TP 5 : Graphes pondérés

Fichiers à télécharger. Sur la page web, vous trouverez :

- `fichier_annexe_TP5_ocaml.ml` : un fichier contenant les types utilisés dans le TP et quelques graphes en exemple ;
- `file_priorite_ocaml.ml` : une implémentation de la structure de file de priorité min en Ocaml (via l'utilisation d'un tas min), utile pour la section 2.
- `p083_matrix.ml` une matrice au format ml, de taille 80×80, utile pour la section 3.

Utilisation d'un fichier extérieur. Créer un répertoire associé au TP, et placez-y les fichiers précédents. Dans un autre script, vous pouvez appeler un de ces fichiers via `#use "nom_du_fichier.ml" ;;`.

1 Implémentation de l'algorithme de Floyd-Warshall

On utilise les types suivants :

```
type zbar=Z of int | Inf ;;
type graphe_pondere_dense = zbar array array ;;
```

Le but est d'implémenter l'algorithme de Floyd-Warshall rappelé ci-après.

Algorithme 1 : Algorithme de Floyd-Warshall

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par sa matrice d'adjacence M
Sortie : La matrice $(\delta(i,j))_{0 \leq i,j < n}$ des plus courtes distances entre deux sommets quelconques du graphe
$A \leftarrow \text{copie}(M)$;
pour *tout* k *entre 0 et* $n-1$ *faire* **faire**
  pour *tout* i *entre 0 et* $n-1$ *faire* **faire**
    pour *tout* j *entre 0 et* $n-1$ *faire* **faire**
      $a_{i,j} \leftarrow \min(a_{i,j}, a_{i,k} + a_{k,j})$
Renvoyer A

Question 1. *Quelques fonctions utilitaires*. Pour implémenter l'algorithme, on propose les petites fonctions suivantes. Codez-les !

- `entier : zbar -> int`, prend en entrée un élément de `zbar` différent de `Inf` et renvoie l'entier associé. On renverra une erreur si `Inf` est passé en argument.
- `inf_strict : zbar -> zbar -> bool`. Avec `a` et `b` deux éléments de $\mathbb{Z} \cup \{\infty\}$, `infstrict a b` teste si $a < b$. Naturellement tout entier est strictement inférieur à l'infini, et `inf_strict Inf Inf` doit s'évaluer en `false`.
- `addition : zbar -> zbar -> zbar` réalise l'addition dans $\mathbb{Z} \cup \{\infty\}$. Naturellement $a + \infty = \infty + a = \infty$ quel que soit a.
- `copie_matrice : 'a array array -> 'a array array` renvoie une copie de la matrice passée en argument. Attention à bien copier *physiquement* les éléments. On pourra utiliser `Array.copy` sur chacune des lignes, ou simplement recopier tous les éléments vers une nouvelle matrice.

Question 2. Implémentez l'algorithme de Floyd Warshall.

```
#floyd_warshall g1 ;;
- : zbar array array =
 [|[|Z 0; Z 1; Z -3; Z 2; Z -4|]; [|Z 3; Z 0; Z -4; Z 1; Z -1|];
   [|Z 7; Z 4; Z 0; Z 5; Z 3|]; [|Z 2; Z -1; Z -5; Z 0; Z -2|];
   [|Z 8; Z 5; Z 1; Z 6; Z 0|]|]
#floyd_warshall gneg ;;
- : zbar array array =
 [|[|Z -1; Z 1; Z -2; Inf; Inf|]; [|Z -3; Z -1; Z -4; Inf; Inf|];
   [|Z -1; Z 1; Z -2; Inf; Inf|]; [|Z 1; Z 3; Z 0; Z 0; Z 5|];
   [|Z -3; Z -1; Z -4; Inf; Z 0|]|]
```

Question 3. Modifiez l'algorithme pour tester l'existence d'un circuit de poids total strictement négatif.

```
#floyd_warshall gneg ;;
Uncaught exception: Failure "cycle de poids strictement negatif"
```

Rappel : s'il y a un tel circuit, l'un des coefficients diagonaux de la matrice donnée par l'algorithme de Floyd-Warshall est strictement négatif.

Question 4. Modifier l'algorithme pour calculer en même temps la *matrice de liaison* $\Pi = (\pi_{i,j})_{0 \leq i,j < n}$, définie par :

$$\pi_{i,j} = \begin{cases} \text{le prédecesseur de } j \text{ dans un plus court chemin de } i \text{ à } j, \text{ s'il en existe un} \\ -1 \text{ sinon (on pourrait mettre autre chose)} \end{cases}$$

```
#floyd_warshall g1 ;;
- : zbar array array * int array array =
 [|[|Z 0; Z 1; Z -3; Z 2; Z -4|]; [|Z 3; Z 0; Z -4; Z 1; Z -1|];
   [|Z 7; Z 4; Z 0; Z 5; Z 3|]; [|Z 2; Z -1; Z -5; Z 0; Z -2|];
   [|Z 8; Z 5; Z 1; Z 6; Z 0|]|],
 [|[|0; 2; 3; 4; 0|]; [|3; 1; 3; 1; 0|]; [|3; 2; 2; 1; 0|];
   [|3; 2; 3; 3; 0|]; [|3; 2; 3; 4; 4|]|]
# floyd_warshall g2 ;;
- : zbar array array * int array array =
([|[|Z 0; Z 6; Inf; Z 3; Z (-1); Inf|];
   [|Z (-1); Z 0; Inf; Z 2; Z (-2); Inf|]; [|Z 1; Z 2; Z 0; Z 4; Z 0; Z 6|];
   [|Z (-3); Z 3; Inf; Z 0; Z (-4); Inf|]; [|Z 1; Z 7; Inf; Z 4; Z 0; Inf|];
   [|Z (-3); Z (-2); Z (-4); Z 0; Z (-4); Z 0|]|],
 [|[|0; 4; -1; 4; 0; -1|]; [|3; 1; -1; 1; 0; -1|]; [|3; 2; 2; 1; 0; 2|];
   [|3; 4; -1; 3; 0; -1|]; [|3; 4; -1; 4; 4; -1|]; [|3; 2; 5; 1; 0; 5|]|])
```

Rappel : pour ce faire, il suffit d'intialiser la matrice Π comme suit : $\pi_{i,j} = \begin{cases} i \text{ si } (i,j) \in E \\ -1 \text{ sinon} \end{cases}$. Lorsque dans l'algorithme, on a $a_{i,j} > a_{i,k} + a_{k,j}$, on réalise l'affectation $a_{i,j} \leftarrow a_{i,k} + a_{k,j}$ et parallèlement $\pi_{i,j} \leftarrow \pi_{k,j}$.

Question 5. Écrire une fonction `impression pi i j` d'impression à l'écran d'un plus court chemin entre i et j s'il en existe un, avec `pi` la matrice de liaison.

```
#let _,pi = floyd_warshall g1 in imprimer_chemin pi 0 1 ;;
0 4 3 2 1
- : unit = ()
#let _,pi = floyd_warshall g2 in imprimer_chemin pi 0 2 ;;
Exception: Failure "pas de chemin !".
```

On utilisera `print_string` et `print_int`. Il est nécessaire de construire d'abord la liste des sommets avant d'imprimer.

2 Implémentation de l'algorithme de Dijkstra

Les graphes en représentation creuse sont donnés par le type `(int * int) list array`, un couple (u, ω) dans la liste d'adjacence d'un somme t signifie qu'il y a un arc (t, u) dans le graphe, de poids ω. L'algorithme de Dijkstra rappelé ci-dessous permet de calculer les plus courts chemins depuis une origine donnée, pour un graphe dont les arcs ont des poids positifs.

Vous trouverez sur le site web une (longue) implémentation en Ocaml des files de priorité (min), permettant de gérer des couples (e, p) où e est un élément et p sa priorité. Les éléments présents doivent tous être distincts. Les fonctions sont les suivantes :

Algorithme 2 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par listes d'adjacences, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s
Sortie : Les distances $\delta(s, t)$ pour tout $t \in V$
$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;
$E \leftarrow \emptyset$; $F \leftarrow \{s\}$;
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F;
  pour *tout voisin* v *de* u **faire**
    si v *n'est ni dans* F *ni dans* E **alors**
      Ajouter v à F
    $d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$
  Ajouter u à E
Renvoyer d

— `PrioQueue.create : unit -> 'a t` crée une file de priorité vide ;
— `PrioQueue.is_empty : 'a t -> bool` teste si une file de priorité est vide ;
— `PrioQueue.add 'a t -> 'a -> int -> unit`. L'appel `PrioQueue.add f e p` rajoute à la file `f` l'élément `e` avec priorité `p`. Si `e` est déja présent, l'exception `AlreadyMemberInPrioQueue` est levée.
— `PrioQueue.remove_prio : 'a t -> 'a`, supprime et renvoie l'élément prioritaire de la file (celui dont la priorité est la plus petite). Lève l'exception `PrioQueue_is_empty` si la file est vide.
— `PrioQueue.change_prio : 'a t -> 'a -> int -> unit` permet de changer la priorité d'un élément. Lève l'exception `NotMemberInPrioQueue` si l'élément n'est pas présent.
— `PrioQueue.mem 'a t -> 'a -> bool` permet de tester la présence d'un élément dans la file de priorité.

Les opérations sont en $O(1)$, sauf les opérations classiques [1] de files de priorité qui s'exécutent en temps $O(\log n)$. (n étant le nombre d'éléments présents).

Question 6. Implémentez l'algorithme de Dijkstra. La fonction `dijkstra g s` doit renvoyer un tableau de distances de type `int array` contenant les distances entre s et tous les nœuds accessibles du graphe, et -1 pour les non accessibles. (Comme les poids sont positifs, il n'y a pas ambiguïté. Pas besoin de travailler dans `zbar` !).

```
#dijkstra g3 0;;
- : int array = [|0; 8; 9; 7; 5|]
#dijkstra g4 0 ;;
- : int array = [|0; 5; 1; 8; 3; 6|]
#dijkstra g4 3 ;;
- : int array = [|-1; -1; -1; 0; -1; -1|]
```

3 Project-Euler 83

Dans la matrice ci-dessous, la somme minimale d'un chemin entre le coin en haut à gauche et le coin en bas à droite est 2297. Un trajet correspondant est indiqué en gras (naturellement, on ne peut se déplacer qu'horizontalement et verticalement, pas en diagonale).

131	673	**234**	**103**	**18**
201	**96**	**342**	965	**150**
630	803	746	**422**	**111**
537	699	497	**121**	956
805	732	524	**37**	**331**

Sur mon site web, vous trouverez une matrice `m80` au format ml de taille 80×80. Déterminez la somme minimale d'un chemin entre le coin en haut à gauche et le coin en bas à droite, et résolvez ainsi le problème 83 du site Project-Euler [2]. Indication : écrire une fonction `sol_pb_euler m` prenant en entrée une telle matrice carrée, construisant le

1. On a recourt dans l'implémentation à des tables de hachages et à une structure de tableau redimensionnable, ces complexités sont donc amorties et sous réserve de hachage uniforme...
2. `https://projecteuler.net/problem=83`

graphe d'ordre n^2 associé (il est très creux !), et lançant l'algorithme de Dijkstra depuis le sommet associé à la case $(0, 0)$. Ne pas oublier de rajouter le coefficient en haut à gauche à votre distance.

```
# sol_pb_euler m5 ;;
- : int = 2297
# sol_pb_euler m80 ;;
- : int = ... à vous de trouver !
```

Vous pouvez aussi résoudre les problèmes 81 et 82. Ils sont en fait plus simples et une solution basée sur la programmation dynamique est possible, ce qui évite le recours aux graphes. C'est un bon exercice de révision.