du problème est représentée par un fichier de $4+M+J$ lignes ayant la forme suivante.

```
J
M

d(0,0) d(0,1) ... d(0,M)
d(1,0) d(1,1) ... d(1,M)
...
d(M,0) d(M,1) ... d(M,M)

o[0]
o[1]
...
o[J]
```

De façon similaire, une solution est représentée par un fichier 2 lignes, où $s[m]$ est le site associé à la machine m.

```
C
s[0], s[1], ..., s[M]
```

Cet exercice a pour objectif de vous montrer une implémentation concrète des métaheuristiques de dégradation vues au cours (*simulated annealing*, recherche tabou, et algorithmes génétiques). Tous les codes vous sont fournis, à l'exception de celui associé à deux scénarios, où vous serez invités à expérimenter plusieurs choix de conception vus au cours. Pour les autres scénarios, il vous est demandé de comprendre l'implémentation et d'expérimenter avec. Notez que ces implémentations comportent plusieurs sources d'inefficacités, que ce soit au niveau des choix de conception, de l'équilibre diversification/intensification, et de l'implémentation. Un de vos objectifs est d'apporter un regard critique sur ces implémentations et d'en déceler les possibilités d'amélioration sur base des techniques vues au cours. Plusieurs fichiers vous sont fournis :

— `atelier.py` : définition du problème de placement de machines.
— `local_search.py` : implémentation d'une recherche locale pour le problème.

- `simulated_annealing.py` : implémentation d'un algorithme de *simulated annealing*.
- `tabu_search.py` : implémentation d'une recherche tabou non optimisée.
- `tabu_search_advanced.py` : implémentation de votre recherche tabou, sur base du code précédent. Vous pouvez par exemple rajouter un mécanisme de mémoire à moyen/long terme, un critère d'aspiration, une taille de liste dynamique, ou une version probabiliste.
- `genetic.py` : implémentation d'un algorithme génétique non optimisé.
- `genetic_advanced.py` : implémentation de votre algorithme génétique, sur base du code précédent. Vous pouvez par exemple essayer une autre fonction de sélection (roulette améliorée, tournoi, etc.), un autre opérateur de reproduction, une autre fonction de mutation, une autre façon de gérer la taille de votre population ou encore une intensification.

Vous avez également à votre disposition 4 fichiers d'instances nommés selon le schéma `atelier_J_M.txt`. Deux petites instances pour comprendre le code, un instance moyenne pour voir plus en détail l'influence des paramètres et enfin une grande instance pour les plus courageux.

Cet exercice est divisé en 7 scénarios que vous devez exécuter via la commande suivante en remplaçant *Si* par l'un des scénarios (**S1**, . . ., **S7**). Ces scénarios sont détaillés dans le code. Chaque méthode a finalement plusieurs paramètres que vous pouvez calibrer.

```
1    python3 main.py --scenario Si
```

Pour chaque scénario, faites attention à bien identifier les différentes étapes clés propres à chaque algorithme. Faites varier les paramètres et amusez vous !