esiea
INGÉNIEUR·E·S D'UN NUMÉRIQUE UTILE

RAPPORT DE PROJET

OPTIMISATION COMBINATOIRE

Année 2023-2024

GROUPE :

MARTY Cédric

TAROT Bastien

LE ROUX Guénaël

AUBERT Antonin

Table des matières :

I. Recherches

Après première lecture et recherche concernant le problème, il semblerait que celui-ci s'apparente à un problème de maximisation de clique. Nous avons donc effectué quelques recherches afin de se documenter sur le problème. Tous les liens que nous avons consultés pour comprendre le problème se trouvent dans l'annexe de ce rapport. Les premières lectures sur le sujet nous informent de différentes méthodes utilisées couramment pour résoudre ce genre de problème ainsi que la difficulté de le résoudre en un temps efficace.

II. Description et résolution du modèle linéaire

a. Description

Nous prendrons P, l'ensemble des personnes présentes dans le jeu de données et R l'ensemble des relations décrites dans le même jeu.

Nous recherchons à maximiser le poids des personnes sélectionnées pour la soirée. Cela nous donne un objectif z défini ci-après :

$$Max\ z = \sum_{i=1}^{n} x_i w_i$$

Avec n, le nombre de personnes dans l'ensemble P, $x_i \in \{0,1\}$, xi = 1 si la personne a été choisie dans le groupe, 0 sinon et enfin, w_i le poids de la personne i.

Pour la contrainte ; nous avons choisi dans un premier temps de dire que deux personnes différentes qui ne se connaissent pas ne peuvent être choisies toutes les deux. Donnant :

$$s.t.\ x_i + x_j \leq 1\ , \forall (P_i, P_j) \in \bar{R}$$
$$i \neq j\ , x_i \in \{0,1\}\ \forall\ i \in P$$

b. Résolution

Une fois cette contrainte trouvée, nous avons écrit le fichier convertion.py qui permet de créer le fichier GLPK. Cependant, lors des premiers tests avec GLPK, nous écrivions toutes les contraintes selon ce procédé dans le fichier.

Exemple : c0 : x0 + x1 <= 1 et c426 : x1 + x0 <= 1

Ces contraintes sont exactement les mêmes. Cela doublait inutilement le nombre de contraintes et le temps d'exécution qui était multiplié par entre 2 et 4 fois avec 800Mb de mémoire utilisée (Avec une multiplication par 3 de la mémoire utilisée en moyenne). Pour résoudre ce problème nous avons comparé l'identifiant de la personne i avec celui de la personne j et avons écrit la contrainte dans le fichier seulement si i < j. Ainsi lorsque la contrainte est écrite par le premier invité la rencontrant, elle ne sera plus jamais comptabilisée. Nous avons donc un premier modèle linéaire fonctionnel utilisé pour trouver les solutions des instances suivantes :

NOM DES INSTANCES	TEMPS RESOLUTION	MEMOIRE UTILISEE	RESULTAT OBTENU
INSTANCE 1	4945.9 secs	288.0 Mb	73
INSTANCE 2	7494.8 secs	375.3 Mb	91
INSTANCE 3	9339.7 secs	464.1 Mb	84
INSTANCE 4	7905.8 secs	416.7 Mb	83
INSTANCE 5	7799.6 secs	400.2 Mb	81
INSTANCE 6	11000.8 secs	527.6 Mb	80
INSTANCE 7	7849.6 secs	411.4 Mb	81
INSTANCE 8	9338.4 secs	468.0 Mb	85
INSTANCE 9	8604.6 secs	450.7 Mb	77
INSTANCE 10	7711.8 secs	395.9 Mb	91

Table des résultats de GLPK

Les résultats ont été calculés sur un serveur et pour gagner du temps nous les avons tous lancé en même temps. D'où le temps de résolution qui peut être grand. Cependant, il est possible d'optimiser encore plus le nombre de contraintes en faisant la sommes des contraintes ayant les mêmes premières personnes, par exemple :

$$x_0 + x_1 \leq 1 \text{ Et } x_0 + x_2 \leq 1$$

$$Deviennent : 2x_0 + x_1 + x_2 \leq 2$$

Cela nous donne une seule contrainte par convive, ce qui permet considérablement de réduire le nombre de contraintes. Voici les résultats obtenus avec cette nouvelle méthode :

NOM DES INSTANCES	TEMPS RESOLUTION	MEMOIRE UTILISEE	RESULTAT OBTENU
INSTANCE 1	572.9 secs	37.0 Mb	73
INSTANCE 2	1077.2 secs	52.5 Mb	91
INSTANCE 3	1567.2 secs	63.2 Mb	84
INSTANCE 4	1942.9 secs	57.5 Mb	83
INSTANCE 5	1469.4 secs	56.7 Mb	81
INSTANCE 6	2224.5 secs	71.9 Mb	80
INSTANCE 7	994.5 secs	52.2 Mb	81
INSTANCE 8	1826.4 secs	67.6 Mb	85
INSTANCE 9	1398.0 secs	58.5 Mb	77
INSTANCE 10	1189.0 secs	52.9 Mb	91

Table des résultats de GLPK avec contraintes optimisées

Les calculs ont été lancés sur la même machine que pour le premier jet. On voit bien la différence sur la mémoire et le temps. Avec un meilleur processeur, il est possible de descendre vers 150 à 300 secs pour les 5 premières instances. Nous avons aussi cherché à résoudre les autres petites instances disponibles avec GLPK. Vous pouvez trouver les résultats associés à ces instances en annexes.

III. Algorithme Glouton

a. Description

L'algorithme glouton est défini comme un algorithme qui prend des décisions locales optimales à chaque étape dans le but d'obtenir une solution globale optimale au problème.

Pour la première implémentation nous avons implémenté l'algorithme glouton ainsi que la fonction heuristique proposée dans le sujet dont le pseudo-code est le suivant :

Algorithme 1 – Algorithme glouton	**Algorithme 1** – Heuristique du glouton

Tableau de personnes C ← Tous les convives
Tableau de booléen S ← Tous les convives (0 si non invité / 1 sinon)
Tant que C ≠ ∅ faire
  Entier i ← Heuristique(C)
  Si i = -1 faire
  Arrêter Tant que
  S[i] ← 1
  Mettre à jour C (suppression de i)
Fin Tant que
	Entier Maximum ← -1
	Entier Index ← -1
	Entier Id du premier convive de la solution <- S[0].id
	Pour convive dans les relations du premier convive faire
	  Si le convive est déjà invité faire
	  Continuer
	  Flottant Valeur du convive ← Poids * Nombre de relation
	  Si Valeur du convive > Maximum faire
	  Pour invité dans S faire
	  Si invité n'est pas ami de convive
	  Arrêter Pour
	  Sinon faire
	  Maximum ← Valeur du convive
	  Index ← Id du convive
	Retourner Index

L'algorithme glouton est en général rapide dans son exécution mais va se limiter à un optimum local qui est toujours prédéfini par son choix premier, il y a donc de nombreux cas où celui-ci n'atteindra pas le résultat optimal par manque de possibilité de suivre un autre chemin optimal. Les résultats trouvés sur chaque petite instance sont détaillés dans la partie ci-après.

b. Résultats

NOM DE L'INSTANCE	SCORE ATTENDU	SCORE OBTENU	GAP	TEMPS
INSTANCE 1	73	69	5%	0.01s
INSTANCE 2	91	90	1%	0.02s
INSTANCE 3	84	71	15%	0.02s
INSTANCE 4	83	67	19%	0.01s
INSTANCE 5	81	80	1%	0.02s
INSTANCE 6	80	73	9%	0.01s
INSTANCE 7	81	61	25%	0.01s
INSTANCE 8	85	69	19%	0.01s
INSTANCE 9	77	68	12%	0.01s
INSTANCE 10	91	72	21%	0.01s

Table des résultats de l'algorithme glouton

IV. Algorithme Bron-Kerbosch

a. Description

Lors de nos recherches sur le problème, nous avons découvert l'algorithme de Bron-Kerbosch. Il s'agit d'un algorithme d'énumération qui permet en listant l'ensemble des cliques de l'instance d'en récupérer la maximale existante. Cet algorithme sort du concept du projet étant donné qu'il ne s'agit ni d'un algorithme glouton, ni d'une méta-heuristique mais il nous paraissait intéressant d'implémenter cet algorithme afin de bien comprendre l'énoncé.

Après optimisation par notre part, voici les résultats obtenus sur chaque instance standard du sujet. Cet algorithme a l'avantage de trouver à chaque fois la solution optimale ; néanmoins, il n'est pas efficace étant donné qu'il liste absolument toutes les cliques disponibles. Il est donc impossible de le lancer sur une instance large et espérer obtenir un résultat rapidement.

b. Résultats

NOM DE L'INSTANCE	SCORE ATTENDU	SCORE OBTENU	GAP	TEMPS
INSTANCE 1	73	73	0.0	1.67
INSTANCE 2	91	91	0.0	2.96
INSTANCE 3	84	84	0.0	4.11
INSTANCE 4	83	83	0.0	3.22
INSTANCE 5	81	81	0.0	3.30
INSTANCE 6	80	80	0.0	4.62
INSTANCE 7	81	81	0.0	2.66
INSTANCE 8	85	85	0.0	4.09
INSTANCE 9	77	77	0.0	2.93
INSTANCE 10	91	91	0.0	2.89

Table des résultats de l'algorithme Bron-Kerbosch

V. Algorithme génétique

a. Description

L'algorithme génétique est un algorithme d'évolution métaheuristique qui se caractérise par sa capacité à explorer une vaste et diverse population en se basant sur le processus de sélection naturelle observé dans la nature. Les meilleures solutions d'une itération vont produire des enfants qui seront meilleurs au fur et à mesure du temps. L'ensemble des résultats obtenus via l'algorithmie génétique sont disponibles ci-dessous pour les instances standard et larges. L'implémentation de l'algorithme génétique nécessite l'utilisation d'aléatoire dans le processus de croisement et de mutation ce qui va grandement apporter de la diversité et faire varier les possibilités et les implémentations de cet algorithme. En paramètre, l'algorithme nécessite la taille de la population à générer (le nombre de solutions à calculer), le nombre d'individus sélectionnés pour la reproduction ainsi que la probabilité de croisement et de mutation qui permettent de rendre plus aléatoire la génération actuelle et ses suivantes. Afin de gérer notre croisement de reproduction nous avons implémenté un croisement multipoints (aléatoire en 2 et 4 points). L'algorithme a reçu les mêmes paramètres d'entrées pour chaque instance, variables qui sont définies ci-dessous.

b. Résultats

Valeurs d'entrée :

Taille de la population : 400

Taille de la sélection pour reproduction : 100

Probabilité de croisement : 0.8

Probabilité de mutation : 2/N avec N nombre de convives possibles

Temps accordé à l'algorithme : 120 secondes

NOM DE L'INSTANCE	SCORE ATTENDU	SCORE OBTENU	GAP
INSTANCE 1	73	71	3%
INSTANCE 2	91	91	0%
INSTANCE 3	84	82	2%
INSTANCE 4	83	83	0%
INSTANCE 5	81	81	0%
INSTANCE 6	80	78	3%
INSTANCE 7	81	81	0%
INSTANCE 8	85	82	4%
INSTANCE 9	77	75	3%
INSTANCE 10	91	87	4%

Table des résultats de l'algorithme génétique (résultats minimaux)

NOM DE L'INSTANCE	SCORE ATTENDU	SCORE OBTENU	GAP
INSTANCE 1	73	72.2	1.4%
INSTANCE 2	91	91	0%
INSTANCE 3	84	82.8	1.5%
INSTANCE 4	83	83	0%
INSTANCE 5	81	81	0%
INSTANCE 6	80	78.4	2%
INSTANCE 7	81	81	0%
INSTANCE 8	85	84	1.2%
INSTANCE 9	77	75.6	1.9%
INSTANCE 10	91	90.2	0.9%

Table des résultats de l'algorithme génétique (résultats moyens)

NOM DE L'INSTANCE	SCORE ATTENDU	SCORE OBTENU	GAP
INSTANCE 1	73	73	0%
INSTANCE 2	91	91	0%
INSTANCE 3	84	84	0%
INSTANCE 4	83	83	0%
INSTANCE 5	81	81	0%
INSTANCE 6	80	80	0%
INSTANCE 7	81	81	0%
INSTANCE 8	85	85	0%
INSTANCE 9	77	77	0%
INSTANCE 10	91	91	0%

Table des résultats de l'algorithme génétique (résultats maximaux)

NOM DE L'INSTANCE	SCORE MIN	SCORE MOYEN	SCORE MAX
LARGE 1	618	620	625
LARGE 2	623	626	629
LARGE 3	657	663	673
LARGE 4	629	634	637
LARGE 5	593	602	608
LARGE 6	618	620	622
LARGE 7	608	615	622
LARGE 8	607	613	616
LARGE 9	576	582	591
LARGE 10	655	665	675

Table des résultats de l'algorithme génétique sur les instances larges

Étant donné les résultats ci-dessus, l'algorithme génétique semble être une bonne solution pour calculer la solution optimale du problème, nous avons cependant sur les instances normales des problèmes d'optimum local. En effet l'algorithme génétique se retrouve souvent bloqué à converger dans un optimum local qui n'est pas l'optimum du problème. Peut-être qu'il serait intéressant de le rendre plus divergent afin d'autoriser une sortie de l'optimum plus simple. Cependant sur des instances larges, il est l'algorithme qui obtient les plus grands résultats en un laps de temps court.

VI. Algorithme de colonies de fourmis (ACO)

a. Description

Nous nous sommes ensuite penchés sur une seconde méta-heuristique et avons choisi d'implémenter l'ACO (Ant Colony Optimization). Cet algorithme utilise le comportement naturel des fourmis et des phéromones qu'elles déposent afin de trouver une solution envisageable. Dans le cadre de notre problème, il faut trouver une solution optimale ayant le poids le plus élevé tout en respectant les contraintes des invités.

Les résultats ci-dessous ont été obtenus avec l'algorithme ACO lancé 5 fois avec à chaque fois 20 secondes données à l'algorithme pour trouver une solution optimale. Les paramètres utilisés pour calculer les solutions de ces petites instances sont notées ci-dessous. Lorsqu'une fourmi se déplace sur une personne elle dépose une quantité de phéromone donnée, multipliée par la valeur de son poids et de son nombre de relations. Ce calcul, inspiré du glouton permet de mettre en valeur la personne lorsqu'elle a été sélectionnée pour une solution. Nous avions également pensé à inclure le score total de la solution dans le dépôt de phéromones mais après quelques tests, cela ne s'est pas avéré concluant.

b. Résultats

Valeurs d'entrée :

Valeur maximale de phéromones sur une personne donné : 20

Valeur minimale de phéromones sur une personne donné : 0.1

Quantité de phéromone déposée lors du passage d'une fourmi : 0.1

Taux d'évaporation des phéromones au fil du temps : 0.9

Nombre de fourmis : 100

Nombre de cycles itératifs maximum : 10000

NOM DE L'INSTANCE	SCORE ATTENDU	SCORE MIN	MOYENNE	SCORE MAX
INSTANCE 1	73	73	73	73
INSTANCE 2	91	90	90.6	91
INSTANCE 3	84	82	83	84
INSTANCE 4	83	81	82.6	83
INSTANCE 5	81	80	80.8	81
INSTANCE 6	80	77	79	80
INSTANCE 7	81	79	80.6	81
INSTANCE 8	85	82	84.4	85
INSTANCE 9	77	76	76.8	77
INSTANCE 10	91	87	90.2	91

Table des résultats de l'algorithme ACO pour les instances standard

Nous avons appliqué la même méthode pour les instances larges, nous avons accordé 120 secondes pour chaque lancement étant donné que les instances larges sont plus conséquentes.

NOM DE L'INSTANCE	SCORE MIN	MOYENNE	SCORE MAX
LARGE 1	524	536	549
LARGE 2	532	537	547
LARGE 3	557	564	568
LARGE 4	522	534	549
LARGE 5	513	520	531
LARGE 6	530	533.8	541
LARGE 7	517	527.4	541
LARGE 8	516	526.8	532
LARGE 9	498	506	510
LARGE 10	566	575	592

Table des résultats de l'algorithme ACO pour les instances larges

Étant donné les résultats observés et donnés par l'algorithme génétique, nous pouvons conclure que l'ACO n'est pas le meilleur algorithme pour les grandes instances. Nous avons essayé de jouer avec les différents paramètres (nombre de fourmis notamment), mais cela n'a pas eu d'influence significative dans les résultats. Ceci s'explique également par le fait que l'algorithme est souvent coincé dans un optimum local après quelques itérations et ce malgré la diversité que nous lui avons imposée (choix aléatoire en fonction des phéromones notamment). Nous retenons cependant que pour les petites instances, l'algorithme de colonies de fourmi est plutôt précis et donne de bons résultats dans un laps de temps assez court.

VII. Conclusion

Après élaboration et recherches fructueuses, nous avons pu établir un bon nombre d'algorithmes qui permettent de résoudre le problème d'une clique maximale. Nous avons aussi trouvé plusieurs algorithmes que nous n'avons pas pris le temps d'implémenter (Dynamic Local Search, Liste Taboue, etc…) mais qui pourraient avoir un meilleur résultat que les nôtres. Nous avons cependant pu identifier un problème récurrent à nos algorithmes, à savoir que ceux-ci ne sortent pas assez de leur optimum local. En effet nos solutions restent bloquées dans des solutions similaires malgré l'évolution et la diversité qu'ils imposent.

Un autre point d'optimisation serait aussi de modifier l'implémentation du code et de l'utilisation des données, nous avons effectué un énorme travail d'optimisation qui pourrait être maximisé en remplaçant la structure actuelle par des matrices d'adjacence notamment et en compressant les graphes.

Notre algorithme le plus puissant actuellement est l'algorithme génétique qui itère énormément au vu de son optimisation et qui trouve des résultats plutôt convergents.

VIII. Annexes

a. Données d'instances larges

NOM DES INSTANCES	TEMPS RESOLUTION	MEMOIRE UTILISEE	RESULTAT OBTENU
INSTANCE 11	349785.9 secs (95h)	756.1 Mb	114
INSTANCE 12	470276.3 secs (130h)	862.2 Mb	122
INSTANCE 13	283472.6 secs (80h)	699.5 Mb	121
INSTANCE 14	520185.3 secs (145h)	965.0 Mb	121
INSTANCE 15	616279.6 secs (170h)	1064.0 Mb	122
INSTANCE 16	389080.2 secs (110h)	819.6 Mb	118
INSTANCE 17	193346.9 secs (53h)	539.0 Mb	111
INSTANCE 18	247569.3 secs (70h)	666.3 Mb	110
INSTANCE 19	211827.3 secs (58h)	549.4 Mb	119
INSTANCE 20	320254.4 secs (88h)	700.4 Mb	111
INSTANCE 41	192422.9 secs (53h)	638.2 Mb	109
INSTANCE 42	220243.8 secs (60h)	673.5 Mb	102

Table des résultats de GLPK pour les autres instances standard

b. Références

[1] Christine Solnon, Serge Fenet, *A study of ACO capabilities for solving the Maximum Clique Problem*, 2006

[2] MinuteLabs.io, PRIMER, *the Evolution Simulator*

[3] kmutya, *Maximum Clique Problem: Linear Programming Approach*, 2019

[4] György Katona, Filipe Mariano, *Python implementation of the CLIQUE subspace clustering algorithm*, 2019

[5] Petter Strandmark, *A shallow fork of GLPK made re-entrant with CMake build files. Tests pass with Visual Studio, Clang, and GCC*, 2017

[6] *GLPK# integer programming*, 2013

[7] Mohammad Soleilmani-Pouri, Alireza Rezvanian, Mohammad Reza Meybodi, *Finding a Maximum Clique using Ant Colony Optimization and Particle Swarm Optimization in Social Networks*

[8] Patric R.J. Östergard, *A fast algorithm for the maximum clique problem*, 2002

[9] Milos Seda, *The Maximum Clique Problem and Integer Programming Models, Their Modifications, Complexity and Implementation*, 2023

[10] Yang Wang, Jin-Kao Hao, Fred 3, Glover, Zhipeng Lü, Qinghua W, *Solving the maximum vertex weight clique problem via binary quadratic programming*

[11] Jocelyn Bernard, Hamida Seba, *Résolution de problèmes de cliques dans les grands graphes*, 2015

[12] André Rossi, Yi Zhou, Jin-Kao Hao, *Programmation linéaire en variables binaires pour le problème de la clique maximum équilibrée dans un graphe biparti*

[13] http://www.cs.ecu.edu/karl/6420/spr16/Notes/NPcomplete/clique.html

[14] Dóra Kardos, Patrik Patassy, Sándor Szabó & Bogdán Zaválnij, https://link.springer.com/article/10.1007/s10100-021-00776-z, 2021

[15] https://fr-academic.com/dic.nsf/frwiki/1374650

[16] Zeynep Ertem, Eugene Lykhovyd, Yiming Wang, Sergiy Butenko, https://www.researchgate.net/publication/326880954_The_maximum_independent_union_of_cliques_problem_complexity_and_exact_approaches, 2020

[17] Satoshi Shimizu, Kazuaki Yamaguchi, Sumio Masuda https://www.sciencedirect.com/science/article/abs/pii/S1572528620300177 ; 2020

[18] Johnson Cordeiro https://www.scribd.com/document/267805406/Maximum-Clique-Problem