Rapport d’Optimisation combinatoire en Python

Membres du projet :

Guillaume BOULBEN

Alexis JORRE

Clément MAILLARD

Hugo TRICOIRE

Sommaire

Réalisation du modèle linéaire

Réalisé par Hugo Tricoire

1- Analyse du contexte

Le problème est le suivant :

Nous sommes chargés d'inviter des convives pour une fête. Pour cela, nous avons à notre disposition une liste de convives potentiels détaillant les relations de connaissances entre eux et un nombre représentant l'importance d'inviter chaque convive.
Notre objectif est de maximiser la somme de ces nombres tout en gardant à l'esprit que chaque invité doit connaître tout le monde.

2- Modélisation du problème

Déclaration des variables

$Soit\ N\ l'ensemble\ des\ convives\ potentiels$

$Soit\ i\ \in\ N,\ représentant\ le\ convive\ n°i$

$Soit\ c_i\ un\ nombre\ entier\ positif\ représentant\ la\ rentabilité\ d'inviter\ le\ convive\ n°i$

$Soit\ V_i \subseteq N^N\ l'ensemble\ des\ connaissances\ de\ i$

L'objectif est donc de maximiser l'expression suivante :

$$\sum_{i=1}^{n} c_i x_i$$

Avec x_i égal à 1 si le convive n°i est invité ou 0 si le convive n°i n'est pas invité.

Définition des contraintes

Dans un premier temps, voici le raisonnement que j'ai pu avoir :

Ayant la liste des connaissances de chaque convive, il nous faudra alors ajouter une nouvelle variable y_{ij} égale à 1 si le convive i connait le convive j et 0 sinon.

Mon but a été de dire que si $x_i = 1$, alors tous les autres convives j tels que $x_j = 1 \; et \; j \neq i$ doivent connaître le convive i.
Voilà ce que ça donnerait en terme mathématique :

$$\forall i \in N \;\; x_i \times \sum_{j=1}^{n} y_{ij} \geq x_i \times \sum_{j=1}^{n} x_j$$

Malheureusement, j'ai remarqué que cela ne couvrait pas toutes les situations. Je me suis donc penché sur une autre approche : isoler les cas problématiques plutôt que les cas satisfaisant les conditions de départ.
Cela donne alors quelque chose de plus simple qui dit que si 2 personnes ne se connaissent pas, alors l'une seule d'entre elles peut être présente à la fête.
Si on introduit $y_{i,j}$ une variable égale à 1 si les convives i et j se connaissent et 0 sinon, voici ce que cela donne en termes mathématiques :

$$\forall i, j \in N \;\; tels \; que \; i \neq j, \; y_{i,j} = 0 \Rightarrow \; x_i + x_j \leq 1$$

Conclusion

Voici donc le modèle linéaire de notre projet :

Maximiser : $$z = \sum_{i=1}^{n} c_i x_i$$

S.c :

$$i \neq j, \; y_{i,j} = 0 \Rightarrow \; x_i + x_j \leq 1$$

$$x_i \in \{0, 1\}$$

3- Programmation du code python de génération de fichier lp

Voici comment j'ai réalisé le code python de l'algorithme de conversion d'instances (fichier texte) en fichier lp.
Avant toute opération, j'ai commencé par déterminer n, le nombre de convives dans l'instance, ce qui m'aidera pour la suite (à savoir que le convive n°i dans le fichier d'instance txt correspond au convive n°(i+1) dans le fichier lp.
Ensuite, j'entame la création du fichier lp et écrit l'équation à maximiser en prenant en compte le poid de chaque convive.

Ensuite, la création des contraintes a été la partie la plus délicate. Il a fallu déterminer à quelle partie du fichier commencer l'exploration des connaissances de chaque convive. J'ai remarqué qu'il n'était pas nécessaire de prendre en compte celles du dernier convive car on va les lister grâce aux autres convives. On crée d'abord L une liste de listes qui recevra des 0 ou 1 correspondant aux connaissances de chaque convives (exemple : L[i] [j]=1 signifie que les convives (i+1) et (j+1) se connaissent.
Grâce à cela, j'ai pu facilement déterminer ceux qui ne se connaissaient pas et écrire l'équation de contrainte définie plus haut pour chaque couple de convives ne se connaissant pas. Pour chaque L[i] [j] = 0 traité, j'ai passé le L[j] [i] à 1 pour éviter qu'une contrainte apparaisse 2 fois.

Pour finir, je définis tous les x_i comme étant des "Binaries" pour que glpk comprennent qu'elles ne peuvent qu'être égales à 0 ou 1.
Pour finir, la création d'un main pour l'exécution de la fonction comme demandé.

L'algorithme fonctionne bien et renvoie la bonne valeur à presque chaque instance, sauf pour l'instance n°3 qui, pour une raison inconnue, renvoie 81 au lieu de 84.

Résolution des 10 premières petites instances avec GLPK	
N° de l'instance	Solution optimale trouvée
1	73
2	91
3	81
4	83

5	81
6	80
7	81
8	85
9	77
10	91

Algorithme Glouton

Réalisé par Clément MAILLARD

Exécution de l'algorithme : python Glouton.py ./[Nom de L'instance]

Pour cet algorithme j'ai repris Le squelette donnée dans le sujet :

Algorithme 1 – Algorithme glouton

Solution S ← {}
Entier C ← Tous les convives
Tant que C ≠ ∅ faire
    Entier i ← Heuristique(C)
    S ← S ∪ {i}
    Mettre à jour C
Fin Tant que

Avec comme heuristique le critères de choix défini par Ci * |Vi| mais non pas Vi en fonction de l'intégralité des convives mais Vi en fonction des convives déjà présents dans la solution.

Contrairement à l'heuristique proposé celui choisi as un petit problème qui est le choix du premier convive car il n'y a encore personne dans la solution son score est donc de 0. J'ai donc choisi d'utiliser Ci * |Vi| en fonction de tout les convives pour effectuer le premier choix et ensuite passé à l'autre heuristique, ce qui permet de régler le problème du premier convive dans la solution.

Résolution des 10 premières petites instances avec L'algorithme Glouton		
N° de l'instance	Solution trouvée	Gap
1	67	0.08
2	75	0.18
3	74	0.12
4	73	0.12
5	63	0.22
6	73	0.09
7	68	0.16
8	73	0.14

9	68	0.12
10	80	0.12

On peut voir les résultats de notre algorithme glouton dans le tableau ci-dessus. On constate que malgré le petit changement pour l'heuristique il n'y a toujours aucune solution optimale à trouver et que pour la majorité des solution on observe un écart d'environ 12% avec un maximum pour l'instance 5 avec 22% d'écart. Notre algorithme reste donc assez peu fiable.

J'ai aussi essayer de prendre les chose sous un autre angle en choisissant un heuristique drastiquement différente qui consisterait à prendre tous les convives en solution puis de supprimer celui qui connais le moin de convives restant en solution un par un mais il s'est avéré que cette heuristique donnais des résultat encore moin fiable que le premier j'ai donc abandonné ce choix.

Algorithme Génétique

Réalisé par Guillaume BOULBEN

1- Organisation générale

Exécution de l'algorihtme : python genetique.py [TEMPS LIMITE] ./instance.txt [NOM DE VOTRE FICHIER]

Pour cet algorithme je suis passé par 2 programmes différents, le premier en cherchant simplement les fonctions qui composent l'algorithme grâce au cours que nous avons eu avant de commencer ce projet j'ai pu facilement comprendre quoi chercher. Le deuxième programme est un peu plus simple, j'ai simplement repris le source code du sujet et je l'ai implémenter.

2- L'algorithme génétique

Un algorithme génétique est une méthode d'optimisation qui s'inspire du processus évolutif de la nature. Son fonctionnement se déroule de cette manière :

On commence par créer une ***population initiale*** de solutions potentielles au problème. Chaque solution est généralement représentée sous forme de chaîne de gènes ou plutôt dans notre cas une liste. Chaque solution de la population est ***évaluée*** en comparant sa performance par rapport à une fonction objective. Cette fonction mesure à quel point une solution est "bien adaptée" ou viable au problème que l'algorithme cherche à résoudre.
Les solutions de la population sont ***sélectionnées*** pour former une nouvelle génération. Ces dernières, ayant une meilleure performance ont une probabilité plus élevée d'être sélectionnées, imitant ainsi le processus de sélection naturelle.

Des paires de solutions sélectionnées sont ***combinées/croisées*** pour créer de nouvelles solutions, appelées enfants (child). Cela simule le croisement encore une fois la génétique dans la nature. Les parties des solutions parentes sont échangées pour produire des solutions enfants. Certaines des nouvelles solutions peuvent subir des ***mutations*** aléatoires, où des parties de leurs gènes sont modifiées. Cela introduit de la diversité dans la population et aide à trouver de nouvelles possibilités en termes de solution finale.

La nouvelle génération, composée de solutions parentes, d'enfants(child) issus du croisement, et éventuellement de solutions mutées, ***remplace*** l'ancienne génération.
La toute dernière étape est de ***répéter*** pour plusieurs générations jusqu'à ce qu'une solution satisfaisante soit trouvée ou qu'un certain critère d'arrêt soit atteint.

3- Limites de l'algorithme

L’algorithme génétique peut avoir du mal à résoudre des problèmes complexes, en particulier si la taille de l’espace de recherche est grande. Dans notre cas, le nombre minimal est personne dans les tests est de 300 personnes avec une affiliation avec certaines personnes. Les interactions complexes que nous avons avec les relations des 300 convives peuvent influencer la dynamique de l'évolution génétique. Des mécanismes de reproduction et de sélection appropriée doivent être définis pour refléter ces relations, et pour l’instant nous ne nous allons pas trouver des fonctions capables de le faire.

Notre programme utilise aussi le principe de l’aléatoire pour permettre d’avoir des mutations qui nous aident à trouver des solutions plus rapidement. Mais malheureusement, cette méthode n’est pas un bon moyen d’obtenir le résultat escompté. Le problème de mutation et croisement génétique font que nos résultats ne cessent de changer à chaque nouveau test.

Métaheuristique

Réalisé par Alexis JORRE

La métaheuristique retenue est l'algorithme du chaos. Cette méthode consiste à utiliser un système dynamique chaotique afin de créer une perturbation dans la résolution linéaire dans le but de ne pas rester sur un extremum local.
Ce système dynamique chaotique consiste en fait en une équation non linéaire, qui peut générer un comportement chaotique, c'est-à-dire sensible aux conditions initiales, et présentant une complexité apparente. La fonction logistique est souvent utilisée $xn+1=r \cdot xn \cdot (1-xn)$ $X_n + 1 = rX_n(1 - X_n)$.
Ici, les solution de cet équation chaotique sont utilisées afin de perturber la population, ce qui permet en théorie de plus explorer l'espace de solution. Une fois cette perturbation effectuée, on vérifie si la population choisie correspond à ce que l'on cherche (les personnes se connaissent-elles toutes?) et on garde les personnes qui vérifient cela.
On calcule ensuite la somme des intérêts pour cette itération, et on recommence en générant aléatoirement une nouvelle population, afin de varier les résultats de l'équation chaotique.
Malheureusement, il semblerait que cette méthode ne soit pas bien adaptée à notre problème, puisque même après un changement d'équation et de paramétrisation, la solution s'approche des 2000 d'intérêt en somme total pour la soirée, au lieu des 73 données dans le sujet (tout de même bien mieux que les 360 000 obtenus avec la première équation utilisée).

Pour expliquer le code en lui-même, on commence (dans la fonction main) par lire le fichier, afin de récupérer les données d'initialisation du problème. Puis, on lance la fonction chaotique optimization. Celle-ci commence par noter le temps actuel, puis commence sa première itération. A l'intérieur de cette itération, on commence par générer une population aléatoire à l'aide de la fonction chaotique choisie, puis on évalue cette population, si celle ci est meilleur que la précédente, on garde cette solution, enfin, on vérifie combien de temps s'est écoulé depuis le début, et on finit par retourner le résultat si le temps max est atteint. On finit ensuite par écrire les résultats dans un fichier .txt.

Algorithme Hybride

Réalisé par Hugo TRICOIRE

1- Organisation générale

Exécution de l'algorihtme : python hybride.py ./instance.txt

Pour l'algorithme hybride, j'ai décidé de mélanger 3 algorithmes différents :

- Un algorithme glouton
- Un métaheuristique
- Un algorithme génétique

Pour l'algorithme glouton, je n'ai fait que reprendre notre propre algorithme réalisé. Au niveau du code python, il correspond à la fonction solution_initiale. Pour ce qui est du métaheuristique, j'ai opté pour un algorithme original : l'algorithme des lucioles. Enfin de ce qui est de l'algorithme génétique, j'ai opté pour un algorithme immunitaire.

2- L'algorithme des Lucioles

Ce dernier se base sur le déplacement des lucioles qui ont tendance à se rapprocher des zones plus lumineuses que les autres. Voici la manière dont je l'ai implémenté :
La fonction algorithme_luciole implémentée a pour but de générer plusieurs solutions en faisant des modifications aléatoires sur la solution initiale générée par le glouton (tout en vérifiant que les solutions générées vérifient bien les conditions du problème, d'où la fonction check_Connaissances).
La fonction mouvement_lucioles permet de générer une de ces solutions. Elle prend en paramètres notamment la solution générée par le glouton, qui est une liste contenant des 0 et des 1 selon si les n convives potentiels sont invités ou non. Conceptuellement, une luciole correspond à une case égale à 1 dans cette liste, et un endroit plus lumineux est un convive ayant un poid supérieur à celui de la luciole. Dans ce cas, en appliquant une formule spécifique afin de déterminer le taux d'attractivité de la luciole vers cette case de la liste, on va ou non la déplacer vers cette case. Cette fonction est répétée récursivement tant que la solution générée ne satisfait pas les contraintes de connaissances.
Finalement, l'algorithme des lucioles retourne une liste de x solutions générées. Chaque solution est une liste contenant 2 éléments :

- Une liste de 0 et de 1 correspondant aux invitations ou non des n convives potentiels
- Le total des poids des convives invités

3- L’algorithme génétique

Le but de cet algorithme est de générer des solutions parents en choisissant au hasard des solutions générées par l’algorithme des lucioles. De ces solutions, le but est de les faire muter aléatoirement suivant un taux de mutation qui va freiner ou accentuer la mutation pour chaque génération (tout en vérifiant que les mutations générées satisfassent également les contraintes des connaissances). On va ensuite ajouter ces solutions enfants à la liste de toutes les solutions générées depuis le début. On va ensuite trier la liste des solutions en fonction du total des poids des convives invités et ressortir la meilleure solution, ce qui nous donne la solution finale.

4- Limites de l’algorithme

L’algorithme étant original et une bonne association des 3 types d’algorithmes, j’avais trouvé intéressant de l’implémenter. En revanche, je me suis rendu compte qu’avec son utilisation massive de l’aléatoire l’algorithme n’est absolument pas optimisable sans en changer son fonctionnement intrinsèque. De plus, Python n’est pas capable de supporter la grande récursivité des fonctions implémentées, notamment pour la satisfaction de la contrainte des connaissances de chaque solution générée.