POLYTECHNIQUE MONTRÉAL

TECHNOLOGICAL UNIVERSITY

INF6102

Devoir 1 - Etudes de marchés

Ecole Polytechnique de Montréal

2311468 - Barreto Jean

2 Mars 2025

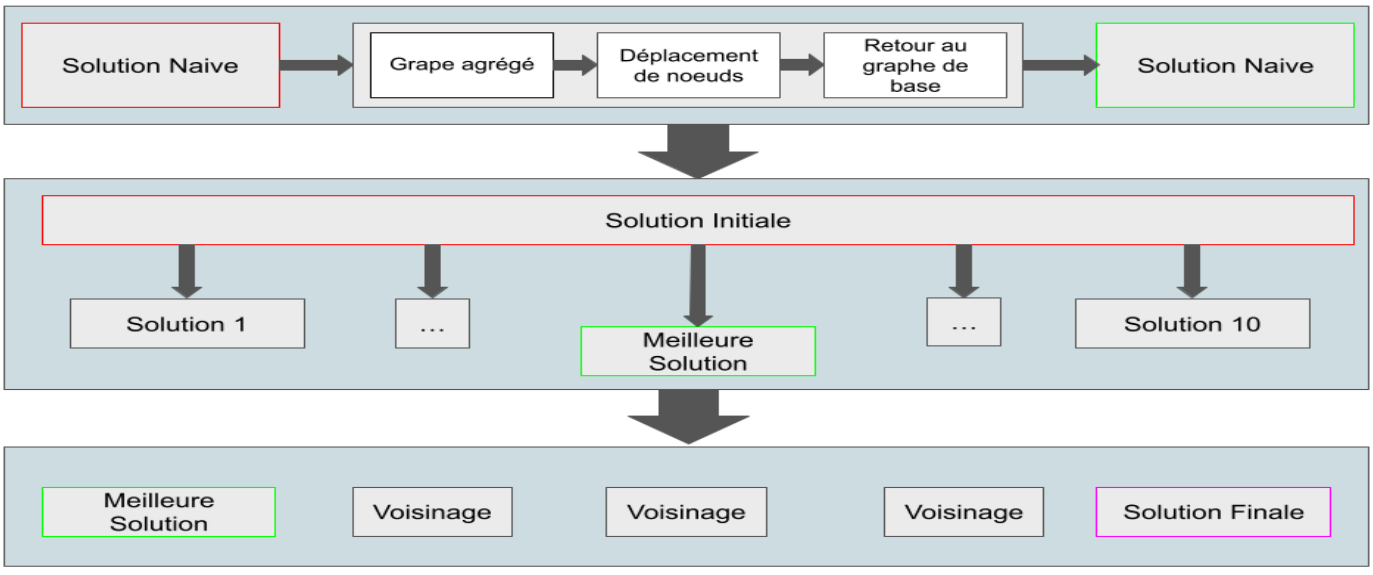
Sommaire :

1- Architecture globale

La conception de l'algorithme repose sur une architecture assez simple. On part de la solution naïve pour construire une *Solution Initiale* via des enchaînements de graphe agrégé et de déplacement de nœuds.

Ensuite la *Phase Grossière* génère une dizaine de solutions se basant sur la *Solution Initiale* en fusionnant des communautés entre elles.

On garde la meilleure et on l'envoie à la *Phase Détaillée* qui retravaille la solution en bougeant des nœuds de communauté via des enchaînements de voisinages.



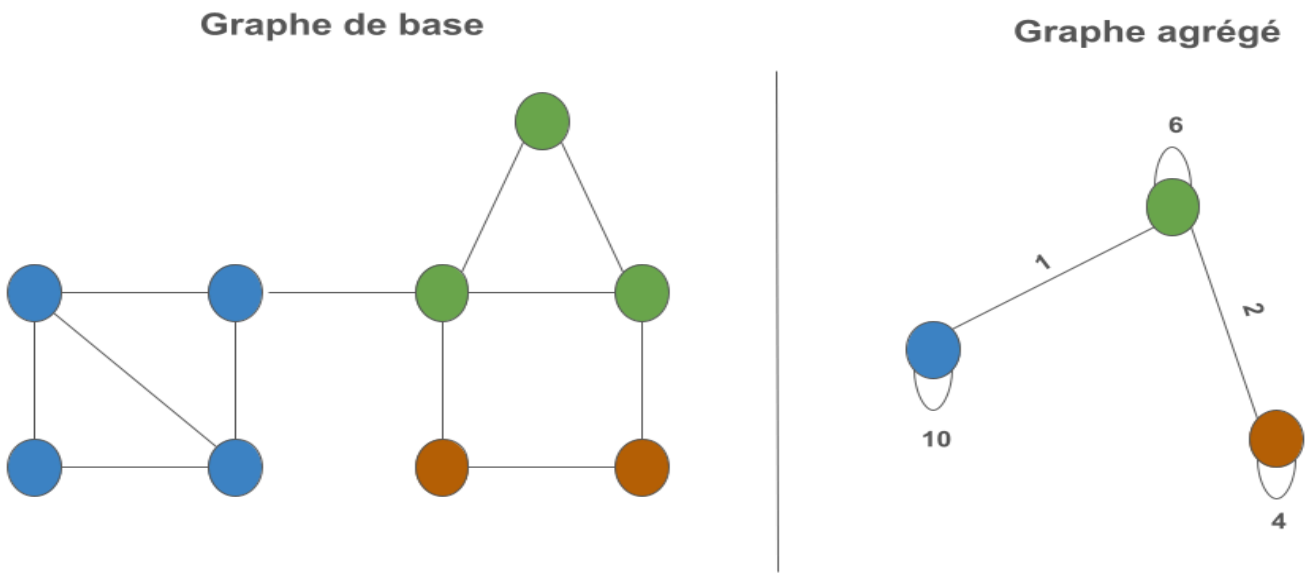
Note : J'ai dû apporter des modifications à la classe *Instance* et *Edges* pour qu'elle supporte la notion de graphe agrégé (cf *Solution Initiale*) ainsi qu'à la façon dont la méthode *solution_value()* fonctionne dans le cas d'un graphe agrégé uniquement, ce qui n'impact pas ma solution finale puisqu'elle n'est pas un graphe agrégé et donc qu'elle se calcule via la méthode originelle.

2 - Solution Initiale

La solution initiale est construite en enchaînant 3 étapes en boucle jusqu'à ce que le nombre de groupes soit réduit à une centaine ou que la modularité n'évolue plus.

Etape 1 : Soit *S* une solution valide pour le graphe *G*, on construit son graphe agrégé *G'* dans une nouvelle instance selon les principes suivants :

- Chaque communauté *c_i* de la solution *S* est transformée en un nœud *N_i* dans la nouvelle instance.
- Deux nœuds *N_i, N_j* de la nouvelle instance sont reliés par *m_extra_ij* arêtes, *m_extra_ij* étant le nombre d'arêtes reliant les deux communautés *c_i* et *c_j.*
- Chaque noeud *N_i* possède 2**m_intra_i* arêtes, *m_intra_i* étant le nombre d'arêtes interne à la communauté *c_i*
- Chaque noeud *N_i* de la nouvelle instance est placé dans sa propre communauté *C_i*



Etape 2 : On parcourt les nœuds *n_i* du nouveau graphe *G'* dans un ordre aléatoire. Soit *n_i I* le nœud en cours d'étude, parmi toutes les communautés *C_j* des voisins de *n_i*, on place *n_i* dans la communauté qui améliore le plus la modularité du nouveau graphe. On continue ainsi jusqu'à ce que plus aucune amélioration ne soit trouvée.

Étape 3 : A partir de la solution *S'* du graphe *G'* on retrouve une nouvelle solution *S* du graphe *G* en fusionnant les communautés *c_i* et *c_j* si les nœuds *N_i* et *N_j* appartiennent à la même communauté dans la solution *S'*.

3 - Phase Grossière

Dans cette phase, on va créer *P* solutions en ne faisant que de la fusion depuis la *Solution Initiale*. Voici les étapes permettant la construction d'un solution *p* parmis *P* :

Initialisation : Soit *S* la solution de départ et *V*, *k* des entiers.

Étape 1: On génère *V* voisins à la solution *S*, chaque voisin étant obtenu en essayant de fusionner *k* paire de communauté aléatoire (mais connectés).

Etape 2 : Soit *v** le meilleur voisin de *V*, si v* améliore *S* alors *S* devient *v** et on revient à l'étape 1. Sinon on décrémente *k* d'une unité, si *k* > 0 on revient à l'étape 1 sinon on s'arrête.

On garde ensuite la meilleure solution *p* de *P*.

4 - Phase Détaillé

Cette phase est une recherche local classique. Soit *S* une solution on choisit *k* noeuds au hasard, pour chaque noeud *n_i* choisi on note *H_i* la liste des communautés de ses nœuds voisins. Il y a donc |*H_1*|*x ... x* |*H_k*| changements possible, on prend le meilleur et cela nous donne un voisin *v*. On recommence jusqu'à avoir *V* voisins. A noter qu'il est possible que certains nœuds ne changent pas de communauté (car il hérite de la communauté d'un de ses voisins qui peut avoir déjà la même communauté).

Si on ne trouve pas de voisins améliorant dans le voisinage *V* on diminue *k* de 1 et lorsque *k*=0 on sauvegarde notre solution et on sélectionne un voisin aléatoire parmi les 10 meilleurs du derniers voisinages *V* afin de diminuer les chances de tomber dans un minimum local. A la toute fin on récupère la meilleure solution *S* parmis toutes celles sauvegardées.

5 - Calcul rapide de la modularité

La fonction fournie pour le calcul de modularité est en O(*M*), *M* étant le nombre d'arêtes. Cela devient problématique pour de grandes instances lorsque l'on veut comparer beaucoup de solutions entre elles. En réalité toute mes modifications de solution ne se résume qu'à deux briques élémentaires : fusionner 2 communautés et déplacer un nœud. Ainsi au lieu de recalculer la modularité au complet après de telles modifications je me contente d'en calculer le gain (positif ou négatif). Pour ce faire je m'inspire de la façon dont la fonction fourni calcul la modularité.

Fusion de 2 communautés : Soit *c_i, c_j* deux communautés et *d_i, d_j* la somme des degrés de leurs nœuds. Si je fusionne *c_i* et *c_j* je me retrouve avec une communauté de *(d_i + d_j)* degrés. Ainsi la variation à une constante près de ce terme est *-(d_i + d_j)^2 - (-d_i^2 -d_j^2) = -2*d_j*d_i*. De plus s' il y avait *L_i, L_j* lien interne dans les communautés *c_i*, c_j il y en a maintenant *L_i + L_j + L_ij* où *L_ij* est le nombre d'arêtes entre *c_i* et *c_j,*. La variation à une constante près de ce terme est donc *L_ij.* Finalement la variation totale de la modularité est donné par *dQ = L_ij/M - 2*d_j*d_i/((2*M)^2).*

Déplacement d'un nœud : Soit *c_i, c_j* deux communautés, *d_i, d_j* la somme des degrés de leurs nœuds et *n_k* un nœud à *d_k* degrés à déplacer de *c_i* vers *c_j*. Après le déplacement *d_i* devient *d_i - d_k* et *d_j* devient *d_j + d_k.* Ainsi la variation à une constante près de ce terme est *-(d_i - d_k)^2 -(d_j + d_k)^2 - (-d_i^2 -d_j^2) = -2*d_k*(d_i - d_j - d_k)*. De plus, si il y avait *L_i, L_j* lien interne dans les communautés *c_i*, c_j alors *L_i* devient *L_i - l_ki* et *L_j* devient *L_j + l_kj* où *l_ki* (respct. *l_kj)* est le nombre de voisins de *n_k* à être dans *c_i* (respct. *c_j)*. La variation à une constante près de ce terme est donc *l_kj - l_ki.* Finalement la variation totale de la modularité est donné par *dQ = (l_kj - l_ki)/M -2*d_k*(d_i - d_j - d_k).*

Ainsi en gardant trace de la somme des degrés de chaque communautés j'ai une complexité en O(max(|*c_i*|,|*c_j*|)) pour la fusion et en O(|*d_k*|) pour le déplacement ce qui est bien mieux.

6- Diverse complexité

Dans cette partie je vais essayer d'expliquer la complexité de mes fonctions clefs.

createSuperInstance(): Correspond à l'étape 1 de la *Solution Initiale.* Une seule boucle sur les arêtes du graphes de base => O(*M*)

nodesShift(): Correspond à l'étape 2 de la *Solution Initiale*. Complexité très dur à calculer.

retrieveSolFromSuperSol(): Correspond à l'étape 3 de la *Solution Initiale*. Une boucle sur le nombre de communautés dans le graphe agrégé par dessus une boucle sur le nombre de nœuds dans la communauté. Soit *N'* le nombre de noeuds et |*C*| le nombre de communautés, on suppose (par simplification) que la taille d'une communauté est en moyenne de *N'*/|*C*| noeuds => O(*N'**|*C*|/*N'*) = O(*N*) or *N'* représente le nombre de communautés |c| dans l'instance de base donc O(|*c*|)

buildSolution(): Créer une instance Solution à partir d'un dictionnaire de taille *N* (nombre de nœuds). Une seule boucle sur *N* => O(*N*)

groups_of_node_dict_to_groups_dict(): Construit un dictionnaire de communauté à partir d'un dictionnaire de noeud. Une seule boucle sur *N* => O(*N*)

La solution étant représentée par un dictionnaire de clé l'identifiant d'un nœud et de valeur sa communauté, toute modification des communautés se fait au plus en O(*N*).

7 - Résultats

Je présente ici divers résultats

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Solution initiale	0.36135	0.48330	0.42276	0.36138	0.37948	0.48734	0.76594	0.80569	0.87136	0.93420	0.848910	0.96423	0.89261	0.88341	0.84611
Post Phase 1	0.41880	0.52048	0.44307	0.38911	0.43575	0.54048	0.80592	0.86404	0.87593	0.93553	0.84891	0.96423	0.89914	0.88341	0.84611
Post Phase 2	0.419789	0.52082	0.44514	0.39230	0.44138	0.54220	0.81447	0.86944	0.879 4918	0.93638	0.84986	0.96435	0.89914	0.88371	0.84625
Nombre de groupes	4	4	4	5	10	12	10	18	26	38	1377	95	32	99	1255
Baseline	0.39	0.51	0.44	0.375	0.43	0.54	0.78	0.84	0.85	0.92	0.84	0.95	0.87	0.86	0.82

On peut voir que ma solution initiale bats la baseline des instances de plus de 4000 nœuds mais pas en dessous. Cela pourrait s'expliquer par le fait que plus il y a de nœuds, plus la solution naïve à de communautés et que donc ma solution finale à le temps de boucler plus de fois avant d'atteindre son critère d'arrêt de 100 communautés.
On note également que je bats toutes les baseline des premières instances après ma Phase 1 à l'exception de l'instance E. Cela s'explique sûrement par une structure interne atypique (comme un nœud central par exemple).
Enfin le nombre de communautés finale augmente avec la taille de l'instance, ce qui est logique si l'on suppose des graphes arbitraires, ce qui ne doit pas être le cas de l'instance K et O qui explose son nombre de communautés finale (probablement une structure avec beaucoup de noeuds isolé i.e. un nombre de voisins moyen par nœud inférieur).

8 - Hyperparamètres

On peut discuter de l'incidence et du paramétrages de divers hyperparamètres tel que :

Critères d'arêt de solutionInitial(): Je l'ai fixé à 100 communautés arbitrairement mais il pourrait être intéressant de l'augmenter. On aurait alors une solution initiale sans doute plus faible mais aussi moins rigide (i.e. mon voisinage pourrait atteindre des nouveaux minima locaux peut être encore meilleure). Ou bien le diminuer pour avoir une phase 1 plus rapide et donc plus de temps alloué à la phase 2, ceci est à contrebalancer avec le temps supplémentaire que prendrait *solutionInitial()* pour s'arrêter

P nombre de solutions à la fin de la Phase 1 : Actuellement fixé à 10, des tests ont montré qu'au-delà la phase 1 prenait trop de temps sur les grosses instances et n'en laissait pas assez à la phase 2. De plus l'intérêt principal d'augmenter P et de lisser l'effet aléatoire par le haut de la phase 1 hors avec $P = 10$ on ne le ressens quasiment plus.

V mergeKComm(): Nombre de voisins générés dans la phase 1, actuellement fixé à 30, ses effets son similaire à *P.*

V mergeKNode(): Nombre de voisins générés dans la phase 2, actuellement égale à 60. L'augmenter aurait pour effet de lisser l'effet aléatoire par le haut et de faire de plus grand saut de modularité de solution en solution mais ralentirait aussi la phase 1.

k mergeKComm() et k mergeKNode(): Fixé à 5, son augmentation aurait pour effet de ralentir énormément l'algorithme, surtout la phase 1 qui a une complexité en puissance de *k* (O(nombre de voisins moyens^k)) alors que la complexité de la phase 2 est linéaire en *k.* De plus l'augmenter pourrait nous faire passer à des solutions bien meilleures mais pouvant nous projeter dans des minimas locaux.