Projet de programmation fonctionnelle (IPF3)

Jérémie Spiesser

Table des matières

Structures de données utilisées 1
Structure des `Noeuds_euclidiens` 1
Structure des `Graphes_euclidiens` 2
Structure du projet et compilation 3
Fonctions d'itération 3
Points délicats 4
Choix d'un élément aléatoirement dans un set 4
Politique de déplacement aléatoire de points 4
Pistes d'améliorations 4

Il s'agissait dans ce projet de programmation fonctionnelle, de résoudre le problème de l'arbre de Steiner dans deux cas précis : un cas rectilinéaire et euclidien.

Devant l'ampleur du travail a faire, j'ai choisi de me consacrer essentiellement au cas euclidien : c'est donc celui-ci que je vais expliquer içi.

Ce projet a été développé avec ocaml 4.11.1.

Structures de données utilisées

Le problème fait intervenir 3 notions : - la notion de noeud ou de point : il s'agit de points dans l'espace - la notion d'arrête : il s'agit de couples de points représentant une liaison entre deux points - la notion de graphe ou d'arbre : il s'agit avant tout d'un ensemble d'arrêtes

J'ai alors utilisé deux structures de données : une pour représenter les points, une pour représenter les graphes.

Ces deux structures de données sont englobées dans deux modules :

- le premier : Noeud_euclidien
- le deuxieme : Graphe_euclidien qui est obtenu grâce à un foncteur

Structure des `Noeuds_euclidiens`

La structure est la suivante :

```
module Noeud_euclidien =
struct
    module Set_int = Set.Make(Int)
    type t = {
        x: float;
        y: float;
        id: int; (* Index du noeud dans la map des noeuds *)
        voisins: Set_int.t; (* liste des index des noeuds voisins *)
        relai: bool (* true si point relai, false sinon *)
```

```
    }

    (**
    @requires rien
    @ensures retourne un noeud_euclidien avec aucun voisins
    @raises rien
    *)

    let createNoeud x y id relai =

(* suite du code*)
```

La structure du noeud euclidien comprend

- les coordonnées du point (qui sont en réalité des entiers dans le sujet, cependant je ne me suis rendu compte de cette information qu'après avoir fini le module de graphe) ,
- un booléen pour indiquer si le point est un point relai ou non,
- un Set d'entiers correspondant aux voisins du point en question.

La notion de poids et de distance apparaît au module suivant.

Structure des Graphes_euclidiens

```
module Graphe_euclidien(X:Noeud) = struct
  module Map_int = Map.Make(Int)
  type t = {
    poids: float; (* Poids du graphe = somme des poids des arrêtes du graphe*)
    a: X.t Map_int.t; (* map int -> noeud_euclidien contenant tous les points *)
    distance: X.t -> X.t -> float; (* fonction qui calcule la distance entre 2 noeuds *)
    relais : X.Set_int.t;(* Ensemble des indices des points relais du graphe *)
    depart: X.Set_int.t;(* Ensemble des indices des points de départ du graphe*)
    compteur : int ; (* Entier qui se doit d'être >0 et d'être supérieur à l'id maximal présent dans a,
  }

let empty_graph = {
  poids = 0.0;
(* suite du code*)
```

Un graphe euclidien est alors la donnée

- d'un poids (un flottant que l'on conserve afin de ne pas avoir a le recalculer lors des itérations d'améliorations)
- d'une map qui à un indice de point renvoie le point en question
- d'un set d'entiers qui correspond à l'ensemble des points relai du graphe
- d'un autre set d'entiers qui correspond à l'ensemble des points de départ du graphe
- d'un compteur qui sert de référence pour les identifiants des points, lors de leur insertion
- d'une fonction de distance pour calculer la distance entre 2 points (içi il s'agit de la distance euclidienne sur R^2 classique)

Cette structure de donnée nous permet alors

1. De pouvoir tirer facilement un point de départ ou de relai sans faire un filtre sur toute la map avec tous les points
2. Si l'on veut éditer un attribut d'un point (par exemple ses coordonnées), il suffit de le faire dans la map et d'actualiser le poids total du graphe

La signature de X est la suivante :

```
module type Noeud = sig
    module Set_int :
      sig
        type elt = Int.t
        type t = Stdlib__set.Make(Int).t
        val empty : t
        val is_empty : t -> bool
        val mem : elt -> t -> bool
        val add : elt -> t -> t
        val singleton : elt -> t
        (* reste de la signature d'un set d'entiers *)
        (* ... *)
       val add_seq : elt Seq.t -> t -> t
        val of_seq : elt Seq.t -> t
  end
      (* Partie de la signature du type Noeud_euclidien *)
  type t = {
    x : float;
    y : float;
    id : int;
    voisins : Set_int.t;
    relai : bool;
  }
  val createNoeud : float -> float -> int -> 'a -> t
  val addVoisin : t -> Set_int.elt -> t
  val removeVoisin : t -> Set_int.elt -> t
  val memVoisin : t -> Set_int.elt -> bool
  (* Reste de la signature de Noeud_euclidien*)
```

Structure du projet et compilation

Je n'ai pas réussi a compiler "proprement" le projet. En effet, j'obtenais l'erreur : "Error: Unbound module Noeud_euclidien" lorsque j'essayai d'importer un module dans un fichier qui n'était pas celui ou le module était déclaré.

Par conséquent, j'ai lancé mon code avec la commande :

`cat graphe_euclidien.ml noeud_euclidien.ml main.ml > projet.ml ; ocaml < projet.ml`

Ainsi, j'ai mis mon code dans main.ml puis lancé avec l'interpréteur `ocaml` le fichier projet.ml ainsi généré. La fonction `euclidian : (int*int) list -> ((float * float) * (float * float)) list` se trouve dans le module Graphe_euclidien.

Fonctions d'itération

Il y a trois itérations possibles :

- ajouter un point relai : fonction `ajoutRelaiTriangle`
- supprimer un point relais en le fusionnant avec un de ses voisins : fonction `transfoFusionVoisin`
- déplacer un point relai : fonction `transfoDeplacerRandomRelai`

Toutes ces fonctions prennent comme unique argument un graphe_euclidien et renvoient un nouveau graphe_euclidien.

Le choix a la fois de la transformation a effectuer ainsi que le choix du graphe a conserver est fait part la fonction `evolveGraph : int -> Graphe_euclidien.t -> Graphe_euclidien.t`.

Enfin, cette fonction est à nouveau englobée dans la fonction `eulidian`, qui s'occupe a partir de la liste de points passée en argument, de créer un graphe, le remplir de points de départ puis de le faire évoluer.

Points délicats

Choix d'un élément aléatoirement dans un set

L'algorithme que j'ai utilisé pour sélectionner aléatoirement un entier dans un set d'entiers est le suivant :

- prendre la taille du set : j'imagine qu'elle est stockée dans le set donc l'appel a la fonction cardinal de Set.Make doit avec une complexité en O(1)
- tirer un indice aléatoirement dans le set (entre 0 et la taille du set exclue)
- prendre le i ième élément du set récursivement en obtenant puis supprimant le plus petit élément du set (ceci i fois)

C'est cette dernière étape qui est réellement coûteuse car elle implique une recopie du set implicite, et ce i fois avec i entre 0 et le cardinal du set.

Politique de déplacement aléatoire de points

J'ai choisi la politique de déplacement de point suivante :

- pour le point a déplacer, prendre le maximum de sa coordonnée x et y (notons la m)
- déplacer le point dans un carré centré en sa position initiale, de taille 2m

Ceci permet d'éviter de tirer des points situés a l'opposé du graphe et permet une approche plus progressive.

Pistes d'améliorations

- Implémenter le dernière méthode de transformation : relier autrement un point de départ

On pourrait prendre aléatoirement un point de départ (qui ne soit pas un point relai), puis sélectionner un nouveau point a utiliser en faisant un Depth First Search avec son voisin puis les voisins de son voisin.

- Faire en sorte que le projet fonctionne avec une vraie compilation
- Faire fonctionner le module graphique pour avoir une indication visuelle de ce que le programme fait.

Actuellement, il est possible - grâce au fait que l'on utilise l'interpréteur intéractif de ocaml - de lire le poids du graphe ainsi que les points et arrêtes du graphe.

Cependant, c'est très peu pratique.