Rapport de projet

Dziga CZAJA

29 décembre 2024

Table des matières

1 Description

Le projet consiste à proposer une solution au problème de l'arbre de Steiner. Étant donné un ensemble de points, il faut trouver un arbre de poids minimal reliant tout les points.

Pour cela, on construit un arbre candidat, c'est à dire un arbre qui relie tout les points. Ensuite, on modifie aléatoirement l'arbre, en rajoutant des points relais entre les points de base, et en modifiant aléatoirement les différents noeuds et points relais.

Nous allons proposer différents algorithmes afin d'essayer de trouver avec le moins de variance possible, un arbre de Steiner.

Sujet.

2 Architecture et chronologie

Le projet est composé de deux modules majeurs. Le module Graph, qui décrit une structure de donnée qui m'a paru la plus adéquate pour le problème posé. Suivi du module Algo, qui pour une question de simplicité est extrait du module graph. Dans cette deuxième partie, on retrouve tout les algorithmes et fonctions nécessaires à la résolution du problème.

Je me suis d'abord penché sur la structure de graphe. Ensuite, une fois que la structure était fonctionelle, s'en est suivi la mise en place des algorithmes. En commençant par l'algorithme naif de hill climbing. S'en est suivi celui de backtracking.

Je me suis alors rendu compte que le résultat dépendait énormément de l'arbre de départ choisi. C'est pour cela que j'ai écrit le troisième algorithme, ou l'on effectue l'algorithme de backtracking sur plusieurs arbres de départ différents.

3 Modules

3.1 Module Graph

La façon d'implémenter les graphes a été choisie grâce à la méthode vue en cours. J'utilise donc une map de noeuds, ou chaque noeud contient un set de clés qui représente ses voisins. A cela on rajoute certains éléments pour gagner en temps d'exécution. Les types graph et node sont donc défini ainsi :

```
module NodeSet : Set.S with type elt = int
module NodeMap : Map.S with type key = int

type node = { is_r : bool; coords: (float*float); succs : NodeSet.t}
type graph = { poids: float; nodes : node NodeMap.t; pts_r: NodeSet.t }
```

Le graphe possède un set, regroupant les clés de tout les noeuds qui sont des noeuds relais. De cette façon, on peut directement utiliser les modules de Set pour récupérer le problème de base d'un graphe, ou encore accéder rapidement aux noeuds concernés par les modifications. Les noeuds possèdent eux aussi un booléen confirmant leur rôle de noeud relais. Cela permet de gagner en temps d'exécution et en temps de confirmation de la nature des noeuds.
De cette façon, certaines informations clés sont disponibles en 0(1), en contrepartie de l'augmentation en taille physique de la structure de données. De plus, la structure de map permet un parcours des noeuds en temps linéaire, et un temps d'accès constant, ce qui est utile dans ce cas, étant donné que les modification du graphe sont sur des zones précises.

Dans le code, l'utilisation des sets n'a surement pas été optimale, étant donné que beaucoup d'algorithmes sont en $0(n)$ avec n le nombre de noeuds ou d'arêtes. Cela aurait potentiellement pu être évité ou optimisé en s'appuyant sur la librairie ocaml des Set, qui présent des algorithmes en $O(log_2 n)$.

3.2 Module Algo

Le module Graph est inclu dans le module algo pour pouvoir récupérer les données des graphes et des noeuds plus rapidement, en ne pas passer par des fonctions auxiliaires, qui alourdiraient le code à mon sens. De plus, la division est fait plus par confort de lisibilité et de clarté, étant donné que la strucutre de données a été faite "sur mesure" pour le problème.

3.3 Visualisation et récupération des données

La visualisation et la récupération des données se fait à l'aide des modules donnés dans le sujet. Une simple fonction a été ajoutée pour pouvoir récupérer les données du problème de départ dans un fichier.

4 Algortihmes proposés

4.1 Hill climbing

On effectue un algorithme naif qui consiste à modifier aléatoirement l'arbre de trois façons différentes, en gardant la modification seulement si elle améliore le poids de l'arbre.

4.2 Backtracking

L'algorithme consiste à essayer toutes les combinaisons possibles de modification de l'arbre candidat, et de choisir le parcours qui offre le meilleur arbre possible.

4.3 Backtracking sur plusieurs candidats

On réitère l'algorithme précédent mais sur plusieurs arbres candidats différents.

On remarque après plusieurs essais, que un des meilleurs moyens pour faire baisser la variance des résultats, est d'augmenter le nombre d'arbres de départ, plutôt que d'augmenter la profondeur

de l'arbre de backtracking. Cela pourraît laisser penser que un des plus gros travails à faire serait de trouver les types d'arbres candidats qui mènent au meilleur résultat.

5 usage

Build :

```
$ make  # for make all
$ make exec # for the main project
$ make test # for some basic tests
```

Requirements : library **graphics**

Usage :

```
$ ./test
$ ./exec [options]
```

exec options :

```
None : launch basis Steiner function
-s <x> <y> : change the visualisation size. default : 800 800
-w : print the weight of the final chosen tree. default : doesn't
-a [1-3] [options]: choose which algorithm to use
    default : -a 3 -n 30 -prof 9 (Steiner function in main)
    algorithms :
    1 : hill_climbing
        options :
            -n <nb> : number of iterations.
    2 : backtracking
        options :
            -d <depth> : depth of the backtracking tree.
    3 : multiple treees backtracking
        options :
            -d <depth> : depth of the backtracking tree.
            -n <nb> : number of beginning trees.
-f <file> : choose the file to import the points used. if not specified, uses stdin to get the in
    You can use -f arbre1.txt as test values.
-o : print the algorithm and parameters used
```