math spé option info TP 6

L'algorithme de Dijkstra

Stéphane Legros Année scolaire 2024-2025

Le but de ce TP est de programmer l'algorithme de Dijkstra, qui calcule, dans un graphe orienté pondéré (à masses entières positives), les plus courts chemins d'une origine s_0 fixée à chaque sommet s du graphe. Nous considérons des graphes de la forme $G = (S, A, P)$ où S est un ensemble (fini), A une partie de $S \times S$ ne contenant aucun point de la diagonale et P une application de A dans $\mathbb{N}$: S est l'ensemble des sommets de G, A l'ensemble des arcs de G et chaque arc $a = (s, t) \in A$ est de poids $P(a)$. Nous supposerons dans tout le problème que $S = \{0, 1, \ldots, n-1\}$ et que le graphe G est défini par la donnée d'un vecteur g contenant les listes d'adjacence : pour tout $i \in \{0, \ldots, n-1\}$, $g.(i)$ est une liste contenant tous les couples $(j, P(i, j))$ qui représente les arcs d'origine i.

Pour $s, t \in A$, nous noterons $\mathcal{P}_{s,t}$ l'ensemble des chemins simples (i.e. sans boucle) allant de s à t.

Pour $s \in S$, nous noterons $\mathcal{C}_s$ l'ensemble des sommets t tels que $\mathcal{P}_{s,t} \neq \emptyset$.

Le poids $P(\sigma)$ d'un chemin étant la somme des points de ses arêtes, nous souhaitons calculer, pour s_0 fixé, le vecteur d des distances minimales, défini par :

$$\forall s \in S,\ d.(s) = \begin{cases} \min\{P(\sigma),\ \sigma \in \mathcal{P}_{s_0,s}\} & \text{si } s \in \mathcal{C}_{s_0} \\ \infty & \text{sinon} \end{cases}$$

ainsi qu'un vecteur π définissant une arborescence des plus courts chemins :

$$\forall s \in S,\ \pi.(s) = \begin{cases} t & \text{si } s \in \mathcal{C}_{s_0} \setminus \{s_0\}, \text{ où } t \text{ est le prédécesseur de } s \text{ dans un plus court chemin de } s_0 \text{ à } s \\ -1 & \text{sinon} \end{cases}$$

Pour trouver un plus court chemin de s_0 à $s \in \mathcal{C}_{s_0}$, il suffira donc de remonter dans l'arbre défini par π, puisque le chemin $(s_0, s_1, \ldots, s_k)$ avec $s_k = s$, $s_{k-1} = \pi.(s_k)$, …, $s_0 = \pi(s_1)$ sera un plus court chemin de s_0 à s.

I - Arithmétique dans $\mathbb{Z} \cup \{\infty\}$

Nous modélisons l'ensemble $\mathbb{Z} \cup \{\infty\}$ par le biais du type :

```
type nombre = Infini | N of int;;
```

1) Écrire les fonctions `inférieur: nombre -> nombre -> bool` et `add: nombre -> nombre -> nombre` telles que :

- `inférieur a b` renvoie le booléen `true` si et seulement si $a < b$;
- `add a b` renvoie le nombre $a + b$.

où $<$ et $+$ ont été prolongées à $\mathbb{Z} \cup \{\infty\}$ de façon naturelle.

2) Nous aurons besoin de calculer la position du minimum de trois éléments pour certaines relations d'ordres. Écrire une fonction `match_trois: ('a ->'a -> bool) -> 'a -> 'a -> 'a -> int` telle que si `'a` est un type permettant de représenter un ensemble totalement ordonné A et si `ordre: 'a -> 'a -> bool` est une fonction qui, appliquée à deux éléments $a, b \in A$, renvoie `true` si $a < b$ et `false` sinon, alors l'appel `match_trois ordre a b c`, renvoie :

- 1 si $a \leq b$ et $a \leq c$;
- 2 si $b < a$ et $b \leq c$;
- 3 sinon.

II - Une structure de file d'attente

Considérons un ensemble fini A et une application $d : A \to E$ où E est un ensemble totalement ordonné. Nous souhaitons définir une structure dynamique permettant de stocker les éléments de A de façon à répondre rapidement aux demandes suivantes :

- si A est non vide, calculer un élément a de A tel que $d(a)$ soit minimal, le supprimer de l'ensemble A et le renvoyer ;
- remettre en ordre la structure après avoir diminué une des valeurs $d(a)$, où a est un des éléments de A.

Dans notre problème, A est une partie dynamique de l'ensemble fixé $S = \{0, 1, \ldots, n-1\}$. Nous pourrons donc gérer A grâce à un tas :

- si, à un moment donné du calcul, A contient k éléments $a_0, a_1, \ldots, a_{k-1}$, ceux-ci seront rangés dans les k premières cases d'un vecteur `t` de taille n ;
- pour tout j compris entre 1 et $k-1$, $d(a_i) \leq d(a_j)$ où i est le père de j. Ici, les nœuds étant numérotés de 0 à $k-1$, on rappelle que le fils gauche de i est $2i+1$ (si $2i+1 < k$), son fils droit est $2i+2$ (si $2i+2 < k$) et son père est $\left\lfloor \frac{i-1}{2} \right\rfloor$ (si $i \geq 1$).

Pour chaque a de A, nous devrons être capable de calculer l'indice i tel que $a = a_i$: nous utiliserons pour cela un vecteur `dico` de taille n tel que pour tout $a \in A$, la case `dico.(a)` contient l'indice i tel que $a = a_i$. Nous utiliserons donc le type :

```
type file = {mutable k: int; ordre: int -> int -> bool; t: int array; dico: int array};;
```

où `ordre` est une fonction qui, appliquée à deux élément s et t de A, renvoie `true` si $a < b$ et `false` sinon.

3) Écrire une fonction `echange: file -> int -> int -> unit` telle que l'appel `echange f i j` échange dans la file `f` les contenus des nœuds numéros i et j. On n'oubliera pas de modifier également le vecteur `f.dico`.

4) Écrire une fonction `entasser: file -> int -> unit` telle que l'appel `entasser f i`, entasse le nœud i dans la file d'attente f. On rappelle que si les fils (éventuels) gauche et droit de i respectent la propriété des tas au début du calcul, il faut qu'à la sortie du calcul, l'arbre enraciné en i possède cette propriété et qu'il contienne exactement les même nœuds qu'au début du calcul.

5) Écrire une fonction `remonter: file -> int -> unit` telle que l'appel `remonter f i` remonte la valeur contenue dans le nœud i dans la file d'attente f. Plus précisément, on suppose qu'au début du calcul, tous les couples (a, b) où a est le père de b dans la file f vérifient la propriété de tas, excepté éventuellement quand $i = b$: il faut qu'à la fin du calcul, f possède la propriété de tas et contiennent évidemment les mêmes nœuds qu'au début du calcul.

6) Écrire une fonction `extraire_min: file -> int` telle que l'appel `extraire_min f` supprime le premier élément de la file f et le renvoie.

7) Question supplémentaire : écrire une fonction `faire_tas: -> file -> unit` qui transforme une file quelconque en tas.

III - L'algorithme de Dijkstra

Si g est un graphe, nous aurons besoin de construire :

- le vecteur des distances d, initialisé à $d.(i) = \begin{cases} 0 & \text{si } i = s_0 \\ \infty & \text{sinon} \end{cases}$
- les vecteurs t et *dico* associés au tas initial : il contient initialement tous les sommets du graphe, s_0 étant en première position.

Par exemple, si $g = [|[1, 3; 4, 8]; [3, 1]; [3, 2]; [4, 1]; [2, 1]|]$ et si $s_0 = 2$, nous avons :

$$d = \begin{pmatrix} \infty \\ \infty \\ 0 \\ \infty \\ \infty \end{pmatrix}, \ t = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 3 \\ 4 \end{pmatrix} \text{ et } dico = \begin{pmatrix} 2 \\ 1 \\ 0 \\ 3 \\ 4 \end{pmatrix}.$$

8) Écrire une fonction `initialiser` qui, appliquée à un graphe g et à un sommet s_0, renvoie le triplet $(d, t, dico)$ qui leur est associé.

9) Écrire une fonction `dijkstra` qui, appliquée à un graphe g et à un sommet s_0, renvoie un couple de fonctions $(distance, chemin)$ tel que, pour tout sommet s connecté à s_0, $distance(s)$ est la distance minimale de s_0 à s et $chemin(s) = [s_0; \ldots; s]$ est un chemin optimal de s_0 à s. On utilisera par exemple l'analyse suivante :

- on initialise les vecteurs d, t et *dico* et on crée le tas *tas* ;
- on initialise un vecteur π qui contiendra les pères dans une arborescence de plus courts chemins (au début du calcul, le vecteur ne contient que des -1) ;
- $n - 1$ fois, on extrait le minimum de *tas* et on met à jour la structure en étudiant tous les successeurs de ce minimum ; on pourra pour cela écrire une procédure auxiliaire `met_à_jour` qui, appliquée à un sommet a et à une liste l de successeurs de a, met à jour le vecteur d, le tas et le vecteur des pères (en diminuant les valeur $d.(b)$ pour chaque élément b de la liste l, quand on a trouvé un meilleur chemin pour aller de s_0 à b en passant par a).
- à la fin du calcul, la matrice d contient les distances minimales et le vecteur π les pères dans une arborescence de plus courts chemins : il reste à définir les fonctions *distance* et *chemin* et à les renvoyer.