université de BORDEAUX

Collège Sciences et Technologies
UF Mathématiques et Interactions

Projet de Méta-heuristiques

Lucas Villenave et Martin Debouté

Unité d'Enseignement : Outils Logiciels Pour l'Optimisation (4TOD809U)

CMI OPTIM & ROAD
2021–2022

Engagement de non plagiat

Nous, Lucas Villenave, Martin Debouté, déclarons être pleinement conscient que le plagiat de documents ou d'une partie d'un document publiés sur toutes formes de support, y compris l'internet, constitue une violation des droits d'auteur ainsi qu'une fraude caractérisée.

En conséquence, nous nous engageons à citer toutes les sources que nous avons utilisées pour produire et écrire ce rapport.

Fait à Talence le 17 avril 2022,

Signature

Lucas Villenave, Martin Debouté

Table des matières

Introduction

Ce document constitue notre rapport sur le projet effectué au cours du semestre de printemps 2022 au sein de l'UE : Outils Logiciels Pour l'Optimisation (4TOD809U).

Le sujet de ce projet était l'implémentation et la comparaison des performances de divers méta-heuristiques. Le problème qui nous a été donné est un problème de partitionnement de graphe non orienté éventuellement pondéré dans R en k classes "à peu près équitables". La fonction objectif à minimiser étant la somme des poids des arêtes inter-classes. Des instances de graphes à partitionner nous ont été fournies ainsi que des fiches explicatives portant sur les différentes méta-heuristiques.

Pour notre projet, une grande liberté nous a été donnée. Entre autres choses, nous avions à définir :

- Notre définition de "à peu près équitable"

- Le nombre de classe d'une partition k

- Nos structures de données

- La nature du voisinage d'une solution ainsi que sa taille

- La seconde méta-heuristique que nous avions à développer (parmi une liste restreinte)

- Les indéterminés de nos algorithmes.

Nous avons donc implémenté les algorithmes suivants :

- Une énumération explicite de toutes les solutions

- Un algorithme de descente de gradient sans optimisation (on explore tout le voisinage)

- Un recuit simulé

- Une méthode tabou sans aspiration

- À cela, s'ajoute les améliorations des algorithmes de descente de gradient et des méta-heuristiques

1 Définitions et contexte

1.1 Espace de recherche

Parmi les libertés qui nous ont été offertes, les premiers choix que nous aillons eu à faire furent de définir le nombre de k-classes de nos futures solutions, la notion de "à peu près équitable" ainsi que notre voisinage.

Pour k, nous avons fait le choix de développer le projet de sorte à pouvoir le changer comme bon nous semble. Tous nos algorithmes fonctionnent donc parfaitement avec un $k \in [2; n]$ (avec n le nombre de sommets de notre graphe). Cependant, pour notre analyse comparative de résultats, nous avons choisis d'effectuer nos tests avec $k = 2$ et $k = 5$.

Pour le "à peu près équitable" nous avons fait le choix de nous donner que peu de liberté. En notant q le quotient et r le reste de la division euclidienne de n par k, on aura les classes $[1; k]$ de taille $q + 1$ et les classes $]k; n]$. Nous avons donc pris la décision d'avoir des classes ayant un cardinal le plus égal possible puisque c'est ce qui correspond le mieux à notre définition de "à peu près équitable". De plus, cette décision a été guidée par notre choix de voisinage.

Dans une réflexion similaire à notre définition de "à peu près équitable", nous avons opté pour la méthode de voisinage qui nous paraissait la plus naturelle. Nous avons donc choisi la méthode *swap*, le principe étant de choisir deux sommets de classes différentes que nous intervertissons afin de créer une nouvelle solution voisine.

De plus, nous avons décidé de focaliser nos efforts sur l'implémentation de ces méthodes ainsi que l'amélioration de nos algorithmes plutôt que sur une étude de l'impact du type de voisinage sur les résultats. En effet, le *swap* ne change pas le cardinal de nos classes, nous pouvons trivialement définir notre notion de "à peu près équitable" comme ci-dessus et être sure que les solutions voisines d'une solution réalisable sont réalisables.

Les algorithmes de descente de gradient, recuit simulé et tabou ont besoin d'une solution initiale pour pouvoir fonctionner, nous avons donc décidé de leur donner en solution initiale une solution parfaitement aléatoire.

1.2 Structure du projet

La structure de données est le coeur de tout projet informatique puisqu'elle impact directement le temps d'execution des différents algorithmes d'un programme. Il est donc important de choisir une structure de données adaptée à notre problème afin de ne pas impacter négativement nos performances. Lors de ce projet nous avons eu à créer deux structures, la première pour nous permettre de représenter un graphe et la seconde pour représenter une solution donnée au problème.

La structure de graphe est composé de n et m (respectivement le nombre de sommets et le nombre d'arêtes), le degré de chaque sommet ainsi q'une matrice d'adjacence. Nous avons développé des méthodes très utiles comme $getNeighbors(vertex)$ ou encore $getDegree(vertex)$ dont les noms sont suffisamment explicites pour ne pas avoir à vous décrire en détails ce qu'elles font. Bien entendu l'intégralité de notre code suit des conventions et est commenté afin de facilement le comprendre.

Il est bon de noter qu'au début du projet nous avions utilisé une liste d'adjacence ainsi q'une structure pour représenter une arete. Le problème étant qu'avec un voisinage de type *swap* nous avions souvent à vérifier si deux sommets étaient adjacents. Or savoir si deux sommets sont adjacents a un coût temporel en $O(n)$ pour une liste d'adjacence mais un coût en $O(1)$ pour une matrice d'adjacence. La matrice d'adjacence (de surcroît symétrique car c'est un graphe non orienté) étant certe plus volumineuse à stocker en mémoire (et donc plus lente à créer) nous l'avons finalement adopté car bien plus efficace concernant certain calcul comme le coût d'une solution etc...

Nous aurions pu garder la liste d'adjacence en plus de la matrice d'adjacence pour définir le graphe mais nous avons préféré ne pas le faire jugeant son coût en mémoire trop élevé pour une si faible utilité *in fine*. Après réflexion l'ajout d'une liste d'ajacence bien implémentée à notre structure de données aurait pu éventuellement nous faire économiser du temps sur certains calculs précis.

La structure de solution est composée d'une partition (affectation des sommets aux différentes classes), du nombre de classe et du graphe assossié au problème. Nous avons fait le choix de contenir le graphe dans la structure de solution car le graphe est nécessaire pour le calcul de coût et de réalisabilité de la solution. De plus, pour certaines fonctions il était plus agréable de n'avoir qu'un paramètre (solution). En effet avoir des méthodes permettant de calculer le coût et la faisabilité d'une solution dans la structure de cette dernière et donc qu'on peut appeler sur l'objet était aussi plus élégant plutôt que d'avoir un évaluateur et un checkeur de solution externe. Nous avons donc construit notre projet sur une approche objet plutôt que fonctionnel, cette approche a en plus l'avantage d'être facilement modifiable et maintenable.

La vraie question de la structure de solution restait néanmoins la forme de la partition. Nous avons choisi d'opter pour des tableaux de tableaux, la partition contient des tableaux de classes qui contiennent eux-mêmes les indices des sommets qui la composent. Nous connaissons parfaitement les limitations de cette structure de partition puisque nous avons aussi considéré une autre structure telle que la partition serait un tableau de taille n et contiendrait des entiers k qui indiqueraient la classe du sommet associé à l'indice n ($partition[n] = k$ indique que n appartient à k). Nous savons que la deuxième structure ne génère que k^n solutions là ou la structure que nous utilisons génère $n!$ solutions et $k^n < n!$ (avec $k < n$). Nous avons tout de même choisi d'utiliser la première structure de partition car elle nous semblait plus naturelle à manipuler étant donné le fait que nous l'avions utilisé en cours. Pour compenser notre algorithme d'énumération explicite parcourt lui-même ses solutions grâce à la deuxième méthode et convertit ensuite sa solution. Pour les autres algorithmes, on remarquera que le nombre suplémentaire de solution de la première structure de partition vient de la symétrie interne des sommets dans une classe. Or notre voisinage ne tient pas compte de cette symétrie, de ce fait la taille du voisinage *swap* des deux structures reste la même et le nombre de solution n'est pas si impactant ici.

1.3 Contexte de développement

Nous avons développé notre projet en Python avec une architecture mixte répondant aux principes de bases de la programmation orientée objet et de la programmation fonctionnelle. Nous avons choisi Python car c'est un langage très portable malgré son exécution assez lente comparé à certains langage de programmation plus performant. Et nous avons utilisé Git comme gestionnaire de version, notre code était hébergé sur un dépôt GitHub. Les tests, eux, fûrent effectués sur un MacBook Pro avec un processeur 2,4 GHz Intel Core i5 quatre cœurs et 8 Go 2133 MHz LPDDR3 de mémoire vive.

Ci-dessous les tableaux de coûts de solutions aléatoires avec $k = 2$ pour comparaison :

n	4	5	10	15	17	20	21	22	23	24
Cost	3	6	16	49	64	50	107	117	118	132

FIGURE 1.1 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	154	203	271	980	3740	4992	5000845

FIGURE 1.2 – Big half instances

Ci-dessous les tableaux de coûts de solutions aléatoires avec $k = 5$ pour comparaison :

n	10	15	17	20	21	22	23	24
Cost	27	79	99	77	174	177	190	210

FIGURE 1.3 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	246	323	400	1638	6015	8009	8001640

FIGURE 1.4 – Big half instances

2 Développement des algorithmes requis

2.1 Énumération

2.1.1 Développement

L'énumération était le premier algorithme qui nous était demandé d'implémenter. Nous avons réalisé une énumération explicite des k^n solutions possibles générées pas un tableau de taille n où chaque entier contenu dans le tableau représente la classe du sommet d'indice correspondant.

2.1.2 Performances

Ci-dessous les tableaux de performances de l'énumération avec $k = 2$ et un temps maximal de $t = 60$ secondes :

n	4	5	10	15	17	20	21	22	23	24
Cost	2	4	13	43	60	40	107	103	116	131
Time (sec)	0.04	0.04	0.05	0.73	3.18	14.15	*timeout*	*timeout*	*timeout*	*timeout*

FIGURE 2.1 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	152	190	$infini$	$infini$	$infini$	$infini$	$infini$
Time (sec)	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*

FIGURE 2.2 – Big half instances

Ci-dessous les tableaux de performances de l'énumération avec $k = 5$ et un temps maximal de $t = 60$ secondes :

n	10	15	17	20	21	22	23	24
Cost	25	$infini$	$infini$	$infini$	$infini$	$infini$	$infini$	$infini$
Time (sec)	43.89	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*

FIGURE 2.3 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	$infini$	$infini$	$infini$	$infini$	$infini$	$infini$	$infini$
Time (sec)	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*

FIGURE 2.4 – Big half instances

2.2 Descente de gradient

2.2.1 Développement

La descente de gradient est le deuxième algorithme que nous avons eu à implémenter. Le but est de partir d'une solution initiale, regarder l'intégralité de son voisinage et prendre le meilleur voisin améliorant comme nouvelle solution. On recommence ainsi de suite jusqu'à ce qu'on ne trouve plus aucune solution ne possédant aucun voisin améliorant (optimum local).

2.2.2 Performances

Ci-dessous les tableaux de performances de la descente de gradient avec $k = 2$ et un temps maximal de $t = 60$ secondes :

n	4	5	10	15	17	20	21	22	23	24
Cost	2	4	13	43	61	40	107	104	116	131
Time (sec)	0.01	0.04	0.01	0.04	0.05	0.05	0.01	0.05	0.02	0.05

FIGURE 2.5 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	152	192	191	851	3560	5060	4998099
Time (sec)	0.06	0.03	0.18	2.19	*timeout*	*timeout*	*timeout*

FIGURE 2.6 – Big half instances

Ci-dessous les tableaux de performances de la descente de gradient avec $k = 5$ et un temps maximal de $t = 60$ secondes :

n	10	15	17	20	21	22	23	24
Cost	25	76	98	73	173	175	188	210
Time (sec)	0.04	0.04	0.05	0.05	0.01	0.05	0.04	0.05

FIGURE 2.7 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	246	320	344	1429	5875	7998	8000065
Time (sec)	0.06	0.07	0.21	2.65	*timeout*	*timeout*	*timeout*

FIGURE 2.8 – Big half instances

2.3 Recuit simulé

2.3.1 Développement

Le recuit simulé est un algorithme se raprochant de la descente de gradient. Il cherche de meilleures solutions se trouvant dans son voisinage, néanmoins le but de la méta-heuristique est de réussir à se sortir d'optimums locaux. Pour cela une température vient s'ajouter à notre modèle. La température commence à un certain niveau et dimunue progressivement, plus la température est élevée, plus nous avons la chance d'accepter une nouvelle solution non améliorante.

2.3.2 Paramétrage de la température (boucle externe)

La température est importante car, si elle est trop élevée, nous allons faire grandement augmenter le nombre d'itérations "chaotiques" ce qui ne serait qu'une perte de temps. En revanche si elle est trop faible alors nous n'arriverons pas à nous sortir de nos optimums locaux. Pour trouver la meilleure valeur de la température initiale, la méthode analytique telle que décrite au dessus de l'équation (1) de la fiche recuit fut implémentée. La température optimale trouvée avec un taux d'acceptation fixé à $\tau = 0.8$ et pour une centaine de pioche aléatoire dans un voisinnage de taille 1000 pour une solution aléatoire initialement donnée était d'environ 36 degrés pour quasiment toutes les instances à l'exception de la dernière, 230 degrés. Bien évidemment il restait à définir le taux de refroidissement ainsi que la température finale qui allait directement définir le nombre d'itération de la boucle externe. Nous avons donc choisi une suite géométrique de raison $q = 0.09$ pour le refroidissement et une température finale de 0.01 degré, pour un nombre d'environ 10 itérations comme conseillé dans la littérature.

2.3.3 Paramétrage du nombre d'itération maximale (boucle interne)

Le nombre d'itérations internes correspond au nombre d'itérations avant de diminuer la température. La boucle interne cherche à trouver un optimum à température constante. C'est elle qui met en œuvre l'algorithme de Metropolis pour éviter d'être englué dans un optimum local. Il faut que cette boucle soit suffisamment longue pour permettre de récupérer la propriété des chaînes de Markov de stabilité (état stationnaire) sinon l'algorithme perd tout son intérêt. Comme suggéré dans la fiche recuit nous avons effectué plusieurs tests (polynôme de degré compris entre 1 et 3) et nous avons convenu que $10 * n$ était un bon nombre d'itérations internes.

2.3.4 Performances

Ci-dessous les tableaux de performances du recuit simulé avec $k = 2$ et un temps maximal de $t = 60$ secondes :

n	4	5	10	15	17	20	21	22	23	24
Cost	2	4	13	46	60	40	107	106	117	131
Time (sec)	0.04	0.1	0.01	0.05	0.06	0.02	0.02	0.03	0.06	0.03

FIGURE 2.9 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	152	193	221	970	3721	4868	5000093
Time (sec)	0.06	0.04	0.6	0.23	4.30	16.72	*timeout*

FIGURE 2.10 – Big half instances

Ci-dessous les tableaux de performances du recuit simulé avec $k = 5$ et un temps maximal de $t = 60$ secondes :

n	10	15	17	20	21	22	23	24
Cost	25	76	98	74	173	175	189	209
Time (sec)	0.01	0.05	0.06	0.02	0.02	0.03	0.06	0.03

FIGURE 2.11 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	246	324	397	1588	5966	8043	7999854
Time (sec)	0.06	0.04	0.04	0.13	2.28	9.15	*timeout*

FIGURE 2.12 – Big half instances

2.4 Tabou

2.4.1 Développement

La méthode tabou est la dernière méta-heuristique que nous aillons eu à implémenter. Une fois de plus le raisonnement est similaire à celui de la descente de gradient, néanmoins ici nous continuons d'itérer jusqu'à atteindre le nombre d'itération maximale. Ainsi nous ne prenons plus le premier voisin améliorant mais le meilleur voisin. De plus, pour nous sortir des minimums locaux, une liste tabou est implémentée. Elle sert de mémoire des derniers mouvements effectués que nous ne voulons pas reproduire par peur de revenir sur nos pas. Une fois le nombre d'itération maximale atteint, nous renvoyons la meilleure solution trouvée par la méthode tabou. Ce qui est stocké dans la méthode tabou est le *swap* effectué entre deux sommets, ainsi un *swap* réalisé entre deux sommets, ne pourra pas être reproduit avant "taille de de la liste tabou" itérations.

2.4.2 Paramétrage de la taille de la liste tabou

Malgré tout nos efforts, quelque soit le nombre d'itérations, le *timeout* ou l'instance utilisée l'impact de la taille de la liste tabou (tant que la taille n'est pas de 0) n'avait aucun impact sur la valeur de la solution trouvée ou le temps d'exécution de l'algorithme. Nous pensons que cela est dû à une taille de voisinage trop grande. Nous avons donc laissé une valeur par défaut de 7.

2.4.3 Paramétrage du nombre d'itérations

Le nombre d'itérations maximale est important car s'il est trop faible nous pouvons ne pas trouver de meilleures solutions accessibles car nous restons bloqué dans des optimums locaux et s'il est trop grand nous allons continuer d'itérer dans le vide sans trouver de meilleures solutions. Nous sommes parti du principe que le nombre d'itération devrait dépendre du nombre de solutions et donc de la taille de notre instance. Nous avons donc décidé d'avoir un nombre d'itérations maximale comme étant un facteur de n (le nombre de sommets du graphe).

Nous avons essayé différentes valeurs et nous avons trouvé empiriquement que globalement sur les petites instances (moins de cent sommets), n itérations semblent amplement suffire à trouver des solutions optimales ou très proches de l'optimal (de valeurs identiques à l'énumération du moins) en un temps négligeable (moins d'une demi seconde). Le tabou pour l'instance de cent sommets met 7 secondes pour 100 itérations, la question de la recherche du nombre d'itérations minimum pour la convergeance a donc commencé à se poser à l'instance "centSommets". En réduisant progressivement le nombre d'itérations maximale nous nous sommes rendu compte que 30 itérations suffisait à la convergeance sur cent sommets. Malheureusement le coût en temps de la méthode tabou en fonction du nombre d'itérations et de la taille des instances semble croître bien trop vite et une simple itération

sur l'instance de taille cinq cent sommets dure 8 secondes et 10 itérations ne suffisent évidemment pas à sa convergeance. C'est pourquoi nous avons décidé de fixer le nombre d'itérations maximale à n. Sur les instances de $n = 4$ à $n = 100$, le temps de calcul reste raisonnable et la convergeance est assurée, cependant sur les instances de cinq cent à dix mille sommets, le facteur temps semble être la grande limite.

2.4.4 Performances

Ci-dessous les tableaux de performances de l'algorithme tabou avec $k = 2$ et un temps maximal de $t = 60$ secondes :

n	4	5	10	15	17	20	21	22	23	24
Cost	2	4	13	43	60	40	107	103	116	131
Time (sec)	0.04	0.04	0.04	0.05	0.06	0.06	0.04	0.08	0.05	0.08

FIGURE 2.13 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	152	189	199	835	3690	4970	4998344
Time (sec)	0.09	0.14	0.72	10.94	*timeout*	*timeout*	*timeout*

FIGURE 2.14 – Big half instances

Ci-dessous les tableaux de performances de l'algorithme tabou avec $k = 5$ et un temps maximal de $t = 60$ secondes :

n	10	15	17	20	21	22	23	24
Cost	25	76	99	73	173	175	188	210
Time (sec)	0.01	0.05	0.05	0.06	0.03	0.04	0.07	0.07

FIGURE 2.15 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	246	320	344	1415	5876	7988	8000615
Time (sec)	0.09	0.12	0.52	7.36	*timeout*	*timeout*	*timeout*

FIGURE 2.16 – Big half instances

3 Algorithmes améliorés

3.1 Descente de gradient : première solution améliorante

3.1.1 Développement

La descente de gradient regarde tout son voisinage avant de prendre la meilleure solution améliorante. Nous nous sommes donc dit que sur les plus grandes instances, parcourir tout le voisinage est très couteux en temps. De plus prendre une solution améliorante plutôt que la meilleure reste dans l'esprit de la descente de gradient puisque le programme ne s'arrêtera que s'il n'y a aucunes solutions améliorantes dans son voisinage. Ceci est donc une implémentation de la descente de gradient tel que nous itérons sur le premier voisin améliorant, ne parcourant ainsi pas tout le voisinage. Les résultats ne méritaient pas de nouveaux tableaux à eux seuls (de plus, nous manquions de temps). Néanmoins ce nouvel algorithme améliorait sensiblement le temps d'exécution de la descente (1.5 à 2 fois plus rapide sur 100 sommets).

3.2 Améliorations de Tabou

3.2.1 Tabou avec aspiration

La méthode du tabou possède une exception, l'aspiration. De prime abord un mouvement ayant été effectué est mis dans la liste tabou et n'en ressort qu'une fois qu'il a été libéré du à la finitude de la liste tabou. Néanmoins une amélioration de tabou consiste à autoriser un mouvement de la liste tabou s'il est suffisamment améliorant. Ici notre définition de "suffisamment améliorant" eut été que le mouvement soit autorisé si la solution donnée fut strictement meilleure que la solution pour laquelle il fut placé dans la liste tabou.

3.2.2 Grandes instances : voisinage partiel

La première amélioration que nous avons décidé de mettre en place fut un voisinage partiel. Nous avons créé une fonction qui à partir d'une solution donnée et un entier n passé en paramètre, nous renvoie aléatoirement une liste de n voisins. L'intérêt est que la fonction ne calcule pas le voisinage entièrement (qui peut être trop grand sur les dernières instances). Ainsi en ne regardant pas tout le voisinage d'une solution mais seulement 100 voisins (nombre de voisins qui par expérimentation nous semble un bon compromis entre vitesse et résultat), nos méta-heuristiques pourraient être grandement accélérée. L'impact sur la descente de gradient ne fut pas très grande (surtout comparé à son amélioration) et le recuit simulé ne demandait pas à être accéléré au prix de rater de meilleures solutions. De fait les résultats ne sont intéréssants que pour l'algorithme tabou.

3.2.3 Performances

Ci-dessous les tableaux de performances de l'algorithme tabou avec aspiration, $k = 2$, un voisinage de taille $s = 100$ et un temps maximal de $t = 60$ secondes :

n	4	5	10	15	17	20	21	22	23	24
Cost	2	4	13	43	60	40	107	103	116	131
Time (sec)	0.04	0.01	0.03	0.06	0.07	0.04	0.08	0.08	0.08	0.08

FIGURE 3.1 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	152	189	191	847	2900	3533	4980940
Time (sec)	0.09	0.07	0.14	0.54	9.64	36.92	*timeout*

FIGURE 3.2 – Big half instances

Ci-dessous les tableaux de performances de l'algorithme tabou avec aspiration, $k = 5$, un voisinage de taille $s = 100$ et un temps maximal de $t = 60$ secondes :

n	10	15	17	20	21	22	23	24
Cost	25	76	98	71	173	175	188	210
Time (sec)	0.02	0.06	0.06	0.03	0.03	0.06	0.03	0.06

FIGURE 3.3 – Small half instances

n	25	30	50	100	500	1000	10000
Cost	246	320	342	1428	4915	6141	7971704
Time (sec)	0.04	0.08	0.08	0.27	4.75	15.83	*timeout*

FIGURE 3.4 – Big half instances

4 Conclusion

Avec $k = 2$ nous savons avoir trouvé les solutions optimales pour les instances 4 à 20 sommets puisque l'énumération explicite renvoie la solution optimale si elle est bien codée. Nous pensons aussi avoir trouvé les solutions optimales pour les instances 21 à 50 sommets bien que nous n'ayons pas de garanties. Cependant il est important de souligner le fait que nos algorithmes n'ont jamais pu trouver de solutions avec un coût plus bas même avec des temps de recherche bien plus élevés et de multiples solutions de départ différentes.

L'énumération est, comme on pouvait s'en douter, la première à trouver ses limites en terme de taille d'instance et de coût en temps. Elle nous aura tout de même permis de vérifier la cohérence de nos autres algorithmes sur les petites instances.

La descente de gradient est plus performante que ce que nous espérions en réalisant la taille des instances, elle permet de trouver des solutions de très bonne qualité jusqu'à l'instance 100 sommets où il est capable de trouver un optimum local avant son temps limite. À l'instance 500 sommets on commence à apercevoir une baisse de performance sur le temps d'exécution et sur l'instance 1000 sommets avec un temps limite la descente de gradient ne fait guère mieux que l'aléatoire. La descente de gradient améliorée est relativement plus rapide et donne donc de meilleures solutions sur l'instance 500 sommets.

Le recuit simulé a l'avantage d'être notre algorithme le plus rapide, la qualité des solutions qu'il produit pour résoudre ce problème est certes faible mais il améliore toujours la solution initialement donnée en paramètre et ce très rapidement. Si nous avions eu plus de temps nous aurions pu essayer de l'améliorer. Une idée serait de lui passer en entrée une bonne solution trouvée au préalable grâce à une autre heuristique.

La méthode tabou est la plus efficace et nous donne les meilleurs résultats. Néanmoins si on ne lui accorde aucunes améliorations, elle dépasse son temps limite dès l'instance 500 sommets, la rendant aussi lente que la descente de gradient et aussi performante que le recuit simulé. C'est grâce aux améliorations portant sur un voisinage à taille variable et au processus d'aspiration que la méthode tabou brille puisqu'elle trouve très nettement les meilleures solutions de tout nos algorithmes et termine son execution en 25 secondes sur l'instance 1000 sommets. Cependant l'instance 10 000 sommets reste bien trop grosse et bien que nous ayons pu trouver des solutions de l'ordre de $4.7 * e^6$ assez rapidement, il faudrait très certainement un temps non raisonnable à l'algorithme tabou pour terminer son execution normalement.

Ouverture

Cette partie correspond à ce que nous aurions voulu faire si nous avions eu plus de temps à consacrer à ce projet. Le sujet nous ayant plu, il y a au final beaucoup de choses que nous aurions voulu faire si nous avions eu plus de temps. Nous avions déjà mentionné le fait que nous aurions voulu améliorer notre recuit simulé pour qu'il puisse nous donner de meilleurs résultats concernant la qualité des solutions trouvées. Comme évoqué un peu plus haut, nous pensions notamment au fait de donner en entrée une bonne solution (résultat de la descente de gradient effectué sur un plus petit voisinage par exemple) au recuit simulé pour voir si nous pouvions trouver une solution encore meilleure.

Nous aurions aussi voulu pouvoir faire plus de tableaux de résultats en faisant varier k. Il est vrai que notre code n'est pas du tout dépendant du $k = 2$ fixé et fonctionne en l'état très bien avec $k \in [2; n]$, nous n'avons simplement pas eu le temps des tests complets sur toutes les instances que nous avions avec un k différent de 2 ou 5.

De plus, si nous avions eu plus de temps nous aurions aussi voulu améliorer l'énumération, en faisant peut-être une énumération implicite de Branch & Bound avec une higher bound initiale correspondant à la meilleure solution trouvée par l'une de nos méthodes (sûrement l'algorithme tabou ou la descente de gradient). De ce fait nous aurions ainsi sûrement put trouver et valider les optimums sur un spectre bien plus large d'instances. Enfin si nous avions eu plus de temps, nous aurions pu essayer d'améliorer d'avantage la méthode tabou dans l'objectif final de réussir à trouver une bonne solution de l'instance 10 000 sommets en un temps raisonnable.