Informatique MP.
TD / TP / Colle n.

Antoine MOTEAU.

17 mars 2019

Multi-graphes (non orientés) eulériens
Recherche d'une chaîne eulérienne.

. . . /Euler-C.tex (2003)
. . . /Euler-C.tex Compilé le 17 mars 2019 à 12h 12m 04s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Ordre du TP : possible en premier tiers d'année (TP 2 ou 3)

Il me semble que j'avais un problème (de cohérence, d'intérêt ?) avec cet énoncé ...
Mais je ne me souviens plus ...
C'est peut-être avec les graphes multipartis (à plusieurs composantes connexes disjointes) ?
Ou alors mon problème était réglé ?

Eléments utilisés :

- notion (élémentaire) de multi-graphe
- théorème d'Euler (admis)
- listes
- récursivité
- exception (gestion élémentaire)
- ressources de format (`printf`)

Documents relatifs au TP :

- Texte du TP : .tex, .dvi, .pdf (version -C et -S)
- Pas de Squelette de programme Caml
- Programme corrigé Caml : EulerC.ml

Recherche d'une chaîne eulérienne dans un multi-graphe (non orienté) eulérien.

Un ***multi-graphe*** non orienté est constitué par la liste de ses ***arêtes***, une arête entre deux ***sommets*** x et y étant représentée par le couple $(\{x,y\},k)$, formé par l'ensemble des extrémités et l'identifiant (numéro) d'arête k.
Si le multi-graphe est ***pondéré***, une arête est un triplet $(\{x,y\},k,p))$, où p est la ***pondération*** de l'arête.

Dans un multi-graphe non orienté,

- on autorise les ***boucles*** (arêtes dont les extrémités coïncident : $(\{x,x\},k)=(\{x\},k))$;
- une arête $(\{x,y\},k)$ peut être parcourue dans les deux sens (de x vers y ou de y vers x) ;
- il peut y avoir plusieurs arêtes entre deux sommets x et y (mais elles sont d'identifiants distincts).
  Dans le cas où il y a au plus une seule arête entre deux sommets, le graphe est dit ***simple***.

Quelques définitions relatives à un (multi-)graphe non orienté G (voir aussi le chapitre Graphe du cours) :

- Le ***degré*** d'un sommet s est le nombre d'arêtes ayant une extrémité égale à s (remarque : une arête $(\{s,s\},k)$ compte pour deux dans le calcul du degré de s).
- Une ***chaîne (d'arêtes*** [1] ***)*** est une séquence d'arêtes consécutives $(\{x,y\},k)$, dans laquelle, si x (ou y) est sommet d'une arête, l'autre y (ou x) est sommet de l'arête suivante (lorsqu'elle existe).
  Une chaîne va du sommet x au sommet y si, en suivant sa séquence d'arêtes, on passe de x à y.,
- G est ***connexe*** si, pour tout couple (x,y) de sommets, il existe une chaîne de x à y .
- La ***composante connexe*** d'un sommet x de G est le plus grand sous-graphe connexe extrait de G et qui contient x.
- Une ***chaîne eulérienne*** est une chaîne (d'arêtes) qui passe par tous les sommets du graphe, en empruntant une et une seule fois chaque arête et cette chaîne est un ***cycle eulérien*** si ses arêtes extrémités sont contigües.
  Remarques. Une chaîne eulérienne peut passer plusieurs fois par un même sommet.
  Une chaîne vide est une chaîne eulérienne du graphe vide.
- Un graphe est ***eulérien*** s'il contient une chaîne eulérienne.

Pour un (multi-)graphe orienté,

- la notion d'arête est remplacée par la notion d'***arc***, un arc du sommet x au sommet y étant représenté par le couple $((x,y),k)$, formé par le couple des extrémités et l'identifiant (numéro) d'arc k.
  Si le multi-graphe orienté est pondéré, un arc est un triplet $((x,y),k,p))$, où p est la pondération de l'arc.
- Une séquence d'arcs concécutifs s'appelle un ***chemin (d'arcs*** [1] ***)***.
- Un chemin dont le premier arc est consécutif au dernier est un ***circuit***.

Pour des raisons pratiques, les (multi-)graphes non orientés seront codées comme des (multi-)graphes orientés, avec des arêtes assimilées à des arcs. La seule différence sera dans le traitement des "arcs" : sommets pris dans l'ordre pour les graphes orientés et dans un ordre indifférent pour les graphes non orientés. *On pourrait aussi dédoubler une arête $(\{x,y\},k,p)$ en deux arcs $((x,y),k,p)$ et $((y,x),k,p)$, de même pondération (et de même numéro ?).*

1 Multi-graphe eulérien

Théorème 1.1. d'Euler *(Euler-Hierholzer, 1736-1873)*

Un multi-graphe non orienté connexe admet une chaîne eulérienne si et seulement si le nombre de sommets de degré impair est
- *0 (existence d'un cycle eulérien dont les extrémités sont un même sommet de degré pair),*
- *ou 2 (existence d'une chaîne eulérienne dont les extrémités sont les deux sommets de degré impair).*

La démonstration du théorème d'Euler est une démonstration constructive basée sur le principe de coupure d'un graphe par suppression d'arêtes. On recherche les chaînes eulériennes dans les composantes connexes (eulériennes) de la coupure et les chaînes rencontrées sont ensuite connectées entre elles.

Définition 1.1.

Un (multi-)***graphe eulérien*** est un multi-graphe non orienté connexe, ayant 0 ou 2 sommets de degré impair.

On se propose ici de chercher les chaînes ou cycles eulériens dans un multi-graphe eulérien.

1. La description comme une séquence de sommets ne serait perninente que pour les graphes simples.

Exemple 1.1. Représenter graphiquement les multi-graphes suivants :

$$G_1 = [(\{1,2\},1);(\{2,3\},2);(\{3,4\},3);(\{4,2\},4);(\{1,5\},5);(\{1,5\},6);$$
$$(\{5,2\},7);(\{5,6\},8);(\{6,7\},9);(\{7,5\},10)]$$
$$G_2 = [(\{1;2\},1);(\{2;4\},2);(\{4;5\},3);(\{5;6\},4);(\{6;4\},5);(\{4;3\},6);(\{3;1\},7);$$
$$(\{1;7\},8);\quad (\{7;8\},9);(\{8;9\},10);(\{9;7\},11)]$$
$$G_3 = [(\{1;2\},1);(\{2;4\},2);(\{4;5\},3);(\{5;6\},4);(\{6;4\},5);(\{4;3\},6);(\{3;1\},7);$$
$$(\{7;8\},9);(\{8;9\},10);(\{9;7\},11)]$$ biparti, déduit de G_2 par suppression de l'arête $(\{1;7\},8)$.

Exemple 1.2. Le problème des sept ponts de Königsberg (Euler-Hierholzer, 1736-1873).

La ville de Königsberg est traversée par le fleuve Pregel et possède 7 ponts (Figure 1). Un piéton peut-il, en se promenant, parcourir chaque pont une et une seule fois ? Si oui, peut-il le faire en revenant à son point de départ ?

Le problème est représenté par le multi-graphe :

$$G_K = [(\{1;2\},1);(\{1;3\},2);(\{1;4\},3);(\{2;4\},4);$$
$$(\{2;4\},5);(\{3;4\},6);(\{3;4\},7)]$$

- les arcs sont les ponts,
- les sommets sont les quartiers de la ville.

Le théorème d'Euler apporte la réponse (qui est ?).

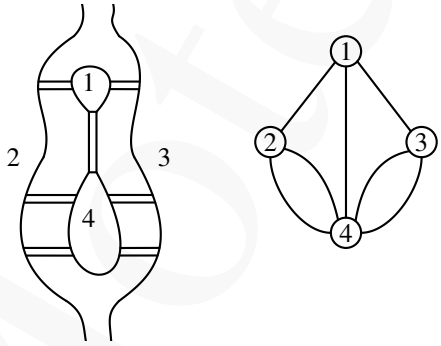


FIGURE 1 – Les 7 ponts de Koenigsberg et le multigraphe associé.

Exemple 1.3. Les promos de la chaîne de restauration Mac'Aronny.

Vous achetez un premier plat au choix (prix unique), et la possession d'un plat vous donne droit à un ticket gratuit pour certains autres plats, mais vous n'avez le droit de prendre qu'un seul plat de chaque sorte.
Les règles sont les suivantes :

- **boisson** donne droit à **charcuterie**, **crudité**, **fromage**, **glace** ou **viande**,
- **charcuterie** donne droit à **boisson**, **crudité**, **fromage**, **glace** ou **viande**,
- **crudité** donne droit à **boisson**, **charcuterie**, **fromage**, **glace**, **omelette**, **poisson** ou **viande**,
- **fromage** donne droit à **boisson**, **charcuterie**, **crudité**, **fruit**, **glace** ou **poisson**,
- **fruit** donne droit à **fromage**, **glace**, **omelette** ou **poison**,
- **glace** donne droit à **boisson**, **charcuterie**, **crudité**, **fromage**, **fruit** ou **poisson**,
- **omelette** donne droit à **crudité**, **fruit**, **poisson** ou **viande**,
- **poisson** donne droit à **crudité**, **fromage**, **fruit**, **glace**, **omelette** ou **viande**,
- **viande** donne droit à **boisson**, **charcuterie**, **crudité**, **omelette** ou **poisson**,

Par exemple, après avoir payé le plat **omelette**, vous pouvez avoir gratuitement **fruit**, puis, grâce à **fruit**, **glace**, puis, ...

1. Avec les règles initiales,
   (a) Y-a t'il un choix initial qui permette de prendre tous les plats au prix d'un seul ?
   (b) Y-a t'il un choix initial qui ne permet pas de prendre tous les plats au prix d'un seul ?
2. La direction supprime **omelette** du menu (et des règles). Que disent les clients qui sont gros mangeurs ?
3. La direction rétablit **omelette** au menu, avec les règles initiales, à l'exception du droit à **fruit** depuis **omelette** (et réciproquement). Les clients gros mangeurs sont-ils satisfaits ?

Indications : On peut utiliser un multi-graphe dont les arêtes sont les plats (un plat ne peut pas être pris plus d'une fois) et dont les sommets sont ...

2 Ressources élémentaires (OCaml)

2.1 Modules (bibliothèques) OCaml

`open Printf;;` module OCaml, éditions formatées, voir exemple d'utilisation ci dessous.

2.2 Codage d'un multi-graphe

On suppose que les sommets des (multi-) graphes sont des entiers (que l'on peut choisir consécutifs à partir de 0 ou de 1).

Une arête (ou un arc, pour un multi-graphe orienté) est codée comme un couple $((x,y),k)$, où

- x et y sont les sommets extrémités de l'arête (origine et extrémité pour un arc),
- k est le numéro de l'arête (ou de l'arc).

La seule différence entre un arc et une arête sera dans le traitement : sommets pris dans l'ordre pour un graphe orienté et dans un ordre indifférent pour un graphe non orienté. Un graphe non orienté sera codé comme un graphe orienté.

Un (multi-)graphe "orienté" est codé comme la liste[a] de ses arcs, de type : `((int * int) * int) list`.
Une chaîne est codée comme une liste d'arêtes consécutives[b], première arête en tête de liste.

a. Ce n'est pas le codage le plus pertinent, mais il est assez simple et permet d'interpréter les chaînes comme des sous-graphes.
b. Une chaîne est alors un (sous-)graphe ("ordonné") et, si ses arêtes sont distinctes, une chaîne eulérienne d'elle même (en tant que (sous-)graphe eulérien).

2.3 Fonctions élémentaires

Les fonctions à écrire peuvent être récursives ou faire appel à des fonctions auxiliaires récursives, mais ne doivent pas utiliser de variable référence.

Ecrire la fonction `is_sommet`, de type `'a -> (('a * 'a) * 'b) list -> bool`, telle que `is_sommet s g` renvoie **true** si `s` est un sommet du multi-graphe `g` et **false** sinon.

Remarque. On rappelle que dans une arête, les sommets extrémités de l'arête sont écrits sous forme de couple, sans préjuger de l'ordre.

Ecrire la fonction `print_graphe`, de type `((int * int) * int) list -> unit`, telle que `print_graphe g` "imprime" (en vrac) la liste des arcs du multi-graphe `g`.

Remarque. Un arc $((x,y),k)$ pourra être "imprimé" ainsi : `printf "((%3d,%3d),%3d) ;" x y k`.

- `"((%3d,%3d),%3d) ;"` est la chaîne de format décrivant l'édition :
  une parenthèse ouvrante, un parenthèse ouvrante, un entier sur 3 places, une virgule, un entier sur trois places, une parenthèse fermante, une virgule, un entier sur 3 places, une parenthèse fermante, un espace et un point virgule.
- Par exemple, `printf "((%3d,%3d),%3d) ;" 10 24 136` donne l'édition `(( 10, 24),136) ;`

Ecrire la fonction `print_chaine`, de type `((int * int) * int) list -> unit`, telle que `print_chaîne ch` "imprime" la chaîne `ch`, liste d'arêtes du genre ch = $[\cdots((s_k,s_{k+1}),k_i);((s_{k+1},s_{k+2}),k_{i+1});\cdots]$ mais aussi
ch = $[\cdots((s_k,s_{k+1}),k_i);((s_{k+2},s_{k+1}),k_{i+1});\cdots]$, puisque les extrémités d'arêtes sont décrites à l'ordre près, sous la forme de "fléches marquées" ordonnées : $\ldots\ s_k --(k_i)--> s_{k+1} --(k_{i+1})--> s_{k+2}\ \ldots$.

Remarque. On pourra s'inspirer du format d'impression utilisé dans `print_graphe` .

Ecrire la fonction `degre`, de type `'a -> (('a * 'a) * 'b) list -> int`, telle que `degre s g` renvoie le degré de `s` (nombre d'arêtes ayant une extrémité égale à `s`) dans le multi-graphe `g`.

Rappel : une arête $((s,s),k)$ augmente de deux le degré de s.

Ecrire la fonction `sommets`, de type `(('a * 'a) * 'b) list -> 'a list`, telle que `sommets g` renvoie la liste des sommets du multi-graphe `g`, sans répétition.

Ecrire la fonction `sommets_impairs`, de type `(('a * 'a) * 'b) list -> 'a list`, telle que `sommets_impairs g` renvoie la liste des sommets de degré impair du multi-graphe `g`, sans répétitions.

Ecrire la fonction `supp_arete`, de type `'a -> (('a * 'a )* 'b) list -> 'a * 'b * (('a * 'a) * 'b) list`, telle que `supp_arete s g` renvoie le triplet `(u, k, h)` où `h` est le sous-graphe du multi-graphe `g` obtenu par la suppression d'une arête d'extrémités `s` et `u`, de numéro `k` dans `g` (s'il y a plusieurs arêtes ayant `s` comme extrémité, on ne précise pas quelle arête doit être choisie).
Renvoie l'erreur `Not_found` si une telle arête n'existe pas dans `g`.

3 Recherche d'une chaîne eulérienne dans un graphe eulérien.

Si au départ le graphe est supposé eulérien, en cours d'algorithme, du fait des coupures, on est amené à ne traiter qu'une partie du graphe, partie non forcément connexe, dont on explorera les composantes connexes.

On utilise donc un algorithme de recherche d'une chaîne eulérienne entre deux sommets x et y d'une même composante connexe, présumée eulérienne, d'un graphe G non forcément connexe, algorithme qui, **en cas de succès**, renvoie un couple :

- chaîne eulérienne de x à y, dans (et égale à) la composante connexe eulérienne de x et y dans G,
- reste de G : graphe déduit de G par suppression de la composante connexe eulérienne de x (et de y).

3.1 Algorithme

Recherche d'une chaîne eulérienne entre deux sommets x et y, dans une même composante connexe eulérienne du graphe G :

- Si G est vide, G est eulérien, de chaîne eulérienne [ ], du sommet x à lui même et le reste est vide.
- Si G est réduit à l'arête $(\{u,v\},k)$, G admet une chaîne eulérienne d'extrémités x et y ssi $\{x,y\}=\{u,v\}$, donc si $\{x,y\}=\{u,v\}$, on renvoie le couple (arête $(\{u,v\},k)$, [ ]) et **sinon, c'est une erreur !**
- Sinon, **x et y étant réputés être les extrémités d'une chaîne eulérienne de $\mathscr{C}$, composante connexe de G** :

  ⋆ Si $x \neq y$, pour que x et y soient extrémités d'une chaîne eulérienne de $\mathscr{C}$, x et y doivent être des sommets effectifs de G, les deux seuls de degré impair dans $\mathscr{C}$ (cf théorème).
  Il doit exister une arête $(\{x,u\},k)$ d'extrémité x dans G, **sinon c'est une erreur !**

  On supprime une arête $(\{x,u\},k)$ (de $\mathscr{C}$) dans G , pour obtenir le sous graphe G_x.

  - Si $u=y$, il se peut que x et/ou y ne soient pas dans G_x ou se trouvent dans des composantes connexes (éventuellement vides) distinctes ou confondues de G_x.
    De plus (suppression de l'arête), x et y sont de degré pair (éventuellement nul) dans G_x et il n'y a pas de sommets de degré impair dans $\mathscr{C}$ privé de l'arête $(\{x,u\},k)$.

    On prend
    - une chaîne eulérienne de x à x (cycle) dans (égale à) la composante connexe de x dans G_x et on note H le reste de G_x (G_x privé de la composante connexe de x),
    - puis une chaîne eulérienne de $u=y$ à y (cycle) dans (égale à) la composante connexe de y dans H et on note K le reste de H (H privé de la composante connexe de y).

    S'il n'y a pas eu d'erreur, on obtient, en connectant ces deux chaînes par l'arête $(\{x,u\},k)$, une chaîne eulérienne de x à y dans la composante connexe de x et y dans G et le reste K du graphe G.

  - Si $u \neq y$, alors
    - y est dans G_x, avec ses arcs incidents inchangés, donc y est de degré impair dans G_x ;
    - si $u=x$, u était de degré impair ($\geqslant 2$) et supprimer l'arête $(\{x,u\},k)$ fait baisser son degré de 2, si $u \neq x$ (et $u \neq y$), u était de degré pair ($\geqslant 1$) dans $\mathscr{C}$ (x et y étant les deux seuls de degré impair) et supprimer l'arête $(\{x,u\},k)$ fait baisser le degré de u de 1 et le degré de x de 1.

    Donc u et y sont les deux seuls sommets de degré impair, d'une même composante connexe de G_x.

    On prend une chaîne eulérienne de u à y dans (égale à) la composante connexe de u et y dans G_x.
    S'il n'y a pas eu d'erreur, en prolongeant l'arête $(\{x,u\},k)$) par cette chaine, on obtient une chaîne eulérienne de x à y dans la composante connexe de x et y dans G, et le reste du graphe G.

  ⋆ Si $x=y$, x est sommet (effectif ou non), extrémité d'un cycle eulérien de $\mathscr{C}$, de degré pair (cf théorème), éventuellement nul, et il n'y a pas de sommet de degré impair dans $\mathscr{C}$ (cf théorème).

  - S'il n'y a pas d'arête d'extrémité x dans G, c'est que le chemin de x à $y=x$ dans G est vide.
    On prend comme chaîne la chaîne vide (eulérienne d'un graphe vide) et comme reste G lui-même.
  - Sinon, on supprime une arête $(\{x,u\},k)$ (de $\mathscr{C}$) dans G , pour obtenir le sous graphe G_x.
    - Si $u=y=x$, le degré de $u=y=x$ baisse de 2, reste pair (éventuellement nul).
    - Si $u \neq y=x$, u était de degré pair (cf théorème), $\geqslant 2$, le degré de $x=y$ baisse de 1 et le degré de u baisse de 1, x et u deviennent tous deux de degré impair (ce sont les seuls dans $\mathscr{C}$ privé de $(\{x,u\},k)$).

    On prend une chaîne eulérienne de u à $y=x$ dans (égale à) la composante connexe de u et $y=x$ de G_x.
    S'il n'y a pas eu d'erreur, en prolongeant l'arête $(\{x,u\},k)$) par cette chaîne, on obtient une chaîne (cycle) eulérienne de x à $y=x$ dans la composante connexe de x et y dans G, et le reste du graphe G.

3.2 Initialisation, exploitation du résultat

Pour initialiser l'algorithme, on cherche la liste des sommets impairs du graphe G, ce qui permet de savoir si G peut ou ne peut pas être (globalement) eulérien.

- Si G ne peut pas être eulérien, il pourrait avoir des composantes connexes eulériennes ..., mais il faudrait aller "à la pêche", de façon empirique ....
- Si G peut être eulérien,

  cas 1 : exécuter l'algorithme à partir des deux seuls sommets distincts de degré impair x et y,

  cas 2 : exécuter l'algorithme à partir d'un sommet x quelconque (lorsque tous sont de degré pair), la composante connexe de x étant forcément eulérienne.

  Au retour de l'algorithme, si il n'y a pas eu d'erreur,
  - si le graphe G est eulérien, on aura une chaîne eulérienne de G et un reste vide,
  - si le graphe G n'est pas connexe, on aura une chaîne eulérienne d'une composante connexe de G (égale à cette composante connexe) et un reste non vide.

  et, en cas d'erreur, ce qui peut arriver uniquement dans le **cas 1**, cela signifie que x et y ne sont pas dans une même composante connexe du graphe G (donc que G n'est pas connexe) et que G n'est pas (globalement) eulérien.

  Remarque. Dans les deux cas, erreur ou pas, on pourra décider si G est connexe ou pas.

3.3 Ecriture de l'algorithme

C'est bien plus long à expliquer qu'à mettre en œuvre !

Les fonctions à écrire peuvent être récursives ou faire appel à des fonctions auxiliaires récursives, mais ne doivent pas utiliser de variable référence.

1. Ecrire la fonction `euler2`, de type `'a -> 'a -> (('a * 'a) * 'b) list -> (('a * 'a) * 'b) list * (('a * 'a) * 'b) list`, telle que `euler2 x y g` renvoie une chaîne eulérienne (éventuellement vide) de `x` à `y` dans le graphe `g` et le multi-graphe déduit de `g` par la suppression de cette chaîne dans `g`.
   `euler2` doit renvoyer l'exception (Failure "Pas de telle chaîne eulérienne") en cas d'erreur.

   Remarque. `euler2 x y` correspond rigoureusement à l'algorithme précédent.
2. Ecrire la fonction `euler`, de type `(('a * 'a) * 'b) list -> (('a * 'a) * 'b) list`, telle que `euler g` renvoie une chaîne eulérienne du multi-graphe graphe réputé eulérien `g`.
   `euler` doit renvoyer l'exception (Failure "graphe non eulérien") si `g` ne remplit pas les conditions nécessaires pour être eulérien et l'exception (Failure "graphe non connexe") si `g` remplit ces conditions mais n'est pas connexe.

   Remarque. `euler` exploite les résultats de `euler2` (x et y à choisir).

4 Questions

1. Représenter (à la main) le dessin des exemples G_1, G_2 et G_3.
   Préciser s'il s'agit de graphes eulériens, à composantes connexes eulériennes, ...
2. Les ponts de Köenisberg ...
3. Les promos de Mac'Aronny ...
4. Ecrire, en Caml, les fonctions récursives ou faisant appel à des fonctions auxiliaires récursives :
   `print_graphe` ; `print_chaine` ; `degre` ; `sommets` ; `sommets_impairs` ; `supp_arete` .
5. Ecrire, en Caml, les fonctions
   `euler2` (fonction récursive et on peut utiliser la concaténation `@`) puis `euler` .
   Remarque. Dans `euler2`, on pourra utiliser la gestion d'exception sous la forme :

   ```
   try let u, _, qx = supp_arete x q in ...
   with Not_found -> failwith "Pas de telle chaîne eulérienne"!
   ou
   try  let u, _, qx = supp_arete x q in ...
   with Not_found -> [ ], q
   ```
6. Prouver la terminaison de `euler2` . et aussi tester, sur des exemples simples puis plus complexes.
7. Quelle est la complexité de `euler2` ?

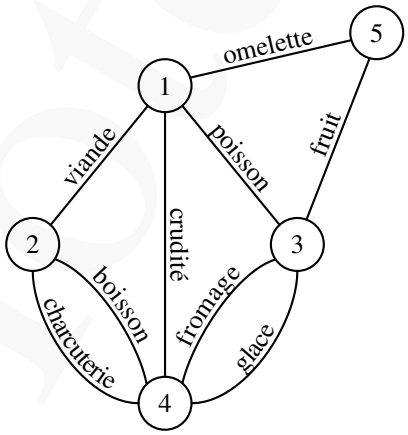
< $\mathscr{FIN}$ > (énoncé)

5 Corrigé (TP Euler)

Les promos de la chaîne de restauration Mac'Aronny

On crée un multi-graphe dont les arêtes sont les plats et les sommets sont les points communs (graphe connexe).

- **viande** et **poisson** sont deux arêtes concécutives.
  viande et **poisson** se relient au sommet 1,
- **charcuterie** est concécutif à **viande** mais pas à **poisson**.
  charcuterie et **viande** se relient au sommet 2,
- **boisson** est concécutif à **viande** mais pas à **poisson**.
  boisson et **viande** peuvent se relier au sommet 2, ce qui
  n'est pas contradictoire avec **charcuterie** et ne nécessite pas de duplication d'arc,
- **fromage** est concécutif à **poisson** mais pas à **viande**
  fromage et **poisson** se relient au sommet 3,
- etc ...



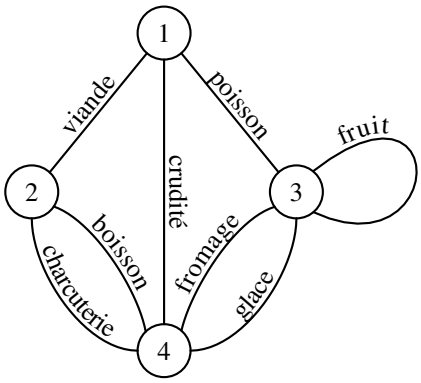
1. On a alors un graphe connexe cohérent où :
   - 1 est de degré 4, pair
   - 2 est de degré 3, impair
   - 3 est de degré 4, pair
   - 4 est de degré 5, impair
   - 5 est de degré 2, pair

   Il y a exactement deux sommets de degré impair.

   D'après le théorème d'Euler, pour le prix d'un plat,

   (a) En démarrant d'un des deux sommets de degré impair (2 ou 4), on peut prendre tous les plats (chemin hamiltonien entre 2 et 4).

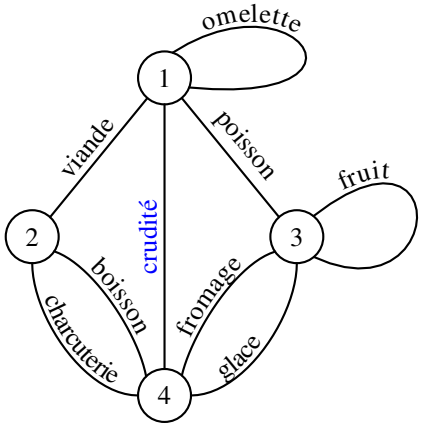
   (b) Si on démarre d'un sommet de degré pair (1, 3, 5), ce n'est pas possible.



2. Si on supprime **omelette**, on obtient le graphe :
   - 1 est de degré 3, impair
   - 2 est de degré 3, impair
   - 3 est de degré 5, impair
   - 4 est de degré 5, impair

   Il y a quatre sommets de degré impair.

   Il n'est plus possible de prendre tous les plats pour le prix d'un seul.

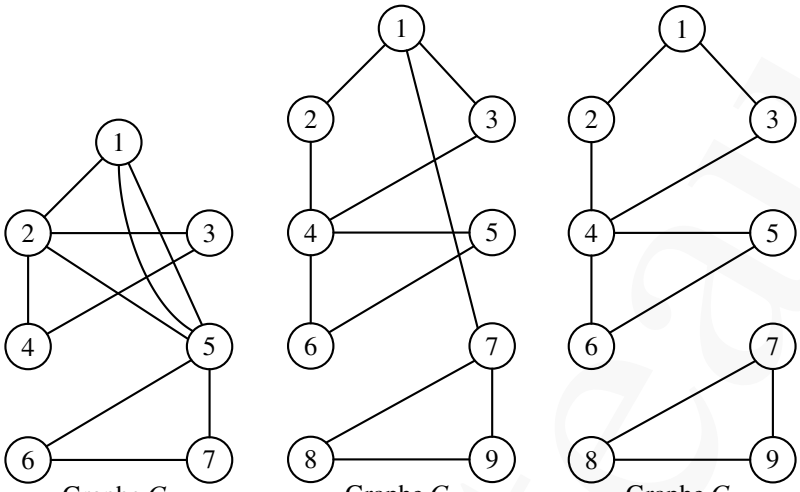


3. En rétablissant **omelette**, avec les règles modifiées, on obtient le graphe :
   - 1 est de degré 5, impair
   - 2 est de degré 3, impair
   - 3 est de degré 5, impair
   - 4 est de degré 5, impair

   Il y a quatre sommets de degré impair.

   Il n'est pas possible de prendre tous les plats pour le prix d'un seul.

Graphes exemples (G_1, G_2, G_3)



Graphe G_1 Graphe G_2 Graphe G_3

Fonctions (OCaml)

```
open Printf;;
(*==== Exemple de multigraphes euleriens =====*)
let g1 = [  ((1,2),1); ((2,3),2); ((3,4),3); ((4,2),4); ((1,5),5); ((1,5),6);
            ((5,2),7); ((5,6),8); ((6,7),9); ((7,5),10) ];;

let g1a = [ ((1,2),1); ((2,3),2); ((3,4),3); ((4,2),4);           ((1,5),6);
            ((5,2),7); ((5,6),8); ((6,7),9); ((7,5),10) ];;
(* g1a = connexe, simple, par suppression de ((1,5),5) dans g1 *)

let g2 = [ ((1,2),1); ((2,4),2); ((4,5), 3); ((5,6), 4); ((6,4),5); ((4,3),6); ((3,1),7);
           ((1,7),8); ((7,8),9); ((8,9),10); ((9,7),11) ];;

(*==== Exemple de multigraphes non connexes, à composantes connexes euleriennes paires *)
let g3 = [ ((1,2),1); ((2,4),2); ((4,5), 3); ((5,6), 4); ((6,4),5); ((4,3),6); ((3,1),7);
                      ((7,8),9); ((8,9),10); ((9,7),11) ];;
(* g3 = biparti, par suppression de ((1,7),8) dans g2 *)

(*==== Exemple de multigraphes connexes non euleriens : Ponts de Koenisberg *)
let gk = [((1,2),1);((1,3),2);((1,4),3);((2,4),4);((2,4),5);((3,4),6);((3,4),7)];;

(*======== Utilitaires d'édition =========*)
let rec print_graphe = function  (* édition intermédiaire pour mise au point *)
   | []           -> print_newline ()
   | [((a,b),n)]  -> printf "((%2d,%2d),%2d);" a b n; print_newline ()
   | ((a,b),n)::q -> printf "((%2d,%2d),%2d);" a b n; print_graphe q
;;
(* print_graphe : ((int * int) * int) list -> unit = <fun> *)

(* chaine = liste arêtes concécutives, extrémités pas forcément dans le bon ordre ! *)

let rec print_chaine = function
 | []        -> print_newline ()
 | [(a,b),n] -> printf "%2d--(%2d)-->%2d" a n b; print_newline ()
 | ((a,b),n)::((a',b'),n')::q when b=a'  -> printf "%2d--(%2d)-->" a n; print_chaine (((a',b'),n')::q)
 | ((a,b),n)::((a',b'),n')::q when a=a'  -> printf "%2d--(%2d)-->" b n; print_chaine (((a',b'),n')::q)
 | ((a,b),n)::((a',b'),n')::q when b=b'  -> printf "%2d--(%2d)-->" a n; print_chaine (((b',a'),n')::q)
 | ((a,b),n)::((a',b'),n')::q  (* a=b'*) -> printf "%2d--(%2d)-->" b n; print_chaine (((b',a'),n')::q)
;;
(* print_chaine : ((int * int) * int) list -> unit = <fun> *)

let rec chaine_to_chemin = function        (* avec mise dans le bon ordre ! *)
  | []           ->  []
  | [((a,b),k)]  ->  [((a,b),k)]
  | ((a,b),k)::((a',b'),k')::q when b = a' -> ((a,b),k):: (chaine_to_chemin (((a',b'),k')::q))
  | ((a,b),k)::((a',b'),k')::q when a = a' -> ((b,a),k):: (chaine_to_chemin (((a',b'),k')::q))
  | ((a,b),k)::((a',b'),k')::q when b = b' -> ((a,b),k):: (chaine_to_chemin (((b',a'),k')::q))
  | ((a,b),k)::((a',b'),k')::q (* a = b'*) -> ((b,a),k):: (chaine_to_chemin (((b',a'),k')::q))
;;
(* val chaine_to_chemin : (('a * 'a) * 'b) list -> (('a * 'a) * 'b) list = <fun> *)
```

```
let gg = [(1,5),1 ; (3,5),2 ; (3,4),3 ; (1,4),4 ; (3,1),5 ; (2,3),6];;
chaine_to_chemin gg;; (* [((1,5),1); ((5,3),2); ((3,4),3); ((4,1),4); ((1, 3),5); ((3,2), 6)] *)

let gg = [(1,5),1 ; (5,3),2 ; (3,4),3 ; (4,1),4 ; (1,3),5 ; (3,2),6];;
print_graphe gg;;
print_chaine gg;; (* 1--( 1)--> 5--( 2)--> 3--( 3)--> 4--( 4)--> 1--( 5)--> 3--( 6)--> 2 *)

let gg2 = [(1,5),1 ; (5,3),2 ; (4,3),3 ; (1,4),4 ; (1,3),5 ; (3,2),6];;
print_graphe gg2;;
print_chaine gg2;; (*  1--( 1)--> 5--( 2)--> 3--( 3)--> 4--( 4)--> 1--( 5)--> 3--( 6)--> 2 *)

(*========= Utilitaires d'information ========*)
let rec degre s = function
  | []           -> 0
  | ((a,b),n)::q -> let d = degre s q in
                    d + (if a=s then 1 else 0) + (if b=s then 1 else 0)
;;
(* degre : 'a -> (('a * 'a) * 'b) list -> int = <fun> *)

let sommets g =
  let rec add s = function
    | []         -> [s]
    | t::q as tq -> if t = s then tq else t::(add s q)
  and     sommets = function
    | []           -> []
    | ((a,b),n)::q -> add a (add b (sommets q))
  in sommets g
;;
(* sommets : (('a * 'a) * 'b) list -> 'a list = <fun> *)

g1;;
sommets g1;;  (* [5; 7; 6; 2; 1; 4; 3] *)

let rec is_sommet s = function
  | []           -> false
  | ((a,b),n)::q -> (a=s) || (b=s) || (is_sommet s q)
;;
(* is_sommet : 'a -> (('a * 'a) * 'b) list -> bool = <fun> *)

let sommets_impairs g =
  let f u s =  if (degre s g) mod 2 = 1 then s::u else u in
  fold_left f [] (sommets g)  (* it_list *)
;;
(* sommets_impairs : (('a * 'a) * 'b) list -> 'a list = <fun> *)

sommets_impairs g1;;    (* [1;5] *)
sommets_impairs g1a;;   (* [2;5] *)
sommets_impairs g2;;    (* [1;7] *)
sommets_impairs g3;;    (* [] *)
sommets_impairs gk;;    (* [1; 2; 3; 4] *)

(*========= Suppression d'arête ========*)
(* suppression de la premiere arête ayant une extrémité égale à a    *)
(* avec renvoi de l'autre extrémité, du numéro et du reste du graphe *)

let rec supp_arete a = function
  | []           -> raise Not_found
  | ((u,v),k)::q -> match a with
                    | s when s = u -> v,k, q
                    | s when s = v -> u,k, q
                    | _            -> let ss,kk,qq = supp_arete a q
                                      in ss, kk, ((u,v),k)::qq
;;
(* supp_arete : 'a -> (('a * 'a) * 'b) list -> 'a * 'b * (('a * 'a) * 'b) list = <fun> *)

let gg =                [((1,5),1); ((1,5),2); ((5,2),3); ((5,6),4); ((6,7),5); ((7,5),6)];;
supp_arete 2 gg;; (* (5, 3, [((1,5),1); ((1,5),2); ((5,6),4); ((6,7),5); ((7,5),6)]) *)
```

```
(*================================================================*)
(* Chaine eulérienne d'extrémité x, y dans une composante connexe  *)
(*================================================================*)
let rec euler2 x y = function
   | [] -> [], []

   | [(u,v),k] -> if ((u,v) <> (x,y)) && ((u,v) <> (y,x))
                  then failwith "Pas de telle chaine eulerienne"
                  else [(x,y),k], []

   | q  -> (* q n'est pas forcément connexe *)
           if x <> y
           then (* x <> y, sommets impairs de q, sommets effectifs de q, supposés *)
                (*         être dans une même composante connexe eulérienne de q  *)

                try let u, k, qx = supp_arete x q
                    in
                    if u = y
                    then (* on peut avoir supprimé x et/ou y                    *)
                         (* qx peut être non connexe, x et y sont pairs dans qx *)
                         (*    x et y sont dans des composantes connexes de qx  *)
                         (*                          distinctes ou confondues    *)
                         (*                          éventuellement vides        *)

                         let chx, rx = euler2 x x qx in
                         let chy, ry = euler2 u y rx in
                         chx @ (((x,u),k)::chy), ry

                    else (* y resté intact est impair dans qx *)
                         (*    u=x -> impair >=2 pair dans q et u<>x -> pair >= 1 dans q *)
                         (*    u est impair (-2) ou (-1) dans qx                         *)
                         (*    u est relié à y (si chaine eulérienne de x à y)           *)
                         (* u et y sont deux sommets distincts, impairs                  *)
                         (*   situés dans une même composante connexe (eulérienne) de qx *)

                         let chuy, ruy = euler2 u y qx in
                         ((x,u),k)::chuy, ruy
                with Not_found -> failwith "Pas de telle chaine eulerienne"

           else (* x=y, sommet pair (>= 0) de q, n'existe pas forcément dans q,        *)
                (*      appartient à une composante connexe C de q, qui peut être vide *)
                (*      et tous les sommets de C doivent être pairs                    *)

                try let u, k, qx = supp_arete x q   (* qx, de sommets impairs u et y(=x) *)
                    in
                    let chuy, ruy = euler2 u y qx
                    in ((x,u),k)::chuy, ruy
                with Not_found -> [], q
;;
(* euler2 : 'a -> 'a -> (('a * 'a) * 'b) list
     -> (('a * 'a) * 'b) list * (('a * 'a) * 'b) list = <fun> *)

(*=== test avec graphes (très) élémentaires --*)
let g0 = [(1,1),1];;
let a, r = euler2 1 1 g0;;  (* [((1, 1), 1)] ,  [] *)
let a, r = euler2 1 2 g0;;  (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 2 2 g0;;  (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 2 3 g0;;  (* Exception: Failure "Pas de telle chaine eulerienne". *)

let g00 = [(1,2),1; (2,3),2];;
let a, r = euler2 1 1 g00;;     (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 1 3 g00;;     (*  [((1, 2), 1); ((2, 3), 2)]  , [] *)
let a, r = euler2 1 2 g00;;     (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 2 2 g00;;     (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 2 3 g00;;     (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 3 1 g00;;     (*  [((3, 2), 2); ((2, 1), 1)] , [] *)
```

```
(*=== test avec des graphes eulériens, impairs, départ sur les deux sommets impairs --*)
let g1 = [  ((1,2),1);((2,3),2);((3,4),3);((4,2),4);((1,5),5);((1,5),6);
            ((5,2),7);((5,6),8);((6,7),9);((7,5),10) ];;

let a, r = euler2 1 5 g1;;   (* g1 connexe eulérien *)
print_chaine a;;  (* 1--( 1)--> 2--( 2)--> 3--( 3)--> 4--( 4)--> 2--( 7)--> 5--( 5)--> 1--( 6)-->
                     5--( 8)--> 6--( 9)--> 7--(10)--> 5 *)
let a, r = euler2 5 1 g1;;
print_chaine a;;
(*-----------*)
let g2 = [ ((1,2),1); ((2,4),2); ((4,5),3); ((5,6),4); ((6,4),5); ((4,3),6);((3,1),7);
           ((1,7),8); ((7,8),9); ((8,9),10); ((9,7),11) ];;

let a, r = euler2 1 7 g2;;   (* g2 connexe eulérien *)
print_chaine a;;
let a, r = euler2 7 1 g2;;
print_chaine a;;

(*==== test avec des graphes eulériens, avec départ sur de mauvais sommets --*)
let a, r = euler2 1 1 g1;;  (* Exception: Failure "Pas de telle chaine eulerienne". *)
let a, r = euler2 4 5 g1;;
let a, r = euler2 4 4 g1;;

(*==== test avec des graphes connexes non eulériens --*)
(* Ponts de Koenisberg *)
let a, r = euler2 1 1 gk;;  (* Exception: Failure "Pas de telle chaine eulerienne". *)

(*==== test avec un graphe non connexe, à composantes connexes eulériennes (paires) --*)
let g3 = [ ((1,2),1);((2,4),2);((4,5),3); ((5,6),4);((6,4),5);((4,3),6);((3,1),7);
           ((7,8),9);((8,9),10);((9,7),11) ];;
(* g3 = biparti, par suppression de ((1,7),8) dans g2 *)

(*-- départ sur un bon sommet : *)
let a, r = euler2 1 1 g3;;
(*   val a : ((int * int) * int) list =
           [((1, 2), 1); ((2, 4), 2); ((4, 5), 3); ((5, 6), 4); ((6, 4), 5);
            ((4, 3), 6); ((3, 1), 7)]
     val r : ((int * int) * int) list = [((7, 8), 9); ((8, 9), 10); ((9, 7), 11)]
*)
print_chaine a;; (*  1--( 1)--> 2--( 2)--> 4--( 3)--> 5--( 4)--> 6--( 5)--> 4--( 6)--> 3--( 7)--> 1 *)

(*-- départ sur un couple de sommets dans des composantes connexes disjointes: *)
let a, r = euler2 1 7 g3;;  (* Exception: Failure "Pas de telle chaine eulerienne". *)

(*====================================*)
(* Recherche d'une chaine eulérienne  *)
(*====================================*)
let euler g =
  let a, r = match sommets_impairs g with
               | []     -> let (x, _), _ = List.hd g in euler2 x x g
               | x::[y] -> euler2 x y g
               | _      -> failwith "graphe non eulerien"
  in if r <> []
     then failwith "graphe non connexe"
     else a
;;
(* euler : (('a * 'a) * 'b) list -> (('a * 'a) * 'b) list = <fun> *)

(*==== TEST ====*)
print_chaine (euler g1);;
print_chaine (euler g2);;

print_chaine (euler g3);; (* Exception: Failure "graphe non connexe".*)
print_chaine (euler gk);; (* Exception: Failure "graphe non eulerien". *)
(*== Fin ==*)
```