Remerciements

Monsieur: HAMAZ Abdelghani
Je souhaite vous remercier très chaleureusement d'avoir accepter de m'encadrer et d'avoir pris en charge mon sujet de memoire. Vous m'avez donné les moyens, les conseils et l'aide précieuse pour la réalisation de ce projet.

Je tiens à adresser mes plus vifs remerciements à M.Aidene, M.Kasdi, M. Saadi, M.Merakeb Abdelkader et surtout M. Oukacha Brahim, pour les efforts qu'ils ont porté pour la formation Recherche Opérationnelle et pour la qualité de cette formation.

Sans omettre notre considération envers tous les enseignants des départements Recherche Opérationnelle et mathématique.

Table des matières

Introduction 3

1 Optimisation combinatoire et méthodes de résolutions exactes 5
1.1 Introduction 5
1.2 Complexité algorithmique [5] 6
1.2.1 Les principales classes de complexité 8
1.2.2 Classes des problèmes 8
1.3 Problème de partitionnement d'un graphe [4] 10
1.3.1 Partitionnement de graphe 10
1.3.2 Balance de partitionnement 11
1.3.3 Fonction objectif 12
1.4 Méthode exacte 12
1.4.1 Méthode d'énumération explicite 12
1.5 Méthode Descente Gradient 13
1.5.1 Principe de la méthode 13
1.5.2 Algorithme de la méthode 14
1.5.3 Convergence de la méthode 14

2 Meta-heuristiques et méthodes approchées 19
2.1 Les meta-heuristiques [3] 19
2.2 Méthode de recuit simulé 20
2.2.1 Principe de recuit simulé 20
2.2.2 Fonction de transition 21
2.2.3 Température initiale 22
2.2.4 Schéma de décroissement de la température 22
2.2.5 Longueur de température 22
2.2.6 Critère d'acceptation des transitions 23

2.2.7 Critère d'arrêt 23
2.2.8 Algorithme du recuit simulé 24
2.3 Méthode Tabou 25
2.3.1 Présentation de la méthode [1] 25
2.3.2 Principe de la méthode Tabou 25
2.3.3 Algorithme de la méthode Tabou 26

3 Application et programmation des méthodes 28
3.1 Présentation du langage 28
3.1.1 Le langage Python 28
3.1.2 Syntaxes et outils 29
3.2 Algorithmes des méthodes 30
3.2.1 Énumération explicite 30
3.2.2 Descente du gradient 34
3.2.3 Recuit simulé 39
3.2.4 Tabou 43
3.2.5 Les tests 56
3.2.6 Les résultats 57

Conclusion 58

Bibliographie 59

Introduction

Défendre ses part de marché dans un environnent concurrentiel devient une préoccupation face au danger que représente la perte de la clientèle au profit de la concurrence, ainsi que la crise économique récente qui est un défit pour les économistes, qui font appel a la recherche opérationnelle et aux méthodes mathématiques.

La recherche opérationnelle est apparue lors de la deuxième guerre mondiale, lorsque l'état major britannique fit appelle à des équipes de mathématiciens et de physiciens pour l'aider à analyser des questions de stratégies militaires (développement d'un réseau de radars, organisation des convois maritimes).
Après la guerre cette approche systématique et scientifique des problèmes de décision a été transportée au monde économique et industriel ou elle a connu de nombreux succès. Depuis, de nouvelles méthodes et de nouveaux champs d'application ont vu le jour.

Les questions aux quelles s'interesse la recherche opérationnelle peuvent être classées en différentes catégories selon les caractéristiques des situations visées, des modèles proposés pour les représenter et des techniques de résolution utilisées on peut par exemple évoquer les problèmes combinatoires, aléatoire ou concurrentiels.
Les problèmes combinatoires apparaissent lorsque les réponses possibles (les solutions réalisables) sont trop nombreuses pour pouvoir être énumérées.

Les problèmes de partitionnment des graphes sont des problèmes combinatoires et qui représentent une modélisation importante a de nombreux problèmes concrets (réseaux et partitionnement des cartes, cheminement...etc).
Étant donnée l'importance de ces problèmes, de nombreuses méthodes de résolution ont été développées en recherche opérationnelle. Elle peuvent être classées sommairement en

deux grandes catégories: les méthodes exactes qui garantissent la complétude de la solution et les méthodes approchée qui perdent la complétude pour gagner en efficacité.
Notre travail consiste à étudier et à comparer les deux méthodes sus-citées. Pour cela, nous avons organisé notre mémoire comme suit: le premier chapitre porte sur des définitions et concepts de base concernant l'optimisation combinatoire et le problème de partitionnement , les graphes, leurs classe de complexité, ainsi les méthodes exactes de résolution pour un problème de partitionnement.
Le second chapitre est une présentation des méthodes de résolution approchées dont on définit deux méthodes qui représentent des labels dans le monde des meta-heuristique et qui sont la méthode Tabou et la méthode de Recuit Simulé. Quant au troisième chapitre , nous avons présenté une application qui est une étude comparative des diverses méthodes de partitionnement en retenant le temps moyen et la solution moyenne comme critères de comparaison. Le travail est achevé par une conclusion .

Chapitre 1

Optimisation combinatoire et méthodes de résolutions exactes

1.1 Introduction

L'optimisation combinatoire occupe une place très importante en recherche opérationnelle, en mathématiques discrètes et en informatique. Son importance se justifie d'une part par la grande difficulté des problèmes d'optimisation et d'autre part par de nombreuses applications pratiques pouvant être formulées sous la forme d'un problème d'optimisation combinatoire.
Autrement dit: l'optimisation combinatoire est une discipline combinant diverses techniques des mathématiques discrètes et de l'informatique afin de résoudre des problèmes d'optimisation dont la structure sous-jacente est discrète (généralement un graphe).

Un problème d'optimisation combinatoire est un problème qui consiste à maximiser (ou minimiser) une certaine fonction linéaire ou non linéaire sur un ensemble fini d'éléments. Un tel problème peut être présenté de la manière suivante:
Étant donné un ensemble fini $E = \{e_1,e_2,...,e_n\}$, une famille $\mathbf{F}$ de sous ensemble de E et un système de poids $(w(e_1),w(e_2),...,w(e_n))$ associé aux éléments de E, trouver un ensemble F de $\mathbf{F}$ de poids $w(F) = \sum_{e_i \in F} w(e_i)$ maximum (ou minimum). Ici la famille $\mathbf{F}$ est l'ensemble des solutions du problème, elle permet de représenter diverses structures combinatoires comme par exemple des chemins, des cycles, des arbres,... etc, dans les graphes. Ce genre de problème apparaît dans des domaines divers que l'industrie, le transport, les télécommunications.

En général, si I est une instance d'un problème de minimisation définie comme étant un couple (X,f) ou $X \in S$ est un ensemble fini de solutions admissible, et f une fonction de coût (ou objectif) à minimiser

$$f : X \to R.$$

Le problème est de trouver $s^* \in X$ tel que $f(s^*) \leq f(s)$ pour tout élément $s \in X$. Notons que d'une manière similaire, on peut également définir les problèmes de maximisation en remplaçant simplement $\leq$ par $\geq$.

1.2 Complexité algorithmique [5]

Définition 1.1. La complexité algorithmique est un moyen d'évaluation du coût d'un algorithme. Cette complexité mesure le nombre d'opérations élémentaires et le coût mémoire. Cela permet surtout de connaître le type de croissance en fonction de la taille des données. La complexité d'un problème est de l'ordre de grandeur de la complexité (dans le pire des cas) du meilleur algorithme connu pour le résoudre. La théorie de la complexité algorithmique s'intéresse à l'estimation de l'efficacité des algorithmes. Elle s'attache à la question: entre différents algorithmes réalisant une même tache, quel est le plus rapide et dans quels conditions?

Remarque 1.1. Quelle complexité cherche-t-on à calculer?

En effet, pour deux données de même taille, l'exécution d'un algorithme peut utiliser des quantités de ressources différentes (les ressources considérées peuvent être le temps, la mémoire et le matériel). On définit donc trois mesures de complexité, la complexité dans le meilleur des cas, la complexité en moyenne et la complexité dans le pire des cas.

La complexité dans le meilleur des cas est définie par:

$$\inf_A(n) = \inf\{\text{cout}_A(d)(d/d \quad \text{de taille } \ n)\}.$$

La complexité dans le pire des cas est définie par:

$$\sup_A(n) = \sup\{\text{cout}_A(d)(d/d \quad \text{de taille } n)\}.$$

Pour définir la complexité en moyenne, il faut disposer pour tous n d'une mesure de probabilité P sur l'ensemble des données de taille n, cette mesure ne fait souvent qu'approximer l'espace réel des données, car il est très dur de poser un modèle de données "réelles". On suppose par exemple souvent que toutes les données de même taille sont équiprobables, ce qui est peu réaliste. La complexité en moyenne est alors donnée par:

$$Moy_A(n) = \sum_{d \text{ de taille } n} P(d) \times cout(d).$$

Quand on parle de la complexité d'un algorithme sans préciser laquelle, c'est souvent de la complexité temporelle dans le pire des cas qu'on parle. La complexité dans le meilleur des cas n'est pas très utilisée, la complexité en moyenne est d'une certaine façon celle qui révèle le mieux le comportement "réel" de l'algorithme à condition de disposer d'un modèle de la répartition des données. Mais bien sûr elle ne garantie rien sur le pire des cas et elle est souvent dure à calculer, même de façon approximative. Son calcul peut nécessiter la mise en oeuvre de techniques mathématiques non élémentaires.

On ne calcule en général pas exactement la complexité mais on se contente de calculer son ordre de grandeur, voire de borner celui-ci. Par exemple, si on fait $n^2 + 2 \times n - 5$ opérations, on retiendra seulement que l'ordre de grandeur est $o(n^2)$.

Règles générales

- Le temps d'exécution (t.e) d'une instruction ou d'un test est considéré comme constant C.
- Le temps d'une séquence d'instruction est la somme des t.e des instructions qui la composent.

1.2.1 Les principales classes de complexité

Complexité	Type	Classe
$0(1)$	temps constant	raisonnable
$0(\log(n))$	logarithmique	raisonnable
$0(\text{n}\log n)$	quasi-linéaire	raisonnable
$0(\text{n}^{\log(n)})$	quasi-polynomiale	raisonnable
$0(\text{n})$	linéiare	raisonnable
$0(\text{n}^2)$	polynomial	raisonnable
$0(\text{n}^k)$	polynomial	pas raisonnable pour $\text{K} \geq 5$
$0(2^n)$	exponentiel	pas raisonnable
$0(\text{n}!)$	factorielle	pas raisonnable

1.2.2 Classes des problèmes

Le but de la théorie de la complexité est la classification des problèmes de décision suivant leur degré de difficulté de résolution.
La théorie de la complexité distingue plusieurs classes de problèmes, mais les plus connues sont les suivantes:

La classe *P*
Un problème de décision est dans P s'il peut être décidé par un algorithme déterministe en un temps polynômial par rapport à la taille de l'instance. On qualifie alors le problème de polynômial. C'est la classe des problèmes dits **faciles**.

La classe *NP* (No deterministic polynomial time)
Cette classe referme tous les problèmes de décision au quels on peut associer un ensemble de solutions potentielles (de cardinal au pire exponentiel) de telle sorte qu'on puisse vérifier en un temps polynômial si une solution potentielle satisfait la question posée. Le terme non déterministe désigne un pouvoir qu'on incorpore à un algorithme pour qu'il puisse deviner la bonne solution.

La classe *NP-complet*
Un problème de *NP* est *NP-complet* si tout problème de *NP* s'y réduit en temps polynômial à ce problème.
De nombreux problèmes d'optimisation combinatoire (la plupart de ceux qui sont vraiment

intéressants dans les applications) ont été prouvés *NP-complet*. Cette difficulté n'est pas seulement théorique et se confirme hélas dans la pratique. Il arrive que des algorithmes exactes de complexité exponentielle se comportent efficacement face à de très grosses instances pour certains problèmes et certaines classes d'instances.

Nous rappelons quelques définitions de la théorie des graphes nécessaires dans la suite.

Définition 1.2. – On appelle longueur d'une chaîne le nombre d'arêtes qui la compose.
- La distance entre deux sommets d'un graphe est la longueur minimale d'une chaîne reliant ces deux sommets.
- Le diamètre d'un graphe est la plus grande distance entre deux sommets quelconques de ce graphe.[2]

Définition 1.3. (**Graphes pondéré**)
- On dit qu'un graphe est pondéré si on a affecté à chaque arête un nombre positif (quel que soit sa signification).
- Ce nombre positif est alors appelé poids de l'arête.
- Le poids d'une chaîne est la somme des poids des arêtes qui la compose.
- La plus courte chaîne entre deux sommets est la chaîne de poids minimal entre ces deux sommets.[2]

Remarque 1.2. Il ne faut pas confondre les notions de plus courte chaîne (qui correspond à une chaîne de poids minimal) et la chaîne de longueur minimal (qui correspond à un nombre d'arêtes minimal).

Définition 1.4. Graphes étiqueté Un graphe étiqueté est un graphe ou chacune des arêtes est affectée d'une lettre, d'un mot, d'un nombre ou d'un symbole.
Ces symboles sont appelés étiquettes.[2]

Remarque 1.3. Un graphe pondéré est donc un cas particulier de graphe étiqueté.

1.3 Problème de partitionnement d'un graphe [4]

Le problème de partitionnement d'un graphe a pour but de découper un graphe en différentes parties (partitions) qui satisfont certaines contraintes (telle que l'équilibre des parties) et qui optimisent une certaine fonction objectif (telle que le coût de coup). Il possède de nombreuses applications comme la conception de circuits intégrés électroniques, la répartition de charge pour les machines parallèles ou la segmentation d'images. Cependant, le partitionnement de graphe est un problème complexe *NP-complet*, dont la solution ne peut pas être trouvée au moyen d'une méthode de résolution exacte. Pour ce type de problème, la recherche locale est une approche adéquate. (dont les méthodes de résolution: le recuit simulé et la recherche tabou font partie, voire" chapitre 2 ").

Définition 1.5. (**Partitionnement**) Le mot "partitionnement" exprime la création d'une partition. La partition est le résultat de la division en parties d'un ensemble. En mathématiques, une partition d'un ensemble S est une famille de sous-ensemble de E tels qu'elles sont disjointes deux à deux et la réunion dont est l'ensemble S.

1.3.1 Partitionnement de graphe

Étant donné un graphe non-orienté G=(V,E), où V est l'ensemble des sommets et E est l'ensemble des arêtes qui relient des paires de sommets. Les sommets et les arêtes peuvent être pondérés, ou $| v |$ est le poids du sommet v et $| e |$ est le poids de l'arête e. le problème du partitionnement de graphe consiste à deviser G en K partions disjointes.
Au point de vue mathématique, on peut partitionner les sommets ou bien les arêtes. Par contre, dans la plupart des applications, on ne s'intéresse qu'au partitionnement des sommets de graphe.

Proposition 1.1. *Soit un graphe G=(V,E) et un ensemble de K sous-ensembles de V,*

noté

$$P_K = \{V_1, V_2, ..., V_K\}.$$

On dit que P_K est une partition de G si :

- *Aucun sous-ensemble de V qui est élément de P_K n'est vide:*

$$\forall i \in \{1,...,K\}, V_i \neq \emptyset.$$

- *Les sous-ensembles de V qui sont éléments de P_K sont disjoints deux à deux:*

$$\forall (i,j) \in \{1,...,K\}, \quad i \neq j \quad , V_i \cap V_j = \emptyset.$$

- *L'union de tous les éléments de P_K est V:*

$$\bigcup_{i=1}^{K} V_i = V.$$

V_i: sont appelées parties de la partition P_K.

1.3.2 Balance de partitionnement

Soit un graphe G=(V,E) et une partition $P_K = \{V_1, V_2, ..., V_K\}$ de ce graphe en K parties. Le poids d'une partie V_i de P_K est :

$$\text{poids}(V_i) = \sum_{v_i \in V_i} \text{poids}(v_i).$$

Le poids moyen d'une partie V_i de P_K est:

$$\text{poids}_{Moy}(V,K) = \sum_{v \in V} \text{poids}(v) \setminus K.$$

La balance de la partition P_K est égale à la division du poids de la partie de poids maximal de P_K par le poids moyen d'une partie:

$$\text{balance}(P_K) = \frac{\max\{\text{poids}(V_1), \text{poids}(V_2), ..., \text{poids}(V_K)\}}{\text{poids}_{Moy}(V,K)}.$$

1.3.3 Fonction objectif

On définit le coût de coupe entre deux parties d'une partition V_a et V_b est la somme des arêtes qui ont deux bouts dans les deux parties:

$$\text{coupe}(V_a,V_b) = \sum_{v_a \in V_a, v_b \in V_b} \text{poids}(v_a,v_b).$$

poids(v_a,v_b) est le poids de l'arête (v_a,v_b), de plus s'il n'existe pas d'arête (v_a,v_b), le poids(v_a,v_b) $= 0$.
La plus simple des fonctions d'objectif est le coût de coupe d'une partition. Le but est de minimiser la somme des poids des arêtes entre les parties de la partition P_K.

Pour la résolution des problèmes, en recherche opérationnelle, le choix de la méthode de résolution constitue une étape cruciale. Il existe deux grande familles de méthodes de résolution. D'un coté, les méthodes exactes(complète) qui garantissent la complètude de la résolution, de l'autre les méthodes approchées heuristique, meta-heuristique (incomplètes) qui perdent en complètude pour gagner en efficacité.

1.4 Méthode exacte

On peut définir une méthode exacte comme une méthode qui garantit l'obtention de la solution optimale pour un problème d'optimisation. L'utilisation de ces méthodes s'avèrent particulièrement intéressante, mais elles sont souvent limitées au cas des problème de petite taille. Il existe différents méthodes exactes pour la résolution de tels problèmes dont l'énumération explicite ou implicite, ici on s'intéresse uniquement à l'énumération explicite.

1.4.1 Méthode d'énumération explicite

Pour la résolution d'un problème de partitionnemnt l'énumération explicite nous permet d'obtenir toutes les solutions possibles du problème.

Le principe de la méthode est très simple, on instancie chaque variable avec une valeur de son domaine (domaine des solutions réalisables). Pour chaque intanciation on teste les

contraintes.
(Ici, c'est le cas de minimisation de la fonction coût), si elle répond au problème (optimise) on l'accepte sinon on la rejette et on refait la procédure jusqu'a obtenir toutes les solutions optimales. On choisira que les solutions qui respecte la condition que toutes les partition du partitionnemnt sont de taille à peu près équitables.

```
On peut résumer sa en une petite forme itérative:\\
1.Soit S0 une solution initiale.
2.Grace à une fonction en énumère toutes les solutions possibles qui satisfait
la contrainte d'a peu près équitables, on aura s_enum
3.Si S_enum meilleur que S0 alors:
     S0:=S_enum
  Fsi.
4.et on passe a l'autre solution énumérée.
```

1.5 Méthode Descente Gradient

Cette section présente ce qu'est une Descente de Gradient(DG). La (DG) s'applique lorsque l'on cherche le minimum d'une fonction dont on connaît l'expression analytique, qui est décidable, mais dans le calcul direct du minimum est difficile. C'est un algorithme fondamental à connaître car utilisé partout sous forme dérive.

1.5.1 Principe de la méthode

Lorsque il est question de minimiser $f : \mathbb{R}^k \rightarrow \mathbb{R}$, une condition nécessaire est d'avoir un gradient nul, mais cette condition n'est pas suffisante.
On peut donc être amené à utiliser des méthodes de descente de gradient.
Une descente produit une séquence x^k telle que :$x^{k+1} = x^k + \varepsilon^k d^k$ où d^k est une direction de descente, $\varepsilon^k > 0$ est le pas, de sorte à avoir $f(x^{k+1} < f(x^k))$ (sauf pour le x^k optimal).
Dans la cadre de la descente de gradient, on choisit $d^k = -\nabla_x f(x^k)$.
Il existe plusieurs stratégies pour définir le pas $\varepsilon^k > 0$:
1.Le pas constant: $\varepsilon^k = \varepsilon$. Dans ce cas, l'algorithme n'est pas toujours convergent.
2.un pas décroissant $\varepsilon^k \propto \frac{1}{K}$ (avec $\sum_k \varepsilon^{(k^2)} < \infty$).

3.La "line search" qui cherche à trouver $\min_{\varepsilon} f(x^k + \varepsilon)d^k$:

- Soit de manière exacte (en pratique, c'est une opération coûteuse et souvent inutile). Toujours convergent.
- soit de manière approchée. Toujours convergent.

1.5.2 Algorithme de la méthode

On construit une suite x_n qui s'approche itérativement du minimum.
1.initialisation:
On part d'un point initialx_0.
2.iteration $(n+1)$:
A partir du point x_n on calcule le point x_{n+1} par:
$x_{n+1} = x_n - \alpha(x_n)$.
Test de convergence : $\|(x_{n+1})\| < \varepsilon$?
(On boucle à l'étape 2 si non convergence).
Quand on est arrivé à convergence on a $x_{min} = x_{n+1}$ et $\|\nabla^{\rightarrow} f(x_{n+1})\| < \varepsilon$ avec ε petit. On vérifie donc bien la condition nécessaire pour avoir un minimum (la dérivée s'annule).
Quelques remarques
1.α est le pas d'avancement à chaque itération (à bien choisir).
2.Sans condition spécifique sur la fonction f, on n'est pas assuré d'obtenir un minimum global. Seulement un minimum local.
Pour illustrer cette méthode, nous présentons, en dimension 1, l'exemple ci dessous.

1.5.3 Convergence de la méthode

Dans le cas de la descente de gradient, on peut garantir asymptotiquement la convergence vers un minimum si les pas de gradient η sont positives et sont graduellement réduit:

$$\sum_{t=1}^{\infty} \alpha_t = \infty.$$

et

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

Exemple 1. *Cet exemple présente la DG sur la minimisation de la fonction f de la figure 1 :*

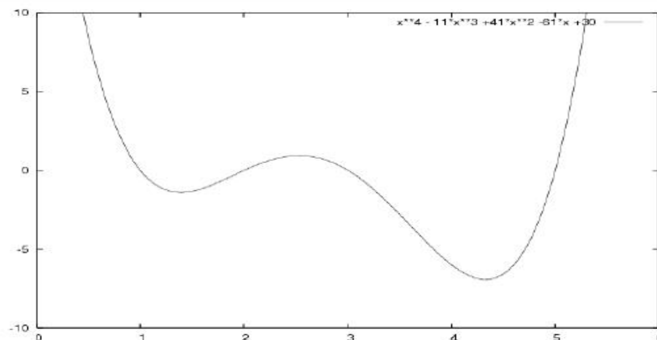

FIG. 1.1 – *La fonction $f(x) = (x-2)(x-3)(x-5)(x-6)$ dessiné sur [0,6]*

Le problème est de trouver la valeur de x qui minimise $f(x)$. Dans cet exemple, on connaît l'expression analytique de la fonction f :

$$\begin{aligned} f(x) &= (x-1)(x-2)(x-3)(x-5) \\ &= (x^4 - 11x^3 + 41x^2 - 61x + 30) \end{aligned}$$

On connaît aussi sa dérivée :

$$f'(x) = 4x^3 - 33x^2 + 82x - 61$$

Pour trouver analytiquement le minimum de la fonction f, il faut trouver les racines de l'équation $f'(x) = 0$, donc trouver des racines d'un polynômes de degré 3, ce qui est "difficile". Donc on va utiliser la DG. La DG consiste à construire une suite x_i (avec x_0 fixé au hasard) de manière itérative :

$$x_{i+1} = x_i - \eta f'(x_i).$$

Remarque: cette formule se note aussi:

$$\Delta x = -\eta f'(x).$$

Δx représente la valeur que l'on ajoute à x à chaque itération.
On peut donner un critère de fin à la DG par exemple si $\Delta x < \varepsilon$ ou si $i > nombre_max$.
Pour comprendre ce que fait effectivement la DG, la suite donne des exemple d'exécutions avec différentes valeurs initiales de x_0 et η.

Si $x_0 = 5$ et $\eta = 0.001$, DG trouve $x = 4.32$ et $f(x) = -6.91$ en 447 itérations.
sur la figure1, verifier que ce résultat ($x = 4.32$,$f(x) = -6.19$) est possible.
Si $x_0 = 5$ et $\eta = 0.01$, DG trouve $x = 4.32$ et $f(x) = -6.91$ en 39 itérations.
Pourquoi le nombre d'itérations a diminue lorsque η a augmenté?
η s'appelle le "pas d'apprentisage". La figure2 montre la convergence de la série de valeur x_i vers le minimum de f.

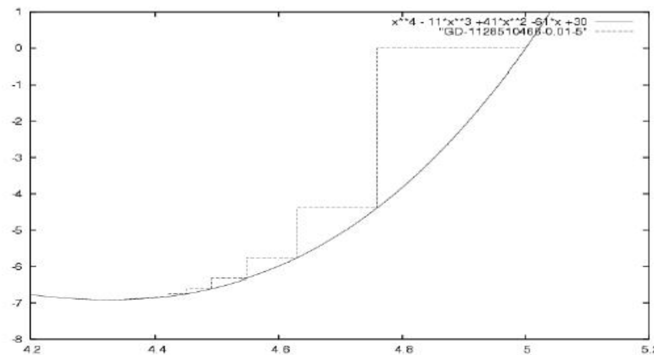

Fig. 1.2 – *La série des valeurs x_i converge vers le point (4.32,-6.91)*

Si $x_0 = 5$ et $\eta = 0.1$, que se passe-t-il? La réponse est donnée par la figure3: à partir de la 20eme itération, la série de valeurs oscille être deux point : $(4.48, -6.63)$ et $(4.10, -6.46)$.
La suite n'arrive pas à atteindre le minimum car le pas d'apprentissage est trop grand.
Si $x_0 = 5$ et $\eta = 0.17$, la figure4 montre ce qui se passe : n'importe quoi,le pas d'apprentissage est grand.
Si $x_0 = 5$ et $\eta = 1$, que se passe t-il?

On peut aussi voir l'effet de la valeur de départx_0.
Si $x_0 = 0$ et $\eta = 0.01$, DG trouve $x = 1.39$ et $f(x) = -1.38$ en 61 itérations.
Pourquoi la valeur finale est elle différente de la valeur finale lorsque $x_0 = 5$?
Réponse sur la figure5. Ici le pas d'apprentissage est suffisamment petit, mais la valeur initiale fait que la série converge vers le minimum local (1.39,-1.38).

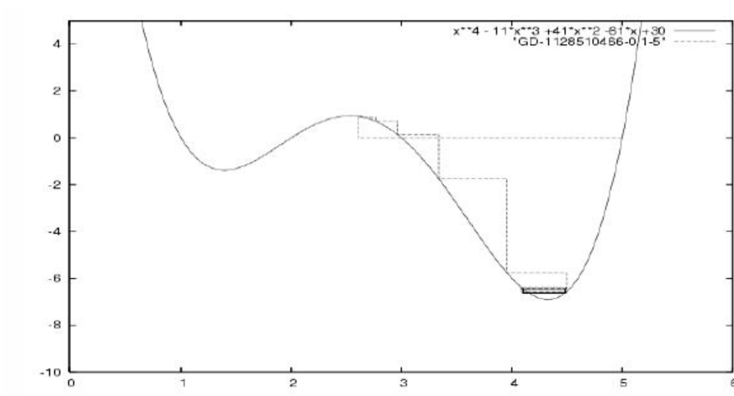

FIG. 1.3 – *si $\mu = 0.10$, la série des valeurs x_i oscille entre les points (-4.48,-6.63) et (4.10,-6.46).*

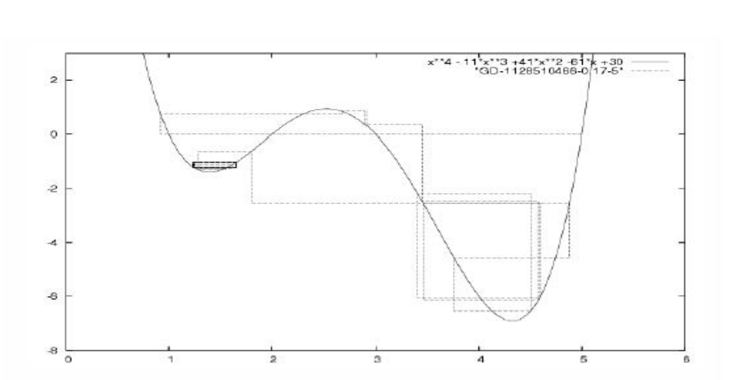

FIG. 1.4 – *si $\mu = 0.17$, la série de valeurs x_i effectue un "joli parcours"*

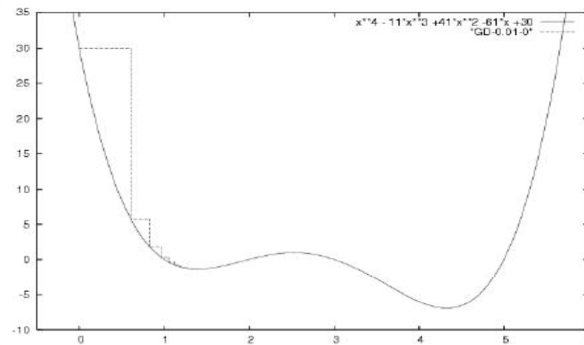

FIG. 1.5 – *si* $x_0 = 0$ *et* $\mu = 0.01$*, la DG converge vers un minimum local.*

En résumé, on voit que la DG trouve le bon résultat si la valeur de départ x_0 *est plus proche du minimum global que d'un minimum local, et si le pas d'apprentissage est suffisamment petit. Si le pas d'apprentissage est trop grand, DG a un comportement chaotique ou diverge. Si la valeur de départ est mal choisie on trouve un minimum local au lieu de trouver le minimum global.*

Chapitre 2

Meta-heuristiques et méthodes approchées

L'objet de ce chapitre est précisément de présenter les approches de résolution de notre problème. La première section présentera d'une manière générale ce que sont les meta-heuristiques, les trois sections qui suivront porteront chacune sur une meta-heuristique particulière.

2.1 Les meta-heuristiques [3]

Les meta-heuristique est une classe de méthodes s'appliquant à toutes sortes de problèmes combinatoires. Elle peuvent également s'adapter aux problèmes continus. Ces méthodes, qui comprennent notamment la méthode du recuit simulé, les algorithme génétiques, la méthode de recherche tabou, les algorithmes de colonies de fourmis, sont proposées dans les années 1980 pour résoudre des problèmes d'optimisation difficiles pour lesquelles on ne reconnaît pas de méthode classique plus efficace.
Le principe de base d'une meta-heuristique est de parcourir l'espace des solutions à la recherche de son minimum global en utilisant des mécanisme pour éviter des minimum locaux.
D'autres termes, les meta-heuristiques tentent de trouver l'optimum global d'un problème d'optimisation difficile, sans être piégé par les optimums locaux.
Le but d'une meta-heuristique est de résoudre un problème d'optimisation donné, si l'on est capable d'attribuer une "qualité" à une solution du problème, alors la meta-heuristique

va rechercher la "meilleure" solution en fonction de ce critère (on parle d'optimum global). En d'autres termes, si le problème est de faire le meilleur gâteaux possible, nous poserons le goût comme critère de qualité. La meta-heuristique va chercher la meilleure combinaison d'ingrédients, une combinaison "optimale", qui permettra d'obtenir le meilleur goût.

Les meta-heuristiques peuvent été classées en deux approches: approche de voisinage et approche à base de population. Les meta-heuristique de voisinage acceptent des solutions dégradées pour s'extraire des minimum locaux tandis que les meta-heuristique de population utilisent un mécanisme collectif pour s'en sortit des minimum locaux.

Ce chapitre présente deux meta-heuristiques: recuit simulé,et la recherche tabou. (Les deux meta-heuristique sont dites aussi méthodes de recherche locale).

2.2 Méthode de recuit simulé

[6] [7] [3] Le recuit simulé a été proposé par "kirkpatrick" et ses collaborateurs [1983], lorsqu'ils ont montré l'analogie entre la recherche d'une solution optimale en optimisation combinatoire et la recherche de l'état d'équilibre thermique d'un solide (énergie minimale) en thermodynamique. Cette méthode est ainsi inspirée de celle de Métropolis [1953] qui était utilisée pour modéliser l'évolution d'un solide vers son état d'équilibre thermique.

2.2.1 Principe de recuit simulé

Dans le domaine de la métallurgie, l'état de la matière dépend de la température: elle est en général à l'état liquide à haute température et à l'état solide a basse température. Deux façons de diminuer la température sont possible:
La première est une baisse très brusque, on obtient ainsi un verre, caractéristique de la technique de trempe. Il s'agit d'une structure avec un minimum local d'énergie.
La deuxième est une baisse progressive de la température, laissant le temps aux atomes d'atteindre l'équilibre, on tendra alors vers une structure de plus en plus régulière avec un minimum global d'énergie, en obtenant ainsi un cristal.

Si l'abaissement n'est pas assez progressif, on obtient une structure avec des défauts. Ceci peut être corrigé par un réchauffement léger de la matière de façon à mettre une mobilité aux sous le nom recuit.
Le recuit simulé cherche donc à imiter le processus précèdent dans la recherche d'un minimum global d'une fonction objectif.
Il fait une analogie, d'une part, entre la fonction objectif f et l'énergie E d'un solide et, d'autre part, entre un paramètre global de contrôle Trs (pseudo température) et la température de recuit T. A partir d'une solution initiale s_0 et d'une température initiale Ts_0 l'algorithme recuit simulé commence à explorer toutes les solutions en procédant par étape.
A chaque étape, l'algorithme fait des transitions de la solution courante dans son voisinage (s). Après une transition, on mesure la variation $\triangle f$. Si elle est négative (la fonction f diminue), alors la nouvelle solution est acceptée, sinon (la fonction f augmente), la nouvelle solution est acceptée avec une probabilité d'acceptation P. Cette acceptation est utilisée pour s'échapper des optimum locaux, elle est analogue à la phase du réchauffement (recuit) effectué pour corriger les défauts de la matière. La probabilité d'acceptation P est basée sur la fonction de Boltzman ($\exp^{-\frac{(\Delta c)}{Trs}}$).
Après un nombre de transition Nrs jugé suffisant pour atteindre tout le voisinage de la solution courante (toutes les configurations de la structure à cette température), la température est abaissée à une température Trs_{K+1}, selon une stratégie *appelée schéma de décroissement de la température*. Le processus s'arrête lorsque l'on atteint la température de recuit (inférieure à une valeur arbitraire égal à 1) ou quand l'évolution de la solution n'est plus possible. L'utilisation de l'algorithme recuit simulé nécessite la définition d'une fonction de transition afin de parcourir l'espace de recherche et la spécification d'un nombre de paramètre , comme la température initiale, le critère d'acceptation de la transition, le schéma de décroissement de la température, la longueur de température et le critère d'arrêt.

2.2.2 Fonction de transition

Une fonction de transition est une fonction qui associe à une solution, une solution $s^{'}$ voisine de s c'est-à-dire $s^{'} \in N(s)$, ou $N(s)$ est le voisinage de s.

2.2.3 Température initiale

C'est le paramètre qui détermine le nombre de solution qui vont être acceptées initialement. Il doit être suffisamment élevé pour favoriser l'acceptation d'un grand nombre de mauvaises solutions (solutions avec valeur de f élevée). La valeur de la température initiale Trs_0 est généralement choisie de manière à avoir un taux d'acceptation initiale compris entre 50% et 80%.
Par exemple $Trs_0 = r \times \max(\Delta f)$ avec $r \gg 1$.

2.2.4 Schéma de décroissement de la température

La température est réduite progressivement au cours du processus selon une méthode de décroissement appelée *schéma de décroissement ou encore schéma de recuit*. Parmi les nombreux schémas de recuit qui ont été proposés deux sont fréquemment utilisée.
Le premier est de type géométrique, il consiste à multiplier l'ancienne température par une constante α positive inférieur à 1 (généralement, α est comprise entre 0,9 et 1).

Ce schéma est donné par :$Trs_{K+1} = \alpha \times Trs_K$ avec $\alpha \simeq 0{,}9995$.
La deuxième est de type arithmétique définit comme: $Trs_{K+1} = Trs - \theta$ avec θ une constante positive.

2.2.5 Longueur de température

Elle représente la longueur du pas ce paramètre représente le nombre de solutions à visiter avant d''atteindre l'équilibre thermique. En optimisation, c'est le nombre de solutions qu'on peut atteindre à partir de la solution courante. La longueur de température est souvent fixée à une valeur déduite d'expériences préalables. Après avoir effectué ce

nombre de transitions, on change la température en utilisant le schéma de décroissance de la température.

2.2.6 Critère d'acceptation des transitions

Les critères les plus connus sont ceux de Métropolis et de glauber. Chacun des deux critères accepte les transitions avec une probabilité P différente:

Métroplois:

$$P = \begin{cases} 1,\text{si} \quad \Delta C \leq 0; \\ \exp^{-\frac{(\Delta c)}{Trs}},\text{sinon}; \end{cases} \tag{2.1}$$

Glauber:

$$P = \frac{\exp^{-\frac{(\Delta c)}{Trs}}}{1 + \exp^{-\frac{(\Delta c)}{Trs}}}$$

2.2.7 Critère d'arrêt

Les plus utilisés sont ceux qui proposent de signaler la fin du processus soit:

- Lorsque la température est inférieure à 1,$\varepsilon = 10^{-2}$.
- En limitant le nombre d'itération à une valeur $Maxiter \approx 10000$.
- En limitant le nombre d'itérations sans changement de coût à une valeur $MaxGel \approx 200$.

2.2.8 Algorithme du recuit simulé

```
Choisir S°,\\ choisir une solution réalisable
Trs:=Trs°,\\initialiser la température Initialiser MaxGel,
Maxiter, \varepsilon,\alpha Stop:=faux S:=S°
Meilleur_sol:=S°,\\initialiser la meilleure solution
Niter:=0,\\initialiser le nombre d'itérations
NGel:=0,\\initialiser le nombre d'itérations sans améliorations\\
Tantque Stop = faux
Faire,\\début du pas de température
            Niter := Niter + 1
            Générer S' dans N(S),\\transiter dans le voisinage de S
            \Delta f := f(S')-f(S)), calculer la variation de la fonction f\\
            Si \Delta f < = 0  alors
                 S:=S',\\accepter la solution
            sinon
                 générer aléatoirement r\in[0,1]
                     Si r\leq \exp^{-\frac{(\Delta c)}{Trs}} alors S:=S',\\accepter
                          suivant le critère de Métropolis
                     FinSi
            FinSi
            Si f(S')\leq f(Meilleur_sol) alors
                 Meilleur_{sol}:= S^{'}
            Finsi
            Si \Delta f=0  alors
                 NGel:=NGel +1
            Sinon
                 NGel:=0
            FinSi
            Trs := \alpha\times Trs,\\abaisser la température
            Si (Trs\leq \varepsilon) ou (Niter=MaxGel) alors
                 Stop:=vraie
            Fait.
```

2.3 Méthode Tabou

2.3.1 Présentation de la méthode [1]

La recherche Tabou est une meta-heuristique d'optimisation présentée par Fred Glover en 1986. On trouve souvent l'appellation recherche avec tabous en français. Cette méthode est une meta-heuristique itérative qualifiée de recherche locale au sens large.

2.3.2 Principe de la méthode Tabou

La procédure de la recherche Tabou est destinée à trouver un optimum global d'une fonction F défini sur un ensemble de solutions réalisables X. Pour chaque solution S de X on définit un voisinage $N(S)$ constitué de toutes les solutions réalisables qui peuvent être obtenues par l'application d'une simple modification m sur S.

La procédure commence par une solution réalisable initiale et tente d'atteindre un optimum global du problème par un déplacement à chaque étape, dés qu'une solution réalisable est obtenue, on génére un sous-ensemble V de $N(S)$ et on se déplace vers la meilleure solution S' dans V. Si l'ensemble $N(S)$ n'est pas très large, il est possible de prendre $V = N(S)$.

Liste Tabou Un élément fondamental de la recherche tabou est l'utilisation d'une mémoire flexible, à court terme, qui garde une certaine trace des dernières opérations passées. On peut y stocker des informations pertinentes à certaines étapes de la recherche Tabou pour en profiter ultérieurement. Cette liste permet d'empêcher les blocages dans les optimums locaux en interdisant de passer à nouveau sur des solutions de l'espace de recherche précédemment visitées.

La durée de cette interdiction, notons la L appelée longueur ou teneur tabou, est l'un des paramètre les plus importants et aussi l'un des plus difficiles à déterminer. Si sa valeur est trop élevée alors des solution non visitées seront injustement inaccessible et la méthode à exploiter le voisinage sera réduite. Inversement, si la valeur est trop faible alors la méthode risque fortement d'être bloquée dans un optimum local. la longueur Tabou permet donc d'éviter tous les cycle de longueur inférieur ou égal à L.

Remarque 2.1. La valeur de L est fixée généralement à 7.

Critère d'aspiration

La liste Tabou pourrait jouer le rôle qui consiste à interdire de se déplacer vers des nouvelles régions susceptibles de contenir de meilleures solutions. Pour éviter cela et dans le but d'avoir une plus grande liberté dans la génération d'un sous-ensemble de solutions réalisables V, il devrait être possible de perdre le statu Tabou d'une modification quand il semble raisonnable de le faire. C'est pourquoi on introduit pour chaque valeur possible Z de la fonction objectif une valeur d'aspiration $A(Z)$. Une solution S dans $N(S)$ qui veut devenir tabou à cause de la liste Tabou peut quand même être prise en compte si $F(S) \leq A(F(S))$, la fonction est appelée fonction**d'aspiration**.

Critère d'arrêt

Comme critère d'arrêt on peut par exemple fixer un nombre maximum d'itérations, ou on peut fixer un temps limite après lequel la recherche doit **s'arrêter**.

Remarque 2.2. Les listes tabou sont générées par le processus **FIFO**(first in firt out),i.e: le plus ancien attribut enregistré fera place au plu récent.

2.3.3 Algorithme de la méthode Tabou

```
0.initialiser:
1.soit S une  solution initiale .\\
2.initialiser la liste tabou $LISTtab$.\\
3.$N_{iter}:=0$.\\
4.$BEST_{iter}:=0$.\\
5.Tantque  (N_{iter}\leq Max_{iter}) faire\\
```

```
      5.1.N_{iter}:= N_{iter} +1\\
      5.2.Tantque (il existe toujours des solutions voisines non
                   explorées et N \leq Nbv) faire\\
            5.2.1.générer S^{'} une solution voisine\\
            5.2.2Si   (f(S^{'})< A(f(S^{'})))  Alors\\
                  Ajouter S^{'} à l'ensemble des solutions voisines \\
                  FinSi\\
           Fait \\
      5.3.retenir la meilleure solution dans l'ensemble des solution voisines\\
      5.4.S:=S^{'}\\
      5.5.metrre à jour la liste tabou LISTtab\\
      5.6.metrre à jour la fonction d'aspiration A\\
      5.7.Si(f(S)< f(S^{0})) Alors\\
            S^{0}:= S\\
            BEST_{inter}:=N_{iter}\\
          FinSi\\
Fait.
\\
```

Chapitre 3

Application et programmation des méthodes

3.1 Présentation du langage

3.1.1 Le langage Python

Python a été créer au début des années 1990 par Guido Van Rossum (CWI)au Stiching Mathematics Centrum au Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclue de nombreuses contributions de la part d'autre personne. La dernière version diffusée par le CWI a été "Python 1.2". En 1995,Guido a continué son travail sur Python au Corporation for Nationaal Research Initiatives(CNRI) de Reseton, en Virginie (USA) où il a diffusé plusieurs versions de ce logiciel. "Python 1.6" été la dernière des versions diffusées par le CNRI. En 2000, Guido et l'équipe de développement centrale de Python sont partis à BeOpen.com pour former l'équipe BeOpen PythonLabs. "Python2.0" a été la première et la seule version diffusée depuis BeOpen.com.

Python est un langage de programmation facile à utiliser et puissant. Il offre des structures de données puissantes de haut niveau et une approche simple mais réelle de la programmation orientée-objet. La syntaxe élégante de python et la typologie dynamique, ajoutées à sa nature interprétée, en font un langage idéal pour le développement rapide d'applications dans de nombreux domaines et sur la plupart des plate-formes.
L'interpréteur python et la vaste librairie standard sont librement disponibles pour toutes les plate formes principales sous forme de sources ou de binaires à partir du site Web de Py-

thon,//www.python.org, et peuvent être distribués librement. Le même site contient aussi des distributions et des pointeurs vers de nombreux modules Python provenant d'autres fournisseurs, des programmes et des outils, et de la documentation supplémentaire.

3.1.2 Syntaxes et outils

Nous définissons quelques instructions et outils qui apparaissent nécessaires pour la lecture du code des méthodes programmées par la suite.

L'instruction if

Peut être l'instruction la plus connue est-elle une instruction de conditionnement: (if,elif).

L'instruction for

L'instruction "for" en Python diffère un petit peu de ce que nous avons pu utiliser en C ou en Pascal. Au lieu d'itèrer toujours dans une progression arithmétique de nombres (comme pascal), ou de laisser l'utilisateur complètement libre dans les tests et les pas d'itérations (comme en C), l'instruction "for" de Python itère parmi les éléments de n'importe quelle séquence, soit une liste ou une chaîne ...etc.

La fonction range

Si nous avons besoin d'itèrer sur une séquence de nombres, la fonction intégrée "range()" vient à point. Elle généré des listes contenant des progressions arithmétique, par exemple:

```
>>>range(10)
[0,1,2,3,4,5,6,7,8,9]
```

Définition de fonctions

Une fonction est définie par le mot clé "def". Il doit être suivi du nom de la fonction e une liste entre parenthèses de paramètres formels. Les instructions qui forment le corps de la fonction commencent sur la ligne suivante. La première instruction du corps de la fonction peut éventuellement être un texte dans une chaîne de caractères , cette chaîne est de documentation de la fonction, ou " docstring".

Le Python est un langage très utilisé dans les problèmes d'optimisation et surtout en optimisation combinatoire, et pour cela nous avons opter à utiliser ce langage pour la programmation des méthodes citées déjà dans la partie théorique. Dans ce mémoire nous utilisons la version (Python2.7).

3.2 Algorithmes des méthodes

3.2.1 Énumération explicite

L'énumération explicite consiste à donner toutes les solutions possibles du partitionnement d'un graphe, on ne choisira que celles qui respectent la notion d'à-peu-près équitable posée au début.
Pour chaque solution trouvée, on calcule le poids interclasse associé et on mémorise le meilleur à chaque itération, pour ce on exécute le code associée **EnumerationVF.py**.

```
#-*-coding: utf8-*- #
==============================================================================
# Algorithme d'Enumeration #
==============================================================================
"""Enumeration : Affiche le partionnement optimale d'un graphe
simple en K classe
                 ainsi que le poid interclasse minimum"""
__author__  = "Mekacher Abderrahmane"
__date__    = "--2012"
__usage__ = """ Note : Le programme s'arrete apres avoir lister
tous les partionnement possible""" #
==============================================================================
import itertools import time #
------------------------------------------------------------------------------
def arete_interclasse(sol,G):
  """fonction qui retourne l'ensemble des arêtes
                        interclasse d'une solution
  donnee en parametre"""
  areteInterclasse=[]
  for i in range(len(sol)-1):
    for j in range(len(sol[i])):
      for h in range (i+1,len(sol)):
        for k in range(len(sol[h])):
          arete1=str(sol[i][j])+str(' ')+str(sol[h][k])
          arete2=str(sol[h][k])+str(' ')+str(sol[i][j])
          z=0
```

```
        E=G[1]
        for l in range(len(G[1])):
          if (arete1==G[1][l]) or (arete2==G[1][l]):
            z+=1
        if z>0:
          areteInterclasse.append(arete1)
  return (areteInterclasse)
#
------------------------------------------------------------------------------
def poid_interclasse(sol,G,poid):
  """fonction qui retourne le poid des aretes inteclasse d'une solution qui
  verifie la condition d'equilibre entre la cardinalite des classes"""
  areteInterclasse=arete_interclasse(sol,G)
  poid_sol=0
  for i in range(len(areteInterclasse)):
    i,j=areteInterclasse[i].split(' ')
    i,j=int(i),int(j)
    poid_sol=poid_sol+poid[i][j]
  return poid_sol

#
------------------------------------------------------------------------------
def enumeration(nbSommets, nbClasses,G,poid):
  """fonction qui enumere les differentes solution et retourne la solution
  optimal avec le poid optimal des aretes interclasses"""
  sommets = xrange(nbSommets)
  support = ( xrange(nbClasses) for _ in sommets )
  monIterateur = itertools.product(*support)
  nbSol_ennumere = 0
  poid_opt=None
  for rep in monIterateur :
    sol_ennumere = [ [] for _ in range(nbClasses) ]
    nbSol_ennumere += 1
    for s,x in enumerate(rep):
```

```
        sol_ennumere[x].append(s)
    z=0
    for i in range(1,len(sol_ennumere)):
      if ((len(sol_ennumere[0])==len(sol_ennumere[i])) or
      ( len(sol_ennumere[0])== len(sol_ennumere[i])+1)
       or ( len(sol_ennumere[0])== len(sol_ennumere[i])-1)):
        z+=1
    if z==(nbClasses-1):
      poid_sol_ennumere=poid_interclasse(sol_ennumere,G,poid)
      if poid_opt==None:
        sol_opt,poid_opt=sol_ennumere,poid_sol_ennumere
      elif poid_sol_ennumere < poid_opt:
        sol_opt,poid_opt = sol_ennumere,poid_sol_ennumere
  return (sol_opt, poid_opt)
#
------------------------------------------------------------------------------
def main():
  """fonction principle de l'algorithme d'enumeration"""
  print "%s\n%s%s\n%s" % ('='*80, __doc__, __usage__, '='*80)
  k=2              #POUR CHANGER LE NOMBRE DE CLASSES?
                       CHANGER LA VALEUR DE K
  E=[]
  while True :
    l=[4, 5, 10, 15, 20, 21]
    z = raw_input("<> Choisissez un nombre de sommet parmis
                      la liste suivante \n%r : "%(l))
    if z == '': break
    s=z+'sommets.txt'
    try:
      t1=time.clock()
      f = file(s)  #POUR ENTRER D AUTRE DONNEE,
                IL SUFFIT DE CHANGER LE NOM DU FICHIER TEXT
      line=f.readline()
      line=f.readline()
```

```
vecteur=line.split(' ')
n=int(vecteur[0]) #recupere le nombre de sommet
print "Nombre de sommets = %d"%n
m=int(vecteur[1]) #recupere le nombre d aretes
print"Nombre de classes = %r"%k
poid=[[0 for j in range (n)]for i in range (n)]
V=range(0,n)
line=f.readline()
line=f.readline()
line=f.readline()
line=f.readline()
while 1:
    vecteur=line.split(' ')
    if vecteur[0]=='#':break
    i=int(vecteur[0])
    j=int(vecteur[1])
    p=int(vecteur[2])
    poid[i-1][j-1]=p
    poid[j-1][i-1]=p
    arete=str(str(i-1)+' '+str(j-1))
    E.append(arete)
    line=f.readline()
G=[]
G.append(V)
G.append(E)
print"\nSolution optimale trouvee pour un partitionnement
                de graphe en %d classes:\n"%k
solution_optimale, poid_optimal = enumeration(n,k,G,poid)
print "Solution_optimale = %r \n    poid_optimal = %r"
                % (solution_optimale,poid_optimal)
print "\n"
t2=time.clock()
temps=(t2-t1)
print"Temps d execution: %r"%(t2-t1)
```

```
      f = file('Test_Enumeration'+s+'.txt','a')  # open 'Test_sommets.txt' in
                          "append" mode (ajoute)
      f.write("temps: %r\nsolution_optimal: %r\n" % (temps,poid_optimal))
      f.close()
    except ValueError:
      print "Error : invalid input value"
#
-------------------------------------------------------------------------------
if __name__ == '__main__':
  main()
#
===============================================================================
```

3.2.2 Descente du gradient

Une méthode qui bloque sur les minima locaux, on pose la condition d'arrêt de la boucle principale est atteinte si le programme trouve une solution moins bonne que celle qu'il a déjà trouvé, pour cela, on exécute le code **DescentegradientVF.py**.

```
===============================================================================
# Algorithme de la descente du gradient #
===============================================================================
"""Descente du gradient : Affiche le meilleur partionnement d'un
graphe simple
                         en K classe ainsi que le poids interclasse minimum
-------------------------------------------------------------------------------
def arete_interclasse(sol,G):# la syntaxe de cette fonction est la
                                  même avec celle de l'énumération

def poid_interclasse(sol,G,poid):#la syntaxe de cette fonction est
                                 la même avec celle du l'énumération
 #-------------------------------------------------------------------------------
def choix_X0(nbSommets,nbClasses):
  """retoune la solution initilial qui est toujours choisis
                de la meme facon"""
  tailleMax = nbSommets/nbClasses
```

```
  X0=[[] for i in xrange(nbClasses) ]
  i=0
  for j in range(nbClasses):
      while len(X0[j]) < tailleMax:
          X0[j].append(i)
          i+=1
  for k in range(i,nbSommets):
      X0[nbClasses-1].append(k)
  return X0

#
------------------------------------------------------------------------------
def choix_voisinage(X0,last_elmt): #Sweep
  """implementation du voisinage sweep"""
  mvt=[]
  if len(last_elmt) <> 0:
    for i in range (len(X0)):
      A=deepcopy(X0[i])
      if last_elmt[i] in A:
        A.remove(last_elmt[i])
      if len(A)==0:
        return [],[],[]
      c=sample(A,1)
      mvt.append(c[0])
    last_elmt=[]
    for i in range (len(mvt)):
      X0[i].remove(mvt[i])

    last_elmt.append(mvt[-1])

    for i in range(1,len(mvt)):
      X0[i].append(mvt[i-1])
      last_elmt.append(mvt[i-1])
```

```
    X0[0].append(mvt[-1])

  else:
    for i in range (len(X0)):
      c=sample(X0[i],1)
      mvt.append(c[0])

    for i in range (len(mvt)):
      X0[i].remove(mvt[i])

    last_elmt.append(mvt[-1])

    for i in range(1,len(mvt)):
      X0[i].append(mvt[i-1])
      last_elmt.append(mvt[i-1])

    X0[0].append(mvt[-1])

  return X0, last_elmt
#
-------------------------------------------------------------------------------
def descente_gradient(nbSommets, nbClasses,G,poid):
  """Methode de la descente du gratient"""
  X0 = choix_X0(nbSommets, nbClasses)
  poid_X0 = poid_interclasse(X0,G,poid)
  poid_opt,sol_opt = poid_X0, X0
  cpt=0
  opt=0
  last_elmt=[]

  while 1:
      Y, last_elmt= choix_voisinage(X0,last_elmt)
      poid_Y = poid_interclasse(Y,G, poid)
      if  poid_Y > poid_X0:
```

```
          break
      else:
          X0 = Y
          poid_opt,sol_opt = poid_Y, Y

  return (sol_opt, poid_opt)
#
-------------------------------------------------------------------------------
def main():
  """fonction principle de l'algorithme descente du gradient"""
  print "%s\n%s%s\n%s" % ('='*80, __doc__, __usage__, '='*80)

  k=2              #nombre de classes     0 < k < nbSommets+1
  E=[]
  while True :
    l=[4, 5, 10, 15, 17, 20, 21, 22,23, 24, 25, 30, 50, 100, 500]
    z = raw_input("<> Choisissez un nombre de sommet
              parmis la liste suivante \n%r : "%(l))
    if z == '': break
    s=z+'sommets.txt'
    try:
      t1=time.clock()
      f = file(s)
      line=f.readline()
      line=f.readline()
      vecteur=line.split(' ')
      n=int(vecteur[0])
      m=int(vecteur[1])
      poid=[[0]*n]*n
      V=range(1,n+1)
      line=f.readline()
      line=f.readline()
      line=f.readline()
      line=f.readline()
```

```
      while 1:
          vecteur=line.split(' ')
          if vecteur[0]=='#':break
          i=int(vecteur[0])
          j=int(vecteur[1])
          p=int(vecteur[2])
          poid[i-1][j-1]=p
          poid[j-1][i-1]=p
          arete=str(str(i-1)+' '+str(j-1))
          E.append(arete)
          line=f.readline()
      G=[]
      G.append(V)
      G.append(E)
      print"""\nMeilleure solution trouvee par la methode
                           de descente du gradient
      pour un partitionnement de graphe %d classe:\n"""%k
      meilleure_solution, meilleur_poid = descente_gradient(n,k,G,poid)
      print "meilleure_solution = %r\n     meilleur_poid = %r\n"
            % (meilleure_solution, meilleur_poid )
      t2=time.clock()
      temps=t2-t1
      print"Temps d execution: %r"%(temps)
      f = file('Test_DescenteGradient'+s,'a')
      f.write("temps: %r\nmeilleure_solution: %r\n\n\n\n" % (temps,meilleur_poid))
      f.close()
    except ValueError:
      print "Error : invalid input value"
#
------------------------------------------------------------------------------
if __name__ == '__main__':
  main()
#
======================================================================
```

3.2.3 Recuit simulé

Dans notre programme, les indéterminés de cet algorithme sont :

- La solution initiale est choisie comme décrite ci-dessus.
- La température initiale a été choisie arbitrairement avec $T = 1000$.
- La condition d'arrêt de la boucle principale est telle que la température devient très faible, on la suppose à $T = 0.01$. La température décrètent grâce à une suite géométrique de raison 0.90.
- La condition d'arrêt de la boucle interne est sous forme d'un polynôme :le nombre de classes multiplié par le nombre de sommets.
- On a rajouté une autre condition qui permet d'arrêter la boucle interne est quand le programme trouve une bonne solution suivie de 5 autres mauvaises consécutives.

Pour cela, on exécute le code **RecuitsimuleVF.py**.

```
#-*-coding: utf8-*- #
==============================================================================
# Algorithme du Recuit Simule #
==============================================================================
"""Recuit Simule : Affiche le meilleur partitionnement d'un graphe
simple en K classe
                 ainsi que le poids interclasse minimum associe"""
-------------------------------------------------------------------
def arete_interclasse(sol,G):# la syntaxe de cette fonction est la
                                  même avec celle de l'énumération
--------------------------------------------------------------------
def poid_interclasse(sol,G,poid):#la syntaxe de cette fonction est
                                 la même avec celle de l'énumération
----------------------------------------------------------------------
def choix_X0(nbSommets,nbClasses):#la syntaxe de cette fonction
                                 est  la même avec celle du
                                 descente gradient
--------------------------------------------------------
def choix_voisinage(X0,last_elmt): #Sweep    ##la syntaxe de cette
                                    fonction est  la même avec celle du
```

```
                                                descente gradient
  """implementation du voisinage sweep"""
  ----------------------------------------------------
# def recuit_simule(nbSommets, nbClasses,G,poid):
  """implementation de l'algorithme du recuit simule"""
  X0 = choix_X0(nbSommets, nbClasses)
  poid_X0 = poid_interclasse(X0,G,poid)
  poid_opt,sol_opt = poid_X0, X0
  T=1000
  cpt=0
  opt=0
  last_elmt=[]
  while T>0.01:
    t=0
    while t<(nbSommets*nbClasses):
      Xt, last_elmt = choix_voisinage(X0, last_elmt)
      nouveau_poid = poid_interclasse(Xt,G, poid)
      delta = nouveau_poid - poid_X0
      if delta<0:
        X0=Xt
        poid_X0=nouveau_poid
        if nouveau_poid - poid_opt<0:
          poid_opt, sol_opt = nouveau_poid, Xt
          opt=1  #il trouve une solution meilleure
          cpt=0  #il initilise le compteur cpt
      elif (random() < exp((-1)*(delta/T))):
        X0, poid_X0 = Xt, nouveau_poid
        cpt +=1  #il incremente cpt a  chaque fois qu
        il trouve une mauvaise solution
      t=t+1
    if (opt==1 and cpt==5):
      break
      #si il prend une mauvaise solution 5 fois de suite
      #apres avoir pri une bonne solution il arrete de tourner
```

```
    T=T*(0.90)
  return (sol_opt, poid_opt)
#
------------------------------------------------------------------------------
def main():
  """fonction principle de l'algorithme du recuit simule"""
  print "%s\n%s%s\n%s" % ('='*80, __doc__, __usage__, '='*80)
  #k=2
  k=2               #POUR CHANGER LE NOMBRE DE CLASSES? CHANGER LA
                                        VALEUR DE K
  E=[]
  while True :
    l=[4, 5, 10, 15, 17, 20, 21, 22,23, 24]
    z = raw_input("<> Choisissez un nombre de sommet
                     parmi la liste suivante \n%r : "%(l))
    if z == '': break
    s=z+'sommets.txt'
    try:
      t1=time.clock()
      f = file(s)  #POUR ENTRER D AUTRE DONNEE, IL SUFFIT DE CHANGER
                                     LE NOM DU FICHIER TEXT
      line=f.readline()
      line=f.readline()
      vecteur=line.split(' ')
      n=int(vecteur[0])
      m=int(vecteur[1])
      poid=[[0]*n]*n
      print "Nombre de sommets = %r"%n
      print"Nombre de classes = %r"%k
      V=range(1,n+1)
      line=f.readline()
      line=f.readline()
      line=f.readline()
      line=f.readline()
```

```
      while 1:
        vecteur=line.split(' ')
        if vecteur[0]=='#':break
        i=int(vecteur[0])
        j=int(vecteur[1])
        p=int(vecteur[2])
        poid[i-1][j-1]=p
        poid[j-1][i-1]=p
        arete=str(str(i-1)+' '+str(j-1))
        E.append(arete)
        line=f.readline()
      G=[]
      G.append(V)
      G.append(E)
      print"""\nMeilleure solution trouvee pour un partitionnement
                        de graphe en %d classes:\n"""%k
      meilleure_solution, meilleur_poid = recuit_simule(n,k,G,poid)
      print "meilleure_solution = %r\n     meilleur_poid = %r\n"
                     % (meilleure_solution, meilleur_poid )
      t2=time.clock()
      temps=t2-t1
      print"Temps d execution: %r"%(temps)
      f = file('Test_RecuitSimule'+s+'.txt','a')
      f.write("temps: %r\nmeilleure_solution: %r\n" % (temps,meilleur_poid))
      f.close()
    except ValueError:
      print "Error : invalid input value"
#
------------------------------------------------------------------------------
if __name__ == '__main__':
  main()
#
==============================================================================
```

3.2.4 Tabou

Il y'a trois versions différentes pour Tabou, dans la première version, on considère les indéterminés de l'algorithme comme suit:

- La solution initiale est retrouvée avec le même principe que le recuit simulé.
- La condition d'arrêt de la boucle principale est sous la forme d'un polynôme: nombre de sommets deviser par 2.
- La taille maximale de la liste Tabou est égale à 7 qui est sous la forme FIFO.
- Le voisinage est choisit exactement comme dans le recuit simulé c'est à dire avec la méthode sweep.
- A chaque itération de l'algorithme on ajoute un mouvement à la liste tabou et on supprime si elle est pleine.

On peut améliorer cette version en ne mettant dans la liste Tabou que les mouvements qui conduisent à une solution moins aussi bonne, ce qui nous amène à la version 2.

Une autre façon de faire est d'utiliser l'aspiration, et ce dans le cas où l'algorithme ne retrouve plus de solution améliorantes au bout d'un certains nombre d'itérations, dans le programme implémenté cette valeur est de 10 mauvaises solutions consécutives et ce à cause de son coût.

Dans ce cas la liste Tabou n'est plus FIFO, on associe a chaque mouvement une étiquette temporelle au moment d'effectuer l'aspiration, tous les mouvements de la liste Tabou sont candidat sauf celui ajouté en dernier. On vérifie d'abord si on peut appliquer le mouvement choisi sur la solution courante, si cela est possible on l'applique sur la solution courante et si on trouve un poids interclasse meilleur, on relâche le mouvement sinon on le garde dans Tabou.

On incrémente toutes les étiquettes temporelles, on supprime celles qui en ont une supérieure à la taille maximale de la liste Tabou.

Les trois versions de Tabou sont implémentées et est peuvent être exécutées, **TabouVFV1.py**, **TabouVFV2.py**, **TabouVFV3.py**.

Tabou V1

```
#-*-coding: utf8-*- #
==============================================================================
# Tabou #
==============================================================================
```

```
"""Tabou V1: Affiche le meilleur partionnement d'un graphe simple
en K classe
                 ainsi que le poid interclasse minimum, sans aspiration"""
__author__  = "Mekacher Abderrahmane"
__date__    = "--2012"
__usage__ = """ Note : Tapez entrer pour arreter le programme""" #
==============================================================================
from random import * import time from copy import * #
------------------------------------------------------------------
def arete_interclasse(sol,G):# la syntaxe de cette fonction est la
                                  même avec celle de l'énumération
-------------------------------------------------------------------
def poid_interclasse(sol,G,poid):#la syntaxe de cette fonction est
                                 la même avec celle de l'énumération
---------------------------------------------------------------------
def choix_X0(nbSommets,nbClasses):#la syntaxe de cette fonction
                                 est  la même avec celle du
                                 descente gradient
--------------------------------------------------------
def choix_voisinage(X0,last_elmt): #Sweep    ##la syntaxe de cette
                                    fonction est  la même avec celle du
                                            descente gradient
  """implementation du voisinage sweep"""
#
------------------------------------------------------------------------------
def tabou(nbSommets,nbClasses,G,poid):
  """implementation de l'algorithme Tabou version 1,
                       sans aspiration"""
  tabou = []
  mvt=[]
  last_elmt=[]
  Tmax = 7
  i=0
  X0 = choix_X0(nbSommets,nbClasses)
```

```
  poid_X0 = poid_interclasse (X0, G, poid)
  poid_opt,sol_opt = poid_X0, X0

  while i< (nbSommets):
      Y, mvt, last_elmt= choix_voisinage(X0,last_elmt)

      #pour un nombre de sommets petit
      if len(Y)==len(mvt)==len(last_elmt)==0:
        break
      else:
        poid_Y = poid_interclasse(Y,G, poid)

        if poid_Y < poid_opt: # on compare Y au Fmin
            sol_opt = Y
            poid_opt = poid_Y

        if len(tabou) == Tmax:
            tabou.remove(tabou[0])
        tabou.append(mvt)
        X0=Y
        i +=1

  return (sol_opt, poid_opt)
#
-------------------------------------------------------------------------------
def main():
  """fonction principle de l'algorithme Tabou V1"""
  print "%s\n%s%s\n%s" % ('='*80, __doc__, __usage__, '='*80)
  k=2              #nombre de classes     0 < k < nbSommets+1
  E=[]
  while True :
    l=[4, 5, 10, 15, 17, 20, 21, 22,23, 24, 25, 30, 50, 100]
    z = raw_input("<> Choisissez un nombre de sommet parmis la liste
                               suivante \n%r : "%(l))
```

```
if z == '': break
s=z+'sommets.txt'
try:
  t1=time.clock()
  f = file(s)
  line=f.readline()
  line=f.readline()
  vecteur=line.split(' ')
  n=int(vecteur[0])
  m=int(vecteur[1])
  print "\nNombre de sommets = %r"%n
  print"Nombre de classes = %r"%k
  poid=[[0]*n]*n
  V=range(1,n+1)
  line=f.readline()
  line=f.readline()
  line=f.readline()
  line=f.readline()
  while 1:
      vecteur=line.split(' ')
      if vecteur[0]=='#':break
      i=int(vecteur[0])
      j=int(vecteur[1])
      p=int(vecteur[2])
      poid[i-1][j-1]=p
      poid[j-1][i-1]=p
      arete=str(str(i-1)+' '+str(j-1))
      E.append(arete)
      line=f.readline()
  G=[]
  G.append(V)
  G.append(E)
  print"""\nMeilleure solution trouvee pour un partitionnement de graphe
            en %d classes:\n"""%k
```

```
      meilleure_solution, meilleur_poid = tabou(n,k,G,poid)
      print "meilleure_solution = %r\n     meilleur_poid = %r\n"
           % (meilleure_solution, meilleur_poid )
      t2=time.clock()
      temps=t2-t1
      print"Temps d execution: %r"%(temps)
      f = file('Test_TabouV1_'+s,'a')  # open 'Test_sommets.txt' in
                                           "append" mode (ajoute)
      f.write("temps: %r\nmeilleure_solution: %r\n" %
                             (temps,meilleur_poid))
      f.close()
    except ValueError:
      print "Error : invalid input value"
#
------------------------------------------------------------------------------
if __name__ == '__main__':
  main()
#
==============================================================================
```

Tabou V2

```
#-*-coding: utf8-*- #
==============================================================================
# Tabou #
==============================================================================
"""Tabou : Affiche le partionnement optimale d'un graphe simple en
K classe
                 ainsi que le poid interclasse minimum, sans aspiration
                mais rend tabou que le mouvement qui a donne
                une mauvaise solution"""
------------------------------------------------------------------
def arete_interclasse(sol,G):# la syntaxe de cette fonction est la
                                  même avec celle de l'énumération
-------------------------------------------------------------------
```

```
def poid_interclasse(sol,G,poid):#la syntaxe de cette fonction est
                                 la même avec celle de l'énumération
---------------------------------------------------------------------
def choix_X0(nbSommets,nbClasses):#la syntaxe de cette fonction
                                 est  la même avec celle du
                                 descente gradient
--------------------------------------------------------
def choix_voisinage(X0,last_elmt): #Sweep    ##la syntaxe de cette
                                    fonction est  la même avec celle du
                                            descente gradient
  """implementation du voisinage sweep"""
#
------------------------------------------------------------------------------
def tabou(nbSommets,nbClasses,G,poid):
  """Methode Tabou version 2, sans aspiration mais met dans tabou que les mouvements
  qui donnent une mauvaise soltution par rapport au min deja trouve"""
  tabou = []
  mvt=[]
  last_elmt=[]
  Tmax = 7
  i=0
  X0 = choix_X0(nbSommets,nbClasses)
  poid_X0 = poid_interclasse (X0, G, poid)
  poid_opt,sol_opt = poid_X0, X0

  while i< (nbSommets):
      Y, mvt, last_elmt= choix_voisinage(X0,last_elmt)

      if len(Y)==len(mvt)==len(last_elmt)==0:
        break
      else:
        poid_Y = poid_interclasse(Y,G, poid)

        if poid_Y < poid_opt: # on compare Y au Fmin
```

```
            sol_opt = Y
            poid_opt = poid_Y

        if len(tabou) == Tmax:
            tabou.remove(tabou[0])

        if (poid_Y > poid_opt):
            tabou.append(tuple(mvt))
        X0=Y
        i +=1

  return (sol_opt, poid_opt)
#
-------------------------------------------------------------------------------
def main():
  """fonction principle de l'algorithme Tabou V2"""
  print "%s\n%s%s\n%s" % ('='*80, __doc__, __usage__, '='*80)
  k=2              #nombre de classes     0 < k < nbSommets+1
  E=[]
  while True :
    l=[4, 5, 10, 15, 17, 20, 21, 22,23, 24, 25, 30, 50, 100]
    z = raw_input("<> Choisissez un nombre de sommet parmis
                    la liste suivante \n%r : "%(l))
    if z == '': break
    s=z+'sommets.txt'
    try:
      t1=time.clock()
      f = file(s)
      line=f.readline()
      line=f.readline()
      vecteur=line.split(' ')
      n=int(vecteur[0])
      m=int(vecteur[1])
      print "\nNombre de sommets = %r"%n
```

```
    print"Nombre de classes = %r"%k
    poid=[[0]*n]*n
    V=range(1,n+1)
    line=f.readline()
    line=f.readline()
    line=f.readline()
    line=f.readline()
    while 1:
        vecteur=line.split(' ')
        if vecteur[0]=='#':break
        i=int(vecteur[0])
        j=int(vecteur[1])
        p=int(vecteur[2])
        poid[i-1][j-1]=p
        poid[j-1][i-1]=p
        arete=str(str(i-1)+' '+str(j-1))
        E.append(arete)
        line=f.readline()
    G=[]
    G.append(V)
    G.append(E)
    print"""\nMeilleure solution trouvee pour un partitionnement
                      de graphe en %d classes:\n"""%k
    meilleure_solution, meilleur_poid = tabou(n,k,G,poid)
    print "meilleure_solution = %r\n     meilleur_poid = %r\n"
                         % (meilleure_solution, meilleur_poid )
    t2=time.clock()
    temps=t2-t1
    print"Temps d execution: %r"%(temps)
    f = file('Test_TabouV2_'+s,'a')  # open 'Test_sommets.txt' in
                                            "append" mode (ajoute)
    f.write("temps: %r\nmeilleure_solution: %r\n" % (temps,meilleur_poid))
    f.close()
  except ValueError:
```

```
      print "Error : invalid input value"
#
-------------------------------------------------------------------------------
if __name__ == '__main__':
  main()
#
===============================================================================
```

Tabou V3

```
#-*-coding: utf8-*- #
===============================================================================
# Tabou avec ASPIRACTION #
===============================================================================
"""Tabou avec ASPIRATION: Affiche le partionnement optimale d'un
graphe simple
                          en K classe ainsi que le poid interclasse minimum"""
-------------------------------------------------------------------
def arete_interclasse(sol,G):# la syntaxe de cette fonction est la
                                  même avec celle de l'énumération
--------------------------------------------------------------------
def poid_interclasse(sol,G,poid):#la syntaxe de cette fonction est
                                 la même avec celle de l'énumération
----------------------------------------------------------------------
def choix_X0(nbSommets,nbClasses):#la syntaxe de cette fonction
                                 est  la même avec celle du
                                 descente gradient
---------------------------------------------------------
def choix_voisinage(X0,last_elmt): #Sweep    ##la syntaxe de cette
                                    fonction est  la même avec celle du
                                            descente gradient
  """implementation du voisinage sweep"""
#
-------------------------------------------------------------------------------
def tabou(nbSommets,nbClasses,G,poid):
  """implementation de l'algorithme Tabou version 1, sans aspiration"""
```

```
tabou = []
mvt=[]
last_elmt=[]
Tmax = 7
opt=0
cpt=0
c=0
i=0
X0 = choix_X0(nbSommets,nbClasses)
poid_X0 = poid_interclasse (X0, G, poid)
poid_opt,sol_opt = poid_X0, X0

while i< (nbSommets/2):
    Y, mvt, last_elmt= choix_voisinage(X0,last_elmt)

    if len(Y)==len(mvt)==len(last_elmt)==0:
      break
    else:
      poid_Y = poid_interclasse(Y,G, poid)

      if poid_Y < poid_opt: # on compare Y au Fmin
          sol_opt = Y
          poid_opt = poid_Y
          opt=1
          cpt=0
      if poid_Y > poid_opt:
        cpt += 1
        new_tabou=deepcopy(tabou)

        for i in range(len(tabou)):
          if tabou[i][2]>=Tmax:
            new_tabou.remove(tabou[i])
        tabou=deepcopy(new_tabou)
        for i in range(len(tabou)):
```

```
    tabou[i][2] += 1
  tabou.append([(mvt),poid_Y,0])

#aspiration
if (opt==1 and cpt==10 and len(tabou)>1):
  s= deepcopy(Y)
  new_tabou=deepcopy(tabou)
  for i in range (len(tabou)):
    if tabou[i][2]>0:#travail sur tous les elements tabou sauf le dernier
      for l in range(len(tabou)):
        c=0
        for j in range (len(s)):
          if (tabou[l][0][j] in s[j]):
            c += 1#si on peu appliquer le mvt sur s
        if c== len(s):#on applique le mvt sur s
          for k in range (len(s)):
            s[k].remove(tabou[l][0][k])
          for k in range(1,len(s)):
            s[k].append(tabou[l][0][k-1])
          s[0].append(tabou[l][0][-1])

          poid_s=poid_interclasse(s,G,poid)
          if poid_s < poid_Y:#poid interclasse de s est meilleur
            if tabou[l] in new_tabou:
              new_tabou.remove(tabou[l]) #on relache le mvt
  tabou=deepcopy(new_tabou)

encien=0
new_tabou=deepcopy(tabou)
if len(tabou) == Tmax:
  for i in range(len(tabou)):
    if tabou[i][2]>encien:
      indice= tabou.index(tabou[i])
  new_tabou.remove(tabou[indice])
```

```
        tabou=deepcopy(new_tabou)
        X0=Y
        i +=1

  return (sol_opt, poid_opt)
#
------------------------------------------------------------------------------
def main():
  """fonction principle de l'algorithme Tabou V3 avec aspiration"""
  print "%s\n%s%s\n%s" % ('='*80, __doc__, __usage__, '='*80)
  k=2             #nombre de classes     0 < k < nbSommets+1
  E=[]
  while True :
    l=[4, 5, 10, 15, 17]
    z = raw_input("<> Choisissez un nombre de sommet parmis la
                         liste suivante \n%r : "%(l))
    if z == '': break
    s=z+'sommets.txt'
    try:
      t1=time.clock()
      f = file(s)
      line=f.readline()
      line=f.readline()
      vecteur=line.split(' ')
      n=int(vecteur[0])
      m=int(vecteur[1])
      print "\nNombre de sommets = %r"%n
      print"Nombre de classes = %r"%k
      poid=[[0]*n]*n
      V=range(1,n+1)
      line=f.readline()
      line=f.readline()
      line=f.readline()
      line=f.readline()
```

```
      while 1:
          vecteur=line.split(' ')
          if vecteur[0]=='#':break
          i=int(vecteur[0])
          j=int(vecteur[1])
          p=int(vecteur[2])
          poid[i-1][j-1]=p
          poid[j-1][i-1]=p
          arete=str(str(i-1)+' '+str(j-1))
          E.append(arete)
          line=f.readline()
      G=[]
      G.append(V)
      G.append(E)
      print"""\nMeilleure solution trouvee pour un partitionnement
                        de graphe en %d classes:\n"""%k
      meilleure_solution, meilleur_poid = tabou(n,k,G,poid)
      print "meilleure_solution = %r\n     meilleur_poid =
         %r\n" % (meilleure_solution, meilleur_poid )
      t2=time.clock()
      temps=t2-t1
      print"Temps d execution: %r"%(temps)
      f = file('Test_TabouV3_'+s,'a')  # open 'Test_sommets.txt' in
                                  "append" mode (ajoute)
      f.write("temps: %r\nmeilleure_solution: %r\n" % (temps,meilleur_poid))
      f.close()
    except ValueError:
      print "Error : invalid input value"
#
-------------------------------------------------------------------------------
if __name__ == '__main__':
  main()
#
===============================================================================
```

3.2.5 Les tests

nombre de sommets	enumeration		descente gradient		recuit simulé	
	SM	TM	SM	TM	SM	TM
4	2.0	0.005	2.0	0.00333	2.0	0.04459
5	4.0	0.00539	4.51	0.00270	4.0	0.04350
10	13.0	0.04950	15.66	0.00269	13.06	0.40149
15	43.0	11.03849	47.74	0.00669	43.32	2.56480
17	60.0	66.8947	62.57	0.01830	60.01	5.04980
20	40.0	300.09769	48.3	0.00860	41.67	6.22759
21	107.0	2350.08999	108.66	0.09400	107.0	15.54270
22	/	/	108.84	0.01460	104.6	17.25749
23	/	/	118.95	0.027500	116.01	21.16560
24	/	/	131.0	0.02619	131.0	28.06369
25	/	/	154.49	0.39920	/	/
30	/	/	202.2	0.05870	/	/
50	/	/	202.2	0.05870	/	/
100	/	/	266.38	1.46829	/	/
500	/	/	3712.22	195.20555	/	/

nombre de sommets	Tabou1		Tabou2		Tabou3	
	SM	TM	SM	TM	SM	TM
4	2.0	0.00419	2.0	0.00570	2.0	0.00520
5	4.0	0.00479	4.0	0.00430	4.16	0.00500
10	14.13	0.00669	14.25	0.00750	14.55	0.00880
15	46.3	0.02010	45.98	0.02009	46.4	0.01659
17	60.97	0.03080	60.92	0.03210	60.92	0.09559
20	46.39	0.04060	46.06	0.04030	/	/
21	107.0	0.08069	107.34	0.08120	/	/
22	107.27	0.09379	107.98	0.09390	/	/
23	117.13	0.11120	117.21	0.11239	/	/
24	131.0	0.13881	131.0	0.13769	/	/
25	152.63	0.17890	152.63	0.18110	/	/
30	188.75	0.37339	198.95	0.39659	/	/
50	240.09	2.1714	239.47	2.16829	/	/
100	973.7	66.62872	972.3	66.19166	/	/
500	/	/	/	/	/	/

3.2.6 Les résultats

- Les tests effectués pour 100 simulations, sauf là où c'est indiqué, comme pour le Tabou version 1, 86 simulations pour 100 sommets.
- Le temps moyens des résultats est en secondes.

L'énumération des solutions coûte trop chère, on arrive à avoir les solution optimales jusqu'au graphe avec 21 sommets.

La descente du gradient est l'algorithme qui nous permet d'aller le plus loin, son coût est très minime. On a réussit à trouver les résultats jusqu'aux graphes 500 sommets mais les solutions sont loin de l'optimum en les comparant avec les solutions trouvées par les autre méthodes.

Le recuit simulé est l'algorithme qui arrive à trouver les solutions les plus proches des solutions optimales mais à partir de 15 ou 17 sommets, il met un peut de temps à les trouver.

Reste le Tabou, il trouve les solutions moins vite que la descente de gradient mais plus vite que le recuit simulé mais moins bonnes que celle du recuit simulé.

On remarque une différence entre les 3 versions de l'algorithme Tabou:

La version 2 avec amélioration arrive à trouver, en moyenne, des solutions meilleures que la version 1 et la différence entre le moyen est très très minime, en ce qui concerne l'aspiration, cette dernière coûte très chère et au bout de 17 sommets on arrive plus a avoir de résultats, peut être qu'il faudra changer la condition qui fait appelle à l'aspiration qui actuellement est de faire une aspiration après avoir trouver une bonne solution puis 10 mauvaise solutions consécutives, passer de 10 à 20 ou plus, pourra peu être changer les tests et permettra d'avoir des résultats.

Conclusion

Bien que les problèmes d'optimisation combinatoire soient souvent faciles à définir, ils sont généralement difficile à résoudre. En effet, la plupart appartiennent à la classe des problèmes NP-difficile et ne possèdent donc pas à ce jour de solutions algorithmique efficaces valables pour toutes les données.
Notre travail s'est soldé par l'implémentation des algorithmes des méthodes cités au début, accompagné d'une comparaison entre les différents algorithmes sur deux importantes échelles qui sont le temps de la recherche de la solution et la qualité de cette dernière (efficacité).

Comme perspectives futures, il serait intéressant de refaire les tests avec une fonction qui choisit la solution initiale aléatoirement à chaque simulation, comme on peut aussi varier les indéterminés des algorithmes pour voire si cela influe vraiment sur le résultat général, on arrive à montrer que l'énumération explicite est bien trop coûteuse et ne peut servir quand l'espace de recherche est très grand, le recuit simulé est le plus proche des solutions optimales mais ne retourne pas le résultat aussi vite que le Tabou.

Bibliographie

[1] Charon.A Germa and O.oudry "Méthode d'optimisation combinatoire", MASSON, paris.
[2] Declic, terminale ES, enseignement et options, Hachette education.
[3] http://fr.wikipedia.org/wiki/RecuitSimulé.
[4] Kuma, Vipin "Gr ph partitioning and application in hight performance computing " University of minnesota (2000).
[5] Philippe lacomme, Christian pris, Marc sevaux "Algorithme et graphes", Edition EY-ROLESS.
[6] Vincent Barichard, thèse doctorat "approche hybride pour les problèmes multiobjectifs ", University Angers(1999).
[7] Yann collette and Patrik siarry "optimisation multiobjectif", Edition EYROLESS.