Informatique MP
Parcours de graphe

Antoine MOTEAU
Lycée Pothier – Orléans – Option Informatique MP
antoine.moteau@wanadoo.fr

. . . /Parcou-C.tex (2003) compilé le lundi 05 mars 2018 à 17h 34m 10s avec LaTeX

. . . /Parcou-C.tex Compilé le lundi 05 mars 2018 à 17h 34m 10s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Ordre du TP : début d'année (TP n^o 1)

Eléments utilisés :

- listes
- vecteurs, matrices
- récursivité

Documents relatifs au TP :

- Texte du TP : .tex, .dvi, .pdf
- Squelette de programme Caml : xxxx-S.ml
- Programme corrigé Caml : xxxx-C.ml

Antoine MOTEAU – Lycée POTHIER – Option Informatique MP
TP n^o

GRAPHES FINIS SIMPLES

Graphes définis par listes d'adjacence : parcours en largeur, en profondeur, tri topologique
Graphes définis par matrice d'adjacence : plus court chemin pondéré pour tous couples de sommets

1 Introduction

Les graphes ont de nombreuses applications (préparation et suivi de chantiers, organisation ou analyse de dépendance entre processus, etc ...).
On suppose que le vocabulaire (imagé) sur les graphes est connu (voir lexique en fin de document).

1.1 Définitions (rappels)

1.1.1 Arcs, arêtes

Dans un graphe dont l'ensemble des sommets est S,

- un ***arc*** est un couple de sommets (u,v), parcouru uniquement dans le sens de u vers v.
  Lorsque $u=v$, l'arc (u,u) est qualifié de ***boucle***.
- une ***arête*** est un ensemble de deux sommets $\{u,v\}$, parcourue indifféremment dans le sens u vers v ou v vers u.
  Lorsque $u=v$, l'arête $\{u,u\}$ $(=\{u\})$ est qualifié de ***boucle***.

Les arcs ou les arêtes peuvent être ***pondérés*** (ou ***valués***), munis d'une valeur : *pondération* (ou *valuation*).
Un graphe dont les arcs ou les arêtes sont pondérées est un ***graphe pondéré*** (ou ***graphe valué***).

1.1.2 Graphes simples ou multiples

Un graphe est dit ***simple*** lorsque, pour tout couple de sommets u et v, il existe au plus un arc ou une arête joignant u à v ; sinon, le graphe est un ***multi-graphe*** (ou graphe multiple) et il faut numéroter les arcs ou arêtes pour les différencier.

1.1.3 Graphes orientés ou non

- Un graphe ***orienté*** est défini par l'ensemble S de ses sommets et l'ensemble A de ses arcs.
- Un graphe ***non orienté*** est défini par l'ensemble S de ses sommets et l'ensemble A de ses arêtes.

Ici, on ne considérera que des graphes **simples**, orientés et pondérés par des entiers [a].

- Un graphe non pondéré sera pondéré artificiellement par la valeur 1.
- Un graphe non orienté sera considéré [b] comme un graphe orienté symétrique ("bi-orienté"), tel que, pour tout arc allant de u à v $(u \neq v)$, il existe un arc similaire, de même pondération, allant de v à u (redondance).

a. les algorithmes sont facilement adaptables à d'autres types de pondération.
b. les algorithmes, écrits pour des graphes simples orientés, fonctionneront aussi avec les graphes simples non orientés.

1.2 Codage des graphes finis

Les n sommets d'un graphe fini seront, en principe, numérotés de façon consécutive à partir de 1 (indicés de 1 à n), la description d'un sommet se réduisant le plus souvent au numéro de ce sommet (mais on peut avoir une information "externe", qui associe au numéro de sommet une description, par exemple : numéro de département $\longrightarrow$ nom du département, nom de la préfecture, etc...).

Si les sommets d'un graphe ne sont pas des numéros, ou ne sont pas numérotés de façon consécutive, on pourra, si besoin, transformer le graphe en numérotant ses sommets de façon consécutive à partir de l'entier k, $k=1$ par exemple (il serait alors judicieux de garder une correspondance entre l'ancienne description et le numéro dans le nouveau graphe).

1.2.1 Codage par liste d'adjacence

Un graphe peut se coder par la liste des couples (s,Ls) où Ls est la *liste d'ajacence* du sommet s, c'est à dire :

graphe simple, non pondéré : liste des sommets u tels qu'il existe un arc de s à u.

graphe simple, pondéré : liste des couples (u,p) tels qu'il existe un arc de s à u pondéré par p,

graphe multiple, non pondéré : liste des couples (u,k) tels qu'il existe un arc de s à u de numéro k,

graphe multiple, pondéré : liste des triplés (u,k,p) tels qu'il existe un arc de s à u de numéro k et pondéré par p.

Remarque. Pour un graphe non orienté, codé comme un graphe orienté symétrique,, si le sommet v apparaît dans la liste d'adjacence de u alors u apparaît aussi dans la liste d'adjacence de v, avec la même pondération.

1.2.2 Codage par matrice d'adjacence

Un graphe simple, à n sommets **numérotés de 1 à n**, assimilé à un graphe orienté, peut se coder par sa *matrice d'adjacence* : matrice carrée indicée par les numéros de sommets, l'élément d'indices (u, v), dans cet ordre, contenant

- 1 ou (0 / ∞) selon qu'il existe ou non un arc allant de u à v (pondération artificielle).
- la pondération de l'arc allant de u à v, dans le cas où les arcs sont pondérés.
  L'absence d'arc pouvant être repérée par une valeur spéciale de la pondération (par exemple ∞).

Remarque. Pour un graphe non orienté, codé comme un graphe (bi-)orienté symétrique, la matrice d'adjacence est symétrique (on pourrait n'en stocker que la moitié supérieure).

Remarque. Dans le cas d'un multi-graphe, la matrice d'adjacence contiendrait des listes de numéros d'arcs, éventuellement pondérés.

1.2.3 Comparaison des deux méthodes de codage

- Le codage par listes d'adjacence est plus économique en taille que le codage par matrice d'adjacence.
- L'utilisation de listes d'adjacence, qui conduit à des parcours de listes, est plus coûteuse en temps d'exécution que le simple accès direct aux éléments d'une matrice d'adjacence.
- Les listes d'adjacence se prêtent plus facilement à l'ajout ou à la suppression de sommets et d'arcs que le codage par matrice d'adjacence.
- ...

Certains algorithmes sont plus adaptés à un codage qu'à l'autre (on sera donc amené à introduire des méthodes de conversion d'un codage à l'autre).

2 Ressources OCaml

2.1 Quelques fonctions utiles de OCaml

2.1.1 Module List (extrait succint)

`val assoc : 'a -> ('a * 'b) list -> 'b`
`List.assoc u g` renvoie le second élément v du premier couple (u, v) rencontré dans la liste de couples g.
Raise Not_found if there is no value associated with u in the list g.

`val map : ('a -> 'b) -> 'a list -> 'b list`
`List.map f g` renvoie la liste des "f s" pour les éléments s successifs de la liste g.

`val iter : ('a -> unit) -> 'a list -> unit`
`List.iter f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`. It is equivalent to
`begin f a1; f a2; ...; f an; () end`.

`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

`val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
`List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.

Autres : `List.length ; List.hd ; List.tl ; List.mem ; List.rev ; ...`
Exemples :

```
List.assoc 3 [1,[2;5;8] ; 2,[] ; 3,[7;2;1;4] ; 4,[2;3] ; 5,[1] ; 6,[] ; 7,[4;1;2]; 8,[3;1] ];;
- : int list = [7; 2; 1; 4]
List.map (function x -> x*x) [1;2;3;4;5;6;7];;
- : int list = [1; 4; 9; 16; 25; 36; 49]
```

2.1.2 Module Array (extrait succint)

`Array.make n a` renvoie un vecteur indicé de 0 à n-1, dont toutes les composantes contiennent a.

`Array.make_matrix p q a` renvoie une matrice, dont les indices (i,j) vérifient $0 \leqslant i \leqslant$ p-1 et $0 \leqslant j \leqslant$ q-1, tous les éléments d'indice (i,j) étant initialisé à la valeur a.

`Array.make_matrix p q a` n'est pas équivalent à `Array.make p (Array.make q a)`

`Array.map f v` renvoie le vecteur des "f s" pour les éléments s successifs du vecteur v.

Autres ; `Array.length ; Array.to_list ; Array.of_list ; ...`
Exemple :

```
Array.map (fun x -> x,x+1) [|1;2;3;4;5;6;7|];;
- : (int * int) vect = [|1, 2; 2, 3; 3, 4; 4, 5; 5, 6; 6, 7; 7, 8|]
```

2.2 Files FIFO (module Queue de OCaml)

The module Queue implements queues (FIFOs), with in-place modification.

`type 'a t`

The type of queues containing elements of type 'a.

`exception Empty`

Raised when take is applied to an empty queue.

`val create : unit -> 'a t`

Return a new queue, initially empty.

`value add : 'a -> 'a t -> unit`

add x q adds the element x at the end of the queue q.

`value take : 'a t -> 'a`

take q removes and returns the first element in queue q, or raises Empty if the queue is empty.

`value peek : 'a t -> 'a`

peek q returns the first element in queue q, without removing it from the queue, or raises Empty if the queue is empty.

`value length : 'a t -> int`

Return the number of elements in a queue.

les autres fonctions, `clear` , `iter`, `...`, ne seront pas utilisées ici.

Exemples d'utilisation et de syntaxe (extraits, avec une File d'entiers) :

```
open Queue;;                                (* ouverture de la bibliothèque *)
...
let q = Queue.create () in Queue.add s q;   (* création puis introduction d'un élément en tête *)
if Queue.length q > 0                       (* length est utilisée comme prédicat *)
then let u = Queue.peek q                   (* copie de la tête de q; q inchangé  *)
     in ... ;
...
try let u = Queue.take q                    (* u est extrait de la tête de q *)
    in ...
with Empty -> ...;
Queue.add v q;
couleur.(Queue.take q) <- 2;                (* utilisation immédiate de l'élément extrait *)
```

2.3 Tri de listes

2.3.1 Ecriture d'un tri (tri "rapide")

Le tri rapide n'est pas le meilleur tri, mais il est simple à écrire et bon en temps moyen ($O(n \log n)$).

`tri_rapide_list : ('a -> 'a -> bool) -> 'a list -> 'a list`

`tri_rapide_list ordre p`, où ordre est un ordre sur l'ensemble des éléments de p, renvoie une liste dont les éléments sont ceux de p, rangés dans l'ordre précisé.

Exemple : `tri_rapide_list (fun x y -> x < y) [7;5;8;0;4;9;3];;` $\longrightarrow$ `[0;3;4;5;7;8;9]`

Remarque. En Caml, l'ordre < est l'ordre générique, naturel, sur le type des données auquel on l'applique.
Pour un type couple, < est l'ordre lexicographique déduit des ordres naturels sur les éléments du couple.

2.3.2 Fonction List.sort de OCam

Le module List de OCaml introduit une fonction de tri de liste :

`val sort : ('a -> 'a -> int) -> 'a list -> 'a list`

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, compare is a suitable comparison function. The resulting list is sorted in increasing order. List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

3 Fonctions élémentaires (graphes simples)

3.1 Graphes codés par listes d'adjacence

Un graphe simple, orienté, pondéré, sera représenté par une liste de couples (sommet, liste d'adjacence du sommet),

- Les sommets étant (en principe) des entiers,
- La liste d'adjacence d'un sommet u étant la liste des couples (v, p) ou v est l'extrémité de l'arc d'origine u (arc $u \longrightarrow v$) et p la pondération de cet arc.

La représentation par listes d'adjacence est compacte et se prête bien à la création d'un graphe par ajouts successifs d'arcs ou de sommets.

3.1.1 Ajout de sommets, d'arcs, d'arêtes

Un élément ne sera ajouté que s'il n'existe pas déjà dans le graphe (unicité !).

`add_sommet : 'a -> ('a * 'b list) list -> ('a * 'b list) list`
`add_sommet s g` renvoie le graphe déduit de g par l'ajout du sommet s, avec une liste d'adjacence vide si s n'était pas déjà dans g.

`add_arc : 'a * 'a * 'b -> ('a * ('a * 'b) list) list -> ('a * ('a * 'b) list) list`
`add_arc (u,v,p) g` renvoie le graphe déduit de g par l'ajout de l'arc (u,v), pondéré par p.
Les sommets u ou v sont crées s'il ne sont pas déjà dans le graphe.

`add_arete : 'a * 'a * 'b -> ('a * ('a * 'b) list) list -> ('a * ('a * 'b) list) list`
`add_arete (u,v,p) g` renvoie le graphe déduit de g par l'ajout des deux arcs (u,v,p) et (v,u,p).
Les sommets u ou v sont crées s'il ne sont pas déjà dans le graphe.

On pourra itérer ces processus (avec `fold_left` ou avec `fold_right`) pour adjoindre, à un graphe, une liste de nouveaux sommets ou arcs et disposer ainsi d'utilitaires confortables pour la création de graphes.

Exemple 3.1. Construction d'un graphe, à partir d'une liste de sommets et/ou d'une liste d'arcs :

```
let g = List.fold_right add_sommet [0 ; 2 ; 3 ;  1 ;  5 ;  4] [];;
let gg = List.fold_right add_arc [(2,4,1) ; (2,3,1) ; (3,1,1) ; (5,4,1)] g;;
```

Exemple 3.2. Exemple de tri pour un graphe, selon deux ordres :

```
tri_rapide_list (fun x y -> x < y)
  (List.fold_left (fun h (s,ls) -> (s,tri_rapide_list (fun x y -> x > y) ls)::h) [] gg);;
```

renvoie un graphe identique à g, où,

- dans la liste principale, les sommets (premier élément du couple) sont en ordre croissant
- dans les listes d'adjacence, les sommets extrémités d'arcs sont en ordre décroissant.

3.1.2 Suppression de sommets, d'arcs, d'arêtes

`remove_sommet : 'a -> ('a * ('a * 'b) list) list -> ('a * ('a * 'b) list) list`
`remove_sommet u g` renvoie le graphe déduit de g par suppression du sommet u et de tous les arcs dont une des extrémités est u.

`remove_arc : 'a * 'b -> ('a * ('b * 'c) list) list -> ('a * ('b * 'c) list) list`
`remove_arc (u,v) g` renvoie le graphe déduit de g par suppression de l'arc (u,v,p) d'origine u et d'extrémité v.
Les sommets u et v, même s'ils deviennent isolés, ne sont pas supprimés.

`remove_arete : 'a * 'a -> ('a * ('a * 'b) list) list -> ('a * ('a * 'b) list) list`
`remove_arete (u,v) g` renvoie le graphe déduit de g par suppression de l'arête d'extrémités u et v (arcs (u,v,p) et (v,u,p)). Les sommets u et v, même s'ils deviennent isolés, ne sont pas supprimés.

3.1.3 Contenus

La liste d'adjacence d'un sommet s du graphe g s'obtient par `List.assoc s g` .

`sommets : ('a * 'b) list -> 'a list`
`sommets g` renvoie la liste de tous les sommets de g .

`arcs : ('a * ('b * 'c) list) list -> ('a * 'b * 'c) list`
`arcs g` renvoie la liste de tous les arcs de g. Dans la liste obtenue, un arc est représenté comme un triplet (sommet origine, sommet extrémité, pondération).

`aretes : ('a * ('a * 'b) list) list -> ('a * 'a * 'b) list`
`aretes g` renvoie la liste de tous les arêtes de g. Dans la liste obtenue, un arête est représentée comme un triplet (u,v,p), et un seul des arcs (u,v,p) ou (v,u,p) sera présent dans la liste,

3.1.4 Numérotation ou renumérotation

Les sommets d'un graphe pourraient ne pas être des numéros ou ne pas être numérotés de façon consécutive (par exemple, après suppression de sommets).

```
renum : int -> ('a * ('a * 'b) list) list -> (int * (int * 'b) list) list
```
`renum k g` renvoie un graphe identique à g, où les sommets sont transformés en leur numéro de rencontre dans la liste principale, pris à partir de k (k compris).

```
max_sommet : (int * 'a) list -> int
min_sommet : (int * 'a) list -> int
```
`max_sommet g` / `min_sommet g`, où g est un graphe à sommets entiers, renvoie le plus grand / petit (numéro de) sommet de g.

3.2 Graphes codés par matrice d'adjacence

Les sommets, ***numérotés de façon consécutive, à partir de*** 1, sont identifiés à leur numéro d'ordre.
Le graphe est représenté par la matrice **bordée** W, telle que

- pour $i = 1,\dots,n$, $W_{0,i} = W_{i,0} = i$ (bordure par les numéros de sommets : confort de lecture),
- pour $i, j \in \{1,\dots,n\}$, $\begin{cases} W_{i,j} = p, \text{ pondération de l'arc } (i,j) & (\text{arc } i \longrightarrow j) \\ W_{i,j} = \infty \text{ s'il n'y a pas d'arc de } i \text{ vers } j. \end{cases}$

Remarque. La représentation par matrice d'adjacence permet de parcourir facilement le graphe à l'envers (de l'extrémité des chemins vers leur origine).

3.2.1 Exemples

1. Graphe non orienté, pondéré artificiellement (0 marque l'absence d'arc et 1 la présence d'arc)

```
let W1 = [| [| 0;1;2;3;4 |];
            [| 1;0;1;0;0|]; [| 2;1;0;1;1 |]; [| 3;0;1;0;1 |]; [| 4;0;1;1;0 |] |];;
```

$$W1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{pmatrix} \quad \text{(matrice \textbf{bordée} symétrique)}$$

2. Graphe orienté, pondéré en entiers relatifs (`max_int` marque l'absence d'arc)

```
let W2 = [| [| 0; 1; 2; 3; 4 |];
            [| 1; max_int;-10;-2;4|]; [| 2; 2;0;4;max_int |];
            [| 3; 1;-1;-4;2 |]; [| 4; 0;1;max_int;3 |] |];;
```

$$W2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & max\_int & -10 & -2 & 4 \\ 2 & 2 & 0 & 4 & max\_int \\ 3 & 1 & -1 & -4 & 2 \\ 4 & 0 & 1 & max\_int & 3 \end{pmatrix} \quad \text{(matrice \textbf{bordée})}$$

3.2.2 Utilitaires élémentaires

```
print_int_matrix : int array array -> unit                (* sens naturel *)
copy_matrix : '_a array array -> '_a array array          (* sens naturel *)
```

Remarques.

- ici, `copy_matrix` est de type faible, en attente d'instanciation : le premier appel fixera le type.
- Si `m1` est une matrice, `let m2 = m1` ne fait pas une nouvelle matrice, mais fait partager à `m1` et à `m2` les mêmes éléments !

3.3 Conversions de codage

On suppose ici que les sommets sont numérotés de façon concécutive, à partir de 1, les arcs étant pondérés par des entiers.

```
madj_of_ladj : (int * (int * int) list) list -> int array array      (* sens naturel *)
ladj_of_madj : int array array -> (int * (int * int) list) list      (* sens naturel *)
```

4 Détection de particularités dans les graphes simples

Certains algorithmes ne s'appliquant qu'à des graphes ayant des propriétés particulières : graphes sans cycles, graphes connexes, etc ..., on met en place des algorithmes élémentaires permettant de détecter la présence de certaines de ces particularités, avec un coût de traitement qui est au plus $(O(|S|)$.

On suppose toujours que les sommets sont numérotés de façon concécutive, à partir de 1.

4.1 Graphes non orientés, codés par listes d'adjacence

Rappel : Les graphes non orientés et non pondérés sont codés comme des graphes orientés, pondérés articiciellement par 1, avec un arc retour $(v,u,1)$ pour tout arc $(u,v,1)$. On peut donc y utiliser (en partie) le vocabulaire des graphes orientés.

4.1.1 Présence de cycles

`have_cycle : (int * (int * 'a) list) list -> bool` A ECRIRE (3+4+6 lignes)

`have_cycle g` renvoie true si et seulement si g possède un cycle.

Principe : Colorier (ou marquer) les sommets au fur et à mesure de leur rencontre, à l'aide des couleurs :

- blanc (ou 0) pour "non marqué" ou "pas encore rencontré",
- gris (ou 1) pour "en cours de marquage" ou "rencontré une fois",
- noir (ou 2) pour "marqué" ou "rencontré deux fois".

Lorsque l'on explore tous les sommets, un sommet u est systématiquement rencontré :

- une première fois, comme sommet origine d'un ou plusieurs chemins (et on le marque "gris")
- au plus une deuxième fois, comme retour d'une extrémité v d'un arc (u,v) (et on le marque "noir")

La rencontre d'un sommet déjà vu au moins deux fois (marqué "noir") dénonce l'existence d'un cycle.

Détail : On utilise un vecteur de marquage, indicé par les sommets et initialisé à la couleur "blanc" (0).

Tant qu'il n'y a pas de cycle, on décrit tous les sommets (`while`) et, pour chaque sommet u,

- s'il est déjà marqué (non "blanc"), "gris" ou "noir", alors il est dans un chemin en cours de traitement et on passe au sommet suivant ;
- s'il est non marqué ("blanc"), alors on traite les chemins d'origine u, en suivant chaque arc (u,v) issu de u (de façon récursive), `List.iter (f u) (List.assoc u g)`, où `f u (v,p)` réalise le traitement :
  - marque u à 1 (état de début de chemin d'origine u, se continuant par v),
  - passe au sommet v (suivi de l'arc (u,v)) :
    - si v est "blanc", alors on traite v, de même que pour u, comme début de chemins ;
    - si v est marqué "gris", alors le marque "noir" : on a eu un deuxième passage en v (les chemins issus de v ont été traités lors du premier passage) ;
    - si v est déjà marqué "noir", c'est qu'il y a eu plus de deux passages en v et cela prouve la présence d'un cycle.

Complexité : chaque sommet étant traité, comme début de chemin, au plus une fois (lorsqu'il n'est pas marqué), cet algorithme a une complexité (maximale) qui est un $O(|S|)$.

4.1.2 Composantes connexes

`nb_connexes : (int * (int * 'a) list) list -> int` A ECRIRE (3+4+2 lignes)

`nb_connexes g` renvoie le nombre de composantes connexes de g (-1 pour un graphe vide).

Principe : de même, on colorie ("blanc" ou "noir") les sommets au fur et à mesure de leur rencontre.
Les sommets étant tous initialement "blanc" (0), le nombre de composantes connexes étant initialement égal à 0 (graphe vide), on explore tous les sommets et, pour chaque sommet u,

- si u est déjà marqué (donc "noir"), alors on passe au suivant
- si u est non marqué ("blanc"), alors u appartient à une nouvelle composante connexe.
  - on marque u en "noir",
  - on suit tous les chemins d'origine u en marquant "noir" les sommets rencontrés.

Complexité : chaque sommet étant traité au plus une fois (lorsqu'il n'est pas marqué), cet algorithme a une complexité (maximale) qui est un $O(|S|)$.

On adapte l'algorithme à la recherche de la composante connexe d'un sommet :

```
comp_connexe : int -> (int * (int * 'a) list) list                A ECRIRE
             -> (int * (int * 'a) list) list
```

`comp_connexe s g` renvoie le sous graphe de g constitué par la composante connexe du sommet s.

4.2 Graphes orientés, codés par listes d'adjacence

4.2.1 Présence de circuits

`have_circuit : (int * (int * 'a) list) list -> bool` A ECRIRE (3+4+2 lignes)

`have_circuit g` renvoie true si et seulement si g possède un circuit.

Algorithme : Les sommets étant initialement "blancs" (0), pour chaque sommet u "blanc",

- mise à "noir" (2) de u avant d'explorer les chemins issus de u
- mise à "gris" (1) de u en fin d'exploration de l'ensemble des chemins issus de u.

Si, en cours d'exploration, on retrouve un sommet marqué "noir" (2), c'est qu'il y a un circuit.

Complexité : chaque sommet étant traité, comme début de chemin, au plus une fois (lorsqu'il n'est pas marqué), cet algorithme a une complexité (maximale) qui est un $O(|S|)$.

4.3 Graphes, orientés ou non, codés par matrice d'adjacence

Pour des graphes (orientés ou non) représentés par matrice d'adjacence, on a des algorithmes identiques, en remplaçant, par exemple, dans l'exploration des chemins d'origine le sommet s,
`List.iter f (List.assoc s g)` par une boucle `for j = 1 to n do if (existe arc (s,j)) then ...`

5 Algorithmes de parcours dans les graphes simples

De nombreux algorithmes sur les graphes (orientés ou non) s'inspirent des méthodes de parcours ou utilisent les résultats d'un parcours. Un parcours de graphe peut être

- exhaustif (parcours complet ou parcours d'une partie accessible), dans le cas de graphe de faible taille ;
- partiel, limité à une "profondeur" d'exploration donnée d'une partie accessible, dans le cas de grands graphes où un parcours complet ne peut pas s'exécuter dans un temps raisonnable.

5.1 Parcours dans des graphes définis par liste d'adjacence

5.1.1 Parcours en largeur, depuis un sommet initial, à l'aide d'une pile FIFO.

Un parcours en largeur explore tous les sommets accessibles situés à la distance k (en nombre d'arc) du sommet initial avant de passer à ceux qui sont à la distance $k+1$. En limitant la valeur de k, on aurait un parcours partiel.

```
parcours_large : int -> (int * (int * 'a) list) list                A ECRIRE (< 18 lignes)
                 -> int array * int array
```

`parcours_large s g` parcours g en largeur à partir du sommet s et renvoie le couple (d,p) où

- $d.(u)$ est le nombre minimal d'arc nécessaires pour relier s à u (∞, s'il n'y a pas de chemin de s à u)
- $p.(u)$ est le père de u dans un plus court chemin reliant s à u ("nil" si u n'a pas de père)

Algorithme : Les sommets étant initialement "blancs" (0), un sommet découvert pour la première fois devient "gris" (1) puis, lorsque tous les sommets qui lui sont adjacents ont été identifiés, il devient "noir" (2).

1. Initialisations.
   - Le vecteur "couleur" est initialisé à "blanc" (0) .
   - Le vecteur d des distances au sommet initial s est initialisé à "∞" (ou -1),
   - Le vecteur p des pères à "nil" (-1) (pas de père).
2. Le sommet initial, "grisé", à distance 0 de lui même, sans père, est introduit dans une pile FIFO neuve.
3. Tant que la pile FIFO n'est pas vide,
   - pour le sommet u ("gris") prêt à sortir de la pile (sans le sortir), chaque sommet encore "blanc", de la liste d'adjacence de u, devient gris, de père direct u, de distance $1+d.(u)$ au sommet initial et on l'introduit dans la pile FIFO (il sera traité à son tour comme "père", le moment venu)
   - puis on dépile le sommet u et on le marque "noir"

Complexité : **A calculer ...**

Remarque. L'utilisation de la pile FIFO garantit que le parcours s'effectue en largeur et le marquage par couleur évite de tourner dans des cycles ou circuits. La coloration "noire", superflue, peut être utilisée à titre illustratif : affichage en "blanc", "gris", "noir" de l'état du parcours, dans un parcours limité en profondeur.

Application : Description d'un plus court chemin, en nombre d'arcs ou arêtes parcourus, d'un sommet u à un sommet v, comme une liste de sommets $[u;\ldots;v]$:

```
court_chemin_arcs : int -> int -> (int * (int * 'a) list) list-> int list        A ECRIRE
```

5.1.2 Parcours en profondeur

Un parcours en profondeur explore systématiquement les arcs issus d'un sommet dès que celui-ci est découvert.

1. **Parcours en profondeur exhaustif, avec reprise sur les sommets non découverts**

   Depuis un des sommets (non précisé), on explore la partie accessible depuis ce sommet et, s'il reste des sommets non vus, on continue l'exploration à partir de ces sommets.

   `parcours_profond : (int * (int * 'a) list) list -> int array * int array * int array` — A ECRIRE (< 16 lignes)

   `parcours_profond g` parcourt g en profondeur et renvoie un triplet de vecteurs (d, p, f) où

   - $d.(u)$ est la "date" de découverte du sommet u
   - $p.(u)$ est le père de u (sommet dans la liste d'adjacence duquel u a été découvert)
   - $f.(u)$ est la "date" de fin de traitement de u

   Algorithme : Les sommets étant initialement "blancs" (0), un sommet découvert pour la première fois devient "gris" (1) puis, lorsque tous les sommets qui lui sont adjacents ont été explorés, il devient "noir" (2).

   (a) Initialisations : dates à -1 ; vecteur des couleurs à "blanc" (0) ; vecteur des pères à "nil" (-1).

   (b) Pour chaque sommet "blanc" u du graphe, on visite u :
   - on marque u à "gris", on incrémente la date et c'est la date de découverte de u
   - pour chaque sommet "blanc" v de la liste d'adjacence de u, on donne u comme père à v, et on visite v
   - puis le sommet u est "noirci", la date est incrémentée et c'est la date de fin de traitement de u.

   Complexité : **A calculer . . .**

2. **Parcours en profondeur de la partie accessible depuis un sommet initial**

   Depuis un sommet initial, on explore uniquement la partie accessible depuis ce sommet.

   Algorithme : c'est le même principe que pour l'algorithme précédent, sauf que l'on part d'un sommet initial donné, sans reprise sur les sommets non accessibles depuis ce sommet.

   Application. Pour un graphe non orienté, on en déduit la description, sous forme de liste de sommets, d'un chemin de visite complète de tous les arcs de la composante connexe contenant un sommet donné, chaque arc étant parcouru une seule fois dans chacun des deux sens :

   `visite_aretes : int -> (int * (int * 'a) list) list -> int list` — A ECRIRE

5.2 Parcours de graphes définis par matrice d'adjacence

Pour des graphes (orientés ou non) représentés par matrice d'adjacence, on a des algorithmes de parcours identiques à ceux des graphes représentés par listes d'adjacence, en remplaçant, par exemple

`"List.iter f (List.assoc s g)"` par une boucle `"for j = 1 to n do if (existe arc (s,j)) then ..."`

6 Tri topologique d'un graphe orienté acyclique (sans circuit)

On utilise le tri topologique pour établir des précédences entre événements. Par exemple,

- pour établir l'enchaînement des incidents qui sont survenus lors d'une catastrophe.
- pour établir une "check list", liste ordonnée des protocoles nécessaires à la mise en route d'un processus.

G étant un graphe représenté par listes d'adjacence, on utilise le résultat du parcours en profondeur exhaustif, vu ci-dessus, pour produire une copie G' de G, où les sommets sont ordonnés linéairement, au sens :

si G' contient un arc (u,v), le sommet u apparaît avant v dans toute liste de G'.

On pourra alors dessiner graphiquement G', sur une droite, avec ses sommets alignés dans l'ordre topologique sur cette droite et ses arcs orientés selon ce même ordre.

Remarque. Si G contient des circuits, il n'est pas possible de créer un tel ordre et le résultat est non sensé.

Algorithme : Soit f le vecteur des dates de fin de traitement (f est un des résultat du parcours en profondeur).

1. On trie les listes d'adjacence selon l'ordre << défini par : `u << v ssi f.(u) > f.(v)`
2. On trie la liste du graphe selon l'ordre <<< défini par : `(u,Lu) <<< (v,Lv) ssi f.(u) > f.(v)`

`tri_topologique : (int * (int * 'a) list) list -> (int * (int * 'a) list) list` — A ECRIRE (4 ou 5 lignes)

`tri_topologique g` où g est un graphe orienté sans circuit, renvoie un graphe dont les sommets apparaissent, dans toute liste, selon l'ordre topologique : u apparaît avant v s'il existe un arc (u, v).

7 Plus courts chemins pondérés d'un graphe simple orienté valué

On considère ici un graphe G, orienté et valué, d'ensemble de sommets $S = \{1,2,\ldots,n\}$, représenté par une matrice d'adjacence (bordée) M, où, pour $i,j \in S$, M_{ij} est le poids de l'arc $(i,j) : i \longrightarrow j$, s'il existe et ∞ sinon.

Le poids d'un chemin entre deux sommets est la somme des poids sur les arcs constituant ce chemin. Les arcs du graphe peuvent avoir des poids négatifs, mais **le graphe ne doit pas posséder de circuit de poids strictement négatif.**

On supprime les boucles éventuelles en donnant, pour tout $i \in S$, la valeur 0 à M_{ii} (**∞ ne convient pas ici !**).

Remarque. Prendre $M_{ii} = 0$ pour tout $i \in S$, revient à supprimer ou ignorer les boucles (i,i), $i \in S$, du graphe, puisqu'elles n'interviennent pas dans le problème : circuits élémentaires de poids positif, qui ne peuvent qu'augmenter les distances pondérées.

7.0.1 Préalables

Les indices utiles pour les matrices sont les numéros de sommets (de 1 à n) et on borde les matrices par les numéros de sommets : $M_{i0} = i$, $M_{0i} = i$ (simple confort de lecture).

- La matrice d'adjacence M est arrangée ainsi : pour $i,j \in S$,

  M_{ij} est la pondération de l'arc (i,j), avec $\begin{cases} M_{ij} = \infty & \text{s'il n'y a par d'arc } (i,j) \text{ (pour } i \neq j) \\ M_{ii} = 0 & \textbf{(l'algorithme ne marche pas avec } M_{ii} = \infty \textbf{)} \end{cases}$

- On convient que, pour tout nombre x, éventuellement égal à ∞, $\begin{cases} \min(x,\infty) = \min(\infty,x) = x \\ x + \infty = \infty + x = \infty \end{cases}$

Remarque. En Caml, pour des valuations entières, on simule ∞ par `max_int` :

```
max_int;;             ---> int = 1073741823  (Caml light) / 4611686018427387903 (OCaml)
max_int + max_int;;   ---> int = -2          surprise!
```

Lemmes (dans les conditions énoncées ci-dessus) :

- Les sous-chemins d'un plus court chemin pondéré sont eux mêmes des plus courts chemins.
- Si le graphe contient n sommets, un plus court chemin pondéré comprends au plus $n-1$ arcs.

7.0.2 Algorithme

On note $D^{(m)}$ la matrice des longueurs des plus courts chemins d'au plus m arcs entre deux sommets.

- Initialement, pour $i,j \in S$, $D^{(1)}_{ij} = M_{ij}$.
- Pour $m > 1$, un plus court chemin d'au plus m arcs entre i et j (éventuellement de longueur ∞) se décompose en
  - un plus court chemin d'au plus $m-1$ arcs de i à k (éventuellement de longueur ∞),
  - un arc de k à j (éventuellement de longueur ∞),

  d'où la relation :

$$\text{pour } i,j \in S, \quad D^{(m)}_{ij} = \min\left(D^{(m-1)}_{ij}, \min_{\substack{k=1\cdots n \\ k\neq i \ k\neq j}} \left(D^{(m-1)}_{ik} + M_{kj} \right) \right)$$
$$= \min_{k=1\cdots n} \left(D^{(m-1)}_{ik} + M_{kj} \right) \quad \text{puisque } M_{ii} = M_{jj} = 0$$

(calculs étendus aux nombres infinis, d'après les conventions introduites ci-dessus)

vérifiée valable pour $i = j$, avec $D^{(m)}_{ii} = 0$ pour tout $m \geqslant 0$.

> Si on introduit de nouvelles opérations sur les nombres, avec les conventions vues ci-dessus :
>
> `+?` définie par `x +? y = min(x,y)` associative, d'élément neutre ∞
> `*?` définie par `x *? y = x + y` associative, d'élément absorbant ∞, d'élément neutre 0,
>
> on obtient une structure de (quasi) anneau et l'égalité ci dessus devient :
>
> $$\text{pour } i,j \in S, \quad D^{(m)}_{ij} = \sum_{k=1}^{n}{}^{?} D^{(m-1)}_{ik} \ *? \ M_{kj} \qquad \text{noté} \qquad D^{(m-1)} *! \ M$$
>
> et on reconnaît $D^{(m)}$ comme le "produit $*!$" de la matrice $D^{(m-1)}$ par la matrice $M = D^{(1)}$.

- Pour $m \geqslant n$, on aura $D^{(m)} = D^{(n)}$, puisque un plus court chemin pondéré contient au plus n arcs.

Il ne reste plus qu'à calculer $D^{(n)}$, puissance n-ième de la matrice $D = M$, au sens des opérations `+?` et `*?` :

- par l'algorithme d'exponentiation rapide (au sens du produit `*!`),
- en calculant la suite des $D^{(2m)} = D^{(m)} \ *! \ D^{(m)}$, à partir de $m = 1$, jusqu'à ce que $2m > n-1$ et alors $D^{(2m)} = D^{(n)}$,

pour avoir dans $D^{(n)}(i,j)$ la longueur d'un plus court chemin de i à j.
Ensuite, on exploite le contenu de $D^{(n)}$ pour obtenir un plus court chemin de i à j.

7.1 Mise en œuvre en Caml

On ne considère que des graphes dont l'ensemble des sommets est $S = \{1, \ldots, n\}$, pondérés par des entiers, sans circuits de poids négatifs, dont on supprime les boucles éventuelles en pondérant par 0 les arcs (u,u).
Par convention, on pose $\infty =$ `max_int` (rappel, en Caml, `max_int + max_int = -2`).

1. Mise en place des nouvelles opérations, `+?` et `*?`, sur les entiers, y compris $+\infty$:

```
let ( +? ) x y = ...                          (* associatif, d'élément neutre max_int  *)
and ( *? ) x y = ...                          (* Associatif, d'élément absorbant max_int *)
;;
(* val ( +? ) : 'a -> 'a -> a = <fun> *)
(* val ( *? ) : int -> int -> int = <fun> *)
```

2. Multiplication `*!` de matrices carrés **bordées** (indicées à partir de 1), au sens de `+?` et `*?` :

```
let mult2_matrix m1 m2 = ...                                    A ECRIRE  (4 ou 5 lignes)
;;
(* mult2_matrix : int array array -> int array array -> int array array *)
let ( *! ) m1 m2 = mult_matrix m1 m2
;;
(* val ( *! ) : int array array-> int array array -> int array array = <fun> *)
```

`mult2_matrix m1 m2` multiplie m1 par m2 au sens des opérations + ? et * ?
On peut écrire `m1 *! m2` à la place de `mult2_matrix m1 m2` .

3. **Algorithme de calcul des plus courtes distances pondérées** :

```
let plus_courtes_distances w = ...                              A ECRIRE  (4 ou 5 lignes)
;;
(* plus_courtes_distances : int array array -> int array array = <fun> *)
```

`plus_courtes_distances w`, où w est la matrice d'adjacence du graphe g, renvoie la matrice (bordée) des plus courtes distance (en poids) d'un sommet à un autre dans le graphe g.

Complexité : **A calculer** , en fonction du nombre n de sommets, en ne comptant que les opérations sur les entiers qui sont coûteuses (`*?`) et en négligeant les autres (`+?`).

7.2 Application : description d'un plus court chemin pondéré d'un sommet à un autre

Pour obtenir la description d'un plus court chemin pondéré d'un sommet u à un sommet v, on introduit la matrice (bordée) ML de liaison du graphe :

$$\text{pour } i,j \in S,\ ML(i,j) = \begin{cases} \text{prédécesseur de } j \text{ dans un plus court chemin pondéré allant de } i \text{ à } j \\ -1 \text{ s'il n'y a pas de chemin de } i \text{ à } j \end{cases}$$

1. La matrice de liaison est déduite de W (`w`), matrice d'adjacence du graphe, et de D (`d`), matrice des plus courtes distance pondérées par un algorithme en $O(n^3)$) :

```
let liaison w d = ...                                           A ECRIRE   (12 lignes)
;;
(* liaison : int array array -> 'a array arrayt -> int array array *)
```

2. On déduit, de la matrice de liaison ml, la liste [u ; . . . ; v] des sommets d'un plus court chemin pondéré de u à v :

```
let court_chemin_poids ml u v = ...                             A ECRIRE  (5 à 6 lignes)
;;
(* court_chemin_poids : int array array -> int -> int -> int list *)
```

8 Dessin de (petits) graphes simples

`dessin_gla : int * int * (int * (int * int) list) list -> unit`

`dessin_gla sizex sizey gla` dessine le graphe orienté, pondéré par entiers, défini par la liste d'adjacence `gla`, dans la fenêtre graphique de dimension `sizex sizey` : numéros de sommets dans des "petits" cercles répartis sur un "grand" cercle, arcs fléchés d'un sommet à l'autre, avec édition succinte des informations sur les arcs.

```
#use "dessine.ml";;
open_graph " 800x800+10+10";;
dessin_gla (size_x ()) (size_y ()) g;;
```

9 Lexique (graphes simples), références

arête (graphe non orienté)
arc (graphe orienté)

boucle : arc ou arête allant d'un sommet à lui même.
degré sortant : nombre d'arcs issus d'un sommet.
degré rentrant : nombre d'arcs aboutissant à un sommet.
degré : nombre d'arcs ou arêtes partant d'un sommet ou y aboutissant (une arête (u,u) compte pour deux dans le calcul du degré de u).
sommet adjacent à un autre (ou sommet incident)
arête ou arc incident à un sommet

chemin (graphe simple orienté) : séquence de sommets reliés par des arcs.
circuit (graphe simple orienté)) : chemin ayant des extrémités identiques et comportant au moins un arc.

chaîne (graphe simple non orienté) : séquence de sommets reliés par des arêtes.
cycle (graphe simple non orienté) : chaîne ayant des extrémités identiques et comportant au moins une arête.

Dans le cas de multi-graphes, il faut décrire les chemins ou les chaînes comme des séquences d'arcs ou d'arêtes (numérotées) concécutives.
Dans les graphes simples, la description des chemins ou des chaînes comme des séquences d'arcs ou d'arêtes concécutives est équivalente à la description comme séquences de sommets.

accessible : v est un sommet accessible à partir de u s'il existe un chemin (ou une chaîne) de u vers v.

graphe connexe (graphe non orienté) : graphe tel que tout couple de sommets soit reliés par une chaîne.
composante connexe (graphe non orienté) : sous-graphe connexe, constitué de sommets et des arcs relatifs à ces sommets (il n'existe pas, dans le graphe, d'arêtes sortant de ce sous-graphe ou y entrant)
(classe d'équivalence de la relation entre sommets : "est accessible à partir de").

graphe fortement connexe (graphe orienté) : Si chaque sommet est accessible à partir d'un sommet quelconque.
composante fortement connexe (graphes orienté) : sous-graphe fortement connexe maximal.

longueur (length) d'un chemin, d'une chaîne : nombre d'arcs, d'arêtes.
poids (weight) d'un chemin, d'une chaîne : somme des pondérations des arcs composant le chemin, la chaîne.

Remarque. Il y a souvent ambigüité dans l'emploi des mots "longueur (length)", "distance", "poids (weight)", "valeur", "valuation (value)", et des expressions "plus court chemin", ...

Ouvrages de référence

[CM95] G.Cousineau, M.Mauny. Approche fonctionnelle de la programmation, Edisciences international Paris 1995.
Quelques éléments sur les graphes, pas mal de choses sur les arbres et rien sur les automates.
Beaucoup de programmation Caml, avec des écritures souvent complexes (parfois inutilement).

[CLR94] T.Cormen, C.Leiserson, R.Rivest. Introduction à l'algorithmique Dunod, 1994.
Introduction to Algorithms, MIT Press 1990 (il y a une édition plus récente, en français).
Très complet sur les problèmes algorithmiques et en particulier sur les graphes et arbres. Peu de choses sur les automates.
Un chapitre intéressant (mais complexe) sur les algorithmes parallèles et les circuits combinatoires

[GM79] M.Gondran, M.Minoux. Graphes et algorithmes, Eyrolles Paris 1979.
Comme le titre le laisse espérer, c'est assez complet sur les graphes, dans un esprit d'application à des problèmes concrets difficiles. (L'édition est déjà un peu ancienne).

< *FIN* >