K-partitionnement de Graphes

Aboudou Hanrifani
Lentali Thomas

20 mars 2015

Chapitre 1

Les algorithmes

1.1 Introduction

Ce projet a pour but de partitionner des graphes connexes non-orientés en K classes afin d'obtenir un minimum d'arêtes interclasses. Nous allons pour cela utiliser un algorithme d'énumération, un algorithme glouton de descente de gradient, et les méta-heuristiques recuit simulé et recherche tabou.

Tous les résultats et les temps d'exécution sont directement stockés dans un fichier (cf annexe).
Pour ce rapport, nous avons décidé d'analyser les performances de nos algorithmes pour un partitionnement en $K = 3$ classes, mais bien entendu, nos programmes peuvent exécuter tout type de partitionnement allant de 1 jusqu'au nombre de sommets du graphe étudié.
Pour une meilleure analyse, nous avons généré 5 fois nos algorithmes afin d'obtenir un temps d'exécution minimum, maximum, et ainsi calculer la moyenne des temps d'exécution.
Il en est de même pour le calcul du nombre des arêtes interclasse.

1.2 Énumération

Nous avons décidé de générer toutes les solutions possibles de l'espace de recherche. On appelle cette méthode énumération explicite. Elle a pour but de parcourir toutes les solutions, vérifier leur validité et enfin stocker la meilleure solution parmi celles qui sont valides.
Nous pouvons constater au travers des résultats, la complexité $K^{nbsommets}$ qui fait exploser le temps d'exécution de l'algorithme, ce qui nous empêche d'obtenir des résultats pour un graphe ayant plus de 17 sommets.
Cette progression du temps d'exécution est illustrée par la courbe suivante :

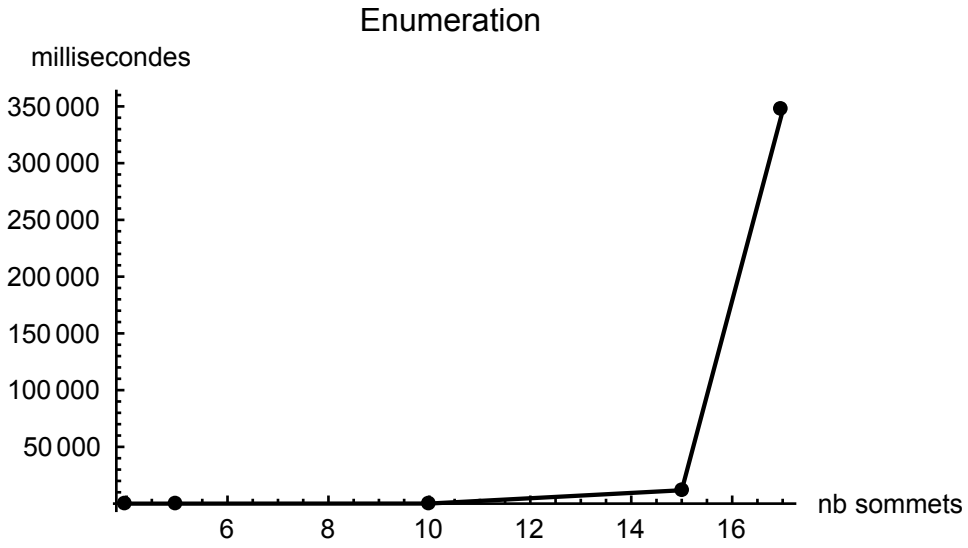


FIGURE 1.1 – Evolution de temps d'execution par rapport au nombre de sommets pour l'énumeration

1.3 Descente de gradient

Cette algorithme glouton a pour but d'améliorer la solution de manière successive jusqu'à ce qu'un minimum local ait été trouvé. Pour avoir plus de chances de tomber sur la solution optimale, nous générons plusieurs fois une solution aléatoire de départ afin de commencer notre descente en différents endroits.
En premier lieu on génère une solution aléatoire qu'on appelle solution de base. Depuis cette solution de base, on explore tout son voisinage afin de trouver le meilleure solution voisine. Pour le voisinage nous avons choisi d'utiliser la méthode swap qui consiste à échanger deux sommets de classes différentes. Nous avons utilisé swap pour tout les algorithmes car il présente l'avantage de ne pas modifier la structure de la solution et nous garantit de trouver une solution valide. Si la solution trouvée améliore le résultat, on recommence le processus jusqu'à ce que la solution soit la meilleure de son voisinage.

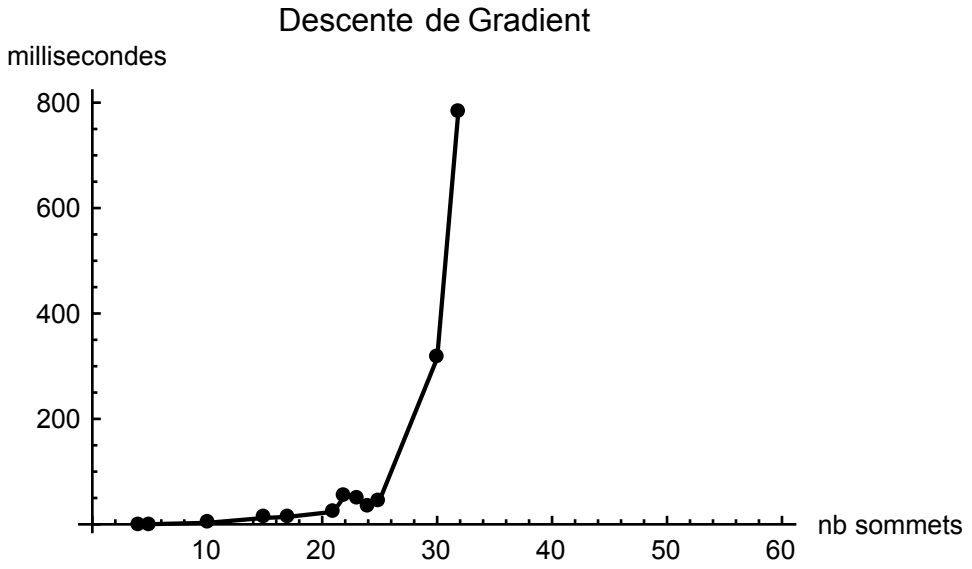


FIGURE 1.2 – Evolution de temps d'execution par rapport au nombre de sommets pour la descente

1.4 Recuit simulé

Cet algorithme est une méta-heuristique qui s'inspire de la physique et plus particulièrement de la métallurgie afin d'améliorer la qualitée d'un métal. En partant d'une haute température à laquelle la matière est devenue liquide, la phase de refroidissement conduit la matière à retrouver sa forme solide par une diminution progressive de la température jusqu'à atteindre un état d'énergie minimale qui correspond à une structure stable du métal. De la même manière, la fonction qui évalue notre solution correspond à l'énergie et on crée un paramètre virtuel afin de gérer la température. En premier lieu on génère une solution aléatoire qu'on appelle solution de base. À partir de cette solution nous allons sur 10 paliers de températures différents, suivant un schéma de refroidissement géométrique ($t_{n+1} = t_n * 0.9$), générer 1000 voisins aléatoires puis pour chaque voisin si la solution est améliorante alors on la prend, sinon on applique le critère de Metropolis afin de savoir si on garde la solution ou pas.
Le critère de Metropolis est une fonction stochastique qui définit la probabilité d'accepter une solution non améliorante. Cette fonction est définie par :

$$CritMetro(x) \rightarrow \begin{cases} 1 & si\ x \leq e^{\frac{-\Delta}{T}} \\ 0 & sinon \end{cases}$$

avec x une variable aléatoire suivant une loi uniforme $[0, 1]$.

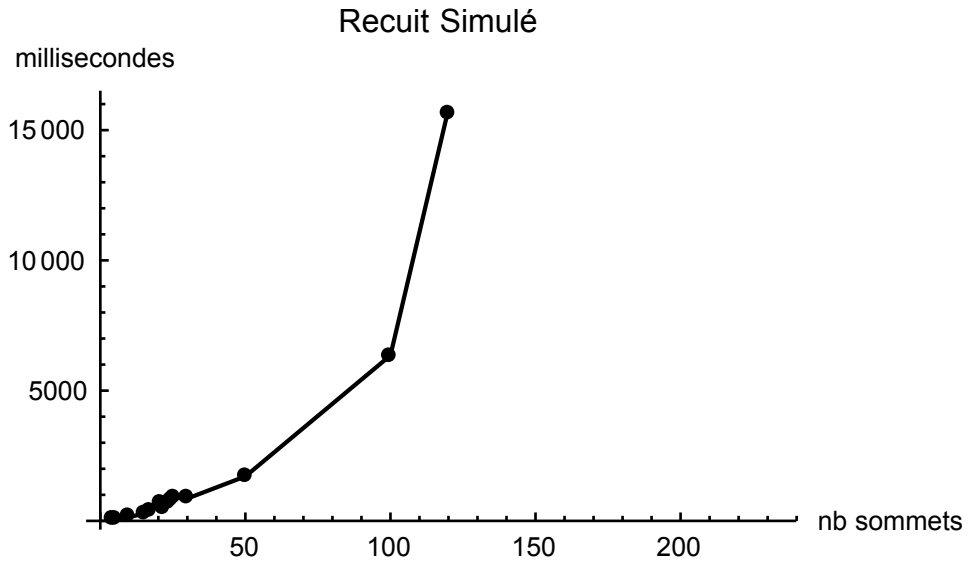


FIGURE 1.3 – Evolution de temps d'execution par rapport au nombre de sommets pour le Recuit

Out[28]=

Temperature –>	0.5	1	3	5	10
4	3	3	3	3	3
5	6	6	6	6	6
10	18	18	18	18	18
15	61	61	61	61	61
17	81	81	81	81	81
21	144	144	144	144	144
22	143	143	143	143	143
23	156	156	156	156	156
24	175	175	175	175	175
25	204	204	204	204	204
30	260	260	260	260	260
50	267	267	270	278	291
100	1140	1132	1136	1157	1223

Après comparaison des résultats obtenus par l'algorithme du recuit simulé on remarque que la température initiale de 1 degré permet d'approcher au plus près du résultat optimal. Pour ce rapport, l'algorithme est donc exécuté avec une température initiale de 1 degré.

1.5 Méthode tabou

Nous avons utilisé la recherche tabou par mouvement avec mémoire à court terme.
La liste tabou utilisée est de taille 8 et de type first in first out.
A chaque itération on parcourt l'ensemble des solutions voisines et choisit la meilleure qui n'est pas interdite, même si elle est plus mauvaise que la solution courante. Son avantage par rapport à la descente de gradient est que la liste tabou permet d'interdire certaines solutions afin de ne pas rester bloqué dans un minimum local et de changer de voisinage.

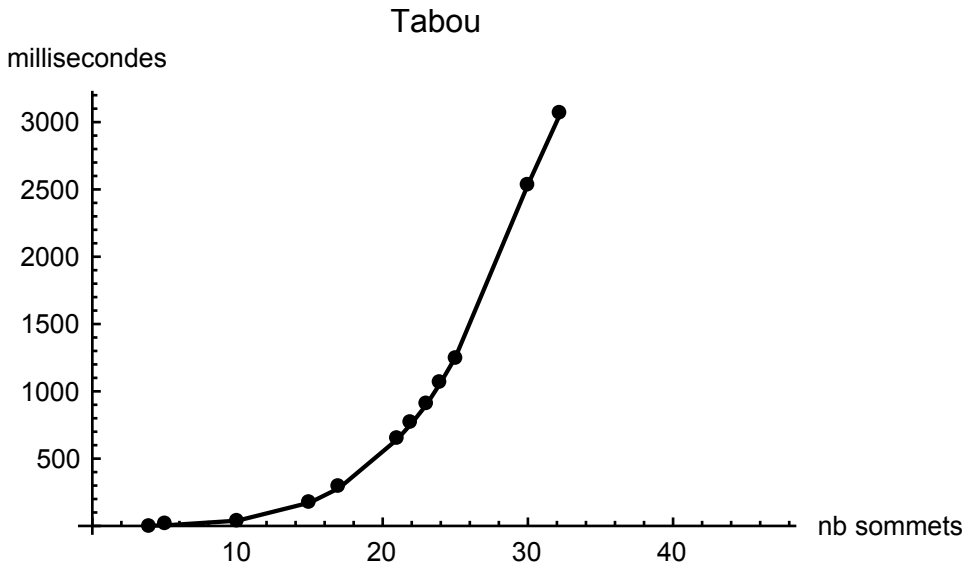


FIGURE 1.4 – Evolution de temps d'execution par rapport au nombre de sommets pour le tabou

Chapitre 2

Comparaison des temps d'execution et des solutions

Afin de pouvoir comparer l'efficacité de nos algorithmes, nous allons nous concentrer sur leurs temps d'exécution et la justesse des solutions trouvées.

2.1 Temps d'execution

Les temps d'exécutions sont très variables d'un algorithme à un autre :

Out[71]=

nb sommets	Enumeration	Descente	Recuit	Tabou
4	0.146	0.2006	87.3418	2.0318
5	0.189	0.2654	101.133	4.3058
10	62.348	2.8678	142.844	39.3174
15	11772.5	12.1844	284.671	174.155
17	346104	14.1558	401.844	281.268
21	–	23.8434	646.635	643.06
22	–	55.628	535.048	762.678
23	–	47.4088	707.709	898.537
24	–	34.4122	837.608	1065.73
25	–	45.962	893.206	1250.56
30	–	315.047	858.84	2534.11
50	–	5391.34	1708.92	19093.8
100	–	170511	6353.65	325717
500	–	–	195557	–
1000	–	–	825357	–

FIGURE 2.1 – Temps d'exécution en millisecondes

Nous allons détailler l'évolution du temps d'exécution des algorithmes pour chaque graphe.

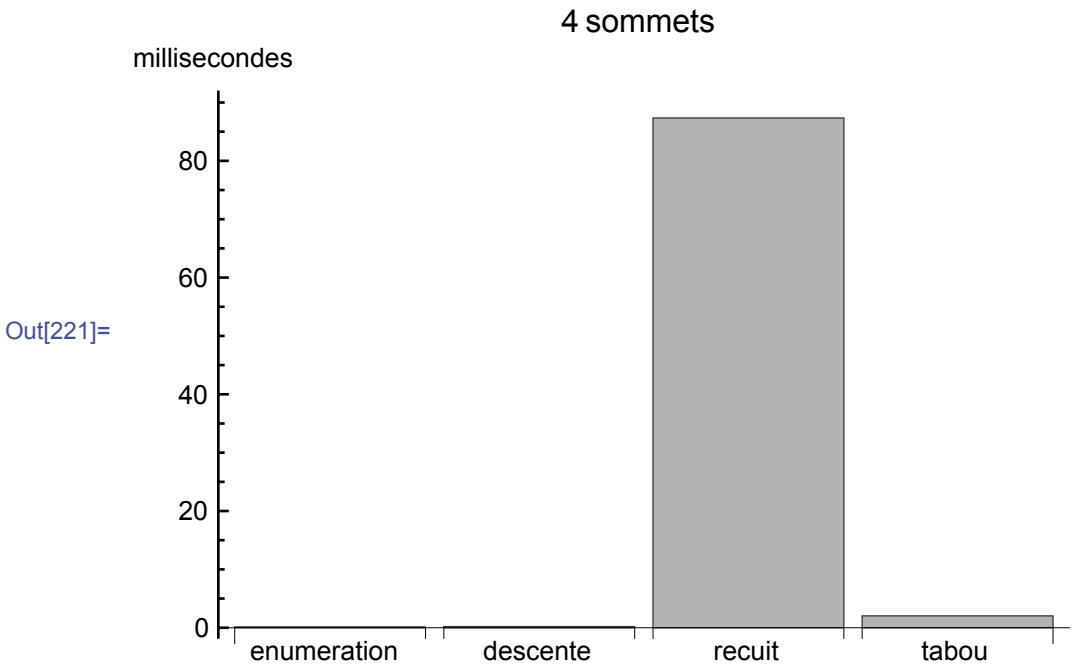


FIGURE 2.2 – 4 sommets

Pour un graphe de quatre sommets, la méta-heuristique du recuit simulé semble être la méthode la moins appropriée dépassant largement en temps les méthodes concurrentes.

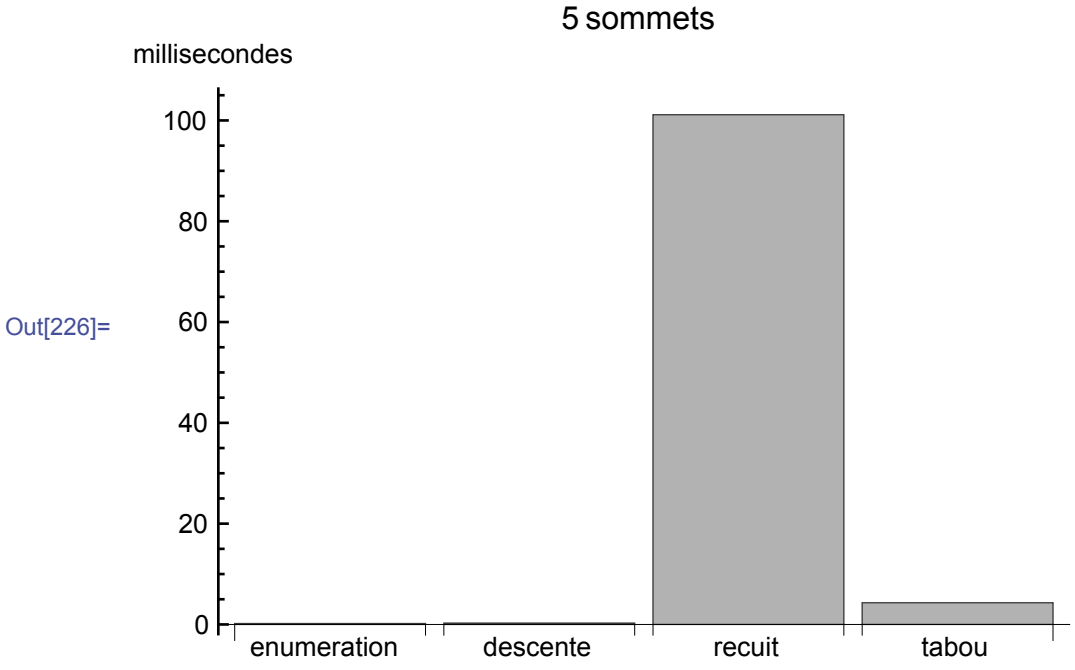


FIGURE 2.3 – 5 sommets

La situation reste la même en ajoutant un sommet au graphe.

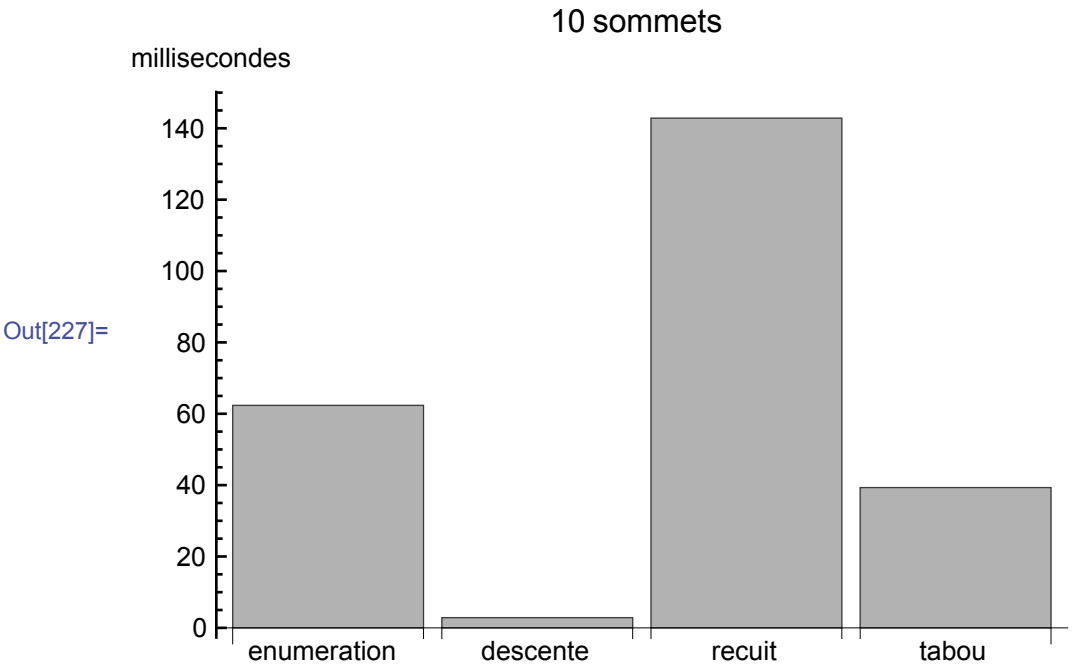
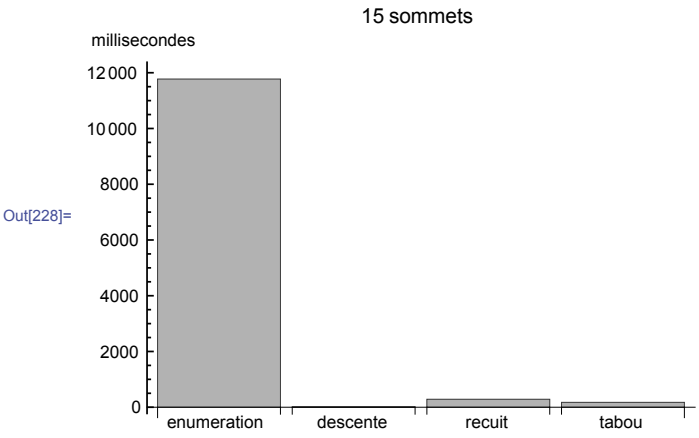
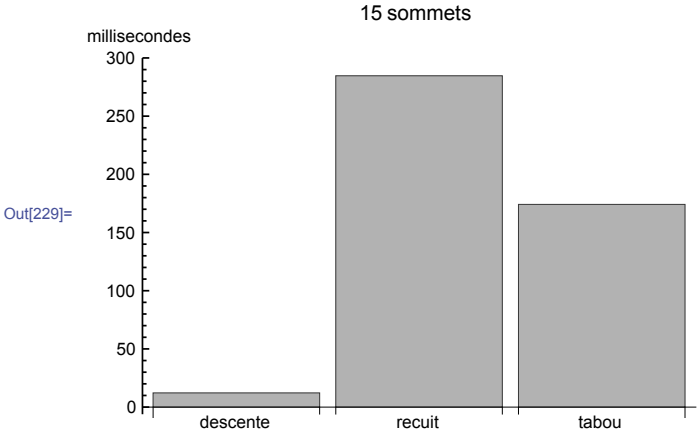


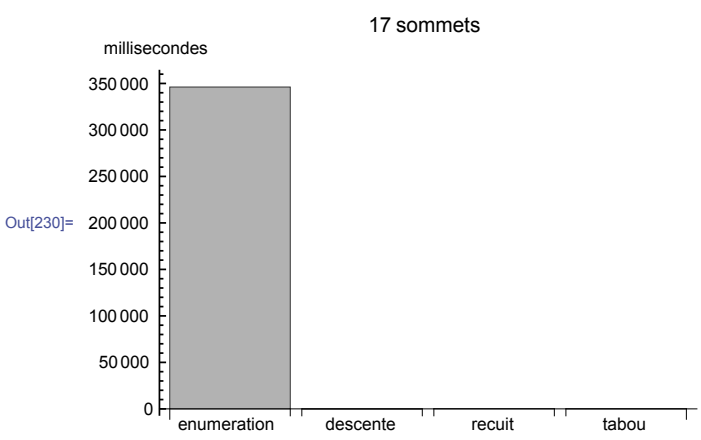
FIGURE 2.4 – 10 sommets

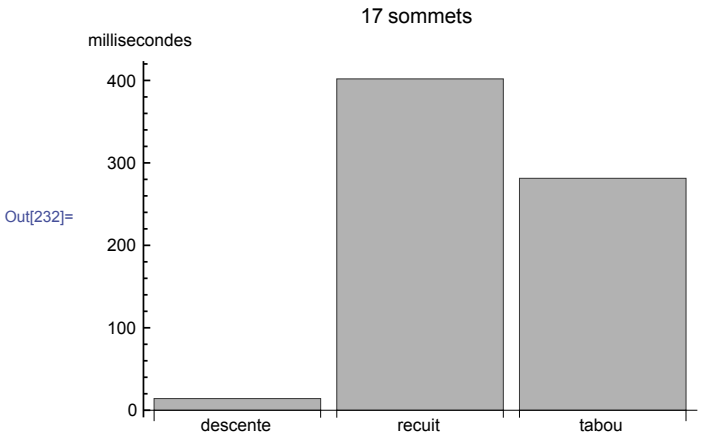
A partir de dix sommets on remarque une hausse des temps d'exécution de l'énumération de la méthode tabou alors que la descente de gradient reste très peu gourmande en temps.





Pour quinze sommets, l'énumération explose en temps, le diagramme de droite montre l'évolution des autre algorithmes rendus quasi-invisibles sur le diagramme de gauche.





Même situation que précédemment pour deux sommets de plus.

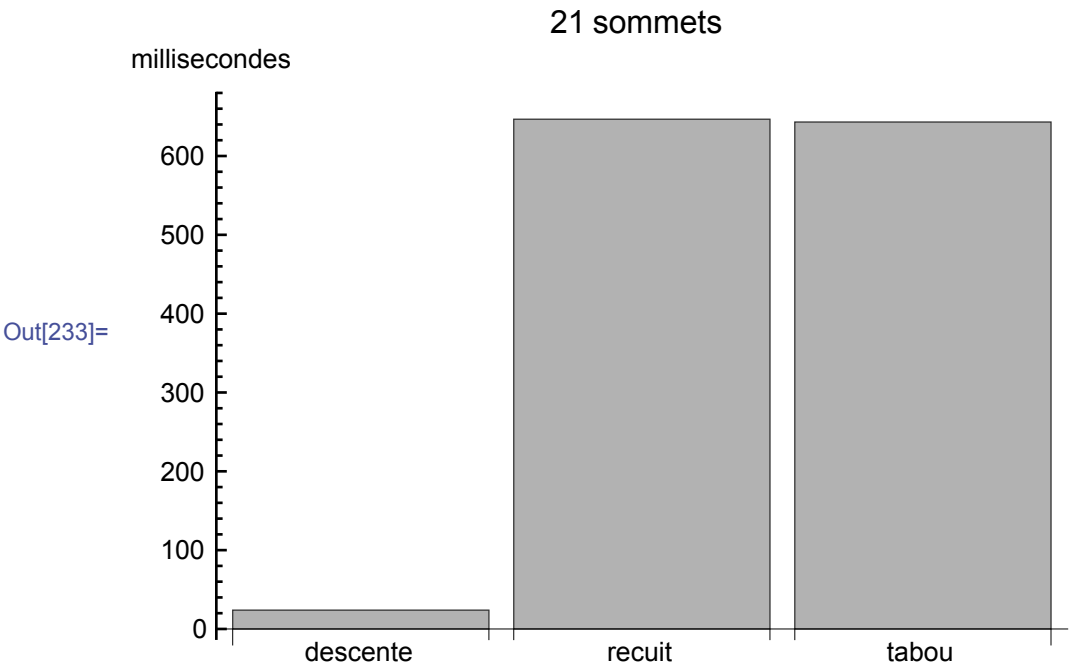


FIGURE 2.5 – 21 sommets

Avec 21 sommets, la méthode tabou demande autant de temps que le recuit simulé pour obtenir l'optimal.

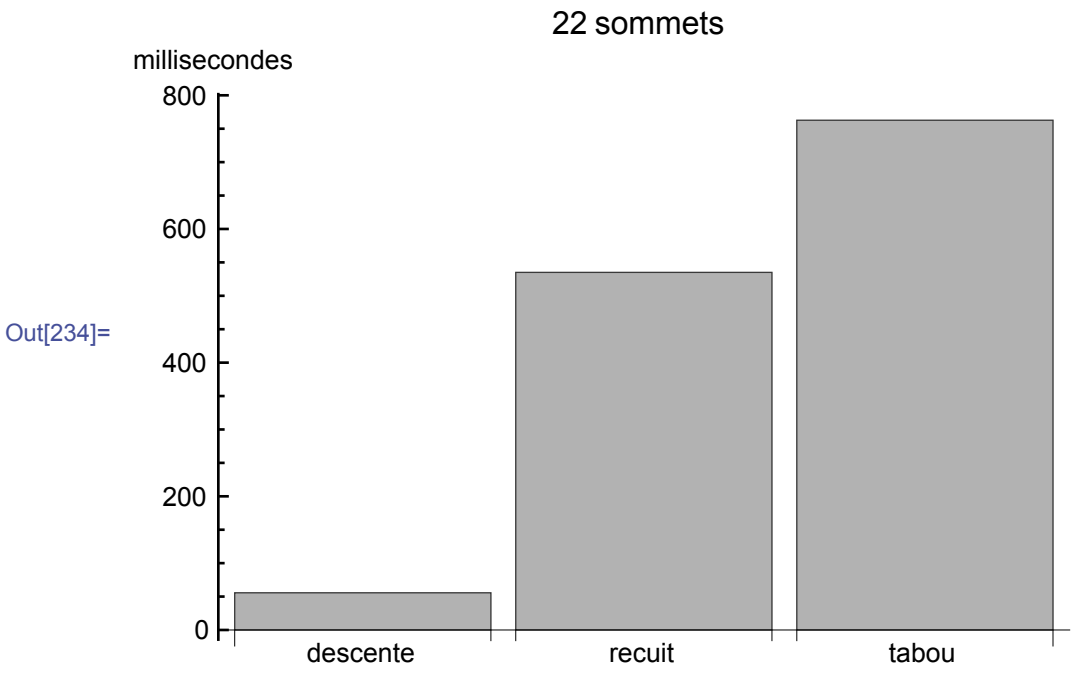


FIGURE 2.6 – 22 sommets

Pour un graphe de 22 sommets la méthode tabou finit par dépasser le recuit simulé, alors que la descente reste la plus rapide.

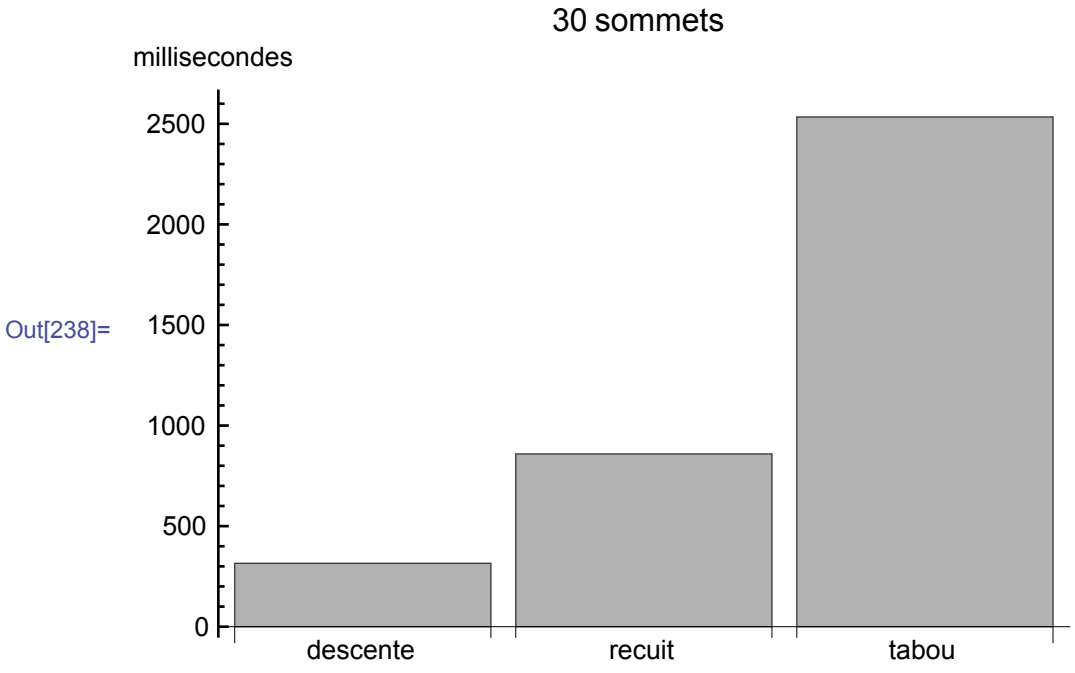
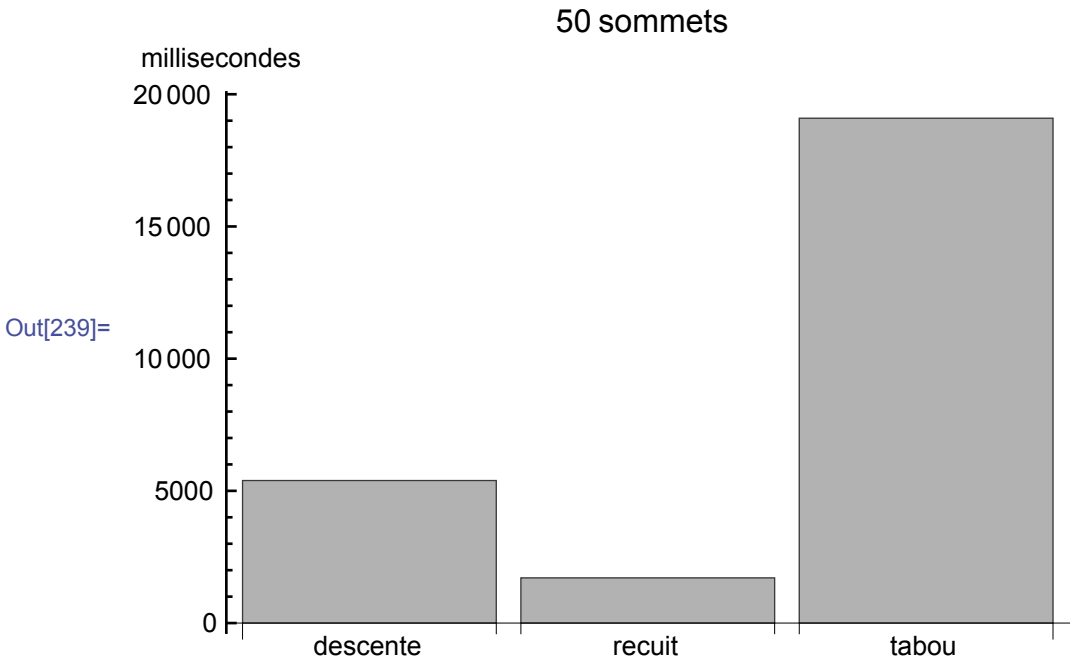


FIGURE 2.7 – 30 sommets

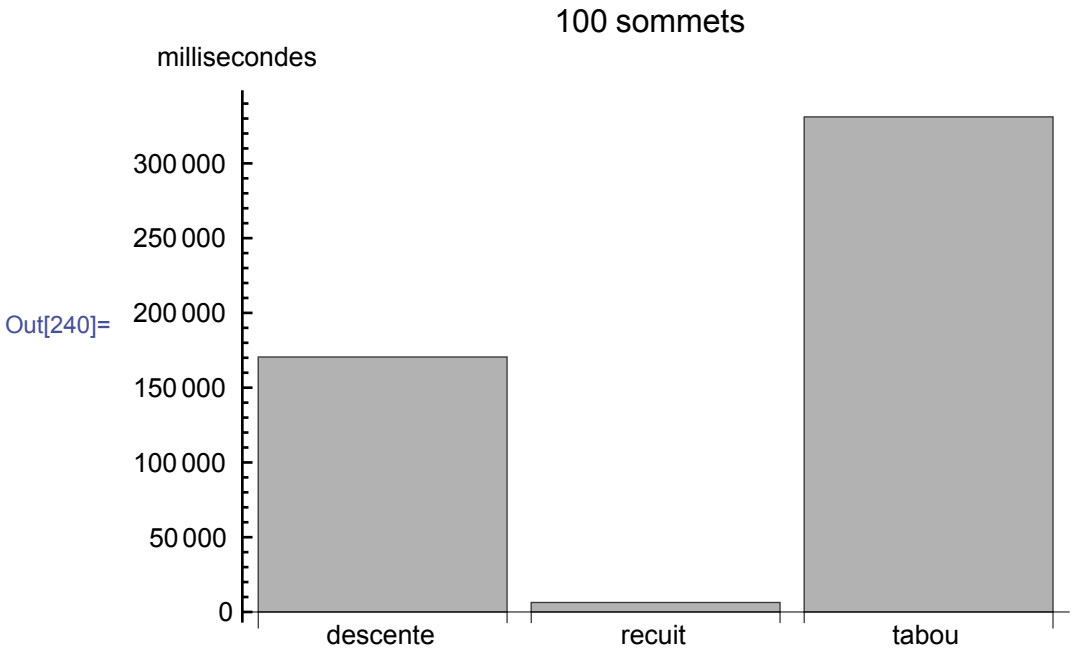
A 30 sommets, la descente commence à prendre plus de temps pour s'exécuter et le tabou est de loin le plus lent.



Out[239]=

FIGURE 2.8 – 50 sommets

Pour 50 sommets, la descente de gradient dépasse le temps nécessaire au recuit simulé.



Out[240]=

FIGURE 2.9 – 100 sommets

Pour 100 sommets, les méthodes de descente et tabou deviennent inappropriées pour trouver l'optimal du problème, au point qu'il devient impossible pour ces deux algorithmes de trouver un résultat pour

unngraphe de 500 sommets, alors que le recuit simulé nous donne un résultat pour des graphes de 500 et 1000 sommets.

Le graphique suivant permet de comparer l'évolution en terme de temps des quatre algorithmes.
Le recuit simulé, bien que le plus long en exécution pour des petit graphes, est le plus efficace lorsque le nombre de sommets des graphes augmente.

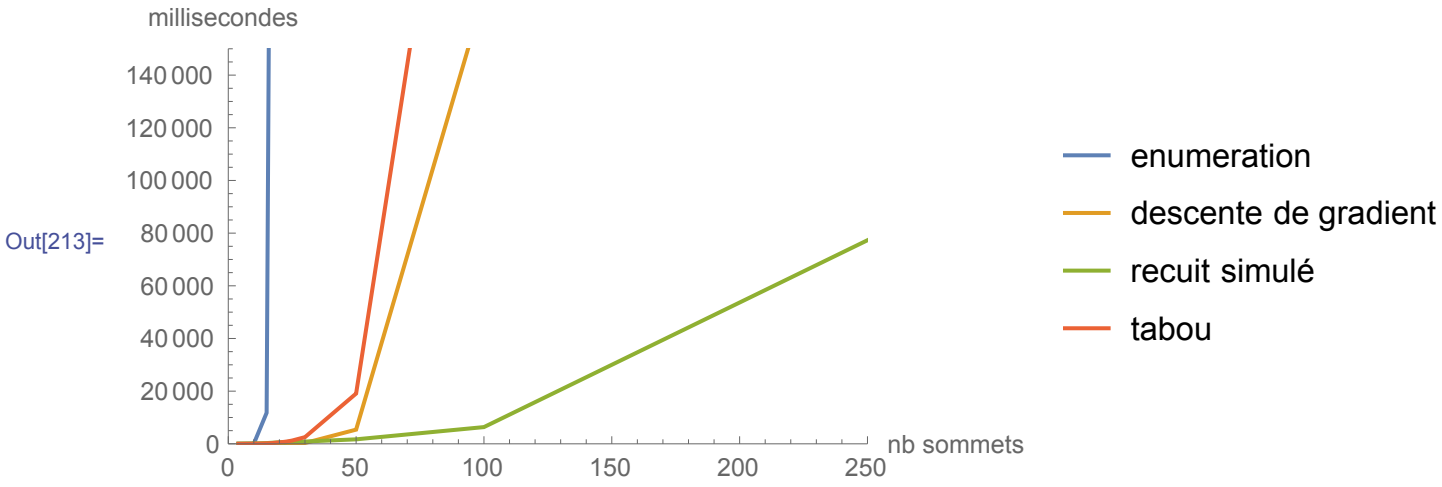


FIGURE 2.10 – Comparaison

2.2 Résultats

Out[69]=

nb sommets	enumeration	descente	recuit	tabou
4	3	3	3	3
5	6	6	6	6
10	18	18	18	18
15	61	61	61	61
17	81	81	81	81
21	–	144	144	144
22	–	143	143	143
23	–	156	156	156
24	–	175	175	175
25	–	204	204	204
30	–	260	260	260
50	–	268	267	270
100	–	1133	1132	1150
500	–	–	3840	–
1000	–	–	4776	–

FIGURE 2.11 – Résultats

Jusqu'au graphe de 17 sommets, on peut affirmer que les algorithmes trouvent tous la solution optimale puisque l'énumération est un algorithme exact.
On peut penser que, du graphe de 21 sommets jusqu'au graphe de 30 sommets, les valeurs trouvées sont optimales puisque tous les algorithmes trouvent les mêmes valeurs.
A partir du graphe à 50 sommets, les méthodes commencent à avoir des résultats différents et c'est le recuit simulé qui atteint les valeurs les plus petites.

Conclusion

Le problème de partitionnement de graphe demande beaucoup de ressources afin d'obtenir un résultat optimal.
Néanmoins il est possible d'approcher au plus près de l'optimal avec des temps raisonnables grâce aux algorithmes gloutons et à des méta-heuristiques.
Ce projet nous convainc de l'efficacité de ces méthodes, notamment celle du recuit simulé qui nous a permis d'avoir un résultat pour un graphe de 1000 sommets (ce qui aurais pris avec l'énumération un temps qui dépasse les limites de ce projet).