Informatique MP
Graphes hamiltoniens, Cycles hamiltoniens dans FRANCE ROUTIERE

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

. . . /Hamilton-C.tex compilé le lundi 05 mars 2018 à 17h 22m 18s avec LaTeX

. . . /Hamilton-C.tex Compilé le lundi 05 mars 2018 à 17h 22m 18s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Ordre du TP : dans le premier tiers d'année (TP n^o 2 ou 3)

Eléments Caml utilisés :

- listes, vecteurs, matrices, ensembles à champs nommés, exceptions
- bibliothèques : Caml light "graphics", "printf" ; OCaml "Graphics", "Printf", "List", "Array"

Documents relatifs au TP :

- Enoncé : Hamilton-C.tex, Hamilton-C.dvi, Hamilton-C.pdf (et version S, par altération de version C)
- Figures recopiées depuis . . . : Hamilton, Petersen, France
- Fichiers de données à inclure (pour France), recopié et aménagé depuis Tpinfo/France/ :
  - contour.ml (aménagé depuis contour0.ml
  - prefect.ml (aménagé depuis prefect0.ml
  - liens.ml (aménagé depuis liens0.ml)
- Squelette de programme Caml : TP-Ham-S.ml
- Programme corrigé OCaml : TP-Ham-C.ml
- Autres (hors TP)
  - Algorithme des fourmis : (adapté pour France, avec souris modifiée)
  - Arbcouvrant, algorithme de parcours exhaustif ...
  - Floyd-Warshall
  - Puissances de la matrice d'adjacence (exemple Hamilton, France) ...

Structure de ce document (dans cet ordre) :

- Texte du TP
- Section 6 : Annexes
  - Annexe 1 : mise en place et déroulement du TP (conseils)
  - Annexe 2 données : (contour.ml, prefect.ml, liens.ml)
  - Annexe 3 squelette : Caml lightam-S.ml)
- Section 7 : corrigé : OCaml (TP-Ham-C.ml)
- Section 8 : HORS TP ; Compléments (OCaml)
  Transformations ; Algorithme des Fourmis ; Arbre couvrant ; Floyd-Warshall ; Calcul matriciel

Distribution pour la réalisation du TP :

- dossier : Hamilton
- contenu :
  - Hamilton-S.pdf **un extrait de** Hamiltone-C.pdf
    - sans la page de garde
    - sans les sections 7 (source du corrigé), 8 (compléments)
  - contour.ml, prefect.ml, liens.ml
  - TP-Ham-S.ml

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

Graphes hamiltoniens.
FRANCE ROUTIERE ET PREFECTURES.

Dans ce TP Caml, on se propose de traiter de problèmes de parcours exhaustifs de graphes.

Les parcours exhaustifs de graphes sont utilisés pour établir des solutions à des problèmes d'ordonnancement, d'optimalisation. Un problème classique est celui du **"voyageur de commerce"** ("contraint") : il s'agit, en empruntant uniquement un réseau routier de ville à ville voisine, de visiter ces villes, en passant une et une seule fois par chaque ville et en revenant à la ville de départ, tout en effectuant un parcours de longueur minimale. Un tel problème, s'il admet une solution, est NP-complet : on ne connaît pas d'algorithme de résolution systématique en un temps polynomial.

A partir de deux exemples (l'un simple, l'autre plus complexe), on cherche une solution au problème de la visite de n villes, en passant une et une seule fois par chaque ville, le long d'un réseau routier reliant certaines villes entre elles, et en revenant à la ville de départ, sans imposer que le parcours soit de longueur minimale. Ce problème moins contraignant, s'il admet une solution, est également NP-complet, mais peut être résolu en temps polynomial dans certains cas particuliers.

Les fichiers de ressources du TP sont initialement dans le dossier MPx/ $\cdots$ /TP-Info/Hamilton/,
à copier dans votre dossier personnel. Ensuite, vous trouverez normalement dans votre dossier Hamilton/

- le texte de ce TP au format .pdf : Hamilton-S.pdf,
- un squelette de programme Caml (fonctions écrites ou à compléter, exemples) : TP-Ham-S.ml,
- des fichiers de données, à inclure, pour l'exemple France routière : prefect.ml, liens.ml, contour.ml.

1 Graphes (simples) finis, non orientés

Un graphe fini non orienté, sans boucles, est un couple $G = (S,A)$ où

- S est un ensemble fini (ensemble des **sommets**)
- A est un sous-ensemble de l'ensemble des parties à 2 éléments distincts de S (ensemble des **arêtes**) (une boucle serait une arête réduite à un élément).

Le graphe peut être **pondéré** (valué) en associant à chaque arête un poids (ou valuation).

Remarque. On assimilera une arête $\{x,y\}$ de poids p aux deux couples ou **arcs** ori
ce qui permet de préciser un ordre de parcours éventuel, $x \longrightarrow y$ ou $y \longrightarrow x$. [1]

Un graphe fini se dessine comme un ensemble de sommets (points, cercles, ...),
reliés par des arêtes (lignes, ...) ou arcs (lignes fléchées, ...).

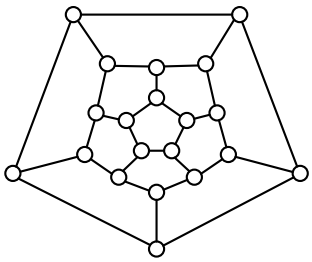

FIGURE 1 – Le voyage autour du monde de Hamilton (hamiltonien).

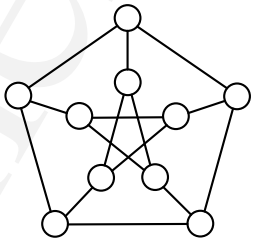

FIGURE 2 – Graphe Petersen (non hamiltonien).

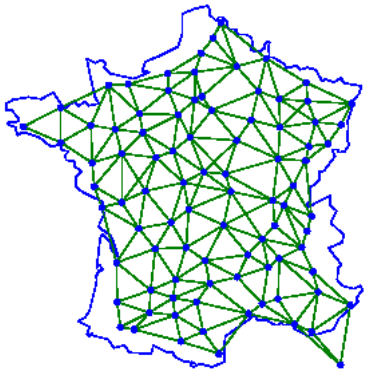

FIGURE 3 – Graphe France routière (hamiltonien).

Remarque. Les identifiants de sommets et valuations d'arc n'ont pas été représentés (voir les données ci-dessous).

Une ***chaîne*** (***chemin*** pour un graphe orienté) ***de sommets***, de ***longueur*** [a] $k \geqslant 0$ (ayant k arêtes ou arcs), est une séquence de $k+1$ sommets, $(x_0, x_1, \cdots, x_k)$, chaque sommet étant lié au sommet suivant par une arête $\{x_i, x_{i+1}\}$ ou un arc $x_i \to x_{i+1}$, tous deux notés (x_i, x_{i+1}) [b] .

Un ***cycle*** (***circuit*** pour un graphe orienté) est une chaîne (chemin) de sommets, $(x_0, x_1, \cdots, x_k)$, de longueur $k > 1$, où $x_k = x_0$, qualifié de **élémentaire** si les sommets $x_1, ..., x_k$ sont distincts.

a. Ne pas confondre avec la longueur en poids, appelée aussi poids, valuation, distance, qui est la somme des poids des arcs.
b. L'arête $\{x_i, x_{i+1}\}$ peut être notée indifféremment (x_i, x_{i+1}) ou (x_{i+1}, x_i).

Les chaînes (chemins), cycles (circuits) peuvent être aussi représentés par des séquences d'arêtes ou d'arcs adjacents :
$\big((x_0,x_1), \ldots, (x_i,x_{i+1}), (x_{i+1},x_{i+2}), \ldots, (x_{k-1},x_k)\big)$.

Un graphe est **hamiltonien** s'il contient un cycle **hamiltonien** (cycle élémentaire passant par tous les sommets).

1. Un graphe non orienté est ainsi considéré comme un graphe orienté symétrique.

Remarques.

- On ne connaît pas de condition nécessaire et suffisante d'existence d'un cycle hamiltonien, mais il existe des conditions suffisantes pour affirmer qu'un graphe est ou n'est pas hamiltonien. Par exemple, le graphe de la Figure 3 (France routière) est hamiltonien puisque deux sommets distincts ont toujours au moins un voisin commun.
- Le problème de l'existence et de la recherche d'un cycle passant une et une seule fois par tous les sommets d'un graphe non orienté est un problème **NP complet** (on ne connaît pas d'algorithme polynomial systématique), mais une solution peut être validée en un temps polynomial (en $O(|S|^2)$) par un algorithme.

2 Codage d'un graphe fini

Pour simplifier le codage (utilisation de vecteurs, de matrices), les n sommets sont numérotés consécutivement de 1 à n et les arcs sont accompagnés de leur seule pondération (**les graphes non pondérés sont pondérés artificiellement par 1**).
Remarque. On pourra associer au graphe (de façon interne ou externe), des informations complémentaires.

Par exemple, dans le graphe de la Figure 3 (France routière),

- les sommets sont les villes préfecture (représentées par le numéro du département),
- les arcs sont des liaisons directes entre préfectures voisines, pondérées par la distance

et on associe à chaque sommet, le "nom de département", le "nom de la préfecture" , le "nom de région" (l'ancienne !) et ses coordonnées géographiques.

2.1 Codage par liste d'adjacence

- La liste **d'adjacence d'un sommet** u est la liste des couples (v, p_{uv}) tels qu'il existe un arc $(u,v) : u \longrightarrow v$ (ou une arête $\{u,v\}$), de pondération p_{uv}.
- Un graphe est codé comme liste des couples (sommet s, liste d'adjacence de s) [a].
  dénommée **liste d'adjacence du graphe**.

a. si les sommets sont numérotés de 0 ou 1 à n, il est parfois plus pratique d'avoir un vecteur de listes d'ajacence (v.(s) = liste d'adjacence de s)

Remarque. Pour un graphe non orienté, codé comme un graphe orienté symétrique, si v est dans la liste d'adjacence de u, u est également dans la liste d'adjacence de v, avec la même pondération (il y a redondance).
Une **boucle** serait un arc (ou arête) (u,u), de poids significatif (u est dans sa propre liste d'adjacence).

Exemples

- Graphe de la Figure 1 (Hamilton), artificiellement pondéré par 1 :

```
[1,[ 2,1;  5,1;  8,1];  2,[ 1,1;  3,1; 10,1];  3,[ 2,1;  4,1; 12,1];  4,[ 3,1;  5,1; 14,1]; 5,[ 1,1;  4,1;  6,1];
 6,[ 5,1;  7,1; 15,1];  7,[ 6,1;  8,1; 17,1];  8,[ 1,1;  7,1;  9,1];  9,[ 8,1; 10,1; 18,1]; 10,[ 2,1;  9,1; 11,1];
11,[10,1; 12,1; 19,1]; 12,[ 3,1; 11,1; 13,1]; 13,[12,1; 14,1; 20,1]; 14,[ 4,1; 13,1; 15,1]; 15,[ 6,1; 14,1; 16,1];
16,[15,1; 17,1; 20,1]; 17,[ 7,1; 16,1; 18,1]; 18,[ 9,1; 17,1; 19,1]; 19,[11,1; 18,1; 20,1]; 20,[13,1; 16,1; 19,1] ]
```

- Graphe de la Figure 2 (Petersen), artificiellement pondéré par 1 :

```
[1,[ 2,1; 5,1; 7,1]; 2,[1,1; 3,1; 8,1]; 3,[2,1;  4,1; 9,1]; 4,[3,1; 5,1; 10,1]; 5,[1,1; 4,1; 6,1];
 6,[ 8,1; 9,1];      7,[9,1; 10,1];     8,[6,1; 10,1];      9,[7,1; 6,1];      10,[7,1; 8,1]  ]
```

- Graphe de la Figure 3 (France routière), pondéré par les distances entre villes voisines ...

Inconvénients :

- obligation de parcours systématique des listes d'adjacence pour obtenir les arcs.
- si le graphe est orienté, non symétrique, il n'est pas immédiat d'obtenir les arcs d'extrémité v.

Avantages :

- on ne stocke que la partie utile de l'information (sauf si l'arête $\{x,y\}$ pondérée par p est représentée par deux arcs : (x,p) est dans la liste d'adjacence de y et (y,p) est dans la liste d'adjacence de x) ;
- pas d'obligation de numéroter les sommets de façon consécutive (mais il est préférable de le faire) ;
- possibilité d'ordonnancement selon divers critères et introduction aisée de nouveaux sommets ou arcs.

2.2 Codage par matrice d'adjacence (sommets numérotés de 1 à n).

La **matrice d'adjacence** du graphe est une matrice M, carrée, d'indices $0,1,\ldots,n$, d'ordre $n+1$, telle que, pour $i,j \in \{1,\ldots,n\}$, $M(i,j)$ contienne l'information (existence, pondération) sur l'arc reliant i à j :

- $M(i,0)$ et $M(0,j)$ sont non utilisés ou prennent la valeur du sommet : matrice "bordée" (lisibilité de la matrice) ;
- Pour $i \neq j$, $\begin{cases} M(i,j) = \text{poids de l'arc } (i,j) \text{ s'il existe un arc de } i \text{ à } j, \\ M(i,j) = \infty \text{ ("valeur impossible") s'il n'existe pas d'arc de } i \text{ à } j \end{cases}$

  si les poids sont des entiers, ∞ peut être représenté par `max_int`
- $M(i,i) = \infty$ (ou $M(i,i) = 0$) [2], s'il n'existe pas de boucle de i à i.

La matrice d'adjacence d'un graphe non orienté est symétrique, et on pourrait ne coder que la partie supérieure de la matrice ....

2. Certains algorithmes matriciels ne marchent (sans acrobaties) qu'avec 0, d'autres ne marchent qu'avec ∞ et il y en a qui marchent avec les deux !

Exemples. On inscrit le numéro de colonne en ligne 0 et le numéro de ligne en colonne 0.

- Le graphe de la figure Figure 1 (Hamilton), artificiellement pondéré, est représenté par la matrice (bordée) :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	1	∞	∞	1	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	1	0	1	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞
4	1	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞
5	∞	1	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞
6	∞	∞	1	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞
7	∞	∞	∞	1	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
8	1	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞
9	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1
10	∞	1	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞
11	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	1	∞
12	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	∞	∞
13	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞	1
14	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞	∞
15	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	∞	∞
16	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞	1
17	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞	∞
18	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	∞	1	0	1	∞
19	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	∞	∞	1	0	1
20	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞	1	∞	∞	1	0

avec $M(i,i) = 0$ lorsqu'il n'existe pas de boucle $i \longrightarrow i$.

- Le graphe de la figure Figure 2 (Petersen), artificiellement pondéré, est représenté par la matrice (bordée) :

	0	1	2	3	4	5	6	7	8	9	10
0	∞	1	∞	∞	1	∞	1	∞	∞	∞	
1	1	∞	1	∞	∞	∞	∞	1	∞	∞	
2	∞	1	∞	1	∞	∞	∞	∞	1	∞	
3	∞	∞	1	∞	1	∞	∞	∞	∞	1	
4	1	∞	∞	1	∞	1	∞	∞	∞	∞	
5	∞	∞	∞	∞	1	∞	∞	1	1	∞	
6	∞	∞	∞	∞	∞	∞	∞	∞	1	1	
7	∞	∞	∞	∞	∞	1	∞	∞	∞	1	
8	∞	∞	∞	∞	∞	1	1	∞	∞	∞	
9	∞	∞	∞	∞	∞	∞	1	1	∞	∞	
10											

avec $M(i,i) = \infty$ lorsqu'il n'existe pas de boucle $i \longrightarrow i$.

- La matrice d'adjacence du graphe de la figure Figure 3 (France), pondéré par les distances, est trop grande (90 sommets) pour être représentée ici.

Avantages :

- accès direct à l'information, à coût constant,
- on trouve facilement les arcs d'origine u,
- on trouve facilement les arcs d'extrémité v.

Inconvénients :

- si le graphe comporte beaucoup de sommets et peu d'arcs, ce qui est fréquent, la matrice est "creuse" et on réserve beaucoup d'espace pour peu de données ;
- la représentation à l'aide d'une matrice se prête mal à l'introduction dynamique de nouveaux sommets.

Remarque. On se servira peu de la matrice d'adjacence, d'autant plus que cela peut poser quelques problèmes (complications, acrobaties) selon le choix fait pour le codage de l'absence de boucle $i \longmapsto i$: 0 ou ∞.

2.3 Type Caml pour le codage des graphes finis dans ce TP.

Les graphes utilisés ici ont pour ensemble de sommets $S = \{1, 2, \ldots, n\}$ et sont définis par leur liste d'adjacence pondérée (par des entiers). On utilisera aussi, pour des raisons de confort et de simplicité, une description (redondante) avec un type **ensemble à champs nommés**,

- permettant de bénéficier des avantages des deux représentations (liste d'adjacence, matrice d'adjacence) ;
- comportant des informations pour dessiner le graphe (coordonnées entières des sommets) ;
- comportant une information pour tracer une ligne polygonale (éventuelle) délimitant le dessin du graphe.

1. Caml light :

```
type int_graphe =
  {ladj: (int * (int * int) list) list; (* liste d'adjacence du graphe *)
   madj: int vect vect;                 (* matrice d'adjacence du graphe *)
   lpos: (int * (int * int)) list;      (* liste des positions graphiques : (s,(x,y) *)
   lbox: (int * int) list               (* ligne d'entourage : coordonnées *)
  }
;;
```

2. OCaml :

```
type int_graphe =
  {ladj: (int * (int * int) list) list; (* liste d'adjacence du graphe *)
   madj: int array array;               (* matrice d'adjacence (bordée) *)
   lpos: (int * (int * int)) list;      (* liste des positions graphiques : (s,(x,y) *)
   lbox: (int * int) list               (* ligne d'entourage : coordonnées *)
  }
;;
```

Autant que possible, on évitera l'utilisation de la matrice d'adjacence.

3 Caml light : Ressources particulières pour le TP

Compléments aux ressources courantes (éléments extraits de l'aide Caml light).

3.1 Fonctions sur les listes, vecteurs, matrices. Extraits :

Fonctions courantes : `list_length ; hd ; tl ...`
`vect_length ; make_vect ; make_matrix ... ; copy_vect ...`
`vect_of_list ; list_of_vect ;...`

Remarque. `make_vect n (make_vect n 0)` ne donne pas une matrice convenable !

Fonctions complémentaires : (**Il faudrait savoir les ré-écrire !**)

`value mem : 'a -> 'a list -> bool`
mem a l is true if and only if a is structurally equal (see module eq) to an element of l.

`value assoc : 'a -> ('a * 'b) list -> 'b`
assoc a l returns the value associated with key a in the list of pairs l. That is, assoc a [... ; (a,b) ; ...] = b if (a,b) is the leftmost binding of a in list l. Raise Not_found if there is no value associated with a in the list l.

`value map : ('a -> 'b) -> 'a list -> 'b list`
map f [a1 ; ... ; an] applies function f to a1, ..., an, and builds the list [f a1 ; ... ; f an] with the results returned by f.

`value subtract : 'a list -> 'a list -> 'a list`
subtract l1 l2 returns the list l1 where all elements structurally equal to one of the elements of l2 have been removed.

`value except : 'a -> 'a list -> 'a list`
except a l returns the list l where the first element structurally equal to a has been removed. The list l is returned unchanged if it does not contain a.

autres : `do_list ; it_list ; list_it ; ...` (mais on peut s'en passer).

3.2 Bibliothèque graphics. Extraits :

Le repère de la fenêtre graphique est tel que { l'axe des x est orienté de gauche à droite (ouest vers est), ; l'axe des y est orienté de bas en haut (sud vers nord), } l'origine du repère étant dans le coin gauche du bas de la fenêtre.

```
#open "graphics";;              (* ouverture de la bibliothèque graphique *)
open_graph " 800x800+10+10";;  (* ouverture de la fenêtre graphique, dimension et position *)
clear_graph ();;               (* efface le contenue de la fenêtre *)
size_x ();; size_y ();;   (* coordonnées entières extrêmes de la fenêtre *)
set_color black;;   (* couleur de tracé (black,white,red,green,blue,yellow,cyan,magenta) *)
moveto x y;;        (* déplacement au point (x,y), sans tracé *)
lineto x y;;        (* déplacement depuis le point courant jusqu'à (x,y) avec tracé *)
draw_circle x y r;; (* trace le cercle de rayon r centré en x y avec la couleur courante *)
fill_circle x y r;; (* remplit le disque de rayon r centré en x y avec la couleur courante *)
draw_string s;;     (* trace le texte de la chaîne s à la position courante *)
```

3.3 Bibliothèque printf (éditions formatées). Extraits :

```
#open "printf";;   (* ouverture de la bibliothèque d'éditions formatées *)
value printf  chaîne_de_formats valeurs_associées -> unit;;      (* syntaxe ... *)
value sprintf chaîne_de_formats valeurs_associées -> string;;    (* syntaxe ... *)
```

Utilisation : voir aide Caml light ; exemples ci-dessous : { édition formatée d'une matrice d'entiers : `print_int_mat`; ; utilisation de la souris : `souris_France ()`. }

Rappel : `print_newline ();; (* passage à la ligne *)`

3.4 Bibliothèque random (nombres pseudo-aléatoires). Extraits :

`value random__init : int -> unit` [3]
Initialize the generator, using the argument as a seed. The same seed will always yield the same sequence of numbers.

`value random__int : int -> int`
random__int bound returns a random number between 0 (inclusive) and bound (exclusive). Bound must be positive and smaller than 2^{30}.

3. le "double souligné" est une déclinaison : fonction `init` de la bibliothèque `random`.

4 OCaml : Ressources particulières pour le TP

Compléments aux ressources courantes (éléments extraits de l'aide OCaml).

4.1 Fonctions sur les listes (List), vecteurs, matrices (Array). Extraits :

Fonctions courantes : `List.length ; List.hd ; List.tl ; List.rev ; ...`
`Array.length ; Array.make ; Array.make_matrix ... ; Array.copy ;...`
`Array.of_list ; Array.to_list ;...`

Remarque. `Array.make n (Array.make n 0)` ne donne pas une matrice convenable !

Fonctions complémentaires : (**Il faudrait savoir les ré-écrire !**)

`val List.mem : 'a -> 'a list -> bool`
List.mem a l is true if and only if a is equal to an element of l.

`val List.assoc : 'a -> ('a * 'b) list -> 'b`
List.assoc a l returns the value associated with key a in the list of pairs l. That is, List.assoc a [... ; (a,b) ; ...] = b if (a,b) is the leftmost binding of a in list l. Raise Not_found if there is no value associated with a in the list l.

`val List.map : ('a -> 'b) -> 'a list -> 'b list`
List.map f [a1 ; ... ; an] applies function f to a1, ..., an, and builds the list [f a1 ; ... ; f an] with the results returned by f. No tail-recursive.

`val List.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 List.filter : ('a -> bool) -> 'a list -> 'a list`
List.filter p l returns all the elements of the list l that satisfy the predicate p. The order of the elements in the input list is preserved.

~~`val subtract : 'a list -> 'a list -> 'a list`~~
~~`val except : 'a -> 'a list -> 'a list`~~ voir `List.filter`
autres : ~~`do_list ; it_list ; list_it`~~ voir `List.iter ; List.fold_left ; List.fold_right ; ....`

4.2 Bibliothèque graphics. Extraits :

Le repère de la fenêtre graphique est tel que { l'axe des *x* est orienté de gauche à droite (ouest vers est), ; l'axe des *y* est orienté de bas en haut (sud vers nord), } l'origine du repère étant dans le coin gauche du bas de la fenêtre.

```
#load "graphics.cma";;          (* chargement de la bibliothèque graphique *)
open Graphics;;                 (* ouverture de la bibliothèque graphique *)
open_graph " 800x800+10+10";;  (* ouverture de la fenêtre graphique, dimension et position *)
clear_graph ();;              (* efface le contenue de la fenêtre *)
size_x ();; size_y ();;       (* coordonnées entières extrêmes de la fenêtre *)
set_color black;;    (* couleur de tracé (black,white,red,green,blue,yellow,cyan,magenta) *)
moveto x y;;         (* déplacement au point (x,y), sans tracé *)
lineto x y;;         (* déplacement depuis le point courant jusqu'à (x,y) avec tracé *)
draw_circle x y r;; (* trace le cercle de rayon r centré en x y avec la couleur courante *)
fill_circle x y r;; (* remplit le disque de rayon r centré en x y avec la couleur courante *)
draw_string s;;      (* trace le texte de la chaîne s à la position courante *)
```

4.3 Bibliothèque Printf (éditions formatées). Extraits :

```
val Printf.printf  chaîne_de_formats valeurs_associées -> unit;;      (* syntaxe ... *)
val Printf.sprintf chaîne_de_formats valeurs_associées -> string;;    (* syntaxe ... *)
```

Utilisation : voir aide OCaml ; exemples ci-dessous : { édition formatée d'une matrice d'entiers : `print_int_mat`; ; utilisation de la souris : `souris_France ()`. }

Rappel : `print_newline ();; (* passage à la ligne *)`

4.4 Bibliothèque Random (nombres pseudo-aléatoires). Extraits :

`val Random.init : int -> unit`
Initialize the generator, using the argument as a seed. The same seed will always yield the same sequence of numbers.

`val Random.int : int -> int`
random.int bound returns a random number between 0 (inclusive) and bound (exclusive). Bound must be positive and smaller than 2^{30}.

5 Le TP (graphes non orientés)

5.1 Données initiales des exemples (selon conventions précédentes)

1. L'exemple "Hamilton" (voyage autour du monde) est donnée initialement par deux listes (fournies en annexe) :
   - la liste d'adjacence (valuée par 1) : `ladj_hamilton` de type `(int * (int * int) list) list` ;
   - la liste des positions : `lpos_hamilton` de type `(int * (int * int)) list`,
     qui associe à un sommet s le couple des coordonnées (x,y) de s utilisable pour dessiner le graphe ;
   - (il n'y a pas de liste pour préciser un contour du dessin du graphe : le contour sera vide).
2. L'exemple "France" (réseau routier de la France) est donnée initialement par les trois listes :
   - la liste d'adjacence valuée : `liens` de type `(int * (int * int) list) list` ;
     qui associe à chaque ville (numéro de département) la liste des couples (ville voisine, distance) ;
   - la liste `prefectures` de type `(int * (string * string * int * int * string)) list`
     qui associe à chaque numéro de département le 5-uplet (nom département, nom préfecture, coordonnée x, coordonnée y, nom région) ;
   - la liste `contour` de type `(int * int) list` qui représente la liste des coordonnées (x,y) de points successifs du contour géographique de la France métropolitaine, selon le sens trigonométrique depuis le point nord extrême.

   Ces listes, définies dans les fichiers liens.ml, prefect.ml, contour.ml, seront incluses dans le programme par
   - Caml light : `include "LIENS.ML";; include "PREFECT.ML";; include "CONTOUR.ML";;`
   - OCaml : `#use "LIENS.ML";; #use "PREFECT.ML";; #uses "CONTOUR.ML";;`

   après avoir défini le chemin d'accès aux fichiers par : `#directory ".... \TP-info\Hamilton";;`

Toutes les coordonnées sont (ont été) adaptées au système de coordonnées de la fenêtre graphique de Caml.

5.2 Construction d'une matrice d'adjacence depuis une liste d'adjacence

Ecrire la fonction `matrice_adj_of_liste_adj` de type

```
(int * (int * int) list) list -> int vect vect     (Caml light)
(int * (int * int) list) list -> int array array   (OCaml)
```

telle que `matrice_adj_of_liste_adj ladj` où `ladj` est la liste d'adjacence d'un graphe pondéré par valeurs entières, de sommets numérotés de 1 à n, renvoie la matrice d'adjacence M, $(n+1)\times(n+1)$, du graphe, bordée par inscription des sommets comme numéros de colonne en ligne 0 et numéros de ligne en colonne 0, avec, pour $i,j \in \{1,\ldots,n\}$, $i \neq j$, $M.(i).(j) = \infty$ (`max_int`) s'il n'existe pas d'arc de i vers j et $M.(i).(i) = 0$ s'il n'y a pas de boucle de i vers i.

Cette fonction pourra faire appel à des fonctions auxiliaires récursives mais ne doit pas utiliser de référence.

Remarque. Les matrices ainsi obtenues pourront être éditées à l'écran par la fonction pré-écrite `print_int_matrix`, sous une forme facile à lire (comme, précédemment, les matrices des graphes Hamilton et Petersen) . . . . . si le nombre de sommets n'est pas trop grand !

5.3 Codage des exemples

5.3.1 Codage de l'exemple Hamilton

Construire la donnée `hamilton`, de type `int_graphe` (on affectera la liste vide au champ nommé `lbox`).

Vérifier, en éditant (avec `print_int_matrix`) la matrice d'adjacence contenue dans cette structure.

5.3.2 Codage de l'exemple France

1. Ecrire (avec une seule instruction) la fonction `prefecture_of_dep`, de type `int -> string`, telle que `prefecture_of_dep n` renvoie le nom de la préfecture du département `n` (ce nom est à extraire de la liste globale `prefectures`, à l'aide d'une fonction adéquate).

   Application : Quelle est la préfecture de la Lozère (48) ?
2. Construire la donnée `france`, de type `int_graphe` (ici le champ `lbox` sera renseigné)

   ~~Vérifier, en éditant (avec `print_int_matrix`) la matrice d'adjacence contenue dans cette structure~~

   . . . . . c'est un peu trop gros pour être lisible.

5.4 Dessin des graphes

5.4.1 Dessin du contour (facile)

Ecrire la fonction `contour_plot`, de type `int_graphe -> unit`, telle que `contour_plot gr` trace, dans la fenêtre graphique, la ligne polygonale formée par les points successifs de la liste `gr.lbox`.

Cette fonction pourra faire appel à une fonction interne récursive mais ne doit pas utiliser de variable référence.

Une indication : c'est plus simple de faire le tracé "à reculons" ....

Application : dessiner (en bleu) le contour de la France.

5.4.2 Dessin des sommets (facile)

Ecrire la fonction `points_plot`, de type `int_graphe -> unit`, telle que `points_plot gr` trace, dans la fenêtre graphique, pour chaque élément $(s,(x,y))$ de la liste `gr.lpos`, un disque centré en (x,y) et de rayon 5.

Cette fonction pourra faire appel à une fonction interne récursive mais ne doit pas utiliser de variable référence.

Application : superposer au graphe précédent, **en** rouge, les points ("yeux") des préfectures de la France.

5.4.3 Dessin des arêtes (assez facile)

1. Ecrire la fonction `trace_to_voisins`, de type `('a * (int * int)) list -> 'a * ('a * 'b) list -> unit`, telle que `trace_to_voisins pos (u,lu)` trace, dans la fenêtre graphique, pour chaque élément `(v,d)` de la liste `lu`, un trait reliant le sommet de numéro `u` au sommet de numéro `v`.
   La liste `pos` contient des éléments de la forme `(s,(x,y))` où `(x,y)` sont les coordonnées géographique du sommet de numéro `s`. La liste `lu` est la liste d'adjacence de `u` et l'élément `d` d'un couple `(v,d)` de `lu` est inutilisé.

   Cette fonction pourra faire appel à une fonction interne récursive mais ne doit pas utiliser de référence.

2. En déduire la fonction `trace_reseau`, de type `int_graphe -> unit`, telle que `trace_reseau gr` trace, dans la fenêtre graphique, tous les traits reliant chaque sommet de `gr` à ses voisins immédiats.

   Cette fonction pourra faire appel à une fonction interne récursive mais ne doit pas utiliser de référence.

Application : superposer au graphe précédent (contour bleu et yeux rouges) les liaisons (en vert) entre les préfectures de la France et leurs voisines.

5.4.4 Dessin global d'un graphe

On en déduit la fonction permettant le dessin global d'un graphe :

```
let trace_int_graphe graph =
   clear_graph ();
   set_color blue;  contour_plot graph;
   set_color green; trace_reseau graph;
   set_color red;   points_plot  graph  (* points en dernier: cache les bouts des traits *)
;;
(* trace_int_graphe : int_graphe -> unit = <fun> *)
```

Application : dessiner le graphe `Hamilton`.

5.4.5 Cadeau : un plus, uniquement pour le graphe France, utilisation de la souris

Le graphe France étant dessiné, après l'exécution de `souris_france ()`, en déplaçant la souris dans la fenêtre graphique, les informations relatives à la préfecture **rouge** située sous le curseur sont éditées, d'après le contenu de la liste `prefectures`, dans le coin gauche du haut de la fenêtre graphique.

La fenêtre graphique se réduit, se déplace, se ferme par les méthodes usuelles et pour reprendre la main, il faut terminer l'exécution de `souris_France ()` en cliquant dans la zone rouge de la fenêtre graphique.

5.4.6 Encore un cadeau : tracé en pointillés d'une chaîne d'arcs fléchés,

Avec la fonction `trace_chaine`, de type `int_graphe -> int list -> unit`, l'appel `trace_chaine_arcs gr ch`, où `ch` est une liste de sommets du graphe `gr`, représentant une chaîne valide, trace en pointillés, sur la fenêtre graphique, les arcs joignants chaque sommet `u` de la liste `ch` à son suivant `v`, comme des segments orientés (par une flèche médiane) de `u` vers `v`.

On pourra ainsi illustrer par le dessin, en superposition avec le dessin du graphe, les chaines ou cycles (orientés) calculés ci-après.

5.5 Chaînes et cycles

5.5.1 List d'adjacence vs matrice d'adjacence

Si on utilise la matrice d'adjacence du graphe, `madj`, il peut y avoir ambiguité selon le codage (0 ou ∞) utilisé pour l'absence de boucle $s \longrightarrow s$. Certains algorithmes matriciels ne marchent (sans acrobaties) qu'avec ∞, d'autres ne marchent qu'avec 0 et il y en a même qui marchent indifféremment avec 0 et ∞. Difficile de faire un choix !
Ensuite, on ne sait plus très bien si `madj.(u).(v)` est la distance de `u` à `v` ou de `v` à `u` ... ce qui est génant dans le cas des graphes orientés non symétriques.

Les choses sont plus claires avec la liste d'adjacence du graphe, puisqu'il n'y a pas de codage pour les boucles inexistantes et que l'identification du sens $u \longrightarrow v$ (v dans la liste d'adjacence de u) ou $v \longrightarrow u$ (u dans la liste d'adjacence de v) semble plus naturelle.

A priori, on convient que si u est énoncé (ou évalué) avant v, il s'agit de l'arc $u \longrightarrow v$ (si on ne considère que des graphes non orientés (bi-orientés symétriques), il y a moins de risques de confusion).

On peut introduire des instructions (ou des fonctions) sécurisées, indépendantes d'un choix fait pour l'absence de boucle, qui fonctionnent comme si ∞ était l'unique codage pour l'absence d'arc ou de boucle :

1. avec la matrice d'adjacence du graphe (`madj`) :

```
is_arete u v = if madj.(u).(v) = 0 then false else madj.(u).(v) < infty
distance u v = if madj.(u).(v) = 0 then infty else madj.(u).(v)
is_commun w u v = (u <> v) && (w <> u) && (w <> v) &&
                  (madj.(u).(w) < infty) && (madj.(w).(v) < infty)   (* u -> w -> v *)
```

2. avec la liste d'adjacence du graphe (`ladj`) :

```
is_arete u v = try let _ = assoc v (assoc u ladj) in true with Not_found -> false
distance u v = try assoc v (assoc u ladj) with Not_found -> infty
is_commun w u v = try let _ = List.assoc w (List.assoc u ladj)       (* u -> w *)
                      and _ = List.assoc v (List.assoc w ladj)       (* w -> v *)
                      in u <> v
                  with Not_found -> false
```

Dans la suite, on utilisera de préférence la liste d'adjacence du graphe.

5.5.2 Longueur en poids (ou valuation) d'une chaîne

Ecrire la fonction `valuation_chaine`, de type `int_graphe -> int list -> int`, telle que `valuation_chaine gr ch`, où `ch` est une liste de sommets représentant une chaine du graphe `gr`, renvoie la somme des poids des arcs de la chaîne `ch`.

Cette fonction doit renvoyer l'exception `Not_Found` si la chaîne `ch` est invalide.

5.5.3 Validation d'un cycle élémentaire/hamiltonien, longueur

Etant donné une séquence $(x_0, x_1, \ldots, x_k)$ de sommets d'un graphe, on cherche à vérifier si cette séquence est un cycle (ou circuit) élémentaire : sommets concécutifs $(x_0 \longrightarrow x_1 \longrightarrow \ldots \longrightarrow x_k)$, fermée $(x_k = x_0)$, passant une et une seule fois par ses sommets $x_1, \ldots, x_k$.

Ecrire la fonction `test_cycle_elementaire`, de type `int_graphe -> int list -> bool`, telle que `test_cycle_elementaire gr cycle` renvoie le booléen `true` si `cycle` est une liste de sommets représentant un cycle élémentaire du graphe `gr`, et le booléen `false` sinon.

indication : on pourra utiliser deux fonctions internes récursives, l'une pour valider l'existence des arêtes (ou arcs) et leur caractère consécutif, l'autre pour valider le passage unique par chaque sommet (**cela suffit-il ?**).

En déduire la fonction `test_cycle_hamiltonien`, de type `int_graphe -> int list -> bool`, telle que `test_cycle_hmiltonien gr cycle` renvoie le booléen `true` si `cycle` est une liste de sommets représentant un cycle hamiltonien du graphe `gr`, et le booléen `false` sinon.

5.6 Recherche d'un cycle hamiltonien

On ne connaît pas d'algorithme polynomial (général) pour décider de l'existence ou pour rechercher un **cycle hamiltonien** (passant une et une seule fois par tous les sommets) d'un graphe fini, et c'est encore plus délicat lorsque l'on cherche un cycle hamiltonien de longueur minimale.

Il s'agit de problèmes **NP complet** : on ne connaît pas d'algorithme polynomial, mais une solution peut être validée en un temps polynomial (en $O(n^2)$ où n est le nombre de sommets) par un algorithme.

La recherche de cycles hamiltoniens (éventuellement de longueur minimale), qui est de complexité exponentielle, est faisable de façon exhaustive quand le nombre de sommets est (très) faible, et sinon ... on fait ce que l'on peut ... (parfois il est possible d'exploiter des particularités du graphe).

5.6.1 Cadeau : recherche exhaustive, pour le graphe Hamilton (OCaml)

Comme le graphe Hamilton a peu de sommets (20), on peut trouver tous ses cycles hamiltoniens par une recherche exhaustive, mais une telle recherche est irréalisable en temps (et espace ?) raisonnable pour le graphe France (90 sommets).

```
let rec print_int_list = function  (* utilitaires de visualiation *)
  | []   -> print_newline ()
  | a::q -> Printf.printf "%4d" a; print_int_list q
and     print_int_int_list = function
  | []   -> print_newline ()
  | a::q -> print_int_list a; print_int_int_list q;;
(* val print_int_list : int list -> unit = <fun> *)
(* val print_int_int_list : int list list -> unit = <fun> *)

let voyage s0 ladj =  (* recherche exhaustive *)
  let nbs = List.length ladj and cycles = ref [] in   (* nombre de sommets; parcours réussis *)
  let rec essai vv = function                (* let rec essai vv la = match la with *)
    | []       -> if (List.length vv >= nbs) &&
                     (List.mem s0 (List.map (function (s,d) -> s) (List.assoc (List.hd vv) ladj)))
                  then cycles := (s0::vv)::(!cycles)  (* fermeture *)
    | (b,d)::q -> essai vv q;
                  if not (List.mem b vv) then essai (b::vv) (List.assoc b ladj)
  in essai [s0] (List.assoc s0 ladj);   (* départ systématique en s0 *)
  !cycles
;;
(* voyage : 'a -> ('a * ('a * 'b) list) list -> 'a list list = <fun> *)

let cycles = voyage 1 ladj_hamilton;;    (* départ systématique en 1 *)
print_int_int_list cycles;;
List.length cycles;;                     (* = 60 solutions *)
```

Justifier la terminaison et la correction de la fonction `voyage`.

Remarque. Pour le graphe Hamilton (20 sommets), voyage est de complexité inférieure à $2^{20} = 1048576$ et pour France (90 sommets), inférieure à $3^{90} = 8.727963568087712e+42 = 872796356808771197066945846595484988840345$.

5.6.2 Recherche d'un seul cycle : méthode spécifique pour le graphe France

Du fait que deux villes voisines ont toujours une autre ville comme voisine commune, le graphe France est un graphe hamiltonien. A l'aide d'un voisin commun, on peut construire un algorithme simple, de complexité linéaire, qui donnera un cycle hamiltonien (mais pas forcément optimal en terme de distance ... loin de là !) :

> dans une chaîne (liste) de villes visitées, on tente d'introduire entre deux villes u et v (reliées par l'arc $u \longrightarrow v$), une ville intermédiaire w qui n'a pas encore été visitée, ce qui, si w existe, conduit à remplacer la séquence u, v par la séquence u, w, v (soit $u \longrightarrow w \longrightarrow v$).

Ainsi, on peut augmenter un cycle élémentaire $[a;a]$, réduit initialement au sommet a (arc "fantôme" $a \longrightarrow a$), jusqu'à obtenir un cycle élémentaire maximal, en nombre d'arcs, donc hamiltonien.

1. Recherche d'une sommet voisin commun à `u` et `v` dans une liste de sommets non visitées `lsnv` :

   Ecrire la fonction `search_new` , de type `int_graphe -> int -> int -> int list -> int` , telle que `search_new gr u v lnv` renvoie un sommet `w`, voisin commun aux sommets voisins `u` et `v` du graphe `gr`, appartenant à la liste de sommets `lnv`, et renvoie l'exception `Not_Found` si un tel voisin n'existe pas.

   Cette fonction pourra faire appel à une fonction interne récursive mais ne doit pas utiliser de variable référence.

2. En utilisant la fonction précédente et la donnée globale `france` :

   Ecrire la fonction `unTourDeFrance` , de type `unit -> int list` , telle que `unTourDeFrance ()`, renvoie une liste représentant un cycle hamiltonien du graphe `france` : `[a;x;...;u;w;v;...;y;a]` correspondant au chemin : `a -> x -> ... -> u -> w -> v -> ... -> y -> a`.

   Cette fonction, peut utiliser une fonction interne récursive, des variables référence, et gérer une exception :

   ```
   try let w = search_new france u v !lsnv
       in ...
   with Not_found -> ...
   ```

3. **Justifier** la terminaison et la correction de `unTourDeFrance ()`
   Tester ! à l'aide de `test_cycle_hamiltonien` et puis par le dessin, avec `trace_chaine`.

5.7 Plus : recherche d'un cycle hamiltonien de longueur (valuation) minimale

A partir d'un ou plusieurs cycles hamiltonien initiaux, de longueurs (en poids) non minimales, on peut chercher à utiliser ces cycles pour en déduire d'autres cycles hamiltoniens qui soient de longueurs inférieures aux précédents.

Remarque. Il n'est pas toujours évident de trouver un ou plusieurs cycles hamiltoniens initiaux et l'amélioration, si elle se produit, pourrait ne pas aboutir à une solution optimale (existence de minima locaux).

5.7.1 Algorithme des fourmis

Des fourmis circulent sur le graphe, de sommet en sommet, en suivant les arcs du graphe.

- Les fourmis qui ont réussi à effectuer un cycle hamiltonien, y déposent suffisamment de phéromones pour inciter leurs congénères à parcourir à leur tour tout ou partie des arcs de ce cycle.
- Des fourmis élitistes renforcent les dépôts de phéromones sur les meilleurs cycles.
- Pour essayer d'éviter de s'enfermer dans des minima locaux,
  - les traces de phéromones s'évaporent au fils du temps,
  - le choix d'un arc par une fourmi ne dépends pas uniquement de la quantité de phéromones qui y est déposée : des fourmis ("aventureuses" ou exploratrices) peuvent être tentées d'emprunter des arcs nouveaux ou non fortement marqués.

Cet algorithme dépends de divers paramètres : nombre de fourmis, amplitude des dépôts, évaporation, ampleur de la contribution des fourmis élitistes, probabilité de choisir un arc plutôt qu'un autre, probabilité d'emprunter un arc faiblement marqué, etc ..., ces paramètres devant être ajustés selon les graphes.

- en fin de tour réussi, dépôts de phéromones et gestion de l'évolution de ces dépôts,
- pendant un parcours, choix aléatoire d'un arc, dépendant des traces de phéromones,
- introduction de perturbations : arcs exclus de l'augmentation des dépôts de phéromones, choix aléatoire d'arcs du meilleur cycle hamiltonien qui seront coupés temporairement ou affaiblis en dépôts de phéromone, etc ... (perturbations qui pourraient éviter de s'enfermer dans des minima locaux).
- éventuellement, mélange de deux approximations, obtenues avec des cycles initiaux différents, pour continuer à partir d'un nouveau cycle initial (ce qui rejoint un peu l'algorithme génétique ci-dessous).

Ce genre d'algorithme peut donner des résultats intéressants (découverte, amélioration notable, pas forcément optimale), même pour des graphes de grande taille,

Dans le cas du graphe `France`, un tel algorithme m'a conduit, en 3-4 heures (sur un modeste vieux PC), à un cycle hamiltonien de longueur 7947, à partir d'un cycle hamiltonien initial de longueur 9865 (découvert par l'algorithme du plus proche voisin commun, `unTourDeFrance`, spécifique à ce graphe, écrit précédemment). NEW ! 7894.

5.7.2 Algorithme génétique (données modifiées génétiquement)

Algorithme qui s'inspire de la théorie de l'évolution : ici, un individu est un cycle hamiltonien, l'ADN de l'individu est la liste des sommets dans l'ordre du parcours, un gène étant le sommet où l'on passe à une certaine position dans le parcours. L'adaptation d'un individu à l'environnement est représentée par l'évaluation de la longueur (en poids) du cycle et on fait des "croisements" et de la "sélection" à l'intérieur de la population :

- on combine deux solutions pour en produire une nouvelle, cette combinaison pouvant
  - subir des mutations (ce qui peut éviter de s'enfermer dans des extrema locaux),
  - être l'objet de réparations de son ADN pour redevenir valide ;
- on élimine les solutions les moins adaptées (attention, il faut quand même laisser des canards boiteux !).

En répétant ce processus, l'adaptation moyenne de la population devrait augmenter et on peut espérer se rapprocher d'une solution optimale du problème. Là encore, il y a de nombreux paramètres à introduire et à ajuster selon les graphes ... .

Ce genre d'algorithme peut aussi donner des résultats intéressants (découverte, amélioration notable, pas forcément optimale), même pour des graphes de grande taille, et, apparemment, à condition de disposer de plusieurs cycles hamiltoniens initiaux, il serait équivalent en termes de performances à l'algorithme des Fourmis.

5.7.3 Autres : Recuit simulé, algorithmes par partitions, ...

WTSP (World Traveling Salesmam Problem) with 1904711-city instance. The tour of length 7515772212 was found on May 24, 2013, by Keld Helsgaun using a variant of his LKH heuristic algorithm. The current best lower bound on the length of a tour for the World TSP is 7512218268, established by the Concorde TSP code (June 5, 2007),

---

$<$ $\mathscr{FIN}$ $>$ (TP Hamilton, énoncé)

6 Annexes

6.1 Annexe 1. Mise en place, déroulement du TP (Caml light, windows)

1. Créer, dans votre répertoire, un dossier `TP-CaML` (pour contenir tous les futurs TPs en Caml)
2. Copier (je dis copier et pas "voler") depuis le dossier `Mpx/OptionInfo/TP-CaML/` le sous-dossier `Hamilton` (et tout son contenu) dans votre dossier `TP-CaML`.

   Contenu du dossier `TP-Caml/Hamilton/`
   - Hamilton-S.pdf (ce document)
   - contour.ml, prefect.ml, liens.ml (fichiers de données pour la France, à incorporer au code)
   - TP-Ham-S.ml (squelette de programme, à compléter)
3. Je déconseille de travailler uniquement avec Camlwin.
   Il est préférable de travailler avec un couple (éditeur, Camlwin).
4. Si vous utilisez CMDCAML (éditeur de texte lié à CamlWin), il y a un problème sur le réseau (confusions dans la résolution de noms) et un problème sous Windows 2000, XP, Vista, Seven (en particulier avec la fonction "Evaluer sélection") :
   - **Attention, il y a un défaut au lancement : "plantage" du lien avec Camlwin !**. Solution :
     1. Dans le fichier "Sans titre 1", écrire `1;;` et l'exécuter, ce qui lance et "plante" Camlwin.
     2. Recommencer l'exécution de `1;;` (et Camlwin n'est plus "planté" ... pour l'instant)
     3. Ouvrir le fichier "TP-Ham-S.ml"
   - **Attention, il y a un défaut en cours d'exécution** : cela peut "replanter" !
     et en cas de replantage, on perd le contenu du fichier en cours (ici TP-Ham-S.ml). Solution :

     Tout d'abord vous devez sauvegarder périodiquement votre fichier !

     1. Fermer (le peu qui reste de) votre fichier "TP-Ham-S.ml", **sans** le sauvegarder
     2. Dans le fichier "Sans titre 1", écrire 1 ; ; et l'exécuter, ce qui peut "dé-planter" Camlwin
     3. Ouvrir le fichier "TP-Ham-S.ml" (dont le contenu est votre ancienne sauvegarde)
   - Ensuite, vous pouvez travailler : écriture dans CMDCAML et exécutions partielles dans Camlwin.
5. Je conseille d'utiliser **un** autre éditeur de texte (par exemple bloc-notes, wordpad) pour écrire le code du programme à partir du fichier "TP-Ham-S.ml".
   Il faut alors aussi lancer séparément Camlwin et faire des "copier-coller" (par petits bouts) de votre code vers Camlwin pour exécutions partielles.

6.2 Annexe 2 : Données du TP

1. "Hamilton" **(Voyage autour du monde)** : listes `ladj_hamilton`, `lpos_hamilton` incluses dans le squelette de programme.
2. "France" **(FRANCE ROUTIERE ET PREFECTURES)** : Evocation du contenu des fichiers à inclure :

```
(*-------------------------------------------------------------------------------*)
(* Contour.ml                                                                    *)
(*   Relevé point par points du contour de la France, selon le sens trigonométrique, *)
(*   depuis le point nord extrême de la France.                                  *)
(*     L'origine du repère est au point (x=0=mini ouest, y=0=mini nord),         *)
(*     l'axe des x est orienté de l'ouest vers l'est (de la gauche vers la droite), *)
(*     l'axe des y est orienté du nord vers le sud (du haut vers le bas).        *)
(*-------------------------------------------------------------------------------*)
let contour = [ ... ];;      (* voir le fichier *)
(*-----------------------------------------------------------------*)
(* Prefect.ml                                                      *) * y compris corse unifiée (20),
(*   Informations sur les départements, préfectures de la France :  *) * dépts 91, 92, 93, 94, 95, ...
(*   (numéro, (département, préfecture, localisation x, y, région)) *)   absorbés par 75, 77, 78.
(*-----------------------------------------------------------------*)
let prefectures = [ ... ];;  (* voir le fichier *)
(*-----------------------------------------------------------------*)
(* Liens.ml                                                        *)
(*   Distances (et liens) entre préfectures voisines               *)
(*  (chaque préfecture est identifiée par son numéro de département)*)
(*-----------------------------------------------------------------*)
let liens =  [ ... ];;       (* voir le fichier *)
```

6.3 Annexe 3. Squelette de programme (Caml light)

```
(*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*)
(*          ENVIRONNEMENT                *)
(*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*)
#open "graphics";;    (* bibliothèque Caml de ressources graphique ET souris *)
#open "printf";;      (* bibliothèque Caml des éditions formatées *)

(*-------------------- Chemin des fichiers du TP -----------------------*)
#directory "MPx\InfoMP\TP-info\Hamilton";;                               (* <=== A ADAPTER *)

let infty = max_int;;

(*=== Type des graphes valués (non orientés) utilisés ici            ===*)
(*           sommets = entiers consécutifs depuis 1, valuation = entier *)
type int_graphe =
  { ladj : (int * (int * int) list) list; (* liste d'adjacence  *)
    madj : int vect vect;                 (* matrice d'adjacence (bordée) *)
    lpos : (int * (int * int)) list;      (* liste des positions graphiques *)
    lbox : (int * int) list               (* ligne d'entourage *)
  }
;;
(*===  Edition formatée (5 places) d'une matrice d'entiers ===*)
let print_int_matrix m =
  print_newline ();
  for i = 0 to vect_length m -1
  do for j = 0 to vect_length m.(0) -1
     do if (m.(i).(j) = infty) or  (m.(i).(j) = -infty)
        then printf "    %c" (char_of_int 64)              (* avec 4 espaces; simule infini par @ *)
        else printf "%5d" m.(i).(j)                       (* Printf.printf avec OCaml *)
     done;
     print_newline ()
  done
;;
(* print_int_matrix : int vect vect -> unit  *)

(*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*)
(*          REALISATION DU TP                *)      (*=========  INCORRECT A PARTIR D'ICI ======*)
(*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*)
(*=====================================================*)
(*== Le TP.1. CONSTRUCTION DE LA MATRICE D'ADJACENCE ===*)
(*=====================================================*)
(* matrice d'adjacence (bordée) depuis la liste d'adjacence *)

let matrice_adj_of_liste_adj ladj =  (* ladj -> madj *)
  let n = list_length ladj in
  ....                                                                 (* 6-7 lignes A ECRIRE *)
;;
(* matrice_adj_of_liste_adj : (int * (int * int) list) list -> int vect vect *)

(*== Le TP.2. Exemples (données initiales) ===*)
(*-------------------------------------------*)
(* Exple1: Voyage autour du monde de Hamilton *)
(*-------------------------------------------*)
let ladj_hamilton =  (* sommet -> liste de (sommet adjacent, distance) *)
[  1, [ 2,1;  5,1;  8,1];   2, [ 1,1;  3,1; 10,1];   3, [ 2,1;  4,1; 12,1];   4, [ 3,1;  5,1; 14,1];
   5, [ 1,1;  4,1;  6,1];   6, [ 5,1;  7,1; 15,1];   7, [ 6,1;  8,1; 17,1];   8, [ 1,1;  7,1;  9,1];
   9, [ 8,1; 10,1; 18,1];  10, [ 2,1;  9,1; 11,1];  11, [10,1; 12,1; 19,1];  12, [ 3,1; 11,1; 13,1];
  13, [12,1; 14,1; 20,1];  14, [ 4,1; 13,1; 15,1];  15, [ 6,1; 14,1; 16,1];  16, [15,1; 17,1; 20,1];
  17, [ 7,1; 16,1; 18,1];  18, [ 9,1; 17,1; 19,1];  19, [11,1; 18,1; 20,1];  20, [13,1; 16,1; 19,1] ]
;;
let lpos_hamilton = (* sommet -> (x,y) coordonnées *)
  [ 1,( 50,340);  2,(210,620);  3,(450,620);  4,(610,340);  5,(330,180); 6,(330,260);  7,(210,300);
    8,(170,380);  9,(210,460); 10,(250,540); 11,(330,540); 12,(410,540); 13,(450,460); 14,(490,380);
   15,(450,300); 16,(370,340); 17,(290,340); 18,(250,420); 19,(330,460); 20,(410,420) ]
;;
```

```
(*-----------------------------------------------------------------*)
(* Exple2: PREFECTURES DE LA FRANCE (de 1 à 90, dont corse = 20) *)
(*-----------------------------------------------------------------*)
(*--------------------- liste de coordonnées : ligne polygonale pour le contour de la France ---*)
include "CONTOUR.ML";; (* let contour = [...(x,y)...] : (int * int) list *)
(*--------------------- liste d'identification des préfectures ---*)
include "PREFECT.ML"   (* let prefectures = [...] : (int * (string * string * int * int * string)) list *)
;;                     (* (n dep, (nom département, nom préfecture, localisation x, y, nom région)) *)
(*--------------------- liste valuée d'adjacence du graphe ---*)
include "LIENS.ML"     (* let liens = [ ... ] : (int * (int * int) list) list *)
;;                     (* (n dep , [... ;(ndep voisin, dist); ...] )  *)
(*===================================*)
(*== Le TP.3. CODAGE DES EXEMPLES ===*)
(*===================================*)
(*-- x.3.1. ------------------*)
let hamilton =  ........                                                    (* A ECRIRE *)

(*-- x.3.2. ------------------*)
let prefecture_of_dep n =
    ..................                                                  (* 1 ligne A ECRIRE *)
;;
(* prefecture_of_dep : int -> string = <fun> *)
prefecture_of_dep 48;;                              (* test *)

let france = ........                                                       (* A ECRIRE *)

(*================================*)
(*== Le TP.4. DESSIN DE GRAPHES ===*)
(*================================*)
(*-- x.4.1. Tracé du contour (ligne polygonale) ---*)
let contour_plot graph =
  ....                                                                  (* 5 lignes A ECRIRE *)
;;
(* contour_plot : int_graphe -> unit *)

(*-- x.4.2. Placement des sommets (point par point) ---*)
let points_plot graph =
  ....                                                                  (* 5 lignes A ECRIRE *)
;;     (* points_plot : int_graphe -> unit *)

(*-- x.4.3. Tracé des arêtes (segments) ---*)
(* positions = liste de (sommet s, couple de coordonnées x,y)
   u = sommet de départ, lu = liste des sommets adjacents à u, avec distance
*)
let trace_to_voisins positions (u,lu) = (* aretes de u à ses voisins *)
  ....                                                                  (* 6 lignes A ECRIRE *)
;;
(* trace_to_voisins : ('a * (int * int)) list -> 'a * ('a * 'b) list -> unit *)

let trace_reseau graph =        (* toutes les arêtes *)
  ....                                                                  (* 5 lignes A ECRIRE *)
;;     (* trace_reseau : int_graphe -> unit = <fun> *)

(*-- x.4.4. Tracé global d'un graphe ---*)
let trace_int_graphe graph =
   clear_graph ();
   set_color blue;  contour_plot graph;
   set_color green; trace_reseau graph;
   set_color red;   points_plot  graph
;;
(* trace_int_graphe : int_graphe -> unit = <fun> *)

(*---------------------------------- tests ---*)
open_graph " 800x800";;
trace_int_graphe hamilton;;
trace_int_graphe france;;
```

```
(*-------------------------------------------------------------------*)
(*-- x.4.5. CADEAU. Interaction de la souris avec France:          *)
(*                  identification de la préfecture sous la souris *)
(*-------------------------------------------------------------------*)
let rec get mx my = function
  | []                    -> 0, "", "", ""
  | (n,(d,p,x,y,r)) :: q -> if abs(mx-x)+abs(my-y) <= 6 then n, d, p, r else get mx my q
;;
(* get:int->int-> (int * (string * string * int * int * string)) list -> int * string * string * string *)

let souris_france () =
  set_color red;
  fill_rect 10 (size_y ()-600) 180 30;
  set_color black;
  moveto 20 (size_y () - 595); draw_string "Click here to finish";

  let stop = ref false in
  while (not !stop)
  do let sta = wait_next_event [Mouse_motion; Button_down] in
     let x = sta.mouse_x and y = sta.mouse_y in
     stop := sta.button && (x > 10) && (x < 200)
                        && (y > (size_y ()-600)) && (y < (size_y ()-570)) ;
     let n,d,p, r = if !stop or (not (mem (point_color sta.mouse_x sta.mouse_y) [red; magenta]))
                    then 0, "", "", ""
                    else get sta.mouse_x sta.mouse_y prefectures in
     if n = 0
     then begin set_color white;
                fill_rect 10 (size_y ()-60) 240 360
          end
     else begin set_color black;
                moveto 10 (size_y () - 20); draw_string p;
                moveto 10 (size_y () - 40); draw_string (sprintf "%2s %s" (string_of_int n) d);
                                      (* draw_string (Printf.sprintf "%2s %s" (string_of_int n) d); *)
                moveto 10 (size_y () - 60); draw_string r
          end
  done;
  set_color white;
  fill_rect 10 (size_y ()-600) 180 30;
  set_color black;
  let _ = wait_next_event [Button_up] in ()    (* il faut "vider" l'événement! *)
;;
(* souris_France  : unit -> unit *)

(*---------------------------------- tests ---*)
trace_int_graphe france;;
souris_france ();;               (* cliquer sur zone rouge pour terminer *)

(*-------------------------------------------------------------------------------*)
(*-- x.4.6. CADEAU : Tracé d'une chaîne d'arcs; de sommets, en pointillés fléchés ---*)
(*-------------------------------------------------------------------------------*)
let trait_pointille (xa,ya) (xb,yb) =
  let x1 = float_of_int xa and y1 = float_of_int ya
  and x2 = float_of_int xb and y2 = float_of_int yb in
  let vx = x2 -. x1 and vy = y2 -. y1 in
  let L = sqrt (vx *. vx +. vy *. vy) in
  let unx = vx /. L and uny = vy /. L in  (* vecteur unitaire dirigeant la flèche *)
  let im = int_of_float (L /. 12.) in     (* 12 = 5 + 7 *)
  let x = ref x1 and y = ref y1 in
  for k = 1 to im
  do moveto (int_of_float !x) (int_of_float !y);
     x := !x +. 5. *. unx; y := !y +. 5. *. uny;  lineto (int_of_float !x) (int_of_float !y);
     x := !x +. 7. *. unx; y := !y +. 7. *. uny  (* préparation suivant *)
  done
;;
(* trait_pointille : int * int -> int * int -> unit = <fun> *)
```

```
let fleche (xa,ya) (xb, yb) =   (* contour polygonal d'une flèche centrée de A -> B *)
  let xxa = float_of_int xa and yya = float_of_int ya
  and xxb = float_of_int xb and yyb = float_of_int yb
  in let n = sqrt( (xxb -. xxa)**2.0 +. (yyb -. yya)**2.0 )
  in let xv = (xxb -. xxa) /. n and yv = (yyb -. yya) /. n
  in let xm =  (xxb +. xxa) /. 2.0 and ym = (yyb +. yya) /. 2.0
  in let fw = 4.0 and fl = 12.0                                    (* ferron 1/2 width et long *)
  in map_vect (function (x,y) -> (int_of_float (floor (x +. 0.5)), int_of_float (floor (y +. 0.5)) ))
       [| (xm +. yv *. fw, ym -. xv *. fw);  (xm -. yv *. fw, ym +. xv *. fw);
          (xm +. xv *. fl, ym +. yv *. fl);  (xm +. yv *. fw, ym -. xv *. fw) |]
;;
(* val fleche : int * int -> int * int -> (int * int) vect = <fun> *)

let trace_chaine graph ls =     (* ls = liste de sommets *)
  let rec trace_seg = function
      []      -> ()
    | [a]     -> let xa,ya = assoc a graph.lpos in draw_circle xa ya 6
    | a::b::r -> trace_seg (b::r);
                 let xa,ya = assoc a graph.lpos and xb,yb = assoc b graph.lpos   (* récup coord *)
                 in trait_pointille (xa,ya) (xb,yb);            (* moveto xa ya; lineto xb yb; *)
                    fill_poly (fleche (xa,ya) (xb, yb));
  in trace_seg ls
;;
(* trace_chaine : int_graphe -> int list -> unit = <fun> *)

(*==================================*)
(*== Le TP.5. CHAINES ET CYCLES ===*)
(*==================================*)
(*-- x.5.1. ----------------------*)

(*-- x.5.2. Longueur (valuation) d'une chaîne ---*)
let valuation_chaine graph ch =
      ...                          (* 3-4 lignes, avec une fonction récursive auxiliaire, A ECRIRE *)
;;
(* valuation_chaine : int_graphe -> int list -> int *)

(*-- x.5.3. Validation d'un cycle hamiltonien ---*)
let test_cycle_elementaire graph cycle =       (* cycle = [a0;..;u; v;..;a0] (u -> v) *)
  let a0 = hd cycle and cs = tl cycle      (* départ et sommets successifs du cycle, non fermé *)
  in  ....                          (* 2 fonctions auxiliaires et 1 ligne d'exploitation A ECRIRE *)
;;
(* test_cycle_elementaire : int_graphe -> int list -> bool *)

let test_cycle_hamiltonien graph cycle =
     ...                                                                  (* 1 ligne A ECRIRE *)
;;
(* test_cycle_hamiltonien : int_graphe -> int list -> bool *)

(*---------------------------------- test ---*)
let vr = [17; 7; 6; 15; 16; 20; 13; 14; 4; 5; 1; 8; 9; 10; 2; 3; 12; 11; 19; 18; 17];;
test_cycle_hamiltonien hamilton vr;;
valuation_chaine hamilton vr;;
test_cycle_hamiltonien hamilton (List.tl vr);;

(*=============================================*)
(*== Le TP.6. RECHERCHE D'UN CYCLE HAMILONIEN ===*)
(*=============================================*)
(*-- x.6.1. Hamilton: recherche exhaustive ---*)
let rec print_int_list = function  (* utilitaires de visualiation *)
  | []   -> print_newline ()
  | a::q -> Printf.printf "%4d" a; print_int_list q
and     print_int_int_list = function
  | []   -> print_newline ()
  | a::q -> print_int_list a; print_int_int_list q
;;
(* val print_int_list : int list -> unit = <fun> *)
(* val print_int_int_list : int list list -> unit = <fun> *)
```

```
let voyage s0 ladj =    (* recherche exhaustive *)
  let nbs = length ladj (* nombre de sommets et parcours réussis *)
  and cycles = ref [] in
  let rec essai vv = function  (* let rec essai vv la = match la with *)
    | []       -> if (length vv >= nbs) &&
                     (mem s0 (map (function (s,d) -> s) (assoc (hd vv) ladj)))
                  then cycles := (s0::vv)::(!cycles)  (* fermeture *)
    | (b,d)::q -> essai vv q;
                  if not (mem b vv) then essai (b::v) (assoc b ladj)
  in essai [s0] (assoc s0 ladj);   (* départ systématique en s0 *)
  !cycles;;
(* voyage : 'a -> ('a * ('a * 'b) list) list -> 'a list list = <fun> *)

let cycles = voyage 1 ladj_hamilton;;           (* départ systématique en 1 *)
print_int_int_list cycles;;
length cycles;;                                 (* = 60 solutions *)
test_cycle_hamiltonien hamilton (hd cycles);;

(*=====================================================*)
(*-- x.6.2. Méthode spécifique pour le graphe france. ---*)
(*=====================================================*)
(*           Ici, il y a toujours un voisin commun à deux sommets *)
(*- 1. Recherche d'un sommet voisin commun à u et v dans lnv liste de sommets non visités *)
let search_new graph u v lnv =       (* stratégie: premier trouvé *)
  let rec search_nouv = function
    | []   -> raise Not_found        (* ou valeur invalide *)
    | w::q -> ....                                          (* quelques instructions A ECRIRE *)
  in search_nouv lnv
;;
(* search_new : int_graphe -> int -> int -> int list -> int = <fun> *)

search_new france 37 36 [47;42;41;43];;   (* 41 *)
search_new france 37 37 [47;42;41;43];;   (* 41 *)
search_new france 37 75 [47;41;42;43];;   (* Exception: Not_found. *)
search_new france 137 36 [47;42;41;43];;  (* Exception: Not_found. *)

(*- 2. Fonction utilisant le graphe France défini ci-dessus *)
let unTourDeFrance () =   (* cycle hamiltonien de a vers a *)
   ....                                   (* 12-13 lignes avec une fonction auxiliaire, A ECRIRE *)
;;
(* unTourDeFrance : unit -> int list = <fun> *)

let vr = unTourDeFrance ();;
test_cycle_hamiltonien france vr;;
valuation_chaine france vr;;
trace_int_graphe france;;
set_color black;;
trace_chaine france vr;;
souris_france ();;

(*=============================================================================================*)
(*== X.7. PLUS: amélioration d'un cycle hamiltonien de longueur 9865, par un algorithme de Fourmis *)
(*=============================================================================================*)
let abest7894 = (* Un bon cycle hamiltonien pour la France, de longueur 7894 *)
[46;47;82;81;12;48;43;7;26;42;69;71;1;74;73;38;5;4;6;20;83;13;84;30;34;66;11;9;31;32;65;64;
 40;33;24;16;86;79;17;85;44;56;29;22;35;53;49;72;61;50;14;27;76;60;80;62;59;8;2;51;55;57;54;
 88;67;68;90;70;25;39;21;52;10;89;77;75;78;28;45;41;37;36;18;58;3;63;23;87;19;15;46];;

test_cycle_hamiltonien france abest7894;;  (* true *)
valuation_chaine france abest7894;;        (* 7894 *)
trace_int_graphe france;;
set_color black;;
trace_chaine france abest7894;;
souris_france ();;
```

7 TP Hamilton. Corrigé (OCaml)

```
(* OCaml 26/11/2017 depuis camllight *)
#load "graphics.cma";;
open Graphics;;
#directory "MPx\InfoMP\TP-info\Hamilton";;  (* <----- Chemin des fichiers du TP, A ADAPTER *)

(*=== ................. ===*)
let infty = max_int;;
let format = Printf.sprintf                                    (* raccourci.. *)
;; (* val format : ('a, unit, string) format -> 'a = <fun> *)

(*=== Type des graphes valués (non orientés = bi-orientés) utilisés ici      ===*)
type int_graphe =           (* sommets = entiers consécutifs depuis 1, valuation = entier *)
  { ladj : (int * (int * int) list) list; (* liste d'adjacence  *)
    madj : int array array;               (* matrice d'adjacence (bordée) *)
    lpos : (int * (int * int)) list;      (* liste des positions graphiques *)
    lbox : (int * int) list  }            (* ligne d'entourage *)
;;
(*=== Edition formatée (5 places) d'une matrice d'entiers ===*)
let print_int_matrix m =
  print_newline ();
  for i = 0 to Array.length m -1
  do for j = 0 to Array.length m.(0) -1
     do if (m.(i).(j) = infty) ||  (m.(i).(j) = -infty)
        then Printf.printf "    %c" (char_of_int 64)     (* 4 espaces + simule infini par @ *)
        else Printf.printf "%5d" m.(i).(j)
     done;
     print_newline ()
  done
;;     (* val print_int_matrix : int array array -> unit  *)

let copy_matrix m =
  let p = Array.length m and q = Array.length m.(0) in
  let c = Array.make_matrix p q m.(0).(0) in
  for i = 0 to p-1 do for j = 0 to q-1 do c.(i).(j) <- m.(i).(j) done done;
  c
;;     (* val copy_matrix : 'a array array -> 'a array array = <fun> *)

let is_symetric m =        (* test de mise au point *)
  let b = ref true and n = Array.length m in
  for i = 0 to n-1
  do for j = 0 to n-1
     do if m.(i).(j) <> m.(j).(i)
        then if !b then begin b := false; Printf.printf "%3d,%3d" i j end
     done
  done;
  !b
;;      (* val is_symetric : 'a array array -> bool = <fun> *)

(*=========================================================*)
(*== X.1. CONSTRUCTION DE LA MATRICE D'ADJACENCE (bordée) ===*)
(*=========================================================*)
let matrice_adj_of_liste_adj ladj =  (* ladj -> madj *)
  let n = List.length ladj in
  let m = Array.make_matrix (1+n) (1+n) (infty) in
  for i = 0 to n do m.(0).(i) <- i; m.(i).(0) <- i done;     (* bords = numéros sommets *)

  let rec remplit = function               (* m.(i).(j) = dist(i,j) *)
    | []      -> ()
    |(v,lv)::q -> remplit q; List.iter (function (u,d) -> m.(v).(u) <- d) lv
  in
  remplit ladj;
  for i = 1 to n do if m.(i).(i) = infty then m.(i).(i) <- 0 done; (* si pas de boucle *)
  m
;;
(* matrice_adj_of_liste_adj : (int * (int * int) list) list -> int aray array *)
```

```
(*=========================================================*)
(*== X.2. + X.3. Exemples (données initiales) + Codage ===*)
(*=========================================================*)
(*-----------------------------------------------*)
(* Exple1.a: Voyage autour du monde de Hamilton *)
(*-----------------------------------------------*)
let ladj_hamilton =  (* sommet -> liste de (sommet adjacent, distance) *)
[  1, [ 2,1;  5,1;  8,1];   2, [ 1,1;  3,1; 10,1];   3, [ 2,1;  4,1; 12,1];   4, [ 3,1;  5,1; 14,1];
   5, [ 1,1;  4,1;  6,1];   6, [ 5,1;  7,1; 15,1];   7, [ 6,1;  8,1; 17,1];   8, [ 1,1;  7,1;  9,1];
   9, [ 8,1; 10,1; 18,1];  10, [ 2,1;  9,1; 11,1];  11, [10,1; 12,1; 19,1];  12, [ 3,1; 11,1; 13,1];
  13, [12,1; 14,1; 20,1];  14, [ 4,1; 13,1; 15,1];  15, [ 6,1; 14,1; 16,1];  16, [15,1; 17,1; 20,1];
  17, [ 7,1; 16,1; 18,1];  18, [ 9,1; 17,1; 19,1];  19, [11,1; 18,1; 20,1];  20, [13,1; 16,1; 19,1] ];;
let lpos_hamilton = (* sommet -> (x,y) coordonnées *)
  [ 1,( 50,340);  2,(210,620);  3,(450,620);  4,(610,340);  5,(330,180);   6,(330,260);  7,(210,300);
    8,(170,380);  9,(210,460); 10,(250,540); 11,(330,540); 12,(410,540);  13,(450,460); 14,(490,380);
   15,(450,300); 16,(370,340); 17,(290,340); 18,(250,420); 19,(330,460);  20,(410,420) ]
;;
let hamilton = { ladj = ladj_hamilton;  madj = matrice_adj_of_liste_adj ladj_hamilton;
                 lpos = lpos_hamilton; lbox = [] };;

(*---------------------------------------*)
(* Exple1.b: Petersen (non hamiltonien) *)
(*---------------------------------------*)
let ladj_petersen =  (* sommet -> liste de (sommet adjacent, distance) *)
  [ 1,[ 2,1;  5,1;  7,1]; 2,[ 1,1;  3,1;  8,1]; 3,[ 2,1;  4,1;  9,1];
    4,[ 3,1;  5,1; 10,1]; 5,[ 1,1;  4,1;  6,1]; 6,[ 8,1;  9,1];
    7,[ 9,1; 10,1]; 8,[ 6,1; 10,1]; 9,[ 7,1; 6,1]; 10, [ 7,1;  8,1]];;
let lpos_petersen = (* sommet -> (x,y) coordonnées *)
  [ 1,(130,500); 2,(330,620); 3,(530,500); 4,(450,260); 5,(210,260);
    6,(290,340); 7,(250,460); 8,(330,540); 9,(410,460); 10,(370,340)]
;;
let petersen = { ladj = ladj_petersen; madj = matrice_adj_of_liste_adj ladj_petersen;
                 lpos = lpos_petersen; lbox = [] };;

(*---------------------------------*)
(* Exple1.c: Chvatal (hamiltonien) *)
(*---------------------------------*)
let ladj_chvatal =  (* sommet -> liste de (sommet adjacent, distance) *)
[ 1, [ 2,1; 4,1;  5,1; 12,1]; 2, [ 1,1; 3,1;  6,1;  7,1]; 3, [ 2,1; 4,1;  8,1;  9,1];
  4, [ 3,1; 1,1; 10,1; 11,1]; 5, [ 1,1; 6,1;  8,1;  9,1]; 6, [ 2,1; 5,1; 10,1; 11,1];
  7, [ 2,1; 8,1; 10,1; 11,1]; 8, [ 3,1; 7,1;  5,1; 12,1]; 9, [ 3,1; 5,1; 10,1; 12,1];
 10, [ 4,1; 9,1;  6,1;  7,1]; 11, [ 4,1; 6,1; 12,1;  7,1]; 12, [ 1,1; 8,1; 11,1;  9,1] ];;
let lpos_chvatal = (* sommet -> (x,y) coordonnées *)
[ 1, (100, 500);  2, (500, 500); 3, (500, 100);  4, (100, 100);  5, (250, 400);  6, (350, 400);
  7, (400, 350);  8, (400, 250); 9, (350, 200); 10, (250, 200); 11, (200, 250); 12, (200,350)]
;;
let chvatal = { ladj = ladj_chvatal;  madj = matrice_adj_of_liste_adj ladj_chvatal;
                lpos = lpos_chvatal; lbox = [] };;

(*---------------------------------------------------------*)
(* Exple2: France, préfectures (de 1 à 90, dont corse=20) *)
(*---------------------------------------------------------*)
(*------------------- liste de coordonnées : ligne polygonale pour le contour de la France ---*)
uses "CONTOUR.ML";;  (* let contour = [...(x,y)...] : (int * int) list *)
(*------------------- liste d'identification des préfectures ---*)
uses "PREFECT.ML"    (* let prefectures = [...] : (int * (string * string * int * int * string)) list *)
;;                   (* (n dep, (nom département, nom préfecture, localisation x, y, nom région)) *)
(*------------------- liste valuée d'adjacence du graphe ---*)
uses "LIENS.ML"      (* let liens = [ ... ] : (int * (int * int) list) list *)
;;                   (* (n dep , [... ;(ndep voisin, dist); ...] )  *)
let france =  { ladj = liens; madj = matrice_adj_of_liste_adj liens;
    lpos = List.map (function (n,(d,p,x,y,r)) -> (n, (x,y)) ) prefectures; lbox = contour }
;;
let prefecture_of_dep n = let d,p,x,y,r = List.assoc n prefectures in p
;;
(* prefecture_of_dep : int -> string = <fun> *)
prefecture_of_dep 48;;
```

```
(*============================*)
(*== X.4. DESSIN DE GRAPHES ===*)
(*============================*)
(*-- X.4.1. Tracé du contour (ligne polygonale) ---*)
let contour_plot graph =
  let rec trace_contour = function
    | [] -> ()
    | [(x,y)]  -> moveto x y                    (* au "dernier", on se positionne .. *)
    | (x,y)::q -> trace_contour q; lineto x y   (* à reculons *)
  in trace_contour graph.lbox
;;
(* contour_plot : int_graphe -> unit *)

(*-- X.4.2. Placement des sommets (point par point) ---*)
let points_plot graph =
  let rec points_place = function
    | [] -> ()
    | (n,(x,y))::q -> points_place q;
                      fill_circle x y 5;
                      (* let s = string_of_int n in moveto (x+5) y; draw_string s *)
  in points_place graph.lpos
;;
(* points_plot : int_graphe -> unit *)

(*-- X.4.3. Tracé des arêtes (segments), avec "rond médian" ---*)
(*          positions = liste de (sommet s, couple de coordonnées x,y) *)
let trace_to_voisins positions (u,lu) =    (* aretes de u à chacun de ses voisins *)
  let xu,yu = List.assoc u positions in
  let rec aux = function
     [] -> ()
    | (v,_)::r -> let xv,yv = List.assoc v positions
                  in moveto xu yu; lineto xv yv; draw_circle ((xu+xv)/2) ((yu+yv)/2) 3;
                  aux r
  in
  aux lu
;;
(* trace_to_voisins : ('a * (int * int)) list -> 'a * ('a * 'b) list -> unit *)

let trace_reseau graph =        (* arêtes de chaque sommet à ses voisins (redondant) *)
   let rec trace_reso = function
     | []   -> ()
     | h::q -> trace_to_voisins graph.lpos h;
               trace_reso q
   in trace_reso graph.ladj
;;
(* trace_reseau : int_graphe -> unit = <fun> *)

(*-- X.4.4. Tracé global d'un graphe ---*)
let trace_int_graphe graph =
   clear_graph ();
   set_color blue; contour_plot graph;
   set_color green; trace_reseau graph;
   set_color red; points_plot graph
;;
(* trace_int_graphe : int_graphe -> unit = <fun> *)

(*-- tests ----------------------*)
open_graph " 800x800";;
trace_int_graphe hamilton;;
trace_int_graphe petersen;;
trace_int_graphe chvatal;;
trace_int_graphe france;;
```

```
(*-------------------------------------------------------------------------------------*)
(*-- X.4.5. CADEAU: Souris pour le graphe France                                       *)
(*   Interaction de la souris avec France: Identification de la préfecture sous la souris *)
(*-------------------------------------------------------------------------------------*)
let rec get_france mx my = function
  | []                   -> 0, "", "", ""
  | (n,(d,p,x,y,r)) :: q -> if abs(mx-x)+abs(my-y) <= 6 then n, d, p, r else get_france mx my q
;;
(* get_france: int->int-> (int*(string*string*int*int*string)) list -> int*string*string*string *)

let souris_france () =
  set_color red;    fill_rect 10 (size_y ()-600) 180 30;
  set_color black;  moveto 20 (size_y () - 595); draw_string "Click here to finish";
  let stop = ref false in
  while (not !stop)
  do let sta = wait_next_event [Mouse_motion; Button_down] in
     let x = sta.mouse_x and y = sta.mouse_y in
     stop := sta.button && (x > 10) && (x < 200)
                        && (y > (size_y ()-600)) && (y < (size_y ()-570)) ;
     let n,d,p,r = if !stop or (not (List.mem (point_color sta.mouse_x sta.mouse_y) [red; magenta]))
                   then 0, "", "", ""
                   else get_france sta.mouse_x sta.mouse_y prefectures in
     if n = 0 then begin set_color white; fill_rect 10 (size_y ()-60) 240 360 end
     else begin set_color black;
                moveto 10 (size_y () - 20); draw_string p;
                moveto 10 (size_y () - 40); draw_string (Printf.sprintf "%2s %s" (string_of_int n) d);
                moveto 10 (size_y () - 60); draw_string r
          end
  done;
  set_color white;  fill_rect 10 (size_y ()-600) 180 30;  set_color black;
  let _ = wait_next_event [Button_up] in ()     (* il faut "vider" l'événement! *)
;;
(* souris_france  : unit -> unit *)
(*------------------------------------------------------------ tests ---*)
open_graph " 800x800+10+10";;
trace_int_graphe france;;
souris_france ();;
(*---------------------------------------------------------------------*)
(*-- X.4.6. CADEAU : Tracé d'une chaîne d'arcs pointillés fléchés ---*)
(*---------------------------------------------------------------------*)
let trait_pointille (xa,ya) (xb,yb) =
  let x1 = float_of_int xa and y1 = float_of_int ya
  and x2 = float_of_int xb and y2 = float_of_int yb in
  let vx = x2 -. x1 and vy = y2 -. y1 in
  let L = sqrt (vx *. vx +. vy *. vy) in
  let unx = vx /. L and uny = vy /. L in  (* vecteur unitaire dirigeant la flèche *)
  let im = int_of_float (L /. 12.) in     (* 12 = 5 + 7 *)
  let x = ref x1 and y = ref y1 in
  for k = 1 to im
  do moveto (int_of_float !x) (int_of_float !y);
     x := !x +. 5. *. unx; y := !y +. 5. *. uny; lineto (int_of_float !x) (int_of_float !y);
     x := !x +. 7. *. unx; y := !y +. 7. *. uny  (* préparation suivant *)
  done
;;      (* trait_pointille : int * int -> int * int -> unit = <fun> *)

let fleche (xa,ya) (xb, yb) =   (* contour polygonal d'une flèche centrée de A -> B *)
  let xxa = float_of_int xa and yya = float_of_int ya
  and xxb = float_of_int xb and yyb = float_of_int yb
  in let n = sqrt( (xxb -. xxa)**2.0 +. (yyb -. yya)**2.0 )
  in let xv = (xxb -. xxa) /. n and yv = (yyb -. yya) /. n
  in let xm =  (xxb +. xxa) /. 2.0 and ym = (yyb +. yya) /. 2.0
  in let fw = 4.0 and fl = 12.0                                  (* ferron 1/2 width et long *)
  in Array.map (function (x,y) -> (int_of_float (floor (x +. 0.5)), int_of_float (floor (y +. 0.5)) ))
       [| (xm +. yv *. fw, ym -. xv *. fw);  (xm -. yv *. fw, ym +. xv *. fw);
          (xm +. xv *. fl, ym +. yv *. fl);  (xm +. yv *. fw, ym -. xv *. fw) |]
;;
(* val fleche : int * int -> int * int -> (int * int) array = <fun> *)
```

```
let trace_chaine graph ls =  (* ls = liste de sommets concécutifs *)
  let rec trace_seg = function
      []      -> ()
    | [a]     -> let xa,ya = List.assoc a graph.lpos in draw_circle xa ya 6
    | a::b::r -> trace_seg (b::r);
                 let xa,ya = List.assoc a graph.lpos
                 and xb,yb = List.assoc b graph.lpos   (* coords *)
                 in trait_pointille (xa,ya) (xb,yb);   (* moveto xa ya; lineto xb yb; *)
                    fill_poly (fleche (xa,ya) (xb, yb));
  in trace_seg ls
;;
(* trace_chaine : int_graphe -> int list -> unit = <fun> *)

(*================================*)
(*== Le TP.5. CHAINES ET CYCLES ===*)
(*================================*)
(*-- X.5.1. ----------------------*)

(*-- X.5.2. Longueur (valuation) d'une chaîne (-> Not_found si invalide) ---*)
let valuation_chaine graph ch =  (* ch = chaine = liste de sommets *)
  let rec long_chaine = function
       []      -> 0
     | [a]     -> 0
     | a::b::q -> (List.assoc a (List.assoc b graph.ladj)) + long_chaine (b::q)
  in long_chaine ch
;;
(* valuation_chaine_arcs : int_graphe -> int list -> int *)

(*-- X.5.3. Validation d'un cycle élémentaire/hamiltonien ---*)
let test_cycle_elementaire graph cycle =     (* cycle = liste sommets a,..,u,v,..,a (arc u->v) *)
  let a0 = List.hd cycle
  and cs = List.tl cycle in
  let rec test_arcs = function
    | []      -> true
    | [a]     -> a = a0
    | a::b::q -> let d = try List.assoc a (List.assoc b graph.ladj) with Not_found -> infty
                 in (d < infty) && (test_arcs (b::q))
  and     test_sommets = function
    | []   -> true
    | a::q -> (not (List.mem a q)) && (test_sommets q)
  in
  (test_arcs cs) && (test_sommets cs)
;;
(* test_cycle_elementaire : int_graphe -> int list -> bool *)

let test_cycle_hamiltonien graph cycle =
  (List.length graph.ladj = (List.length cycle)-1) && (test_cycle_elementaire graph cycle)
;;
(* val test_cycle_hamiltonien : int_graphe -> int list -> bool = <fun> *)

(*=========================================*)
(*== X.6. RECHERCHE D'UN CYCLE HAMILTONIEN ===*)
(*=========================================*)
(*=============================================================================*)
(*-- X.6.1. Recherche exhaustive (de tous les cycles hamiltoniens): Hamilton, Chvatal ---*)
(*=============================================================================*)
let rec print_int_list = function
  | []   -> print_newline ()
  | a::q -> Printf.printf "%4d" a; print_int_list q
and     print_int_int_list = function
  | []   -> print_newline ()
  | a::q -> print_int_list a; print_int_int_list q
;;
(* val print_int_list : int list -> unit = <fun> *)
(* val print_int_int_list : int list list -> unit = <fun> *)
```

```
let voyage s0 ladj =         (* recherche exhaustive *)
  let nbs = List.length ladj (* nombre de sommets *)
  and cycles = ref []        (* cycles (liste de listes de sommets) *)
  in
  let rec essai vv = function  (* let rec essai vv la = match la with *)
    | []       -> if (List.length vv >= nbs) &&   (* c'est fini et on vérifie que l'on peut fermer *)
                     (List.mem s0 (List.map (function (s,d) -> s) (List.assoc (List.hd vv) ladj)))
                  then cycles := (s0::vv)::(!cycles)  (* fermeture (sn = s0) *)
    | (b,d)::q -> essai vv q;
                  if not (List.mem b vv) then essai (b::vv) (List.assoc b ladj)
  in
  essai [s0] (List.assoc s0 ladj);   (* à reculons, départ systématique en sn = s0 *)
  !cycles
;;
(* voyage : 'a -> ('a * ('a * 'b) list) list -> 'a list list = <fun> *)

let cycles = voyage 1 ladj_hamilton;;    (* départ systématique en 1 *)
print_int_int_list cycles;;
List.length cycles;;                     (* = 60 solutions *)
let tr1 = List.hd cycles;; (* [1;8;9;10;11;12;13;14;15;16;20;19;18;17;7;6;5;4;3;2;1] *)
trace_int_graphe hamilton;;
set_color black;;
trace_chaine hamilton tr1;;

let cycles = voyage 1 ladj_chvatal;; (* départs systématique en 1 *)
print_int_int_list cycles;;
List.length cycles;;                      (* = 370 .. . Sur Wikipedia on annonce 360.. *)
let tr1 = List.hd cycles;;
trace_int_graphe chvatal;;
set_color black;;
trace_chaine chvatal tr1;;
```

- La réponse est rapide avec le graphe Hamilton (20 sommets) : $2^{20} = 1048576$, avec 60 cycles trouvés.
- La réponse est rapide avec le graphe Chvatal (12 sommets) : $3^{12} = 531441$, avec 370 cycles trouvés.
- Cela ne finit pas en temps raisonnable pour le graphe France (90 sommets) : $3^{90} = 8.72796356808771197e+42$ = 87279635680877119706694584659548498884 0345.

```
(*=====================================================================*)
(*-- X.6.2. Recherche d'un (seul) cycle hamiltonien pour France. ---*)
(*=====================================================================*)
(* il y a toujours un voisin commun à deux sommets voisins *)
(*- 1. Recherche voisin commun à u et v dans lnv liste de sommets non visités *)
let search_new graph u v lnv =      (* stratégie: premier trouvé *)
  let rec search_nouv = function
    | []   -> raise Not_found       (* ou ... valeur invalide *)
    | w::q -> try let _ = List.assoc w (List.assoc u graph.ladj)
                  and _ = List.assoc v (List.assoc w graph.ladj) in
                  w                                          (* u -> w -> v *)
              with Not_found -> search_nouv q
  in search_nouv lnv
;;
(* search_new : int_graphe -> int -> int -> int list -> int = <fun> *)

(*---------------------------------------------------------------------- tests ---*)
search_new france 37 36 [47;42;41;43];;   (* 41 *)
search_new france 37 37 [47;42;41;43];;   (* 41 *)
search_new france 37 75 [47;41;42;43];;   (* Exception: Not_found. *)
search_new france 137 36 [47;42;41;43];;  (* Exception: Not_found. *)
```

```
(*- 2. Fonction utilisant le graphe France défini ci-dessus *)
let unTourDeFrance () =   (* cycle hamiltonien de a vers a *)
  let lsnv = ref (List.map (function v,lv -> v) france.ladj)  (* liste sommets non visités *)
  in
  let rec ajout_un = function
    | []  -> []   (* ne se produit pas *)
    | [u] -> [u]
    | u::v::q -> try let w = search_new france u v !lsnv in
                     lsnv := List.filter (function x -> x <> w) !lsnv;
                     ajout_un(u::w::v::q)
                 with Not_found -> u::(ajout_un (v::q))
  in
  let a = List.hd !lsnv in
  lsnv := List.tl !lsnv;  (* liste sommets non visités *)
  ajout_un [a;a]
;;
(* val unTourDeFrance : unit -> int list = <fun> *)

(*------------------------------------ test --*)
let vr = unTourDeFrance ();;
test_cycle_hamiltonien france vr;;
valuation_chaine france vr;;
trace_int_graphe france;;
set_color black;;
trace_chaine france vr;;
souris_france ();;
```

$< \mathscr{F}\mathscr{I}\mathscr{N} >$ TP Hamilton-C ; Annexes + corrigé.

8 TP Hamilton. Compléments

8.1 Transformation chaînes et cyles sommets/ arcs

8.1.1 Rappels et conventions

Par défaut, une chaîne, un cycle sont représentés par des listes de sommets concécutils :

Une ***chaîne*** (***chemin*** pour un graphe orienté) ***de sommets*** [a] , de ***longueur*** $k \geqslant 0$ (ayant k arêtes ou arcs), est une séquence de $k+1$ sommets, $(x_0, x_1, \cdots, x_k)$, chaque sommet étant lié au sommet suivant par une arête $\{x_i, x_{i+1}\}$ ou un arc $x_i \to x_{i+1}$, tous deux notés (x_i, x_{i+1}) [b] .

Un ***cycle*** (***circuit*** pour un graphe orienté) est une chaîne (chemin) de sommets, $(x_0, x_1, \cdots, x_k)$, de longueur $k > 1$, où $x_k = x_0$, qualifié de **élémentaire** si les sommets $x_1, ..., x_k$ sont distincts.

a. La description par chaîne ou chemin de sommets ne convient pas dans le cas d'un multi-graphe.
b. L'arête $\{x_i, x_{i+1}\}$ peut être notée indiffféremment (x_i, x_{i+1}) ou (x_{i+1}, x_i).

Il peut être utile d'avoir une représentation comme liste d'arêtes ou arcs concécutifs :

Une **chaîne ou chemin**, d'arêtes ou d'arcs, de **longueur** $k \geqslant 0$ (ayant k arêtes ou arcs), est une séquence de k arêtes ou arcs concécutifs, $(x_0, x_1), (x_1, x_2), \cdots, (x_{k-1}, x_k)$.

Une **cycle ou circuit**, d'arêtes ou d'arcs, de **longueur** $k \geqslant 0$ (ayant k arêtes ou arcs), est une chaîne d'arêtes ou arcs, $(x_0, x_1), (x_1, x_2), \cdots, (x_{k-1}, x_k)$ de longueur $k > 1$, où $x_k = x_0$ et les sommets $x_1, ..., x_k$ sont distincts.

8.1.2 Rectification d'une chaîne d'arêtes en un chemin d'arcs

Si une chaîne d'arêtes est représentée par la liste $[(x_0, x_1); (x_1, x_2); \ldots; (x_i, x_{i+1}); (x_{i+1}, x_{i+2}); \ldots; (x_{k-1}, x_k)]$, cette même chaîne d'arêtes est **aussi** représentée par la liste $[(x_1, x_0); (x_2, x_1); \ldots; (x_i, x_{i+1}); (x_{i+2}, x_{i+1}); \ldots; (x_k, x_{k-1})]$ dans laquelle on identifie moins facilement la représentation équivalente en tant que chaîne de sommets $[x_0; x_1; x_2; \ldots; x_i; x_{i+1}; x_{i+2}; \ldots; x_k]$. La fonction `chaine_to_chemin` transforme la deuxième liste d'arcs en la première :

```
let rec chaine_to_chemin = function        (* avec mise dans le bon ordre ! *)
  | []       ->  []
  | [(a,b)]  ->  [(a,b)]
  | (a,b)::(a',b')::q when b = a' -> (a,b):: (chaine_to_chemin ((a',b')::q))
  | (a,b)::(a',b')::q when a = a' -> (b,a):: (chaine_to_chemin ((a',b')::q))
  | (a,b)::(a',b')::q when b = b' -> (a,b):: (chaine_to_chemin ((b',a')::q))
  | (a,b)::(a',b')::q when a = b' -> (b,a):: (chaine_to_chemin ((b',a')::q))
  | _ -> failwith "Cette liste n'est pas une chaîne d'arcs valide."
;;
(* val chaine_to_chemin : ('a * 'a) list -> ('a * 'a) list = <fun> *)

let gg = [(1,5) ; (3,5) ; (3,4) ; (1,4) ; (3,1) ; (2,3)];;
chaine_to_chemin gg;;  (* ->  [(1, 5); (5, 3); (3, 4); (4, 1); (1, 3); (3, 2)] *)
let gg2 = [(1,5) ; (3,5) ; (3,4) ; (2,1); (1,4); (3,1) ; (2,3)];;
chaine_to_chemin gg2;; (* -> Exception: Failure "Cette liste n'est pas une cha\195\174ne valide". *)
```

8.1.3 Transformation chaînes de sommet / chaînes d'arcs (OCaml)

```
(* Un cycle qui a été construit à l'envers: ajout nouvel arc en tête ... il serait mieux de le renverser: *)
let arcscycle9762 = (* cycle (d'arcs) hamiltonien pour la France, de longueur  9762 *)
  [37,36; 86,37; 87,86; 23,87; 18,23; 89,18; 21,89; 71,21; 69,71; 42,69; 43,42; 15,43; 46,15; 47,46; 33,47; 40,33;
   ....
   19,63; 24,19; 16,24; 17,16; 85,17; 79,85; 49,79; 72,49; 41,72; 36,41]
;; (* 36 -> 41 -> 72 -> 49 -> ... -> 87 -> 86 -> 37 -> 36 *) *)

let arcscycle9762 = List.rev arcscycle9762;;
  [36,41; 41,72; 72,49; 49,79; 79,85; 85,17; 17,16; 16,24; 24,19; 19,63; 63, 3;  3,58; 58,45; 45,77; 77,10; 10,52;
   ....
   42,69; 69,71; 71,21; 21,89; 89,18; 18,23; 23,87; 87,86; 86,37; 37,36]
;; (* 36 -> 41 -> 72 -> 49 -> ... -> 87 -> 86 -> 37 -> 36 *)

let  rec chaine_nodes_to_arcs  = function  (* ls  = match  ls with  *)
   | a::b::q -> (a,b)::chaine_nodes_to_arcs (b::q)
   | _       ->  []
;;
(* val chaine_nodes_to_arcs : 'a list -> ('a * 'a) list = <fun> *)
let chaine_arcs_to_nodes la =
   let lb = chaine_to_chemin la
   in (fst (List.hd lb))::(  List.map (function a-> snd a) lb )
;;
(* val chaine_arcs_to_nodes : ('a * 'a) list -> 'a list = <fun> *)

let nodescycle9762 = chaine_arcs_to_nodes arcscycle9762;;
let barcscycle9762 = chaine_nodes_to_arcs nodescycle9762;;
```

8.2 Algorithme de fourmis

```
(*=========================================*) (* Créé le 10/08/08. modif 10/12/17. *)
(* Algorithme des fourmis (adapté pour France) *)
(*=========================================*)
(* A. PREALABLE: Exécuter le contenu du ficher TP-Haml-C.ml *)
(*-------------------------------------------------------------------------------*)
(* B. Dépots de phéromones: donnée externalisée de l'algo Fourmi pour France, *)
(*    pour être lues sur le graphe par souris_france2 (); (voir ci-dessous)   *)
(*-------------------------------------------------------------------------------*)
let ph = Array.make_matrix (91) (91) (1e-6);; (* phéromones déposés p0 = 1e-6 *)
let re_init_ph p0 =
   for i = 1 to 90 do for j = 1 to 90 do ph.(i).(j) <- p0; ph.(j).(i) <- p0 done done
;; (* val re_init_ph : float -> unit = <fun> *)
(*-----------------------------------------------------------------------*)
(* C. Lecture des données sur le graphe France à l'aide de la souris *)
(*     positionnement sur milieu du trait -> edite villes et dépôts  *)
(*-----------------------------------------------------------------------*)
let rec get2 mx my = function        (* spécifique France *)
  | []        -> 0, 0
  | (a,la)::q -> let (_,_, xa, ya, _) = List.assoc a prefectures in
                 let rec search = function
                    | []       -> get2 mx my q
                    | (b,d)::r -> let (_,_, xb, yb, _) = List.assoc b prefectures
                                  in let x  = (xa+xb)/2 and y = (ya+yb)/2
                                  in if abs(mx-x)+abs(my-y) <= 6 then a, b else search r
                 in search la
;; (* val get2 : int -> int -> (int * (int * 'a) list) list -> int * int = <fun> *)

let souris_france2 () =  (* spécifique France *)
  set_color red; fill_rect 10 (size_y ()-600) 180 30;
  set_color black; moveto 20 (size_y () - 595); draw_string "Click here to finish";

  let stop = ref false in
  while (not !stop)
  do let sta = wait_next_event [Mouse_motion; Button_down] in
     let x = sta.mouse_x and y = sta.mouse_y in
     stop := sta.button && (x > 10) && (x < 200)
                        && (y > (size_y ()-600)) && (y < (size_y ()-570)) ;
     let a,b = if !stop (* or (not (List.mem (point_color sta.mouse_x sta.mouse_y) [green; black])) *)
               then 0, 0 else get2 sta.mouse_x sta.mouse_y liens in
     if a = 0 then begin set_color white; fill_rect 10 (size_y ()-60) 200 60; set_color black end
     else begin let da,pa,xa,ya,ra = List.assoc a prefectures in   (* infos a *)
                moveto 10 (size_y () - 20); draw_string (pa);
                moveto 10 (size_y () - 40); draw_string (format "%2d %s" a da);
                moveto 10 (size_y () - 60); draw_string (ra)
          end;
     if b = 0 then begin set_color white; fill_rect 500 (size_y ()-60) 180 60; set_color black end
     else begin let db,pb,xb,yb,rb = List.assoc b prefectures in   (* infos b *)
                moveto 500 (size_y () - 20); draw_string (pb);
                moveto 500 (size_y () - 40); draw_string (format "%2d %s" b db);
                moveto 500 (size_y () - 60); draw_string (rb)
          end;
     if (a = 0) || (b=0)
     then begin set_color white; fill_rect 550 (size_y ()-350) 180 120; set_color black end
     else begin moveto 550 (size_y () - 270); draw_string (format "%3d ->%3d :" a b);
                moveto 620 (size_y () - 270); draw_string (format "%17.10f" ph.(a).(b));
                moveto 550 (size_y () - 300); draw_string (format "%3d ->%3d :" b a);
                moveto 620 (size_y () - 300); draw_string (format "%17.10f" ph.(b).(a));
                let d = List.assoc b (List.assoc a liens) in  (* distance entre a et b *)
                moveto 550 (size_y () - 330); draw_string (format "%3d" d)
          end
  done;
  set_color white; fill_rect 10 (size_y ()-600) 180 30; set_color black;
  let _ = wait_next_event [Button_up] in ()  (* il faut "vider" l'événement! *)
;;
(* val souris_france2 : unit -> unit = <fun> *)
```

```
(*--------------------------------------------------------------------*)
(* D. Algorithme des Fourmis (légèrement personnalisé pour France)   *)
(*    avec dépots (ph) externalisés, pour examen externe             *)
(*--------------------------------------------------------------------*)
let algoFourmis graph good nbT r e =   (* good = cycle sommets fermé hamiltonien initial *)
                                       (* nbT = nombre de tentatives *)
                                       (* r = evap, e = coeff dépot élitiste *)
   let nbS = List.length graph.ladj (* nombre de sommets *)
   and p0 = 1e-6                    (* valeur trace minimale, initiale, phéromone sur un arc *)
   and a = 1.0 and b = 5.0          (* paramètres tirage sommet suivant *)
   and q = 100.0 in                 (* paramètre dépots de phéromones *)
   re_init_ph p0;
   let s0 = ref 0 and s1 = ref 0    (* s0 ville de départ, s1 ville suivante *)
   and si = ref 1 and sj = ref 1    (* pour tentatives de coupure (avec ph < p0) *)
   and ssi = ref 1 and ssj = ref 1  (* pour tentatives altération de dépots *)
   and exi = [] in                  (* sommets origine d'arcs exclus des évolutions de dépots *)

   let tp = ref (good)                                (* bon tour de départ *)
   and lp = ref (valuation_chaine france good) in     (* et sa longueur *)

   let nbF = 3*nbS in               (* nombre de fourmis *)
   let tf = Array.make (1+nbF) []   (* tf.(f) = parcours de la fourmi f    *)
   and lf = Array.make (1+nbF) 0    (* lf.(f) = longueur parcours fourmi f  *)
   in

   let dist u v = (* graph.madj.(u).(v)   <<== Pb si boucles codée 0 et pas infty *)
                  try List.assoc v (List.assoc u graph.ladj)
                  with Not_found -> infty  (* absence d'arc (u,v) y compris boucle (u,u) *)
   in
   let choice_next u visited =
       (*------------------- sommets possibles depuis u ---*)
       let sp = ref [] and next = ref 0  in
       for s = 1 to nbS
       do (* existe arete (u,s), extrémité s non vue et (u,s) non coupé (ph >= p0) *)
          if (dist u s < infty) && (not (List.mem s visited)) && (ph.(u).(s) >= p0)
          then sp := (s,0.0)::(!sp)  (* -> (sommet, proba) *)
       done;
       sp := (0,0.0)::!sp;   (* pour indices débutant à 1 *)
       (*------------------- tirage extrémité parmi possibles, par roue de loterie *)
       let n = List.length !sp in
       if n > 1 then
       begin
            let vp = Array.of_list !sp in  (* indices à partir de 1 *)
            for s = 1 to n - 1
            do let v, p = vp.(s) in
               let duv = float_of_int (dist u v) in
               vp.(s) <- v, (snd vp.(s-1)) +. (ph.(u).(v) ** a *. ( 1.0 /. duv) ** b)
            done;

            let z = Random.float (snd vp.(n-1)) in
            let s = ref 1 in
            while (!s < n-1) && ((snd vp.(!s)) <= z) do incr s done;
            next := fst vp.(!s)
       end
       else next := 0;  (* convention sommet non trouvé (numéros sommets de 1 à n) *)
       !next
   in
   let pheromones_depots_survivante f =
       (*- dépots de phéromones en fin de parcours individuel réussi *)
       for i = 0 to nbS-1    (* indice dans tf.(f), et pas numéro sommet *),
       do (* supprimé: sauf arcs débutant en villes exclues des évolutions *)
          let u = List.nth tf.(f) i
          and v = List.nth tf.(f) ((i+1) mod nbS) in
          ph.(u).(v) <- ph.(u).(v) +. q /. (float_of_int lf.(f)) /. (float_of_int nbS);
          ph.(v).(u) <- ph.(u).(v)
       done
   in
```

```
let pheromones_evaporation_depots_elite () =
    (*- dépots de phéromones et évaporation (en fin de tour) *)
    for u = 1 to nbS   (* numéro de sommet *)
    do (* sauf arcs débutant en villes exclues des évolutions *)
       for v = 1 to nbS
       do if (dist u v < infty) && (not (List.mem u exi)) then
          begin
              ph.(u).(v) <- p0 +. (1.0 -. r) *. ph.(u).(v); (* résiduel,évaporation *)
              ph.(v).(u) <- ph.(u).(v)
          end
       done
    done;

    for i = 0 to nbS-1    (* indice dans tp, et pas numéro de sommet *)
    do (* supprimé: sauf arcs débutant en villes exclues des évolutions *)
       let u = List.nth !tp i
       and v = List.nth !tp ((i+1) mod nbS) in
       ph.(u).(v) <- ph.(u).(v) +.  e *.  q  /. (float_of_int !lp) /. (float_of_int nbS);
       ph.(v).(u) <- ph.(u).(v)
    done
 in

let alterations () =
   (* -- affaiblisement *)
   let i = Random.int nbS in    (* indice [0 .. nbS-1] *)
   ssi := List.nth !tp i;
   ssj := List.nth !tp ((i+1) mod nbS);  (* arc (ssi, ssj) *)
   ph.(!ssi).(!ssj) <- p0;
   ph.(!ssj).(!ssi) <- ph.(!ssi).(!ssj);

   (* -- coupure temporaire, pour 1 tour: 1/rétablissement 2/coupure *)
   ph.(!si).(!sj) <- p0;
   ph.(!sj).(!si) <- ph.(!si).(!sj);

   let i = Random.int nbS in   (* indice [0 .. nbS-1] *)
   si := List.nth !tp i;
   sj := List.nth !tp ((i+1) mod nbS);  (* arc (si, sj) *)
   ph.(!si).(!sj) <- p0 /. 2.0;
   ph.(!sj).(!si) <- ph.(!si).(!sj);
in

for t = 1 to nbT (* tentatives : cycles de tours de Fourmis *)
do
    (*=== I. altération sur le meilleur tour tp *)
    alterations ();

    (*=== II. chaque fourmi f tente un tour *)
    for f = 1 to nbF
    do
        let s0 = ref (1 + Random.int nbS) in   (* s0 = sommet de départ, début d'arête *)
        let s00 = !s0 in
        tf.(f) <- [!s0]; lf.(f) <- 0;

        (*--- avancement de sommet en sommet *)
        let h = ref 1 in   (* h = nb sommets visités *)
        while (!h < nbS) && (lf.(f) < infty)
        do
            let s1 = choice_next !s0 tf.(f) in  (* 0 = non trouvé *)
            if s1 = 0
            then lf.(f) <- infty  (* longueur infinie *)
            else begin               (* accumulation nouvelle arête (s0 -> s1) *)
                tf.(f) <- s1::tf.(f);              (* "avance" (recule!) en s1 *)
                lf.(f) <- (dist !s0 s1) + lf.(f);  (* longueur du chemin *)
                s0 := s1;                          (* début arête suivante *)
            end;
            incr h
        done; (* while *)
```

```
            (*--- fermeture du tour (si possible) *)
            if lf.(f) < infty then
            begin let d = dist !s0 s00 in       (* !s0 = last , s00 = first *)
                  if d < infty
                  then begin tf.(f) <- s00::(tf.(f)); lf.(f) <- d + lf.(f);
                             pheromones_depots_survivante f
                       end
                  else lf.(f) <- infty;
            end;
        done; (* for f *)

        (*=== III. meilleure performance globale *)
        for f = 1 to nbF
        do
            (* if lf.(f) < infty then        (* infos, mise au point --*)
            begin let s = format "%5d  %5f1" t lf.(f)
                  in moveto 10 (size_y () - 50); draw_string s;
            end; --------------------------*)
            if lf.(f) < !lp then (* meilleur long et tour *)
            begin lp := lf.(f); tp := tf.(f); end
        done;

        (*=== IV. évaporation et renforcement dépots de phéromone sur best *)
        pheromones_evaporation_depots_elite ();

        (*-- edition infos, pour mise au point --*)
        (* let s = format "%5d  %5f0" t !lp in moveto 10 (size_y () - 100); draw_string s; *)
        (* if !New then begin print_string s; print_newline () end; *)
    done; (* t *)

    !tp, !lp
;;
(* val algoFourmis : int_graphe -> int list -> int -> float -> float -> int list * float = <fun> *)

(*--------------------------- test --*)
let nodescycle9762 =
  [36; 41; 72; 49; 79; 85; 17; 16; 24; 19; 63;  3; 58; 45; 77; 10; 52; 70; 88; 54; 55; 51;  2;
   75; 60; 27; 78; 28; 61; 53; 35; 44; 56; 29; 22; 50; 14; 76; 80; 62; 59;  8; 57; 67; 68; 90;
   25; 39;  1; 74; 73; 38;  5;  4;  6; 20; 83; 13; 84; 26;  7; 30; 48; 12; 82; 32; 31; 11; 81;
   34; 66;  9; 65; 64; 40; 33; 47; 46; 15; 43; 42; 69; 71; 21; 89; 18; 23; 87; 86; 37; 36];;

Random.init 17253;;
let cp, lp = algoFourmis france nodescycle9762 50 0.98 15.0;;

test_cycle_hamiltonien france cp;;  (* true *)
valuation_chaine france cp;;        (* 8164 *)

trace_int_graphe france;;
set_color black;;
trace_chaine france cp;;
souris_france2 ();;

(*== meilleur cycle (obtenu par plusieurs exécutions sur des cycles issus de 9762) ==*)
let abest7947 =  (* Un bon cycle hamiltonien pour la France, de longueur 7947 *)
  [76; 27; 14; 50; 61; 72; 49; 53; 35; 22; 29; 56; 44; 85; 17; 79; 86; 37; 41; 45; 28; 78; 75;
   77; 89; 10; 52; 21; 71; 69; 42; 26;  7; 43; 48; 12; 81; 82; 46; 47; 24; 19; 15; 63;  3; 58;
   18; 36; 23; 87; 16; 33; 40; 64; 65; 32; 31;  9; 11; 66; 34; 30; 84; 13; 83; 20;  6;  4;  5;
   38; 73; 74;  1; 39; 25; 70; 90; 68; 67; 88; 54; 57; 55; 51;  2;  8; 59; 62; 80; 60; 76];;

et ... j'ai même trouvé un 7894 ... mais c'était avec le programme Python:
let abest7894 = (* Mieux! Un bon cycle hamiltonien pour la France, de longueur 7894 *)
  [46; 47; 82; 81; 12; 48; 43;  7; 26; 42; 69; 71;  1; 74; 73; 38;  5;  4;  6; 20; 83; 13; 84;
   30; 34; 66; 11;  9; 31; 32; 65; 64; 40; 33; 24; 16; 86; 79; 17; 85; 44; 56; 29; 22; 35; 53;
   49; 72; 61; 50; 14; 27; 76; 60; 80; 62; 59;  8;  2; 51; 55; 57; 54; 88; 67; 68; 90; 70; 25;
   39; 21; 52; 10; 89; 77; 75; 78; 28; 45; 41; 37; 36; 18; 58; 3; 63; 23; 87; 19; 15; 46];;
```

8.3 Arbre couvrant d'un graphe non orienté

Dans un graphe non orienté (bi-orienté symétrique), connexe (où deux sommets sont toujours reliés par une chaîne), d'ensemble de sommets $S = \{1,2,3,\ldots,n\}$, on cherche à supprimer des arêtes ou arcs, de façon à obtenir une structure arborescente (arbre couvrant).
Pour accéder facilement aux listes d'adjacence et les modifier facilement, on transforme la liste d'adjacence du graphe en un vecteur d'adjacence du graphe, indicé par les numéros de sommets.
Par exemple, la liste d'adjacence, ordonnée selon les sommets croissants de 1 à n :

```
ladj = [ 1,[2,50; 5,30; 6,40; 8,25]; 2,[1,50; 3,15; 4,5]; 3,[2,15; 4,11; 6,13]; 4, [2,5; 3,11; 5,12; 6,20];
         5,[1,30; 4,12; 6,17]; 6,[1,40; 3,13; 4,20; 5,17; 7,12]; 7,[6,12]; 8,[1,25] ]
```

est transformée en le vecteur d'adjacence, d'indices utiles les numéros de sommets de 1 à n :

```
vadj = [| 0,[]; 1,[2,50; 5,30; 6,40; 8,25]; 2,[1,50; 3,15; 4,5]; 3,[2,15; 4,11; 6,13]; 4,[2,5; 3,11; 5,12; 6,20];
               5,[1,30; 4,12; 6,17]; 6,[1,40; 3,13; 4,20; 5,17; 7,12]; 7,[6,12]; 8,[1,25] |]
```

par l'instruction : `vadj = Array.of_list ((0,[])::ladj)`.
Remarque. `vadj.(s) = s, ls` et l'information s est redondante ... on pourrait s'en passer, mais cela facilite le retour à une liste d'adjacence de graphe en final.

On réalise un parcours en profondeur du graphe ("visite"), en comptant le nombre de passage en chaque sommet : si un sommet est vu une deuxième fois (depuis un nouveau père potentiel), on "casse" le lien à ce père et on annule le passage.

```
let arbre_couvrant ladj =
  let n = List.length ladj in
  let vadj = Array.of_list ((0,[])::ladj)  (* indices utiles 1..n ; vadj.(s) = (s,ls)*)
  in
  let compteur = Array.make (n+1) 0 in   (* compteur.(s) = nombre de visites du sommet s *)
  let rec visite p f =               (* visite de f fils de p, excluant remontée directe de f à p *)
    compteur.(f) <- compteur.(f)+1;               (* d = dist(p,f) = dist(f,p) *)
    if  compteur.(f) > 1
    then begin (* f vu deux fois -> rupture du lien p f et annulation de la visite *)
               vadj.(p) <- p, (List.filter (function (u,x) -> u <> f ) (snd vadj.(p)));
               vadj.(f) <- f, (List.filter (function (u,x) -> u <> p ) (snd vadj.(f)));
               compteur.(f) <- 1
         end
    else (* visite des fils de f, à l'exclusion du père p de f *)
         let fils = List.map (function (u,x) -> u) (snd vadj.(f)) in
         List.iter (visite f) (List.filter (function u -> u <> p ) fils)
  in
  visite 0 1;      (* père et (fils,dist), 0 = père fictif de 1 au départ, distance bidon = 0 *)
  List.tl (Array.to_list vadj)  (* liste d'adjace du graphe couvrant, sans père fictif *)
;;
(* val arbre_couvrant : (int * (int * 'a) list) list -> (int * (int * 'a) list) list = <fun> *)

(*=== tests ===*)
(*--- avec Hamilton, on obtient un arbre peigne ---*)
let bladj_hamilton =  arbre_couvrant ladj_hamilton;;
let bhamilton = { ladj = bladj_hamilton; madj = matrice_adj_of_liste_adj bladj_hamilton;
                  lpos = lpos_hamilton; lbox = [] };;
trace_int_graphe bhamilton;;

(*--- avec France, il y a quelques branches, mais pas beaucoup ... ---*)
let bliens = arbre_couvrant liens;;
let bfrance =
  { ladj = bliens;
    madj = matrice_adj_of_liste_adj bliens;
    lpos = map (function (n,(d,p,x,y,r)) -> (n, (x,y)) ) prefectures;  (* = pref_coords *)
    lbox = contour };;
trace_int_graphe bfrance;;
souris_france ();;
```

8.4 Floyd-Warshall : plus court chemin pour tous couples de sommets d'un graphe orienté valué

On considère ici un graphe G, orienté et valué, d'ensemble de sommets $S = \{1,2,\dots,n\}$, représenté par une matrice d'adjacence 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 ou ∞ à M_{ii} (0 ou ∞, choix sans incidence ici).

Remarque. Prendre $M_{ii} = 0$ ou $M_{ii} = \infty$ 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.

8.4.1 Calcul des distances minimales

Pour $0 \leqslant k \leqslant n$, on note W_{ij}^k le poids minimal d'un chemin du sommet i au sommet $j \neq i$, n'empruntant que des sommets intermédiaires dans $S_k = \{s,\ s \leqslant k\}$ (on a $S_0 = \{\}$, $S_1 = \{1\}$, …, $S_n = \{1,2,3,\dots,n\}$) et, si un tel chemin n'existe pas, on pose $W_{ij}^k = \infty$. Pour $k = 0$, il s'agit de chemins sans intermédiaires (réduits à un arc) et $W_{i,j}^0 = M_{ij}$.

On note W^k la matrice des W_{ij}^k (W_{ii}^k à préciser). En posant $W_{ii}^0 = M_{ii}$, $W^0 = M$, matrice d'adjacence du graphe G.

1. S'il existe un chemin p entre les deux sommets distincts i et j, de poids minimal, ses sommets intermédiaires, s'ils existent, sont dans un ensemble $S_k = \{s,\ s \leqslant k\}$, avec ~~$0 \leqslant k \leqslant n$~~ $1 \leqslant k \leqslant n$, et
   - soit p n'emprunte pas le sommet k,
   - soit p emprunte exactement une fois le sommet k (car les circuits sont de poids positifs ou nuls) et p est la concaténation de deux chemins, entre i et k et k et j respectivement, tous deux de poids minimal, dont les sommets intermédiaires, s'il existent, sont dans S_{k-1},

   ce qui conduit à la relation : pour $1 \leqslant k \leqslant n$, $W_{ij}^k = \min\left(W_{ij}^{k-1},\ W_{ik}^{k-1} + W_{kj}^{k-1}\right)$ (relation de récurrence sur k).
2. S'il n'existe pas de chemin entre les deux sommets distincts i et j, pour tout $k \geqslant 1$, $W_{ij}^k = \infty$ et on a, soit $W_{ik}^{k-1} = \infty$, soit $W_{kj}^{k-1} = \infty$, et la relation ci-dessus reste valable si on convient que pour tout x, éventuellement égal à ∞,
$$\begin{cases} \min(x,\infty) = \min(\infty,x) = x \\ x + \infty = \infty + x = \infty \end{cases}$$
3. Pour $j = i$, la relation ci-dessus reste valable, $\begin{cases} \text{si on a fait le choix initial } M_{ii} = 0, \text{ avec } W_{ii}^k = 0 \text{ pour tout } k, \\ \text{si on a fait le choix initial } M_{ii} = \infty, \text{ avec } W_{ii}^k = \infty \text{ pour tout } k. \end{cases}$

En introduisant de nouvelles opérations sur les nombres, avec les conventions vues ci-dessus :

`+?`	définie par `x +? y = min(x,y)`	associative, d'élément neutre $+\infty$
`*?`	définie par `x *? y = x + y`	associative, d'élément absorbant $+\infty$, d'élément neutre 0

la relation de récurrence sur k devient : $\begin{cases} k = 0 : \text{pour } i,j \in S, & W_{ij}^0 = M_{ij} \\ k > 0 : \text{pour } i,j \in S, & W_{ij}^k = W_{ij}^{k-1} +? \left(W_{ik}^{k-1} *? W_{kj}^{k-1}\right) \end{cases}$

En final, $W_{i,j}^n$ sera la plus courte distance des chemins de i à j ($W_{i,j}^n = \infty$ s'il n'y a pas de chemin de i à j).

8.4.2 Plus courts chemins

On adapte l'algorithme précédent, avec une matrice P telle que $P_{i,j}$ est le père de j dans un meilleur chemin de i à j. Lorsqu'on découvre un meilleur chemin de i à j passant par k, on remplace le père P_{ij} de j par le père P_{kj} de j.

$$\begin{cases} k = 0 : \text{pour } i,j \in S, & \begin{cases} W_{ij}^0 = M_{ij} \\ P_{ij} = \begin{cases} -1 & \text{si } i = j \text{ ou si } M_{ij} = \infty \\ i & \text{sinon (père de } j \text{ dans le chemin réduit à l'arc } i \longrightarrow j \text{)} \end{cases} \end{cases} \\ k > 0 : \text{pour } i,j \in S, & newD = W_{ik}^{k-1} *? W_{kj}^{k-1} \begin{cases} \text{si } newD < W_{ij}^{k-1}, & \begin{cases} W_{ij}^k = newD \\ P_{ij} = P_{kj} \end{cases} \quad (P_{kj} \text{ meilleur père de } j) \\ \text{sinon}, & \begin{cases} W_{ij}^k = W_{ij}^{k-1} \\ P_{ij} \quad \text{est inchangé.} \end{cases} \end{cases} \end{cases}$$

En final, $\begin{cases} P_{i,j} \text{ sera le père de } j \text{ dans un plus court chemin de } i \text{ à } j, \text{ de longueur } W_{ij}^n. \\ \text{avec } P_{i,j} = -1 \ , W_{ij}^n = \infty \quad \text{s'il n'y a pas de chemin de } i \text{ à } j. \end{cases}$ **et pour obtenir les sommets successifs dans un plus court chemin de i à j, il suffira de suivre (à reculons) les $P_{i,k}$ depuis $k = j$ jusqu'à $k = i$.**

Complexité : $O(|S|^3)$.

Rappel : Dijkstra est de complexité $O\big((|S| + |A|)\log|S|\big)$, mais il ne calcule les chemins que pour une seule origine.

```
(*-- Opérations spéciales sur int --*)
let ( +? ) x y = min x y     (* let prefix +? x y = min x y; si Caml light *)
and ( *? ) x y = if (x = infty) or (y = infty) then infty else x + y
;;
(* val ( +? ) : 'a -> 'a -> 'a = <fun> *)
(* val ( *? ) : int -> int -> int = <fun> *)

(*-- Calcul des plus courtes distances --  w.(i).(j) = poids arc i -> j --*)
let floyd_warshall ladj =
  let n = List.length ladj
  and w = matrice_adj_of_liste_adj ladj in   (* (n+1)x(n+1), bordée *)
  for i = 1 to n do w.(i).(i) <- 0 done;     (* suppression des boucles par 0 (infty) *)

  for k = 1 to n           (* k in first !!! *)
  do for i = 1 to n
     do for j = 1 to n do w.(i).(j) <- w.(i).(j) +? w.(i).(k) *? w.(k).(j) done
     done
  done;
  w
;;
(* val floyd_warshall : (int * (int * int) list) list -> int array array = <fun> *)

(*-- Calcul des plus courts chemins et distances --  w.(i).(j) = poids arc i -> j --*)
let floyd_warshall_parent ladj =
  let n = List.length ladj
  and w = matrice_adj_of_liste_adj ladj in   (* (n+1)x(n+1), bordée *)
  for i = 1 to n do w.(i).(i) <- 0 done;     (* suppression des boucles par 0 (infty) *)

  let parent = Array.make_matrix (n+1) (n+1) (-1) in  (* -1: orphelin, par défaut *)
  for i = 1 to n
  do for j = 1 to n
     do if (i = j) || (w.(i).(j) = infty) then parent.(i).(j) <- -1 else parent.(i).(j) <- i
     done;
  done;
  for k = 0 to n do parent.(k).(0) <- k; parent.(0).(k) <- k done; (* bordures *)

  for k = 1 to n           (* k in first !!! *)
  do for i = 1 to n
     do for j = 1 to n
        do let newD = w.(i).(k) *? w.(k).(j) in
           if newD < w.(i).(j) then
           begin w.(i).(j) <- newD;
                 parent.(i).(j) <- parent.(k).(j)
           end
        done
     done
  done;
  parent  (*, w *)
;;
(* val floyd_warshall_parent : (int * (int * int) list) list -> int array array = <fun> *)

let shortest_path parent i j =   (* parent = matrice des pères *)
  let rec aux u v =
    if u = v then [v]
    else if parent.(u).(v) = -1 then []    (* pas de chemin *)
         else v::(aux u parent.(u).(v))    (* parent.(u).(v) = père de v *)
  in
  List.rev (aux i j)  (* aux j i (renversement) ne marche que si graphe non orienté symétrique! *)
;;
(* val shortest_path : int array array -> int -> int -> int list = <fun> *)

(*--------------------------- tests --*)
let path = shortest_path (floyd_warshall_parent hamilton.madj) 1 13;;   (* [1; 2; 3; 12; 13] *)
trace_int_graphe hamilton;;
let path = shortest_path (floyd_warshall_parent france.ladj) 1 13;;     (* [1; 69; 42; 26; 84; 13] *)
trace_int_graphe france;;
souris_france ();;
```

8.5 Plus courts chemins pondérés, pour tous couples de sommets d'un graphe 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 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, 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.

8.5.1 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.

8.5.2 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.

```
(*-- Opérations spéciales sur int --*)
(* infty (= max_int), ( +? ) et ( *? *) ont déjà été introduits (Flyod-Warshall) *)

(*-- Opérations spéciales sur int matrices bordées (indices utiles à partir de 1) --*)
let mult2_matrix m1 m2 =
  let p = Array.length m1 and q = Array.length m1.(0)
  and r = Array.length m2 and s = Array.length m2.(0) in
  if q <> r then raise (Invalid_argument "index out of bounds");

  let m = Array.make_matrix p s infty in
  for i = 1 to p-1
  do for j = 1 to s-1
     do for k = 1 to q-1
        do m.(i).(j) <- m.(i).(j) +? m1.(i).(k) *? m2.(k).(j) done
     done
  done;
  for k = 0 to p-1 do m.(k).(0) <- k done; (* bordures *)
  for k = 0 to s-1 do m.(0).(k) <- k done;
  m
;;  (* val mult2_matrix : int array array -> int array array -> int array array = <fun> *)

let ( *! ) m1 m2 = mult2_matrix m1 m2
;;  (* val ( *! ) : int array array -> int array array -> int array array = <fun> *)

(*------------------ test --*)
let m1 = [| [|0; 1; 2|]; [|1; 1; 2|]; [|2; 3; 4|] |];;     (* 2 x 2 *)
let m2 = [| [|0; 1; 2|]; [|1; 3; 2|]; [|2; 3; 4 |];;      (* 2 x 2 *)
print_int_matrix (m2 *! m1);   (*  [|[|0; 1; 2|]; [|1; 4; 3|]; [|2; 6; 5|]|] *)
```

Calcul de la matrice $C = D^{(n)}$, telle que, pour $i, j \in S$, C_{ij} est la plus courte distance pour un chemin de i à j.

```
(*--------------------------------  madj.(i).(j) = poids arc i -> j --*)
let plus_courtes_distances madj =           (* version 1: depuis madj *)
  let n = (Array.length madj) - 1           (* calcul de madj^(2m) jusqu'à 2m > n-1 *)
  and d = ref madj in
  for i = 1 to n do !d.(i).(i) <- 0 done;    (* suppression des boucles par 0 (PAS infty) *)
  let m = ref 1 in
  while !m < n -1
  do d := !d *! !d;   (* sans effet de bord sur madj *)
     m := 2 * !m
  done;
  !d
;;
(* val plus_courtes_distances : int array array -> int array array = <fun> *)

(*--------------------------------  madj.(i).(j) = poids arc i -> j --*)
let plus_courtes_distances ladj =          (* version 2: depuis ladj *)
  let n = List.length ladj                 (* calcul de madj^(2m) jusqu'à 2m > n-1 *)
  and d = ref (matrice_adj_of_liste_adj ladj) in  (* (n+1)x(n+1), bordée *)
  for i = 1 to n do !d.(i).(i) <- 0 done;         (* suppression des boucles par 0 (PAS infty) *)
  let m = ref 1 in
  while !m < n -1
  do d := !d *! !d;
     m := 2 * !m
  done;
  !d
;;
(* val plus_courtes_distances : (int * (int * int) list) list -> int array array = <fun> *)

let rec exp2_rap_matrix m = function    (* exponentiation rapide au sens de *! *)
   | 0 -> failwith "problème"
   | 1 -> m
   | p -> let u = exp2_rap_matrix m (p/2) in
          let v = u *! u in                    (* let v = mult2_matrix u u in *)
          if p mod 2 = 0 then v else v *! m    (* mult2_matrix v m *)
;;
(* val exp2_rap_matrix : int array array -> int -> int array array = <fun> *)
```

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

Pour obtenir un plus court chemin pondéré d'un sommet à un autre, on introduit la matrice ML de liaison du graphe :

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

Remarque. ML est égale à la matrice P ("parent") construite par la fonction `floyd_warshal_parent` précédente.

La matrice de liaison est déduite de la matrice d'adjacence `madj` du graphe (version 1) ou de la liste d'adjacence `ladj` du graphe (version 2) et de la matrice C $(= D^{(n)})$ des plus courtes distances pondérées, par un algorithme en $O(|S|^3)$:

> Pour tout couple de sommets (i,j), $j \neq i$, le père de j, dans un meilleur chemin de i vers j, est le meilleur intermédiaire k, entre i et j, qui soit un voisin de j.

sachant que, pour k voisin de j, $C_{k,j}$ est le poids de l'arc $k \longrightarrow j$: $\begin{cases} C_{kj} = \texttt{madj}_{kj} \\ (j, C_{kj}) \in L_k \text{ (liste d'adjacence de } k) \end{cases}$

```
let liaison madj mc =                  (* version 1: madj; madj.(k).(j) = poids arc k -> j *)
  let n = (Array.length mc) - 1 in
  let ml = Array.make_matrix (n+1) (n+1) (-1) in
  for s = 0 to n do ml.(s).(0) <- s; ml.(0).(s) <- s done; (* bordures *)
  for i = 1 to n
  do for j = 1 to n
     do if (i = j) || (mc.(i).(j) = infty)
        then ml.(i).(j) <- -1
        else for k = 1 to n        (* k in last ! *)
             do (* (k <> j) & (k meilleur "entre" i et j) & (arc k -> j) *)
                if (k <> j) && ( mc.(i).(j) = mc.(i).(k) *? mc.(k).(j) )
                            && ( mc.(k).(j) = madj.(k).(j) )
                then ml.(i).(j) <- k  (* k père de j dans le chemin de i à j *)
             done
     done
  done;
  ml
;;
(* val liaison : int array array -> int array array -> int array array = <fun> *)

let liaison ladj mc =                   (* version 2: ladj *)
  let n = List.length ladj in
  let ml = Array.make_matrix (n+1) (n+1) (-1) in
  for s = 0 to n do ml.(s).(0) <- s; ml.(0).(s) <- s done; (* bordures *)
  for i = 1 to n
  do for j = 1 to n
     do if (i = j) || (mc.(i).(j) = infty)
        then ml.(i).(j) <- -1
        else for k = 1 to n        (* k in last ! *)
             do (* (k <> j) & (k meilleur "entre" i et j) & (arc k -> j) *)
                if (k <> j) && ( mc.(i).(j) = mc.(i).(k) *? mc.(k).(j) )
                            && ( List.mem (j, mc.(k).(j)) (List.assoc k ladj) )
                then ml.(i).(j) <- k  (* k père de j dans le chemin de i à j *)
          done
     done
  done;
  ml
;;
(* val liaison :  (int * (int * int) list) list -> int array array -> int array array = <fun> *)
```

On déduit, de la matrice de liaison ML, un plus court chemin pondéré du sommet i au sommet j,
en suivant (à reculons) les $ML(i,k)$, depuis $k = j$ jusqu'à $k = i$ (fonction déjà vue) :

```
let shortest_path parent i j =   (* parent = ml = matrice de liaison *)
  let rec aux u v =
    if u = v then [v]
    else if parent.(u).(v) = -1 then []      (* pas de chemin *)
         else v::(aux u parent.(u).(v))      (* parent.(u).(v) = père de v *)
  in List.rev (aux i j)  (* aux j i (renversement) ne marche que si graphe non orienté symétrique! *)
;;
(* val shortest_path : int array array -> int -> int -> int list = <fun> *)
```

8.5.4 Exemples

```
(*------------------------- Hamilton (avec tous les poids = 1) ------*)
let c = plus_courtes_distances hamilton.ladj;;
let c2 = exp2_rap_matrix hamilton.madj 20;;
print_int_matrix c;;
print_int_matrix c2;;

let ml = liaison hamilton.ladj c;;
print_int_matrix ml;;
let ch = shortest_path ml 1 20;;  (*  [1; 8; 9; 18; 19; 20] *)
trace_int_graphe hamilton;;

(*-------------------------- Hamilton modifié: poids {1, 8} = 30 (symétrique) ------*)
let ladj_hamiltonB =
[ 1, [ 2,1;  5,1;  8,30];  2, [ 1,1;  3,1; 10,1];   3, [ 2,1;  4,1; 12,1];  4, [ 3,1;  5,1; 14,1];
  5, [ 1,1;  4,1;  6,1];   6, [ 5,1;  7,1; 15,1];   7, [ 6,1;  8,1; 17,1];  8, [ 1,30;  7,1;  9,1];
  9, [ 8,1; 10,1; 18,1];  10, [ 2,1;  9,1; 11,1];  11, [10,1; 12,1; 19,1];  12, [ 3,1; 11,1; 13,1];
 13, [12,1; 14,1; 20,1];  14, [ 4,1; 13,1; 15,1];  15, [ 6,1; 14,1; 16,1];  16, [15,1; 17,1; 20,1];
 17, [ 7,1; 16,1; 18,1];  18, [ 9,1; 17,1; 19,1];  19, [11,1; 18,1; 20,1];  20, [13,1; 16,1; 19,1] ];;
let lpos_hamiltonB =
  [  1,( 50,340);  2,(210,620);  3,(450,620);  4,(610,340);  5,(330,180);  6,(330,260);  7,(210,300);
     8,(170,380);  9,(210,460); 10,(250,540); 11,(330,540); 12,(410,540); 13,(450,460); 14,(490,380);
    15,(450,300); 16,(370,340); 17,(290,340); 18,(250,420); 19,(330,460); 20,(410,420) ];;
let hamiltonB = { ladj = ladj_hamiltonB; madj = matrice_adj_of_liste_adj ladj_hamiltonB;
                  lpos = lpos_hamiltonB; lbox = [] };;

let c = plus_courtes_distances hamiltonB.ladj;;
let ml = liaison hamiltonB.ladj c;;
let ch = chaine_court ml 1 20;;  (*  [1; 5; 6; 15; 16; 20] : on ne passe plus par 8, trop coûteux *)
trace_int_graphe hamiltonB;;

(*-------------------------------------------- France ---------*)
let c = plus_courtes_distances france.ladj;;  (* 2 à 3 secondes ... *)
let ml = liaison france.ladj c;;
let ch = shortest_path ml 22 13;;  (*  [22; 35; 53; 72; 41; 18; 3; 42; 26; 84; 13] *)
trace_int_graphe france;;
souris_france ();;
```

$< \mathscr{FIN} >$ TP Hamilton-C ;
+ Annexes + corrigé + compléments.