Algorithme de Kruskal

Jean-Baptiste Rouquier

Pour profiter du canevas de code source, tapez dans une console

```
cp ~jrouquie/sujets/TP9-Kruskal.ml ~/login-TP9.ml
emacs login-TP9.ml
```

1 Rectangles

On pose (`so` (sud-ouest) et `ne` (nord-est) sont les deux coins du rectangle) :

```
type point = {x:int; y:int}
type rect = {so: point; ne: point}
```

Question 1.1. Écrire `distance : point -> point -> int` qui calcule la distance selon la norme infinie.

Question 1.2. Quelle condition doivent vérifier les points `so` et `ne` ? Écrire la fonction `valid_rect : rect -> bool` qui teste cette condition.

Question 1.3 (Manipulation des rectangles). Écrire

```
appartient : point -> rect -> bool
milieu : rect -> point
disjoints : rect -> rect -> bool
```

2 Quadtrees

Un quadtree est une structure de données représentant des points du plan et permettant de trouver rapidement les points d'une région donnée en argument.

Elle consiste à découper le plan en cellules rectangulaires organisées hiérarchiquement. Le plan entier est représenté par une grosse cellule. Chaque cellule qui contient deux points ou plus est subdivisée en quatre sous-cellules de même taille, la subdivision s'arrêtant lorsqu'une cellule contient zéro ou un point.

Le sujet de 1997 de l'école Polytechnique (figure 1) utilisait des quadtrees (il les utilisait pour stocker des «corps», comprendre «points»).

On a donc envie de définir le type suivant :

```
type arbre =
  | Vide
  | Feuille of point
  | Noeud of cellule
and cellule = {
  region: rect;
  fils: arbre array}
```

La région d'une cellule est bien sûr le rectangle qu'elle représente. En fait on va avoir besoin de stocker de l'information sur les points (une «étiquette»), on ajoute donc un type `'a` sur les feuilles :

```
type 'a arbre =
  | Vide
  | Feuille of point * 'a
```

La figure 1 ci-contre montre un exemple de division d'un univers en cellules.

Une représentation structurée de la répartition des corps en cellules et sous-cellules est un arbre dont les nœuds internes ont quatre fils et dont les feuilles contiennent zéro ou un corps. La division en sous-cellules n'est poussée qu'autant que nécessaire, c'est-à-dire que chaque nœud qui a des fils représente une cellule qui contient deux corps ou plus. Un tel arbre est un *quadtree (adaptatif)*.

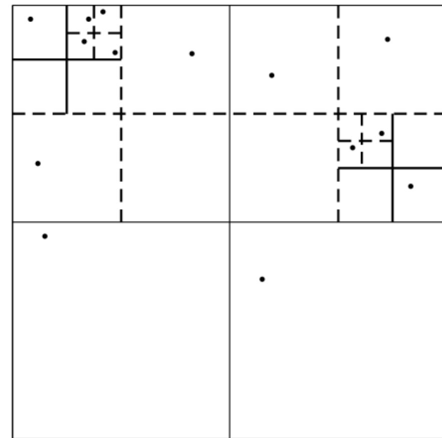


Figure 1

La figure 2 ci-dessous montre la représentation arborescente de l'univers déjà décrit sous forme de schéma sur la figure 1.

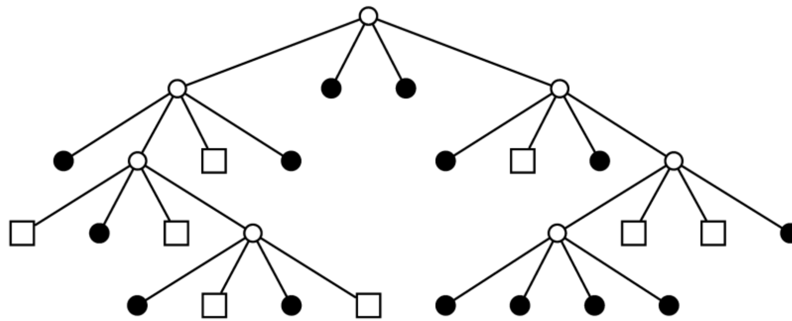


Figure 2

(Dans la figure 2 ci-dessus, les cellules vides sont des carrés et les cellules qui ne contiennent qu'un corps sont des cercles pleins.)

Fig. 1 –

```
  | Noeud of 'a cellule
and 'a cellule = {
  region: rect;
  fils: 'a arbre array}
```

Noter que le type `array` est mutable, ces arbres seront donc modifiés en place.

Question 2.1 (subdivisions). On numérote les régions dans le même ordre que sur la figure :

3	0
2	1

Écrire `sous_region : rect -> int -> rect` qui renvoie la sous-région de l'indice donné.

Écrire `indice_fils : rect -> point -> int` telle que `indice_fils region point` est celle des quatre sous-régions de `region` qui contient `point`.

Question 2.2 (construction). écrire les deux fonctions mutuellement récursives
- `insere_cellule : 'a cellule -> point * 'a -> unit` et
- `insere_arbre : 'a arbre -> rect -> point * 'a -> 'a arbre`

qui ajoutent un point à un quadtree. `insere_cellule` modifie en place la cellule donnée pour lui ajouter le point. `insere_arbre` prend un rectangle en argument : c'est la région recouverte par l'arbre donné. Elle renvoie l'arbre modifié.

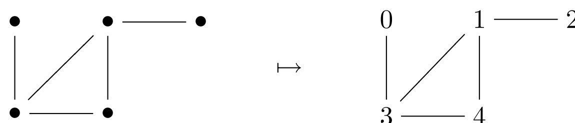
Testez avec le code fourni dans le sujet. (Sélectionnez la fenêtre graphique de Caml et appuyez sur une touche pour faire défiler les 10 tests aléatoires. Laisser la fenêtre Caml se fermer toute seule.)

Question 2.3 (utilisation). Écrire `get_points : rect -> 'a cellule -> (point * 'a) list` qui renvoie les points du quadtree qui appartiennent au rectangle donné. Il ne faut pas parcourir tout l'arbre. On pourra utiliser deux sous-fonctions mutuellement récursives `get_points_cellule` et get_points_arbre.

Remarque. On peut obtenir un découpage du plan qui garantit une profondeur $\log_4 n$ ainsi : on cherche deux droites qui partagent le plan en quatre zones contenant chacune le même nombre de points à un près (montrer qu'il existe deux telles droites), puis on découpe chacune des zones récursivement.

3 Graphes

Informellement, un graphe est un ensemble de points (des *sommets*) dont certains sont reliés par un trait (une *arête*). Pour les distinguer, on numérote les sommets.



Formellement, un graphe non-orienté est un couple (V, E) ou V est l'ensemble des sommets (*vertices*) et $E \subseteq \mathcal{P}_2(V)$ est l'ensemble des arêtes (*edges*). E est un ensemble de paires de sommets.

Deux sommets sont dits voisins s'ils sont reliés par une arête. Une arête est dite adjacente à un sommet si ce sommet est l'une de ses deux extrémités. On peut pour certains problèmes associer un poids (ou coût) à chaque arête, pour ce TP le poids sera simplement sa longueur (en norme euclidienne ou infinie, à votre choix).

La représentation habituelle des graphes se fait par listes d'adjacence : un graphe est représenté par un tableau de longueur $|V|$ dont la case i contient la liste des voisins du sommet i.

Une autre représentation classique mais plus coûteuse en mémoire est la matrice d'adjacence : un graphe est alors représenté par une matrice $M \in \{0,1\}^{|E|\times|E|}$ telle que $m_{i,j} = 1$ si et seulement s'il y a une arête entre i et j.

Nous utiliserons ici la représentation *forward star* qui est proche de la représentation par listes d'adjacence. La seule différence est que toutes les listes d'adjacence sont concaténées dans un tableau. Pour chaque sommet on stocke la position dans ce tableau du début de sa liste d'adjacence. On a donc

```
type sommet = {pos:point; premiere_arete:int}
type arete = {debut:int; fin:int; poids:int}
type graphe = {sommets:sommet array; aretes:arete array}
```

Pour appliquer une fonction `f` à toutes les arêtes adjacentes à un sommet `s`, on peut donc utiliser le code

```
for i = s.premiere_arete to pred graphe.sommets.(succ i).premiere_arete do
  f graphe.aretes.(i)
done;
```

On remarque que ce code ne marcherait pas pour le dernier sommet, on ajoute donc un sommet virtuel au tableau `graphe.sommets`, dont `premiere_arete` est `-1`. Attention donc lorsque l'on veut traiter tous les sommets, il ne faut pas traiter le dernier élément de `graphe.sommets`.

Question 3.1. En utilisant les fonctions fournies par le sujet, écrire `draw_graph : graphe -> unit`.

L'algorithme de Kruskal calcule un arbre couvrant (i.e. un sous-graphe reliant tous les sommets mais ne contenant pas de cycle) de poids minimum. Il consiste à construire la liste L des arêtes triées par poids croissant. On considère alors un graphe G ayant les mêmes sommets que le graphe de départ et initialement aucune arête. Pour chaque arête e de L, l'algorithme l'ajoute à G si et seulement si elle ne crée pas de cycle dans G. On renvoie alors la liste des arêtes retenues.

Pour tester la présence d'un cycle, on utilise la structure *union-find* du TP précédent sur les pointeurs. On stocke dans cette structure les composantes connexes de G. Initialement il y a une composante connexe par sommet puisqu'il n'y a pas d'arêtes. Chaque fois que l'on ajoute une arête, on fusionne les composantes connexes contenant ses extrémités.

Question 3.2 (oralement). Montrer qu'une arête crée un cycle si et seulement si ses deux extrémités sont dans la même composante connexe.

Question 3.3 (correction de l'algorithme, oralement). Soit G le graphe courant à une étape de l'algorithme et e l'arête qu'il s'apprête à considérer (i.e. celle de poids minimal dans L). Si e ne crée pas de cycle dans G, montrer qu'il existe un arbre couvrant de poids minimal, contenant $G \cup \{e\}$.

Ainsi, si le graphe de départ est connexe, l'algorithme construit un ensemble d'arêtes sans cycles et fusionnant toutes les composantes connexes de la structure *union-find*, i.e. un graphe connexe sans cycles, donc un arbre. De plus il existe un arbre couvrant de poids minimal contenant cet arbre, il lui est donc égal.

Question 3.4. Écrire enfin `kruskal : graphe -> arete list` qui renvoie la liste des arêtes de l'arbre couvrant de poids minimal. Testez avec le code donné dans le sujet.

Remarque. Si le graphe n'est pas connexe l'algorithme renvoie un arbre couvrant pour chacune des composante connexes, chaque arbre (et donc la forêt) étant de poids minimal.