Programmation fonctionnelle

Arbres de Steiner

Timothé Rios

08 janvier 2020

Sommaire

Architecture du projet

```
projet_ipf_s3_Timothé_Rios/
|  docs/
|  |  rapport.pdf
|  bin/
|  |  (...)
|  src/
|  |  display.ml
|  |  eucli.ml
|  |  recti.ml
|  |  test_eucli.ml
|  |  test_recti.ml
|  |  tree.ml
|  Makefile
|  README.md
|  (test_eucli.exe)
|  (test_recti.exe)
```

Les parenthèses (...) représentent les fichiers créés par `make`

1 Introduction

Le but de ce projet est d'implémenter deux algorithmes de recherche d'un arbre de Steiner. Plus précisément, il s'agit de prendre en entrée une liste de points et de retourner un arbre reliant tous ces points et ayant le poids le plus petit possible. Le poids d'un arbre étant le poids de ses arêtes, le poids d'une arête se définit de deux façons différentes : la distance de Manhattan entre ses extrémité ou la distance usuelle entre celles-ci.

Deux exécutables sont donc générés par le Makefile. Les deux lisent sur l'entrée standard un entier sur une ligne, correspondant au nombre de points à relier, puis pour chaque point, son abscisse et son ordonée sur une ligne séparées par un espace. Le premier renvoie un arbre de Steiner pour la distance de Manhattan et le deuxième un arbre de Steiner pour la distance usuelle.

2 Implémentation des arbres

La première étape de ce projet était d'implémenter les arbres sur lesquels il fallait itérer pour générer un arbre de Steiner.

Il fallait itérer sur deux types d'arbres : ceux utilisant la distance de Manhattan, les arbres rectilinéaires, et ceux utilisant la distance usuelle, les arbres euclidiens.

Étant donné que les arbres rectilinéaires ne peuvent avoir que des coordonnées entières, contrairement aux arbres euclidiens qui ont des coordonnées réelles, il fallait aussi utiliser deux type de coordonnées.

2.1 Coordonnées

2.1.1 Signature

Afin d'implémenter les deux types de coordonnées différentes, j'ai commencé par écrire une signature de module Coord.

Celle-ci me permet de définir toutes les opérations à faire sur des coordonnées sans préciser le type auquel appartiennent les deux éléments du couple constituant une coordonnée. Au fur et à mesure que j'avançais dans le projet, j'ai ajouté à ce module les fonctions de manipulation de coordonnées dont j'avais besoin.

```
module type Coord =
sig
  type dot
  val zero : dot
  val ( ++ ) : dot -> dot -> dot
  val ( -- ) : dot -> dot -> dot
  val ( *** ) : dot -> dot -> dot
  val d_abs : dot -> dot
  val distance : dot * dot -> dot * dot -> dot
  val dump_coord : dot * dot -> unit
end
```

Ce module contient le type abstrait `dot`, qui sera par la suite implémenté en `int` ou `float`. À partir de ce type, il est possible de déclarer le `dot zero`, les opérations élémentaires d'addition, de soustraction et de multiplication, la valeur absolue ainsi que la distance entre deux couples de `dot`.

La dernière fonction déclarée, `dump_coord`, sert à imprimer une coordonnée sur la sortie standard.

2.1.2 Implémentation des coordonées

L'étape suivante était d'écrire deux implémentations de module ayant pour signature Coord. Par exemple, pour les coordonées des arbres rectilinéaires :

```
  module Rectilinear_Coord : Coord with type dot = int =
struct
  type dot = int
  type coord = dot * dot

  let zero = 0

  let (++) d1 d2 = d1 + d2
  let (--) d1 d2 = d1 - d2
  let ( *** ) d1 d2 = d1 * d2
  let d_abs d = abs d
```

```
  let distance c1 c2 = match c1 with
  |(x1,y1) -> match c2 with
    |(x2,y2) -> (abs (x1-x2)) + (abs (y1 -y2))

  let dump_coord c = match c with
  (x,y) -> Printf.printf "(%d,%d) " x y
end
```

Le module Rectilinear_Coord implémente donc le type `dot` avec `int` et la fonction `distance` déclarée dans Coord avec la distance de Manhattan.

2.2 Module des arbres

Après avoir implémenté les coordonnées, je suis passé à l'implémentation des arbres. Comme pour les coordonnées, je devais traiter deux types d'arbres différents. J'ai donc décidé d'utiliser un foncteur prenant en argument un module de type `Coord` et retournant un module implémentant les arbres.

```
  module TreeMod :
  functor (X : Coord) ->
    sig
      type dot = X.dot
      type coord = dot * dot
      type tree = Node of bool * coord * tree list
      val ( ++ ) : X.dot -> X.dot -> X.dot
      val ( -- ) : X.dot -> X.dot -> X.dot
      val ( *** ) : X.dot -> X.dot -> X.dot
      val zero : X.dot
      val distance : X.dot * X.dot -> X.dot * X.dot -> X.dot
      val dump_coord : X.dot * X.dot -> unit
      val dump : (X.dot * X.dot) list -> unit
      val dump_coord_tree : tree -> unit
      val dump_coord_tree_list : tree list -> unit
      val isin : 'a -> 'a list -> bool
      val uniq : 'a list -> 'a list
      ...
    end
```

Ce foncteur m'a évité d'avoir à traduire et Ãă recopier toutes les fonctions communes aux deux types d'arbres d'un type à un autre. Toute fonction utile à la fois aux arbres rectilinéaires et aux arbres euclidiens est comprise dans TreeMod.

les arbres sont représentés par le type `tree`. Un arbre est donc constitué d'une coordonée, d'un booléen qui indique si cette coordonée est l'un des points que l'on cherche à relier et d'une liste d'arbres (ses sous-arbres "directs").

3 Arbres rectilinéaires

L'algorithme de génération d'un arbre de Steiner rectilinéaire que j'ai implémenté est celui proposé par le sujet. Il s'agit donc de trouver un arbre candidat en partant du "graphe complet" formé avec toutes les arêtes utiles de la grille composée des points à relier et de leurs points relais.

Une fois qu'un tel arbre est généré, il faut le perturber en lui ajoutant un certain nombre d'arêtes, puis retrouver un arbre acyclique en gardant la connexité avec tous les poinst à relier.

3.1 Graphe complet

D'après le projet, dans le cas rectilinéaire, le nombre de points relais utiles est fini. En prenant l'exemple du sujet : Pour une liste de points de base, le graphe complet associé est

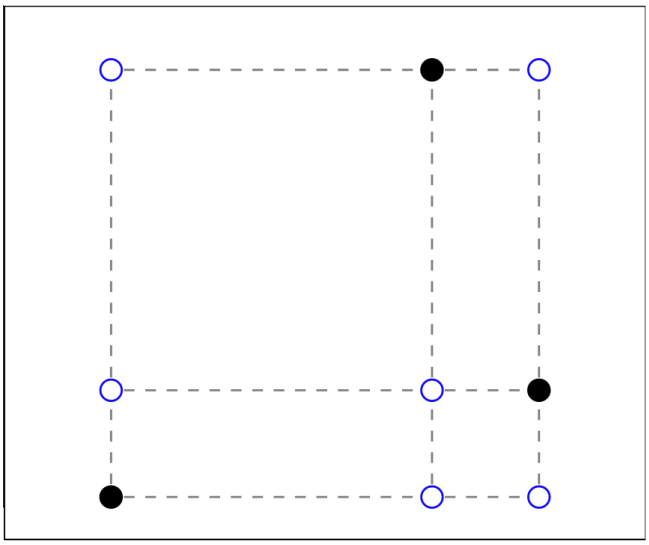

Figure 1: Exemple du sujet

donc l'arbre contenant toutes les arêtes de la grille. La fonction qui s'occupe de créer un graphe complet, `complete_graph`, crée un arbre à partir du point en bas à gauche de la grille et lui ajoute ses points voisins en ne créant que des arêtes qui vont soit vers le haut, soit vers la droite. Quand elle est arrivée au coin en haut à droite, elle a fini.

3.2 Arbres candidats

Lors de la première itération de l'algorithme, on doit générer un arbre candidat à partir du graphe complet.

Pour ce faire, on va supprimer une arête au hasard, puis vérifier la connexité de l'arbre avec les points de départ. S'il n'est plus connexe (avec les points de départ), on annule la suppression d'arête et on réessaye. S'il est connexe, on vérifie s'il contient des cycles, ie si, en le parcourant, on trouve deux sous-arbres avec les mêmes coordonnées. S'il contient encore des cycles, on recommence à supprimer une arête. Sinon, on a notre arbre candidat t_1.

Pour passer d'un arbre candidat t_n au suivant t_{n+1}, on commence par ajouter à t_n un certain nombre d'arêtes. Je détermine ce nombre d'arêtes en fonction du nombre de points possibles.

À partir de n points tous non alignés (ie dont leurs abscisses et leurs ordonnées sont différentes deux à deux), on obtient une grille contenant $2n(n-1)$ points. Le nombre de points -et donc d'arêtes- augmente ainsi proportionnellement au carré du nombre de points de départ non alignés. J'ai donc choisi d'ajouter à chaque étape un nombre d'arête égal à la partie entière de la racine du nombre de points possibles.

Une fois que l'on a ajouté des arêtes à t_n, on obtient t_{n+1} en appliquant le même algorithme que pour obtenir t_1.

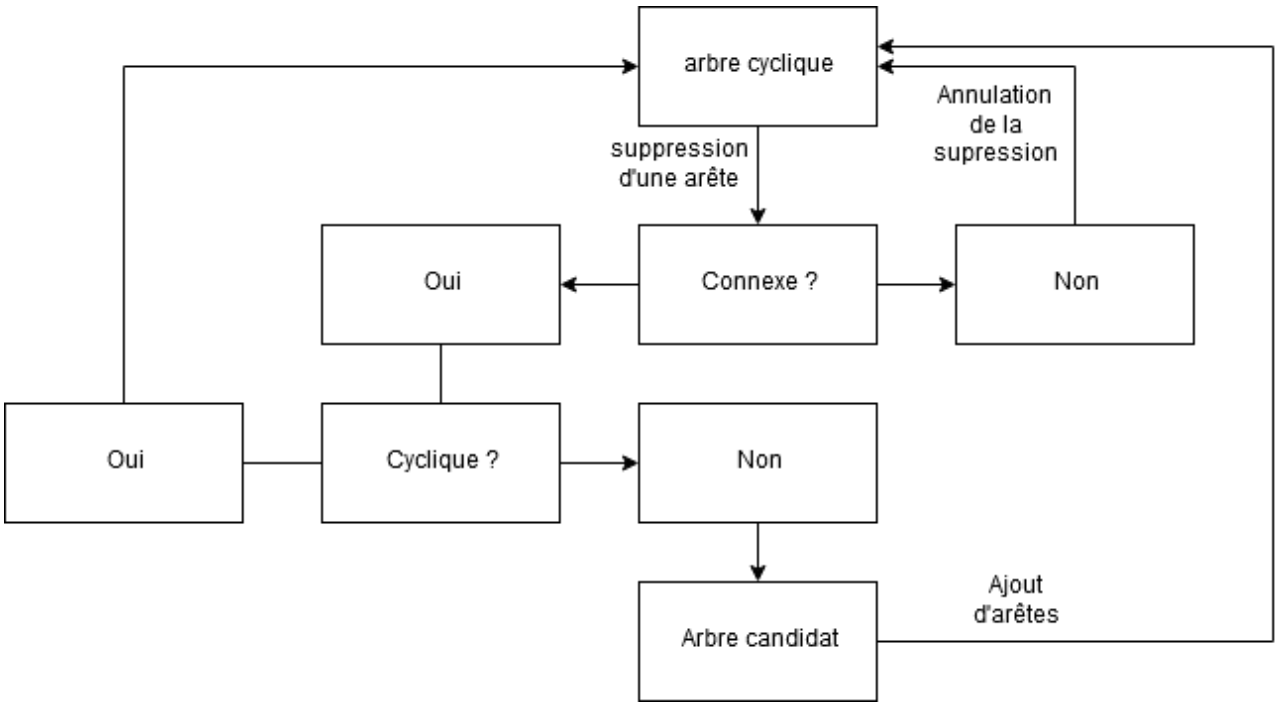

Figure 2: algorithme de génération d'un arbre candidat rectilinéaire

4 Arbres euclidiens

De même que pour les arbres rectilinéaires, l'algorithme que j'ai implémenté pour les arbres euclidiens est celui proposé par le sujet.

4.1 Première itération

Le premier arbre candidat euclidien est plus facile à générer que le premier arbre candidat rectilinéaire. Dans ce cas-ci, il suffit de trouver un arbre reliant tous les points de base entre eux.

Pour cela, on prend au hasard un point parmi les points de base, qui devient la racine de notre arbre t_1. Ensuite, tant que les points de base ne sont pas tous dans t_1, on prend un point de base qui n'est pas dans t_1 au hasard et on l'ajoute à un sous-arbre de t_1 (t_1 inclus), lui aussi pris au hasard. Quand tous les points de la base sont dans t_1 on a notre arbre candidat.

4.2 Altérations

Afin de passer de t_n à t_{n+1}, on effectue sur t_n une modification prise au hasard parmi les quatre proposées par le sujet.

On peut tout d'abord ajouter un point relais à t_n. Pour ce faire, on tire au hasard trois points de t_n p_1,p_2,p_3 tels que p_1 soit relié à p_2 et p_2 à p_3.
Ensuite, on supprime les liaison entre ces trois points avant de tous les relier à un nouveau point relais situé au sein du triangle formé par p_1,p_2 et p_3.

On peut aussi supprimer un point relais : on commence par tirer au hasard un point relais p de t_n, puis trois points de t_n reliés à p. On supprime alors les liaisons entre ces trois points et p avant de relier le premier au deuxième et le deuxième au troisième.

La dernière modification possible pour les points relais est d'en déplacer un. Après avoir tiré au hasard un point relais p de t_n, on tire trois autres points reliés à p, puis on déplace p de façon à ce qu'il soit toujours dans le triangle formé par les trois points tirés.

Enfin, il est possible de modifier la façon dont est relié un point de départ au reste de l'arbre. Pour cela, on tire un point de base b de t_n qui n'a qu'une seule arête. On supprime celle-ci et on en recrée une qui va de b a un point de t_n qui n'est pas celui qui reliait précédemment b.

5 Conclusion

Le code que j'ai produit fonctionne et donne les résultats attendus. Mais avec le recul, je remarque ce qui m'a posé problème et comment j'aurai pu l'éviter.

5.1 Difficultés encontrées

Ce fut lorsque j'ai développé ma fonction générant un graphe complet à partir d'un arbre que j'ai rencontré ma première difficulté.

En effet, mon choix d'implémentation des arbres pose problème dans ce projet qui, en fin de compte, manipule des graphes et en particulier des graphes cycliques. Or, comme l'indique le fait que je parle d'arbres depuis le début, la structure d'arbres que j'utilise possède explicitement une racine et donc une "direction" à la fois des arêtes des arbres et des arbres eux-même.

Manipuler et surtout créer avec cette structure des graphes cycliques demande beaucoup plus de ressources que d'autres implémentations comme la matrice d'adjacence ou d'incidence. De même, trouver trois points pour ajouter ou bouger un point relais pour les arbres euclidiens a demandé beaucoup de travail dans la recherche de l'algorithme en partie à cause du choix d'implémentation.

Il n'empêche que cette implémentation présente aussi des qualités : il est très simple d'ajouter des points à l'arbre, de vérifier la connexité ou la présence de cycles.

5.2 idées d'optimisations

Comme dit dans la partie précédente, je pense que les programmes, et surtout `test_recti.exe` peuvent être plus rapides si l'on change l'implémentation des graphes. Les algorithmes de certaines fonctions en seraient aussi simplifiés.

Il serait possible aussi de calculer la complexité des fonctions en fonction du nombre de points de base, car j'ai pu pensé pour certaines a une implémentation différente après coup. Je pense qu'il est aussi possible de réduire le nombre de parcours de l'arbre pour une génération d'un arbre candidat euclidien ou rectilinéaire en gardant des informations utiles en mémoire.