Steiner Tree Problem Optimization

Louis Cohen

February 11, 2018

Contents

1 Abstract

Dans ce rapport je presenterai différentes approches choisies pour approximer le probleme des arbres de Steiner. Basé sur le papier ref ref pour les choix des voisins.

2 Introduction

Le problème des arbres de Steiner est un problème de graphe NP Complet. Il consiste à, pour un graphe pondéré et un sous ensemble de noeuds (les terminaux), trouver le sous ensemble d'aretes de poids minimal connectant ces terminaux. Ce problème largement étudié trouve de nombreuses applications blablabla. Il n'est évidemmnet pas possible de résoudre completement le problème dans le cas général. Cependant dans certains cas le calcul de l'optimum est possible, par exemple avec un nombre faible de terminaux ou encore pour un graphe avec faible treewidth. Enfin dans le cas général il reste comme approche l'approximation de l'optimum. Dans la majeure partie des cas en effet une approximation est largement appréciable. Dans un premier temps certaines approximations peuvent être deterministes. Nous nous baserons sur l'une d'entre elle **ref ref** . Nous randomiserons ensuite nos approches. Dans la section 1 je décrirais le problèmes et lui donnerais un cadre théorique. La section 2 sera dédiée à la description d'une approximation déterministe. La section 3 expliquera la notion de voisin de solution qui servira dans pour toutes les approximations suivantes. La section 4 se penchera sur des recherches locales tandis que la section 5 présentera des algorithmes biologiques (principalement un algorithme moléculaire). Enfin la section 6 comparera les différentes approximations.

3 Problèmes des arbres de Steiner

Soit $G = (V, E)$ un graphe (V son ensemble de noeuds et E son ensemble d'arêtes, muni d'une fonction de poids $w : E \to \mathbb{R}$. Soit de plus un sous ensemble de noeuds $T \subset V$, les terminaux. Le but est de trouver le sous ensemble d'arêtes de poids minimal connectant tous les terminaux. On dira qu'un sous ensemble est admissible si il connecte bien les terminaux. $\arg\min_{S \subset E/S \quad admissible} \sum_{e \in S} w(e)$.

On appellera graphe induit par une solution le graphe composé des arêtes de l'ensemble solution S et des noeuds appartement a cet ensemble d'arêtes.

On dira par la suite qu'une solution est optimale si elle est admissible et que la suppression de n'importe quel arête n'en fait plus une solution optimale. On peut remarquer qu'une solution est optimale si et seulement si le graphe induit par la solution est un arbre dont les feuilles ne sont que des terminaux.

4 Optimisation déterministes

La plus-part de nos approximations auront pour solution de base une 2-approximation déterministe (dans le code fonction first_solution() dans le fichier local_search()). Son principe est relativement simple. On va créer un nouveau graphe $G_{ter} = T, E_T$. Les noeuds sont donc les terminaux et les arêtes sont étiquetées par le poids du plus court chemin entre les deux terminaux. On va ensuite calculer un arbre couvrant minimal $Tree_{ter}$ de G_{ter}. Enfin va créer notre solution (2-approximation) du problème d'arbre de steiner $First\_solution$: pour chaque arête présente dans $Tree_{ter}$ on ajoute toutes les arêtes du plus court chemin correspondant dans G_{ter} a $First\_solution$. On remarque donc que $First\_solution$ est bien une solution admissible vu que tous les terminaux seront ajoutés. On

peut prouver que cet algorithme donne une deux approximation (voir ...). Je ne ferai pas ici la démonstration mais me contenterai de présenter un cas ou la borne est atteinte : FIGURE A RAJOUTER... On voit que notre algorithme va choisir les arêtes $(1,2)$ $(2,3)$ et $(1,3)$ sans passer par le noeud 4 alors que l'optimum (si $2a < b$ est de choisir les arêtes $(1,4)$ $(2,4)$ et $(3,4)$. Quand b tend vers 2a le ratio entre les deux tends vers 2.

5 Notion de voisin

5.1 Presentation du précédent papier

Par la suite la pluspart de nos optimisation se baseront sur une notion de "voisin" des solutions. Il faut évidemment que ces "voisins" soit proche d'un point de vue de score face au problème. Cette proximité dans le cas de graphes comme les notre sera logiquement proche d'une métrique comparant les ensembles de sommets. Cependant comme expliqué dans REFERENCE, il n'est pas suffisant de regarder les ensembles de d'aretes directement. L'ajout ou la suppression d'arete ne suffit pas. On a donc implémenté un nouveau "modificateur" : l'ajout de chemin. Pour résumer nous avons quatre "modificateurs" d'une solution courante :

L'addition d'aretes : on ajoute un certain nombre d'arêtes "voisine" de notre graphe. C'est à dire d'arêtes dont au moins un des noeuds fait déjà partie de la solution. En effet sinon on risque de souvent rajouter des arêtes complétement déconnectées de notre graphe qui n'ont que très peu D'Intérêt.

La suppresion d'aretes : on supprime des arretes qui laisse notre solution admissible. Apres cette suppression on recalcule les composantes connexes et on ne garde que la composante connexe contenant tous les terminaux (ils sont tous dans la meme car la solution reste faisable)

Add a path : We randomly choose two nodes of our current solution and we add all the edges of a shortest path between this ndoes

Clear : clear make the solution minimal. While it's possible it delete an edge and calculate the connected componnents and only keep the one containing the terminals.

5.2 Neighbour choice

On a de nombreux choix pour comment parcourir notre espace de solution. Nous appellerons "modifications élémentaires" les trois premieres modifications décrites plus haut. Le clear sera généralement utilisé en fin de création d'une solution. J'ai implémenté 3 différentes manière d'obtenir un voisin. Toutes dépendent d'un radius r de recherche qui va correspondre au nombre de modifications élémentaires qu'on va s'autoriser.

La fonction voisin renvoie une solution : la solution qu'on lui donne à laquelle on a appliqué r fois une des fonctions *one_step_search* décrites ci dessous, puis applique clean sur cette solution.

5.2.1 Version 1

Cf one_step_search() dans le fichier local_search.

La version la plus simple qui consiste à tirer aléatoirement l'une des trois modification élémentaire et l'appliquer. Cette méthode est relativement efficace. Seul problème le coup de la suppression d'arête est (dans mon implémentation) très elevée. En effet il faut pour cahque arêtes de la solution en cours calculer toutes les composantes connexes de la solution privée de cette arête. Après calcul il était évident que cette modification élémentaire était plus de 10 fois plus coûteuse en pratique que les autres modifications élémentaires. D'ou l'idée de la seconde version.

5.2.2 Version 2

Cf one_step_search_v2() dans le fichier local_search.

Cette version voulait s'affranchir du problème du coût de suppression... Elle s'est donc complètement affranchie de la suppression d'arête. En effet cette version est environs 5 fois plus rapide mais comme décrit ci dessous, ses résultats sont bien plus faibles. On remarque donc que cette manière d'explorer l'ensemble des solutions semble nécessaire et que un simple clean régulier des solutions ne suffit pas. Fort de ce constat nous avons donc proposé la version 3.

5.2.3 Version 3

Cf one_step_search_v3() dans le fichier local_search.

Cette ultime version n'est pas spécialement plus rapide (bien que légèrement) que la précédente. Elle est légèrement moins randomisée. En effet on remarque que faire des suppression a tour de bras avant d'avoir modifié notre solution ne semble pas pertinent. On va donc forcer une modification d'ajout (aléatoirement ajout d'arêtes ou de chemin) puis une suppression, puis à nouveau une modification d'ajout. Ceci donne de meilleurs résultats que la version 1 entièrement randomisée. Un idéal serait sans doute une version 4 a mi chemin entre les deux, forçant avec haute probabilité des additions puis des suppressions puis des additions mais laissant un léger aléatoire. On peut pour certains tests choisir aléatoirement entre appliquer la version 1 et la version 3.

5.2.4 Comparaison des versions

La figure ... nous présente un exemple de test sur une instance (instance 039) de la fonction local_search_only_better (décrite dans la partie suivante) avec 1000 itérations. Le temps d'exécution est relativement long (environs 25 minutes pour la version 1 et 3 et 5 pour la version 2). Les points rouges sont le gain de notre meilleure solution a l'instant k et les points bleus le gain du voisin calculé pour l'instant k.

On voit clairement que la version 2 n'est pas très efficace : convergence plus lente pour des valeurs équivalentes, convergence finale beaucoup plus haute que les autres versions et gains des solutions visitées bien plus hauts que celles des deux autres.

Les versions 1 et 3 semblent assez proches, bien que la version 3 semble donner de meilleurs résultats.

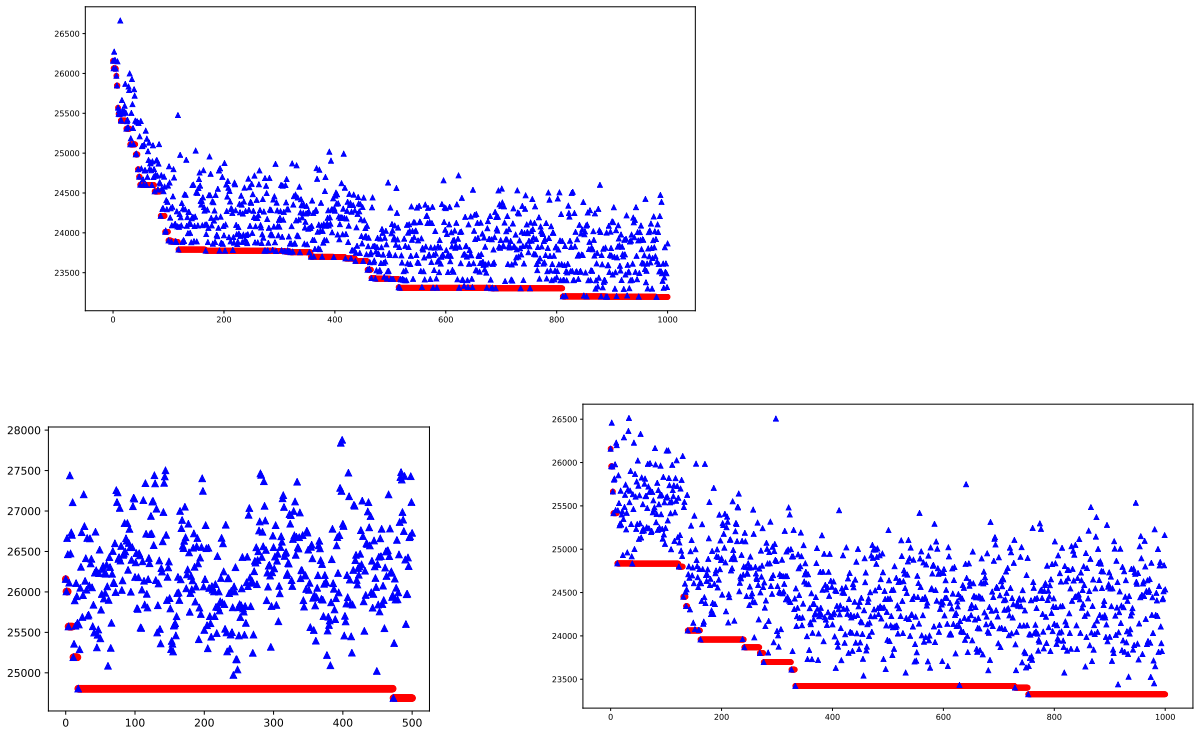

Figure 1: Dans l'ordre de gauche à droite : version 1, 2, 3

6 Local Searchs

On va vouloir parcourir l'ensemble des solutions a la recherche de minimum locaux. Tout le problème de ces parcours est d'arriver a trouver de "bons" minimums locaux, il faut donc en général continuer a trouver un bon trade off entre l'exploration et l'exploitation des points visités.

6.1 Sans erreur

Cf local_search_only_better() dans le fichier simulated_annealing.

Cette première version est la plus "simple". On n'explore pas du tout de nouvelles parties de notre espace de solution et on se contente de choisir de manière greedy la meilleure solution possible à chaque étape. Nous avons largement utilisé cette fonction pour comparer nos version de choix de voisins.

6.2 Avec erreur

Cf local_search_with_errors() dans le fichier simulated_annealing.

Cette fois a chaque tour de boucle on s'autorise le choix d'une mauvaise solution. Ce choix est entierement arbitraire

6.3 Recuit simulé

6.3.1 Présentation basique

6.3.2 Choix de la température

7 Algorithmes biologiques

7.1 Algorithme moléculaire

7.1.1 Choix des