

Optimisation combinatoire : application

Conditions d'évaluation : ce travail est à effectuer en équipe de 4 maximum, les noms de tous les participants seront renseignés sur le rapport. L'évaluation prendra en compte les points suivants :

1. la rédaction du rapport (langue française, structure, pertinence de l'analyse des résultats) ;
2. le travail de recherche (modélisation linéaire, nombre de méthodes, améliorations, ...);
3. la performance de la meilleure méthode (rang parmi les équipes), qui sera bien sûr vérifiée avec un lot inédit d'instances, et après lancement sur la machine du correcteur.

Évidemment, mais autant le mettre noir sur blanc, tout échange de code avéré entre deux (ou plusieurs) groupes annulera le travail rendu par les groupes concernés et déclenchera un conseil de discipline pour tous les étudiants concernés. Il en est de même pour tout utilisation de code extérieur (Internet, GPT, ...).

Conditions de rendu : le rendu doit prendre la forme d'une archive .zip contenant :

1. un rapport (format pdf), avec en entête les noms des membres du groupe, décrivant la modélisation linéaire, les résultats du solveur et les diverses approches algorithmiques étudiées ainsi que leurs résultats ;
2. les sources (.py) des différents algorithmes développés, testés sous Ubuntu 23.10 avec python 3.11, sans bibliothèque tierce sauf sur demande par mail, ainsi que les différents scripts (génération et lancement de lp) ;
3. le script doit se lancer en ligne de commande **exactement** de la manière suivante :

Lancement d'un algorithme

```
python3 main.py temps input.txt output.txt
```

avec temps de nombre de secondes alloués à l'exécution de votre meilleur algorithme, avec pénalisation du rang en cas de dépassement de 5 secondes, en considérant que les temps de chargement de l'instance, de complétion de la dernière itération et d'écriture de la solution seront absorbés par cette tolérance, input.txt le fichier d'instance à traiter, et output.txt le fichier qui contiendra la solution. Ce dernier fichier contiendra **uniquement** N nombres (0 ou 1), séparés par des espaces, qui correspondront aux valeurs d'instanciation des variables pour la solution fournie.

Ce travail doit être déposé sur Moodle au plus tard le vendredi 22 décembre 2023 à 18h.

1 Description du problème et des instances

Le but de cette activité est l'élaboration de méthodes d'optimisation dédiées à un problème d'organisation de soirée. On dispose d'un ensemble de N convives potentiels.

Pour chaque convive potentiel $i \in \{1, \dots, N\}$, on définit V_i l'ensemble de ses connaissances, avec $V_i \subseteq \{1, \dots, N\}$. En d'autres termes, pour tout couple $(i, j) \in \{1, \dots, N\}^2$, si i et j se connaissent, alors $i \in V_j$ et $j \in V_i$. On définira également, pour tout convive potentiel i , une valeur c_i représentant l'intérêt d'avoir cette personne à notre soirée. Pour une instance donnée, on cherchera à réunir le plus de personnes intéressantes possibles à condition que toutes se connaissent : l'objectif sera donc dépendant du coefficient d'intérêt c_i de chacun des convives invités. Par exemple, si i , j et k ont respectivement les coefficients $c_i = 2$, $c_j = 3$ et $c_k = 5$, alors inviter les trois augmentera le score de votre soirée de 10, score que l'on cherchera à maximiser.

Afin de pouvoir exploiter les outils tels que les solveurs, il nous est nécessaire d'établir le modèle linéaire. Une fois celui-ci défini, il sera possible de procéder à des tests, via un ensemble d'instances provenant de la communauté de chercheurs en optimisation combinatoire. Une instance donne une valeur à chaque constante du modèle (N , liens entre chaque convive), et dont le format est donné ci-dessous.

- La première ligne contient N et M deux entiers séparés par un espace, représentant respectivement le nombre de convives potentiels et le nombre de liens d'amitié.
- Les N lignes suivantes contiennent deux entiers séparés par un espace, représentant respectivement i et c_i .
- Les M lignes suivantes contiennent deux entiers i et j séparés par un espace, représentant deux convives potentiels se connaissant (i.e. $i \in V_j$ et $j \in V_i$).

L'objectif sera est de créer un script annexe prenant un fichier d'instance en entrée (ex : "instance1.txt"), et de générer en sortie le fichier ".lp" correspondant (voir section suivante) via une commande du type :

Lancement d'un algorithme

```
python3 conversion.py instance1.txt instance1.lp
```

Attention : les indices commencent à 0 dans le fichier ! Les convives sont donc numérotés de 0 à $N-1$

2 Solveur linéaire GLPK

Le logiciel libre GLPK est un solveur linéaire offrant de très bonnes performances, souvent au niveau de ILOG CPLEX et Gurobi, qui sont deux alternatives commerciales. Voici les procédures d'installation pour Ubuntu et pour Windows :

- dans le cas d'Ubuntu, le logiciel est dans le dépôts, et il y a de grandes chances que ce soit également le cas pour votre distribution favorite :

Installation des paquets

```
sudo apt install glpk-utils
```

- dans le cas de Windows, vous devrez télécharger l'archive via le lien ci-dessous, dans laquelle se trouve un dossier w64 contenant les exécutables et les DLLs nécessaires au fonctionnement du solveur :

Lien vers l'installateur

<https://sourceforge.net/projects/winglpk/>

Une fois le logiciel installé, son utilisation se limitera à la ligne de commande suivante :

Ligne de commande

```
glpsol --lp modele.lp --output solution.txt
```

2.1 Format de fichier lp

Le format lp est un standard permettant l'exécution d'un solveur sur un programme linéaire. D'une manière générale, le format lp est très proche de la manière dont nous avons écrit les programmes linéaires durant tout le cours. Le fichier est décomposée en sections, qui sont les suivantes :

- l'expression de l'objectif, précédée du mot clef Maximize ou Minimize en fonction de sens de l'optimisation ;
- l'expression de toutes les contraintes \geq , \leq ou $=$, précédée du mot clef Subject To ;
- la déclaration des restrictions des domaines de définition sur les variables, si nécessaire, et précédée du mot clef Bounds ;
- la déclaration des variables entières avec le mot clef Generals ;
- la déclaration des variables binaires avec le mot clef Binaries ;
- la conclusion du modèle via le mot clef End.

Si l'on essaie de traduire l'instance de sac à dos $I = (3, 2), (2, 2), (5, 6), (7, 5)$, chaque couple représentant un objet avec sa valeur et son poids, et $W = 8$, on obtient le fichier suivant :

Exemple de fichier

```
Maximize
  z: 3 x1 + 2 x2 + 5 x3 + 7 x4
Subject To
  poids: 2 x1 + 2 x2 + 6 x3 + 5 x4 <= 8
Binaries
  x1
  x2
  x3
  x4
End
```

Première constatation : on met des espaces entre les variables, constantes et opérateurs. De plus, seuls les opérateurs $+$ et $-$ sont autorisés, du fait de la nature linéaire du modèle. La multiplication du constante avec une variable se fait en précédant la variable de la constante, tout en les séparant d'un espace. Ensuite, l'objectif et chaque contrainte ont un nom **unique** : z, poids, ... Enfin les noms de variables, contraintes et objectifs sont des chaînes alphanumériques commençant par une lettre. Si l'on lance l'optimisation de ce

programme via GLPK, l'exécution est quasiment instantanée, et génère un fichier solution dont le début est donné ci-dessous :

Lecture du résultat

Problem:

Rows: 1

Columns: 4 (4 integer, 4 binary)

Non-zeros: 4

Status: INTEGER OPTIMAL

Objective: z = 10 (MAXimum)

No.	Row name	Activity	Lower bound	Upper bound
1	poids		7	8

No.	Column name	Activity	Lower bound	Upper bound	
1	x1	*	1	0	1
2	x2	*	0	0	1
3	x3	*	0	0	1
4	x4	*	1	0	1

Les nombres de lignes et de colonnes représentent respectivement le nombre de contraintes et de variables. Le statut de l'optimisation est ici "INTEGER OPTIMAL", indiquant que l'optimisation a été terminée avec succès, et on lit directement la valeur de la solution optimale, ici $z = 10$. Si l'on s'intéresse à la structure de la solution, le premier tableau énumère les valeurs des parties gauches des contraintes, alors que le deuxième tableau donne les valeurs affectées à chaque variables : ici, seules $x1$ et $x4$ sont à 1.

2.2 Résultats partiels

Instance	Solution optimale	Temps (s)	Mémoire (Mb)
instance1.txt	73	1272	291
instance2.txt	91	2119	397
instance3.txt	84	2588	457
instance4.txt	83	2080	415
instance5.txt	81	2082	390
instance6.txt			
instance7.txt			
instance8.txt			
instance9.txt			
instance10.txt			

TABLE 1 – Résultat de glpk sur les instances

Pour chaque instance, une fois le fichier lp obtenu, il faudra lancer l'optimisation de ce programme via GLPK. Afin de vérifier le bon fonctionnement de votre modèle et de votre script de traduction, vous trouverez ci-dessus les résultats des 5 premières instances.

Attention : il vous est demandé de fournir, dans votre rapport, le modèle linéaire (bien expliqué) ainsi que les résultats de glpk, ou Gurobi, sur les 10 premières instances. Faire un script shell et lancer le travail sur la machine pendant la nuit peut être une bonne idée! Cependant, si vous n'arrivez pas à fournir la solution optimale pour une instance (temps d'exécution trop long), vous pouvez communiquer la valeur de la solution trouvée au bout d'une heure ainsi que la valeur de la borne supérieure.

3 Méthodes approchées

Nous travaillons sur un problème prouvé NP-difficile. Cela implique que n'importe quel algorithme de résolution exacte sera d'une complexité au moins exponentielle. En pratique, cela se traduit par une explosion du temps de calcul lors de l'augmentation linéaire du nombre de variables. En d'autres termes, à partir d'une certaine taille d'instances, les solveurs tels que GLPK seront inutilisables.

La question est alors la suivante : comment générer une solution de bonne qualité pour notre problème, peu importe la taille de l'instance considérée? La réponse réside dans l'étude des algorithmes d'optimisation approchée. Leur but est de s'approcher le plus possible d'une solution optimale, via des mécanismes de convergence ou des méthodes de construction intelligentes.

Remarque : vous pouvez traiter cette partie sans la précédente, il n'est pas nécessaire d'avoir un modèle linéaire pour procéder à l'optimisation des instances fournies avec des métaheuristiques.

3.1 Description de l'algorithme glouton

L'algorithme glouton est un algorithme simpliste. Son but est de construire une solution au problème, et de s'arrêter lorsqu'elle est terminée. Pour notre problème, la construction consiste en l'ajout de convives à une solution tant que certains convives non invités sont éligibles (i.e. ils connaissent tous les convives déjà invités). Le squelette de l'algorithme est donné par l'algorithme 1 :

Algorithme 1 – Algorithme glouton

```
Solution  $S \leftarrow \{\}$ 
Entier  $C \leftarrow$  Tous les convives
Tant que  $C \neq \emptyset$  faire
    Entier  $i \leftarrow$  Heuristique( $C$ )
     $S \leftarrow S \cup \{i\}$ 
    Mettre à jour  $C$ 
Fin Tant que
```

Si l'algorithme en lui-même est simpliste, l'intelligence, et donc la qualité de la solution en sortie, dépendra de l'heuristique de choix du convive à insérer dans la solution à chaque itération. L'heuristique implémente ici le **critère de choix**, c'est à dire le fait de régler une variable de décision à 1, en considérant l'état de la solution actuelle. Une heuristique viable peut être de choisir la variable ayant le score maximum, score lui-même défini par $c_i * |V_i|$, c'est à dire la coefficient d'intérêt de i multiplié par la taille de son groupe d'amis.

Afin de mesurer l'efficacité de notre algorithme, nous calculerons pour chaque instance le **gap** entre la valeur de la solution calculée et la valeur de la solution optimale donnée

par GLPK sur les différentes instances.

$$Gap(S) = \frac{f(S^*) - f(S)}{f(S^*)} \quad (1)$$

Sachant que notre problème est un problème de maximisation, notre mesure du gap ne peut être négative : on ne peut en effet pas trouver de solution réalisable dont le score est supérieur à l'optimal.

Instance	Score	Gap	Instance	Score	Gap
instance1.txt	69	0.05	instance6.txt	???	???
instance2.txt	90	0.01	instance7.txt	???	???
instance3.txt	71	0.15	instance8.txt	???	???
instance4.txt	67	0.19	instance9.txt	???	???
instance5.txt	80	0.01	instance10.txt	???	???

TABLE 2 – Résultat de l'algorithme glouton sur les instances

Les résultats des expérimentations sont visibles dans le tableau 2. Le premier constat est que notre algorithme glouton ne trouve jamais la solution optimale pour les instances 1 à 5. De plus, si pour les instances 2 et 5, on observe un écart très faible (de l'ordre de 1%), on observe également un écart allant jusqu'à 19% sur les autres instances. Cet algorithme est donc peu fiable, et, en fonction de l'instance traitée, risque de produire des résultats médiocre.

3.2 Description de l'algorithme génétique

Cette section a deux objectifs : vous permettre de voir comment adapter un algorithme génétique à notre problème, mais également comment présenter les modalités d'adaptation, les procédures de test et les résultats d'expérimentations dans votre rapport.

L'algorithme génétique fait partie de la famille des **métaheuristiques**. Il s'agit d'algorithmes dont les principes d'optimisation sont facilement adaptables à la plupart des problèmes. De plus, il est possible d'améliorer considérablement leurs performances via des modifications plus ou moins avancées (hybridation, contrôle dynamique des paramètres, ...). Commençons par étudier le principe général de l'algorithme génétique.

Il s'agit d'un algorithme évolutionnaire : le principe est de faire évoluer une population d'individus en reprenant les fondamentaux de la théorie de l'évolution (sélection et mutation). Ici, un individu représente une solution réalisable de notre instance. D'un point de vue codage, on pourra considérer qu'une solution est une structure contenant entre autres le tableau des valeurs des variables de décisions. Le chromosome de l'individu est donc un tableau de N booléens, chacun des octets pouvant être réglé soit à 0, soit à 1.

3.2.a Initialisation de la population

L'initialisation de la population est une étape extrêmement importante : il s'agit de générer des solutions de qualité acceptable afin de permettre une convergence suffisamment rapide, tout en gardant une diversité conséquente afin de guider la recherche dans plusieurs zones de l'espace des solutions. Soit P la population, T le nombre d'individus à générer, on peut considérer l'algorithme suivant :

Algorithme 2 – Initialisation de la population

```

Solution  $S_0 \leftarrow \text{glouton}(I)$ 
 $P \leftarrow P \cup \{S_0\}$ 
Pour  $i$  allant de 1 à  $T-1$  faire
    Solution  $S_i \leftarrow \text{gloutonRandomise}(I)$ 
     $P \leftarrow P \cup \{S_i\}$ 
Fin Pour

```

Commençons par remarquer que la première solution est générée via l'algorithme glouton défini dans la section précédente. Le but est d'introduire dans la population une solution de bonne qualité. Les $T - 1$ solutions suivantes sont quant à elles générées par un algorithme glouton randomisé qui, au détriment de la qualité de la solution fournie, va générer des solutions différentes à chaque appel, afin d'augmenter la diversité de notre population.

3.2.b Sélection pour la reproduction

A chaque itération, sur une population P de taille T , on sélectionne $T' \leq T$ individus pour la reproduction (avec T' pair). Cette sélection doit se faire selon la valeur des solutions, c'est à dire la valeur de la fonction objectif. Afin de ne pas sélectionner toujours les mêmes individus, il est recommandé d'ajouter un peu d'aléatoire afin d'accroître la diversité, en définissant une probabilité de choix d'un individu de la manière suivante :

$$P(S_i) = \frac{f(S_i)}{\sum_{S_j \in P} f(S_j)} \quad (2)$$

Étant donné qu'il s'agit d'un problème de maximisation, plus la valeur de $f(S_i)$ est élevée, plus la solution est de bonne qualité. On veut donc favoriser les solutions avec une valeur $f(S_i)$ élevée au détriment des autres.

3.2.c Croisement de deux individus

Une fois votre sélection de T' individus terminée, il faut maintenant les organiser en couple. Chaque couple d'individus générera deux enfants. Il existe un grand nombre d'opérateurs de croisement dans la littérature, nous utiliserons pour commencer le croisement en un point. Soient P_1 , P_2 , E_1 et E_2 respectivement les deux parents et les deux enfants. Au début de notre croisement, les parents sont définis (on connaît les valeurs composant leurs chromosomes), et les enfants sont vides.

En fonction d'une probabilité de croisement p_c généralement comprise entre 0.7 et 1, il sera décidé au début du croisement s'il faut simplement recopier P_1 et P_2 dans E_1 et E_2 (probabilité faible), ou bien procéder à un croisement (probabilité élevée). Dans le deuxième cas, on génère un point de croisement K aléatoirement entre 1 et N , N étant la taille du chromosome. Tous les gènes numérotés de 1 à K sont copiés du parent P_1 (respectivement P_2) à l'enfant E_1 (respectivement E_2). Sur la Figure 1, on peut voir l'exemple de la première phase de croisement, où la taille de chromosome est $N = 7$ et le point de croisement est $K = 4$.

Lors de la deuxième et dernière phase, tous les gènes numérotés de $K + 1$ à N sont copiés du parent P_1 (respectivement P_2) à l'enfant E_2 (respectivement E_1). Sur la Figure 2, on peut voir la complétion du croisement faisant suite à l'exemple de la Figure 1.

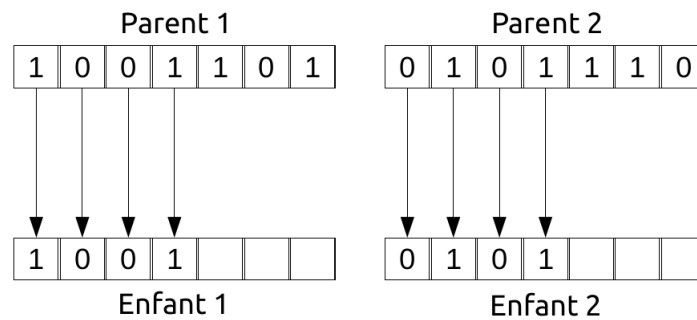


FIGURE 1 – Première phase du croisement

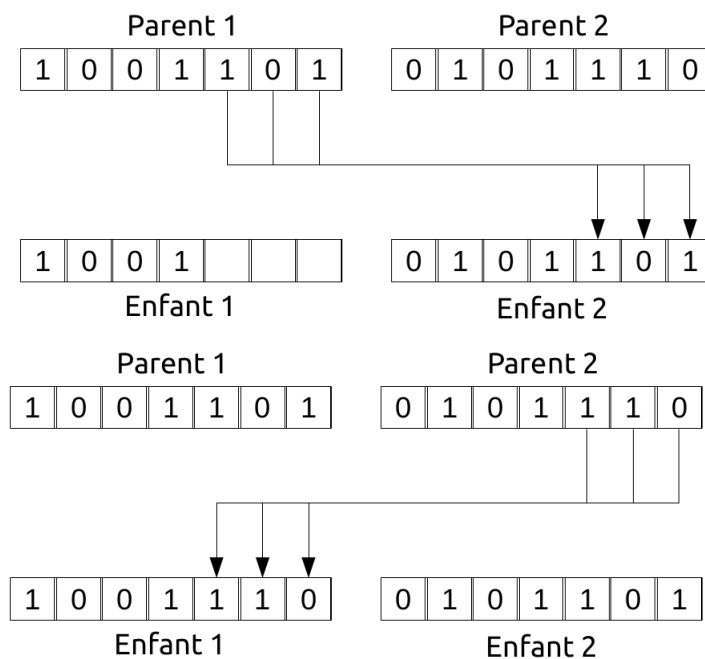


FIGURE 2 – Deuxième phase du croisement

Il s'agit là d'un opérateur parmi une multitude : on peut procéder à un croisement à deux points, multi-points, ou chercher d'autres stratégies plus intelligentes de découpage.

3.2.d Mutation

L'opérateur de mutation est appliqué à chaque solution générée via l'opérateur de croisement. Le but de cet opérateur est d'ajouter une perturbation afin d'éviter un manque de diversité au sein de la population après quelques générations. En effet, si un individu représente une solution trop avantageuse, celle-ci va invariablement finir par prendre le pas sur toute la population. Au final, tous les individus de la population seront des copies, et plus aucune convergence ne sera possible via les croisements. Au vu de la représentation de la solution sous forme de chromosome, l'opérateur de mutation est simple à mettre en place. Il prend la forme d'une boucle itérant de 1 à N , et, pour chaque gène i , détermine via un tirage aléatoire si celui-ci doit muter ou rester inchangé. Comme chaque gène est représenté par une valeur binaire, la mutation consiste simplement à inverser celle-ci.

L'opérateur de mutation prend pour paramètre une solution nouvellement créée, et la

probabilité de mutation. La valeur de celle-ci déterminera l'efficacité de l'algorithme :

- Si elle est trop élevée, les impacts des mutations casseront toute la convergence amenée par la sélection et le croisement. En d'autres termes, les enfants s'éloigneront trop des parents pour profiter de leurs qualités. Ainsi, la convergence de l'algorithme sera aléatoire.
- Si elle est trop faible, les enfants ressembleront trop aux parents pour se dégager d'un éventuel **optimum local**. Au final, toutes les solutions se ressembleront et la zone de l'espace des solutions explorées sera faible.

Dans la littérature, il est recommandé d'utiliser une probabilité de mutation de l'ordre de :

$$p_m = \frac{1}{N}$$

Ainsi, pour une solution, l'espérance du nombre de gènes mutés est de 1.

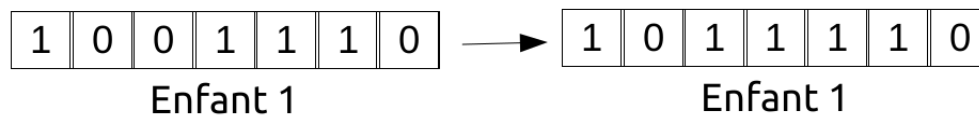


FIGURE 3 – Mutation du gène 3

3.2.e Réparation

Après croisement et mutation, il est courant que la solution générée ne soit pas réalisable. Il s'agit maintenant de vérifier que les contraintes de notre problème soient bien respectées, et corriger la solution si besoin via un opérateur de réparation :

1. on commence par affecter un score de "problème" par convive i , défini simplement par le nombre de convives qu'il ne connaît pas dans le groupe ;
2. on enlève aléatoirement des convives avec un score non nul (on définira un tirage aléatoire intelligent, en fonction du score), jusqu'à arriver à une solution réalisable ;
3. on ajoute des convives à la manière d'un glouton jusqu'à ce que ce ne soit plus possible.

A la fin, vous obtenez une solution réalisable.

3.2.f Sélection pour la survie

Une fois les croisements, mutations et réparations effectuées, notre population compte $T + T'$ individus. Afin de garder un temps d'exécution constant à chaque itération, il est nécessaire de réduire celle-ci à T individus. On procède donc à une nouvelle sélection, qui peut être soit déterministe, soit randomisée :

- dans le premier cas, on trie les individus par ordre croissant de valeur de fonction objectif, et on garde les T meilleurs,
- dans le deuxième cas, on procède à une sélection inspirée de celle pour la reproduction, afin de garder une certaine diversité.

A vous de déterminer laquelle utiliser au cours de vos expérimentations. Pour cela, vous pouvez mesurer la diversité moyenne de votre population à chaque itération via la formule suivante :

$$d = \frac{2}{N \times T \times T - 1} \sum_{i=1}^{T-1} \sum_{j=i+1}^T \sum_{k=1}^N |S_i(k) - S_j(k)|$$

Où d représente la mesure de la diversité, entre 0 et 1, S_i et S_j représentent deux solutions, et $S_i(k)$ représente la valeur du k -ème gène de S_i . Si la diversité de votre population est trop proche de zéro, vous pouvez agir sur la probabilité de mutation ainsi que sur les stratégies de sélection.

3.2.g Synthèse de l'algorithme

Algorithme 3 – Algorithme génétique

Entrées. Instance I

Réel p_c

Réel p_m

Entier T

Entier T'

Entier IterMax

Sortie. Entier f_{best}

Solution[] $P \leftarrow \text{initialisationPopulation}(I, T)$

Entier $f_{best} \leftarrow \max_{S_i \in P}(f(S_i))$

Pour i allant de 1 à IterMax **faire**

Solution[] $M \leftarrow \text{selectionReproduction}(P, T')$

Solution[] $C \leftarrow \text{croisement}(M, p_c)$

$C \leftarrow \text{mutation}(C, p_m)$

$C \leftarrow \text{reparation}(C, I)$

$P \leftarrow P \cup C$

Entier $f'_{best} \leftarrow \max_{S_i \in P}(f(S_i))$

Si $f_{best} > f'_{best}$ **alors**

$f_{best} \leftarrow f'_{best}$

Fin Si

$P \leftarrow \text{selectionSurvie}(P, T)$

Fin Pour

Renvoyer f_{best}

3.3 Expérimentations et résultats

Les expérimentations ont été menées avec un Ryzen 5 3600 avec 16GB de DDR4, sous Ubuntu 21.04 sur un algorithme génétique implémenté en python. Le paramétrage a été effectué de la manière suivante :

- taille de la population : 400 individus,
- taille de la sélection : 100 individus,
- probabilité de croisement : 0.8,
- probabilité de mutation : $\frac{2}{N}$, avec N le nombre de variables de décision,

- le temps accordé à l'algorithme est de 60 secondes.

La sélection pour reproduction est effectuée aléatoirement, en fixant la probabilité de choisir un individu en fonction de sa valeur de fonction objectif. Les couples formés parmi la sélection sont aléatoires. Le croisement est effectué via un opérateur multipoints (entre 2 et 4 points, tirage aléatoire avant chaque croisement). Enfin, la sélection pour survie est déterministe, on sélectionne toujours les 400 meilleurs. L'algorithme étant en grande partie aléatoire, chaque instance est traitée 5 fois afin d'obtenir un score minimum, un score moyen et un score maximum (Tables 4,5,6).

Instance	Score	Gap	Instance	Score	Gap
instance1.txt	73	0.00	instance6.txt	???	???
instance2.txt	91	0.00	instance7.txt	???	???
instance3.txt	84	0.00	instance8.txt	???	???
instance4.txt	81	0.02	instance9.txt	???	???
instance5.txt	81	0.00	instance10.txt	???	???

TABLE 3 – Résultat de l'algorithme génétique sur les instances (minimum)

Instance	Score	Gap	Instance	Score	Gap
instance1.txt	73	0.00	instance6.txt	???	???
instance2.txt	91	0.00	instance7.txt	???	???
instance3.txt	84	0.00	instance8.txt	???	???
instance4.txt	82.2	0.01	instance9.txt	???	???
instance5.txt	81	0.00	instance10.txt	???	???

TABLE 4 – Résultat de l'algorithme génétique sur les instances (moyenne)

Instance	Score	Gap	Instance	Score	Gap
instance1.txt	73	0.00	instance6.txt	???	???
instance2.txt	91	0.00	instance7.txt	???	???
instance3.txt	84	0.00	instance8.txt	???	???
instance4.txt	83	0.00	instance9.txt	???	???
instance5.txt	81	0.00	instance10.txt	???	???

TABLE 5 – Résultat de l'algorithme génétique sur les instances (maximum)

L'interprétation des résultats est ici évidente : l'algorithme génétique partant d'un ensemble de solutions comprenant entre autres celle générée par l'algorithme glouton, il ne peut que faire au moins aussi bien. Dans le pire des cas, il ne trouvera pas de meilleure solution que celle-ci et renverra donc la solution calculée par l'algorithme glouton amélioré. Cependant, on voit ici que, pour toutes les instances, l'algorithme génétique obtient de bien meilleurs résultats que l'algorithme glouton, même lorsque l'on ne tient compte que des pires résultats (scores minimum) fournis par la métaheuristique. Pour toutes les instances traitées ici, l'algorithme génétique trouve également la solution optimale, et, si l'on omet l'instance 4, l'optimal est atteint à chaque lancement. Dans le cas de l'instance 4, on voit que l'optimal est atteint lors d'au moins un des cinq lancements, et présente un gap moyen de 1%.

Il reste donc à vérifier son comportement sur le reste des instances, y compris celles de grandes tailles (archive "instances2.zip"). Voici les résultats partiels obtenus sur ces grandes instances (le temps d'exécution a été fixé à 120 secondes) :

Instance	min	avg	max	Instance	min	avg	max
large1.txt	626	633.4	644	large6.txt	???	???	???
large2.txt	623	631.6	639	large7.txt	???	???	???
large3.txt	664	676.4	685	large8.txt	???	???	???
large4.txt	637	640.0	645	large9.txt	???	???	???
large5.txt	???	???	???	large10.txt	???	???	???

TABLE 6 – Résultat de l'algorithme génétique sur les grandes instances

On remarque que l'algorithme perd en stabilité à mesure que la taille de l'instance augmente, il faudra explorer d'autres pistes afin d'accroître les performances. L'algorithme présenté ici n'est pas "affiné" : les paramètres ont été fixés un peu au hasard, et aucune procédure intelligente ne vient les modifier en cours d'exécution. On pourrait par exemple faire varier les probabilités de croisement et de mutation en fonction de la diversité de la population et du nombre d'itérations passées sans amélioration. On pourrait également tester d'autres opérateurs de croisement, d'autres mutations, ... Enfin, une piste de recherche serait la procédure de création des couples : pour l'instant, elle est aléatoire, mais on pourrait associer les solutions en fonction de leur ressemblance.

3.4 Quelques autres métaheuristiques

Vous avez tous les indices pour commencer votre travail sur les méthodes approchées : commencez par un algorithme glouton puis un algorithme génétique. Il existe cependant un grand nombre de métaheuristiques, qui, pour la plupart, sont adaptables à notre problème.

3.4.a Algorithme à liste taboue

L'algorithme à liste taboue (TS : Tabu Search) est une recherche locale améliorée : on part d'une solution et on effectue un mouvement à chaque itération, tout en maintenant une liste évitant les cycles dans l'espace des solutions. L'idée est la suivante :

- on commence par générer une solution via l'algorithme glouton ;
- à chaque itération, on étudie le **voisinage** de la solution courante, et on choisit la meilleure d'entre elles ;
- on enregistre le mouvement effectué pour passer de la solution courante à la solution suivante dans une liste, afin d'éviter d'annuler ce mouvement au cours des prochaines itérations.

Pour notre problème, le mieux est d'étudier un voisinage de type k – *Flip*, c'est à dire l'inversion de k variables (0 devient 1, et inversement). Dans la littérature, certains auteurs ont effectué des implémentations du 3 – *Flip* donnant des résultats acceptables. Ainsi, à chaque mouvement, on enregistre les index des trois variables impactées, et on interdira de toucher celles-ci pendant un certain nombre d'itérations.

3.4.b Algorithme à colonies de fourmis

L'algorithme à colonies de fourmis (ACO) est un algorithme évolutionnaire probabiliste se déroulant, dans le cas de notre problème, de la manière suivante :

- on initialise les phéromones à une valeur standard pour chaque variable ;
- à chaque itération, chaque fourmi construit une solution, en guidant ses choix en fonction d'une heuristique et des phéromones ;
- on réimprime des phéromones sur les variables en fonction de la valeur objectif, et ce pour (1) la meilleure solution trouvée ou (2) toutes les solutions générées ;
- on procède à l'évaporation des phéromones.

Il s'agit d'un algorithme efficace, mais difficile à paramétrer, il faut en effet déterminer : l'impact de l'heuristique par rapport aux phéromones, le ratio d'évaporation à chaque itération, la quantité de phéromones à ajouter en fonction de la valeur objectif de la solution considérée...

3.4.c Algorithme à essaim particulaire

L'algorithme à essaim particulaire (PSO) est un algorithme évolutionnaire se basant sur la chasse des oiseaux. Chaque individu est une particule se déplaçant dans l'espace selon trois facteurs : (1) l'inertie ou la confiance dans la trajectoire courante, (2) la mémoire de la meilleure position traversée par la particule m et (3) la meilleure position n des particules "amies". Soient V_t , V_{t+1} , P_t et P_{t+1} les vitesses et positions aux itérations t et $t + 1$, on définit :

$$\begin{aligned}V_{t+1} &= wV_t + c_1(m - P_t) + c_2(n - P_t) \\P_{t+1} &= P_t + V_{t+1}\end{aligned}$$

avec w , c_1 et c_2 trois paramètres de l'algorithme, représentant les impacts des trois facteurs sur la trajectoire. Cet algorithme a initialement été développé pour l'optimisation continue, mais est adaptable pour l'optimisation combinatoire de plusieurs manières. Ici, les composantes de la vitesse et de la position sont des réels entre 0 et 1. Il faudra donc générer une solution S_t par particule à chaque mise à jour de la position. Pour chaque composante i , on peut décider de la valeur 0 ou 1 si la valeur de la position est supérieure à un certain seuil. Par exemple, si $P_t[i]$ est supérieure à 0.7, alors on décide de fixer $S_t[i]$ à 1, zéro sinon.

3.5 Pistes d'amélioration

Une fois un algorithme implémenté, il faut essayer de la pousser dans ses retranchements. Pour cela, la première étape est le paramétrage. Dans le cas d'un algorithme génétique, la taille de la population, de la sélection, la probabilité de croisement et la probabilité de mutation sont à déterminer. La seule manière de procéder est de faire des tests en quantité suffisante. Pour chaque paramètre, on détermine une valeur minimum, maximum et un pas d'incrément, et on teste toutes les combinaisons possibles, afin de trouver le meilleur paramétrage.

On fait parfois des choix d'implémentation. J'ai par exemple choisi un opérateur de croisement multi-points pour mon algorithme génétique, mais rien de garanti que celui-ci

soit le plus efficace pour ce problème. Il peut être intéressant de comparer plusieurs versions d'un algorithme pour déterminer quelle est la plus performante pour notre problème.

Enfin, que ce passe-t-il quand on croise du blé et du chocolat¹? Ou un bulbizarre et un carapuce²? On procède à une hybridation! Si deux de vos algorithmes utilisent des mécanismes de convergence différents, il peut être très intéressant de les croiser : à chaque itération, le premier fait son travail (ici une construction de solutions), et fait appel au second, qui dans l'idée est très rapide, pour partir des solutions générées et effectuer son propre processus de convergence. En fait, à chaque itération, le premier algorithme génère le point de départ du second. L'exemple le plus connu dans la littérature est l'appel à un algorithme à liste taboue à chaque itération d'un algorithme génétique, pour améliorer un ou plusieurs enfants générés lors de l'itération.

Algorithme 4 – Principe d'hybridation

Entrées. Liste paramsMethode1
Liste paramsMethode2
Sortie. Entier f_{best}

Solution[] $P \leftarrow \text{initialisationMethode1}()$
Entier $f_{best} \leftarrow \max_{S_i \in P}(f(S_i))$
Pour i allant de 1 à IterMax **faire**
 Solution[] $P \leftarrow \text{iterationMethode1}(P)$
 Entier $f'_{best} \leftarrow \text{methode2}(P)$
 Si $f_{best} < f'_{best}$ **alors**
 $f_{best} \leftarrow f'_{best}$
 Fin Si
Fin Pour
Renvoyer f_{best}

4 Résumé du travail à effectuer

Vous devez réaliser les tâches suivantes :

1. la modélisation linéaire du problème décrit dans ce document ;
2. un programme permettant la traduction de fichiers d'instance en fichier lp, afin de les exploiter avec GLPK (au moins les premières instances, jusqu'à ce que GLPK ne soit plus exploitable) ;
3. un premier algorithme glouton et enregistrer les résultats fournis pour chaque instance, afin de les comparer avec ceux fournis par GLPK, puis avec les méthodes plus avancées ;
4. un algorithme génétique suivant (ou pas) le plan fourni dans ce document, afin d'obtenir des résultats proches (voire meilleurs!) que ceux présentés ici ;
5. au moins une métaheuristique supplémentaire à comparer avec les méthodes précédemment implémentées.
6. des hybridations, des heuristiques gloutonnes différentes, ...

La modélisation, les détails d'implémentation, les procédures de test, les résultats et interprétations devront être consignés dans un rapport d'environ 10 pages (ou plus).

1. des chocapics
2. ???