I - Graphes orientés

1) a)
```
let matrice_vers_liste mat =
  let n = Array.length mat in
  let t = Array.make n [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if mat.(i).(j) = 1 then t.(i) <- j::t.(i)
    done
  done;
  t;;
```

b)
```
let successeurs_vers_predecesseurs t1 =
  let n = Array.length t1 in
  let t2 = Array.make n [] in
  let rec distribue l i = match l with
    |[] -> ()
    |j::q -> t2.(j) <- i::t2.(j);distribue q i
  in
  for k = 0 to n-1 do
    distribue t1.(k) k
  done;
  t2;;
```

2) a) Soit s et t deux sommets reliés par un chemin et considérons un chemin de plus petite longueur entre s et t. Supposons par l'absurde que ce plus court chemin de s à t est de longueur $n' > n-1$. Alors le chemin s'écrit $(s;s_1)(s_1;s_2)...(s_{n'-1};t)$ avec $\forall i \in [\![1;n'-1]\!]$, s_i des sommets de G. Il y a alors 3 cas possibles :

i) s'il existe $i \in [\![1;n'-1]\!]$ tq $s_i = s$, alors le chemin $(s_i;s_{i+1})...(s_{n'-1};t)$ est un chemin de s à t de longueur strictement plus petite que n' ; ce qui est absurde.

ii) s'il existe $i \in [\![1;n'-1]\!]$ tq $s_i = t$, alors le chemin $(s;s_1)...(s_{i-1};s_i)$ est un chemin de s à t de longueur strictement plus petite que n' ; ce qui est absurde.

iii) si on n'est pas dans ces deux premiers cas, alors nécessairement, il existe $(i,j) \in [\![1;n'-1]\!]^2$ tq $i \neq j$ et $s_i = s_j$, alors le chemin $(s;s_1)...(s_i;s_{j+1})...(s_{n'-1};t)$ est un chemin de s à t de longueur strictement plus petite que n' ; ce qui est absurde.

b) $N_{i,j}(1) = \begin{cases} 1 \text{ s'il y a une arête entre } i \text{ et } j \\ 0 \text{ sinon} \end{cases} = \begin{cases} 1 \text{ si } M_{i,j} = 1 \\ 0 \text{ si } M_{i,j} = 0 \end{cases} = M_{i,j}$

c) Chaque chemin de i à j de longueur p est composé d'un chemin de i à un sommet k de longueur $p-1$ et d'un chemin de ce sommet k à j de longueur 1. Toutes les combinaisons de chemin de cette forme sont distinctes. On obtient donc : $N_{i,j}(p+1) = \sum_{k=1}^{n} N_{i,k}(p) * N_{k,j}(1)$

d) Par le principe de récurrence sur la longueur p des chemins, on obtient $N_{i,j}(p) = M^p_{i,j}$.

e) G est fortement connexe $\Leftrightarrow \forall (s,t) \in V^2$ tq $s \neq t$, il y a au moins un chemin de s à t de longueur inférieure à $n-1$.

$\Leftrightarrow \forall (i,j) \in [\![1;n-1]\!]^2$ tq $i \neq j$, il existe $k \in [\![1;n-1]\!]$ tq $M^k_{i,j} > 0$.

$\Leftrightarrow \forall (i,j) \in [\![1;n-1]\!]^2$ tq $i \neq j$, $(T_{1,n})_{i,j} > 0$. (car tous les coefficients des puissances de M sont positifs ou nuls)

Or pour tout $i \in [\![1;n-1]\!]$, $(T_{1,n})_{i,i} \geq 1$, donc G est fortement connexe si et seulement si les coefficients de $T_{1,n} = I + M + M^2 + \cdots + M^{n-1}$ sont tous strictement positifs.

f) Pour le calcul classique du produit de deux matrices de taille n, on fait n^2 produits de vecteurs de taille n. Pour chacun de ces produits vectoriels, il faut faire n multiplications d'entiers et $n-1$ additions d'entiers. On fait donc $n^2 * (n + (n-1))$ opérations, ce qui est un $O(n^3)$.

g) Pour calculer $T_{1,n}$, on calcule consécutivement les différentes puissances de M ($n-2$ produits matriciels) et on somme les n matrices obtenues ($n-1$ sommes matricielles). On fait donc $(n-2) * (n^2 * (n + (n-1))) + (n-1) * n^2 = O(n^4)$ opérations.

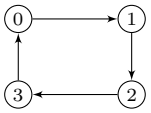
h) Les coefficients binomiaux étant tous strictement positifs, les coefficients de $T_{1,n}$ sont tous non nuls si et seulement si il en est de même des coefficients de $T_{2,n}$

i) Soit $(x, n) \in \mathbb{N}^2$. On peut calculer rapidement x^n en appliquant autant que possible le principe suivant :
— si n est pair, $x^n = (x^2)^{n/2}$;
— si n est impair, $x^n = x * (x^2)^{(n-1)/2}$.
On peut alors calculer x^n en faisant $O(\ln(n))$ produits.

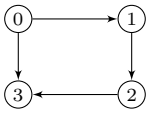
j) Ce principe peut s'appliquer au calcul rapide de la puissance p-ième d'une matrice car le produit matriciel est associatif.
$T_{2,n} = (I + M)^{n-1}$ Il suffit donc de faire une addition matricielle et de l'élever à la puissance $n-1$. Cela fait n^2 additions d'entiers plus $O(\log(n))$ produits matriciels. Le nombre d'additions est négligeable, on obtient donc une complexité en $\mathcal{O}(n^3 \log(n))$.

3) Matrice d'incidence sommets–arcs

a)



$$\begin{pmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ 1 & 0 & 0 & -1 \end{pmatrix}$$



$$\begin{pmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & 0 & 1 \end{pmatrix}$$

b) Pour chaque arête e de G notons :
— col_e la colonne de (a_{ij}) correspondante ;
— s_e l'origine de e ;
— p_e l'extrémité de e.
Supposons que G possède un cycle, alors il existe $(f_1, ..., f_n) \in E^n$ tq :
— $s_{f_1} = p_{f_n}$;
— $\forall i \in [\![1; n-1]\!], p_{f_i} = s_{f_i}$.
Alors pour tout $v \in V$, $\sharp\{f_i | s_{f_i} = v\} = \sharp\{f_i | p_{f_i} = v\}$. En conséquence $\sum_{i=1}^{n} col_{f_i} = 0$. Les colonnes de (a_{ij}) sont donc linéairement liées.

c) Pour chaque sommet s de G, notons lig_s la ligne de a_{ij} correspondante.
— cas 1 $(i = j)$: le coefficient L_{ii} est le résultat du produit $lig_{s_i}\, {}^t lig_{s_i}$ qui est la somme des carrés des coefficients de lig_{s_i}. Les coefficients de lig_{s_i} étant des éléments de $\{0; 1; -1\}$, Lii est le nombre d'arêtes ayant s_i comme origine ou comme extrémité.
— cas 2 $(i \neq j)$: le coefficient L_{ij} est le résultat du produit $v\, {}^t w$ où $v = lig_{s_i}$ et $w = lig_{s_j}$; or

$$\forall k \in [\![1; q]\!], v_k * w_k = \begin{cases} -1 & \text{si } e_k = (v_k, w_k) \text{ ou } e_k = (w_k, v_k) \\ 0 & \text{sinon} \end{cases}$$

L_{ij} est donc l'opposé du nombre d'arêtes entre s_i et s_j.

d)
```
let incidence_vers_adjacence mat_inc =
  let n = Array.length mat_inc in
  let m = Array.length mat_inc.(0) in
  let o = ref 0 in
  let e = ref 0 in
  let mat_adj = Array.make_matrix n n 0 in
  for j = 0 to m-1 do
    for i = 0 to n-1 do
      match mat_inc.(i).(j) with
      | -1 -> o := i
      | 1 -> e := i
      | _ -> ()
    done;
    mat_adj.(!o).(!e) <- 1
  done;
  mat_adj;;
```

e) Supposons par l'absurde que A soit inversible, il est alors nécessaire que A soit une matrice carrée, donc $p = q$. Cela implique également qu'il y a au moins cycle dans G. En conséquence, les colonnes sont linéairement liées, ce qui contredit l'inversibilité de A. A n'est donc jamais inversible.

f) Montrons par récurrence sur la taille que pour toute matrice carrée A' extraite de A, $\det A' \in \{0;1;-1\}$.
- Initialisation : les coefficients de A sont tous dans $\{0;1;-1\}$, donc la propriété est vraie pour les matrices de taille 1×1.
- Hérédité : soit $k \in \mathbb{N}^*$ tq $k < p$ et $k < q$ et supposons que toute matrice A' de taille $k \times k$ extraite de A, $\det A' \in \{0;1;-1\}$. Soit B' une matrice de taille $(k+1) \times (k+1)$ extraite de A, alors il y a 3 cas possibles :
  - **cas 1 :** il y a une colonne de B' avec que des 0.
    Dans ce cas, $\det B' = 0 \in \{0;1;-1\}$.
  - **cas 2 :** il y a une colonne de B' qui contient un seul coefficient non nul (1 ou -1).
    Dans ce cas, il existe A' une matrice carrée de taille k extraite de A tq $|\det B'| = 1 * |\det A'| \in \{0;1\}$. Donc $\det B' \in \{0;1;-1\}$.
  - **cas 3 :** toutes les colonnes de B' contiennent des coefficients nuls et exactement un 1 et un -1.
    Dans ce cas, la somme des lignes de la matrice est nul. Les lignes sont linéairement liées et $\det B' = 0 \in \{0;1;-1\}$.
- Conclusion : pour toute matrice carrée A' extraite de A on a $\det A' \in \{0;1;-1\}$

g) Si $q \neq p-1$ alors G ne peut pas être un arbre (d'après l'exercice 8) et $\tilde{A}$ n'est pas carrée.
Pour la suite, supposons que $q = p-1$.
- $\boxed{\Rightarrow}$ On suppose que G est un arbre. Soit $i_0 \in [\![1,p]\!]$ tel que l'on supprime la ligne i_0. Comme on sait que dans un arbre (non réduit à un unique sommet) il y a au moins deux sommets de degré 1, il existe un sommet différent de s_{i_0} qui est de degré 1. En développant le déterminant selon celle ligne qui contient donc une fois 1 (ou -1) et que des 0, on se ramène au cas d'un graphe qui est encore un arbre mais avec un sommet de moins. On peut alors conclure en rédigeant une récurrence sur le nombre de sommet de G.
- $\boxed{\Leftarrow}$ Comme $q = p-1$, il suffit de montrer que G est connexe. Supposons par l'absurde qu'il ne le soit pas et considérons une composante connexe ne contenant pas le sommet s_{i_0}. En faisant la somme des lignes qui correspondent aux sommets de cette composante on trouve 0 et donc $\det(\tilde{A}) = 0$.

4) Fermeture réflexive et transitive

a) $x\mathcal{R}^*y \Leftrightarrow$ il existe un chemin de x à y.
Soient x, y et z des éléments de V.
- (réflexivité) $x\mathcal{R}^{(0)}x$ donc $x\mathcal{R}^*x$
- (transitivité) Supposons que $x\mathcal{R}^*y$ et $y\mathcal{R}^*z$. Alors il existe n_1 et n_2 tq $x\mathcal{R}^{(n_1)}y$ et $y\mathcal{R}^{(n_2)}z$. Par récurrence directe, on obtient que $x\mathcal{R}^{(n_1+n_2)}z$. En conséquence, $x\mathcal{R}^*z$.
- (antisymétrie) Supposons que $x\mathcal{R}^*y$ et $y\mathcal{R}^*x$. Alors soit $x = y$ soit il y a un chemin de x à y et un chemin de y à x, c'est-à-dire qu'il y a un cycle passant par x et y. $\mathcal{R}^*$ est donc antisymétrique si et seulement s'il n'y a pas de cycle dans G.

II - Graphes non-orientés

5) Soit n le nombre de sommets. Le nombre d'arêtes vaut $k*n/2$. Or k est impair, donc le nombre de sommets est pair.

6) Un graphe non orienté G est un *arbre* si par définition il est connexe et sans cycle.
Notons n le nombre de sommets de G.

a) Montrons par récurrence sur le nombre n de sommets de G que si G est connexe alors G possède au moins $n-1$ arêtes.
- Initialisation : si G possède 2 sommets et qu'il est connexe, alors il contient au moins une arête $(2-1=1)$
- Hérédité : soit $k \in \mathbb{N}$ tq tout graphe connexe à k sommets contient au moins $k-1$ arêtes. Soit G un graphe connexe à $k+1$ sommets. Par hypothèse de récurrence, G contient au moins $k-1$ arêtes. Supposons par l'absurde que G n'a que $k-1$ arêtes. Dans ce cas, la somme des degrés des sommets vaut $2*(n-1) = 2n-2$. Ainsi, il y a au moins un sommet s de degré 1. Le graphe G' obtenu en ôtant s à G est donc un graphe connexe à k sommets avec seulement $k-2$ arêtes ce qui contredit l'hypothèse de récurrence. G a donc au moins k arêtes.

— Conclusion : pour tout $n \in \mathbb{N}^*$, si G est connexe avec n sommets, alors G possède au moins $n-1$ arêtes.

b) Montrons par récurrence sur le nombre n de sommets de G que si G est sans cycle alors G possède au plus $n-1$ arêtes.
— Initialisation : si G possède 3 sommets et qu'il a 3 arêtes, alors il contient un cycle.
— Hérédité : soit $k \in \mathbb{N}$ tq tout graphe à k sommets sans cycle contient au plus $k-1$ arêtes. Soit G un graphe sans cycle à $k+1$ sommets. Supposons par l'absurde que G a au moins $k+1$ arêtes, alors en particulier, G admet un sous graphe $\tilde{G}$ sans cycle à $k+1$ sommets et $k+1$ arêtes.
Si les sommets sont tous de degrés 2, alors $\tilde{G}$ admet une chaîne eulérienne cyclique (exercice précédent), ce qui est en contradiction avec l'hypothèse d'absence de cycle. Il y a donc un sommet s de degré 1. Le graphe G' obtenu en ôtant s à $\tilde{G}$ est donc un graphe sans cycle à k sommets avec k arêtes ce qui contredit l'hypothèse de récurrence. G a donc au plus k arêtes.
— Conclusion : pour tout $n \in \mathbb{N}^*$, si G est connexe avec n sommets, alors G possède au moins $n-1$ arêtes.

c) Un arbre est un graphe connexe sans cycle, donc pour montrer que G est un arbre si et seulement si G est sans cycle mais possède un cycle si on lui ajoute n'importe quelle arête, il suffit de montrer que si G est sans cycle, alors G est connexe si et seulement si lorsqu'on lui ajoute n'importe quelle arête, G possède un cycle.
Soit G un graphe sans cycle.
— Supposons que G est connexe, alors $\forall (s,t) \notin E$, il y a un chemin de s à t. Donc si on ajoute l'arête (s,t) au graphe G, le graphe obtenu possède un cycle.
— Supposons que G n'est pas connexe, alors il existe $(s,t) \in V^2$ tels qu'il n'y a pas de chemin de s à t, donc on peut ajouter l'arête (s,t) à G sans faire apparaître de cycle.
G est donc connexe si et seulement si lorsqu'on lui ajoute n'importe quelle arête, G possède un cycle.

d) Un arbre est un graphe connexe sans cycle, donc pour montrer que G est un arbre si et seulement si G est connexe mais n'est plus connexe dès qu'on lui retire n'importe quelle arête, il suffit de montrer que si G est connexe, G est sans cycle si et seulement si G n'est plus connexe dès qu'on lui retire n'importe quelle arête.
Soit G un graphe connexe.
— Supposons que l'on peut supprimer une arête (s,t) de G en préservant la connexité. Le graphe G' obtenu ainsi contient donc un chemin de s à t, et prolongé par l'arête (s,t) forme un cycle dans G. Ceci contredit l'hypothèse faite.
— Supposons que G admet un cycle, alors on peut supprimer n'importe quelle arête de ce cycle en préservant la connexité.
G est donc sans cycle si et seulement si G n'est plus connexe dès qu'on lui retire n'importe quelle arête.

e) Soit G un graphe.
G est sans cycle et possède exactement $n-1$ arêtes
$\Rightarrow^b$ G est sans cycle et si on ajoute une arête, on crée un cycle
$\Rightarrow^c$ G est un arbre.
Ainsi G est sans cycle et possède exactement $n-1$ arêtes $\Rightarrow$ G est un arbre.
Réciproquement, G est un arbre $\Rightarrow^{def}$ G est connexe et sans cycle. Donc G est un arbre $\Rightarrow^{a,b}$ G a au moins $n-1$ arêtes et G a au plus $n-1$ arêtes.
Ainsi G est un arbre $\Rightarrow$ G est sans cycle et possède exactement $n-1$ arêtes.

f) Montrer que G est un arbre si et seulement si G possède exactement $n-1$ arêtes et est connexe. Soit G un graphe.
G est connexe et possède exactement $n-1$ arêtes
$\Rightarrow^a$ G est connexe et si on enlève une arête, G n'est plus connexe
$\Rightarrow^d$ G est un arbre.
Ainsi G est connexe et possède exactement $n-1$ arêtes $\Rightarrow$ G est un arbre.
Réciproquement, G est un arbre $\Rightarrow^{def}$ G est connexe et sans cycle. Donc G est un arbre $\Rightarrow^{a,b}$ G a au moins $n-1$ arêtes et G a au plus $n-1$ arêtes.
Ainsi G est un arbre $\Rightarrow$ G est connexe et possède exactement $n-1$ arêtes.

7) Graphe biparti

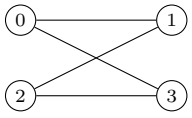
a) Soit G un graphe connexe qui n'a pas de cycle. Soit r un sommet de G que nous allons considérer comme la racine de l'arbre. Alors on définit les fonctions de profondeur $prof : V \to V$ et de parentalité $p : V \setminus \{r\}$ comme suit :
- $prof(r) := 0$
- pour tout s tq $(r, s) \in E$, $prof(s) := 1$ et $p(s) := r$
- pour tout $k \in \mathbb{N}$, pour tout s tq $(s, t) \in E$, $p(s)$ pas définie et $prof(t) = k$, $prof(s) := k + 1$ et $p(s) := t$.

G est connexe, donc $prof$ et p sont bien définies respectivement sur V et $V \setminus \{r\}$.

Montrons que $V_1 = \{s \in V \,|\, p(s)\%2 = 1\}$ et $V_2 = \{s \in V \,|\, p(s)\%2 = 0\}$ forment une partition de V tq $E \subseteq V_1 \times V_2 \cup V_2 \times V_1$.

Supposons par l'absurde qu'il existe une arête $e = (s, t) \in E$ tq $e \in V_1 \times V_1 \cup V_2 \times V_2$. Alors, en remontant de pères en fils, il existe un chemin de s à r et un chemin de t à r n'empruntant pas l'arête e. En conséquence, il y a un cycle qui passe par s et t dans G. Il y a alors contradiction avec le fait que G soit un arbre.

Exemple de graphe biparti, connexe qui n'est pas un arbre :



est biparti ($V_1 = \{0; 2\}$ et $V_3 = \{1; 3\}$), connexe et ce n'est pas un arbre car il contient un cycle $(0 - 1 - 2 - 3 - 0)$.

b) — Soit G un graphe. Supposons, par l'absurde, qu'il existe un cycle de longueur impaire $(x_0, x_1, ..., x_{2p})$ et que G est biparti. Il existe donc une partition de V, V_1 et V_2 tq $E \subseteq V_1 \times V_2 \cup V_2 \times V_1$. Sans perte de généralité, on peut alors supposer que $x_0 \in V_1$ dans ce cas, $x_1 \in V_2$ et par récurrence directe : $\forall k \in [\![0; 2p]\!], x_k \in \begin{cases} V_1 & \text{si } k\%2 = 0 \\ V_2 & \text{si } k\%2 = 1 \end{cases}$ On obtient donc $x_{2p} \in V_1$ ce qui est contradictoire avec l'existence d'une arête entre $x_0 \in V_1$ et $x_{2p} \in V_1$. G ne peut donc pas être biparti et contenir un cycle de longueur impaire.

— Soit G un graphe. Supposons que G n'est pas biparti. Considérons l'algorithme qui parcourt en largeur G depuis un sommet r en attribuant alternativement les sommets à V_1 et V_2. Un sommet s à distance de r (longueur minimale d'un chemin entre r et s) paire sera dans V_1 et les autres dans V_2. G n'étant pas biparti, il existe alors deux sommets s et t tous les deux dans V_1 ou tous les deux dans V_2 tq $(s, t) \in E$. Cela implique qu'il y a un cycle dans G de longueur "distance de s à r"+"distance de t à r"+1 qui est un nombre impair. Donc si G n'est pas biparti, il contient nécessairement un cycle de longueur impaire.

On bien montré qu'un graphe est biparti si et seulement s'il ne contient pas de cycle de longueur impaire.

c)
```
let biparti g =
let n = Array.length g in
let rep = Array.make n 2 in
let rec parcours_liste l k =
  match l with
  |[]->()
  |s::q -> propage s k; parcours_liste q k
and propage i k =
  if rep.(i) = 2 then (rep.(i) <- k; parcours_liste g.(i) (1-k))
  else if rep.(i) <> k then  failwith "pas biparti"
in
let rec initie i =
  if rep.(i) = 2 then propage i 0;
  if i<n-1 then  initie (i+1)
in
initie 0;
rep;;


let biparti_Denis g =
let n = Array.length g in
```

```
  let rep = Array.make n (-1) in
  let rec parcours l k =
    match l with
    |[]->()
    |x::q -> if rep.(x) <> (-1) then ()
        else
        ( rep.(x) <- 1 - t.(k);
        parcours g.(x) x);
        parcours q k
 in rep.(0) <- 0 ; parcours g.(0) 0;
rep;;
```

8) Le théorème de Hierholzer

a) Soit G un graphe. Supposons que G est eulérien sans sommets isolés. Alors il existe $c = (s_1, ..., s_n) \in V^n$ qui emprunte une et une seule fois chaque arête de G.
— pour tout sommet s distinct de s_1 et de s_n, le nombre k_s d'arêtes dont s est le départ est égal au nombre d'arêtes dont s est l'arrivée. Comme la chaîne est eulérienne, le degré de s est alors égal à $2 * k_s$. Le sommet s est donc de degré pair.
— si $s_1 = s_n$, le degré de s_1 est pair pour la même raison.
— si $s_1 \neq s_n$, alors s_1 et s_n sont de degrés impairs. En effet, en dehors de l'arête d'extrémité, on peut appliquer le même raisonnement que pour les autres sommets. Leur degré est alors un nombre pair auquel on ajoute 1 pour le compte de l'arête d'extrémité.

Pour finir, la chaîne étant eulérienne, si elle passe par tous les sommets, alors le graphe est connexe ; sinon, cela implique que les sommets qui ne sont pas visités par la chaîne sont des sommets isolés, ce qui a été exclu.

b) Soit G un graphe avec que des sommets de degré pair et au moins une arête. Supposons que G ne contient pas de cycle.

On considère un chemin construit à partir d'une arête sans réutiliser deux fois la même arête jusqu'à être bloqué. Alors :
— soit on est déjà passé par ce sommet et donc on a exhibé un cycle ;
— soit le degré du sommet d'arrivée étant pair, il y a forcément une autre arête que celle qu'on vient d'emprunter, donc on n'est pas bloqué !

Il y a donc nécessairement un cycle.

Une fois un cycle de G exhibé, si on considère le sous-graphe de G construit en enlevant les arêtes de ce cycle à G, on obtient un graphe qui peut ne pas être connexe, mais dont tous les sommets ont un degré pair. Tant qu'il reste des arêtes non utilisées, on a montré que l'on peut construire un nouveau cycle. On peut ainsi partitionner l'ensemble des arêtes de G en un nombre fini de cycles.

G étant connexe, les cycles de cette partition peuvent être concaténés en une chaîne eulérienne.

c) Dans le cas où G possède exactement deux sommets s et t de degré impair, alors il y a deux cas :
— si $(s, t) \notin E$, on peut considérer le graphe G' obtenu en ajoutant l'arête (s, t). Grâce à la question précédente, on construit un cycle eulérien dans G'. En supprimant (s, t) de ce cycle, on obtient une chaîne eulérienne de G.
— si $(s, t) \in E$, on peut considérer le graphe G' obtenu en ôtant l'arête (s, t) (voire un des deux sommets pour préserver la connexité si l'un des deux est de degré 1, la connexité assure qu'au moins un des deux est relié à une autre arête). Grâce à la question précédente, on construit un cycle eulérien dans G'. En ajoutant (s, t) dans cette chaîne eulérienne de G en cassant le cycle à un passage sur s ou t (il y en a forcément un parce qu'on a au plus ôté un sommet en contruisant G').

III - Graphes pondérés

9) Pondération des sommets

Soit $p' : E \to \mathbb{R}$ qui associe un poids à chaque arête de G de la façon suivante : $\forall (s, s') \in E, p'((s, s')) = p(s')$.

Alors pour tout couple $(s,t) \in V^2$, pour tout chemin ch de s à t, $p(ch) = p'(ch) - p(s)$.

Les chemins de coût minimum avec cette nouvelle pondération seront donc les mêmes qu'avec la pondération initiale.

10) Approvisionnement d'un stock

a) On modélise le stock de l'entreprise par un graphe dont les sommets sont identifiés par des couples (i,x) où $i \in \{0,1,\dots,n\}$ est le numéro du mois et $x \in \{0,1,\dots,S\}$ est le stock en début de mois (**avant** d'acheter le stock pour le mois [1]).

On ajoute alors les arêtes. Précisément de tout sommet (i,x) et tout $\alpha \in \mathbb{N}$ (qui correspond à la somme achetée) il part une arête vers $(i+1, x+\alpha-c_i)$ à condition que
— on ait $x+\alpha \leq S \iff \alpha \leq S-x$ (pour que le stock ne dépasse pas S)
— on ait $x+\alpha-c_i \geq 0 \iff \alpha \geq c_i - x$ (pour que la quantité restante à la fin du mois soit positive).
Le poids d'une telle arête est donnée par $p_i\alpha$.

On cherche alors le chemin de poids minimal de $(0,s_0)$ à $(n,0)$ où s_0 est le stock initial.

Il ne reste plus qu'à implémenter Dijkstra [2] :

b)
```
let stock c pr sinit smax =
  let n = Array.length c in
  let visit = Array.make_matrix (n + 1) (smax + 1) false in
  let antecedent = Array.make_matrix (n + 1) (smax + 1) (-1, -1) in
  let f = creeFileVide () in
    insertFile f ((0, sinit), 0, (- 1, - 1));
    let rec parcours f =
    let ((j, s), p, a) = defile f in
    if not (visit.(j).(s)) && j < n then
        (visit.(j).(s) <- true;
         antecedent.(j).(s) <- a;
         for alpha = max 0 (c.(j) - s) to smax - s do
            if not visit.(j + 1).(s + alpha - c.(j))
            then
            insertFile f ((j + 1, s + alpha - c.(j)), p + pr.(j) * alpha, (j, s))
          done);
          if (j,s) <> (n,0) then parcours f in
    parcours f;
  antecedent;;
```

On reconstruit alors le chemin comme d'habitude

On implémente les files de priorités

```
type  'a filePriorite = {mutable taille : int ; donnee : 'a vect};;

let echange f k p =
  let temp = (f.donnee).(k) in
  (f.donnee).(k) <- (f.donnee).(p);
  (f.donnee).(p) <- temp;;

let rec descente f k =
  let (xk, pk, ak) = f.donnee.(k) in
  let (x2k1,p2k1,a2k1) = f.donnee.(2*k+1) in
  let (x2k2,p2k2,a2k2) = f.donnee.(2*k+2) in
  if (2*k +1 < f.taille && pk > p2k1 || (2*k +2 < f.taille && pk > p2k2)) then
    if (2*k +2 < f.taille && p2k1 > p2k2) then
      (echange f k (2*k+2);
       descente f (2*k+2))
    else
      (echange f k (2*k+1);
```

1. La convention n'est pas la même que celle prise en cours mais cela semble un peu plus simple ainsi
2. Pour ne pas alourdir le code, on a supposé que les prix stockés dans le tableau `pr` sont entiers et non flottants

```
        descente f (2*k+1));;

let estVideFile f = f.taille = 0;;

let defile f =
  if f.taille = 0 then failwith "la file est vide"
  else
    (let u = f.donnee.(0) in
     echange f 0 (f.taille - 1);
     f.taille <- f.taille -1;
     descente f 0;
     u);;

let insertFile  f x   =
  f.taille <- f.taille +1;
  f.donnee.(f.taille -1)<- x;
  let k = ref (f.taille -1) in
  let p = ref ((!k-1)/2) in
  let (xk, pk, ak) = f.donnee.(!k) in
  let (xp,pp,ap) = f.donnee.(!p) in
  while !k <> 0 && pk < pp do
    echange f !k !p;
    k := !p;
    p := (!p -1)/2
  done;;
```

11) Structure Union-Find :

a)
```
let creer n =
  let t = Array.make n 0 in
  for i = 0 to n-1 do
    t.(i) <- i
  done;
  t;;
```

b)
```
let rec rep t i =
  if t.(i)=i then i
  else rep t t.(i);;
```

c)
```
let fusion t i j =
  t.(rep t i) <- t.(rep t j);;
```

d) La fonction `representant` parcourt les représentants successifs jusqu'à la racine, au pire, chaque sommet est traité une fois. La complexité est linéaire en le nombre d'éléments. La fonction `fusion` fait simplement 2 appels à la fonction `representant`. La complexité est donc également linéaire en le nombre d'éléments.

e)
```
let creer2 n =
  let t = Array.make n (0,0) in
  for i = 0 to n-1 do
    t.(i) <- (i,0)
  done;
  t;;

let rec rep2 t i =
  let x = fst(t.(i)) in
  if x=i then i
  else (let y = rep2 t x in
        t.(i) <- (y,snd(t.(i)));
        y);;

let fusion2 t i j =
```

```
let ri = rep2 t i in
let rj = rep2 t j in
if snd(t.(ri)) <= snd(t.(rj)) then
  (t.(ri) <- (rj, snd(t.(ri)));
   t.(rj) <- (fst(t.(rj)), max (snd(t.(rj))) (snd(t.(ri))+1)))
else (t.(rj) <- (ri, snd(t.(rj)));
      t.(ri) <- (fst(t.(ri)), snd(t.(ri))));;
```

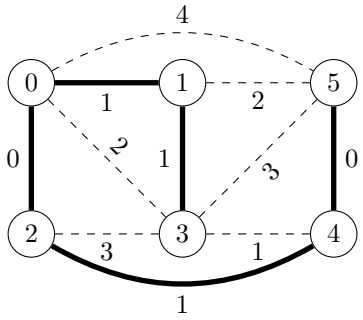
12) Recherche d'arbres couvrants de poids minimal :

a) i) Soit $G_0 = G$, alors G_0 est connexe. S'il n'a pas de cycle, alors c'est un arbre couvrant de G. Sinon, on peut supprimer n'importe quelle arête d'un cycle. En effet, pour tout chemin joignant deux sommets passant par l'arête supprimée, on peut construire un chemin passant par le reste du cycle à la place de cette arête. On peut ainsi supprimer des arêtes tant que G_0 a des cycles en préservant la connexité. À la fin, G_0 est un bien un arbre couvrant de G.

ii) À la fin de l'algorithme, (V, E_m) est sans cycle. Montrons qu'il est connexe. Supposons par l'absurde que (V, E_m) n'est pas connexe. Alors il existe $V_1 \subseteq V$ et $V_2 \subseteq V$ tel qu'il n'y ait pas d'arête entre ces deux ensembles de sommets dans (V, E_m). Or, par connexité de G, il existe $e_i \in E$ reliant ces deux ensembles. En conséquence, lors de l'itération i, le graphe $(V, E_i \cup \{e_i\})$ n'avait pas de cycle, donc e_i aurait dû être placé dans E_{i+1}. D'où l'absurdité !

b) Il y a au moins un arbre couvrant (question a).i)) et au plus $|V| - 1$ parmi $|E|$ (Exercice 8). La fonction poids admet donc au moins un minimum sur l'ensemble des arbres couvrants.

c) i) Les arêtes sont triées par poids croissants :
$[(0, 2, 0); (4, 5, 0); (0, 1, 1); (1, 3, 1); (2, 4, 1); (3, 4, 1); (0, 3, 2); (1, 5, 2); (2, 3, 3); (3, 5, 3); (0, 5, 4)]$



ii) D'après l'exercice 8 et l'hypothèse que (V, E_1) et (V, E_2) sont sans cycle, chaque composante connexe de (V, E_1) et (V, E_2) est un arbre. Notons qu'ajouter une arête entre des sommets de deux composantes connexes distinctes ne peut former de cycle.

Procédons par contraposée. On suppose que pour toute arête $e \in E_2 \setminus E_1$ le graphe $(V, E_1 \cup \{e\})$ a un cycle. Cela implique que toutes les arêtes de E_2 relient deux sommets d'une même composante connexe de (V, E_1).

De ce fait, toute composante connexe de (V, E_2) est inclus dans une composante connexe de (V, E_1). Maintenant, comme les composantes connexes de (V, E_1) sont des arbres et que (V, E_2) est sans cycle, cela implique que $|E_2| \leq |E_1|$.

iii) Soit T un arbre couvrant de poids minimum et R l'arbre construit par l'algorithme de Kruskal. Pour tout $1 \leq k \leq |V| - 1$, on notera respectivement T_k et R_k les sous graphes de T et R contenant uniquement leurs k arêtes de poids minimaux. De plus, on notera $p(G)$ la somme des poids des arêtes d'un graphe G et $p(e)$ le poids d'une arête e.

Montrons par récurrence sur k que pour tout $1 \leq k \leq |V| - 1$, $p(R_k) \leq p(T_k)$.

— (I) pour $k = 1$: l'arête de poids minimum dans R est l'arête minimum dans G, son poids est bien inférieur ou égale au poids de toutes les arêtes de T.

— (H) soit $1 \leq k < |V| - 1$ tel que $p(R_k) \leq p(T_k)$. Remarquons que R_k et T_{k+1} satisfont les hypothèses de la question précédente, donc il y a une arête e de T_{k+1} qui n'est pas dans R_k et qui peut y être ajoutée sans former de cycle. Cette arête est forcément de poids supérieur ou égal au poids de l'arête e' qui va être ajoutée par l'algorithme pour construire R_{k+1}. Ainsi, $p(T_{k+1}) = p(T_k) + p(e') \leq p(R_k) + p(e) \leq p(R_{k+1})$.

— (C) par le principe de récurrence, pour tout $1 \leq k \leq |V|-1$, on a bien $p(R_k) \leq p(T_k)$.
En conséquence, l'arbre couvrant construit par l'algorithme de Kruskal est bien de poids minimal.

iv) On considère ici que le graphe en entrée est codé par une liste des arêtes triée par poids croissants. l'ensemble des composantes connexes de l'arbre en construction est codé par la structure union-find. Ainsi, une arête crée un cycle si et seulement si les deux sommets ont le même représentant.

```
let kruskal g =
  let cc = creer2 (List.length(g)) in
  let rec construction aretes acc=
    match aretes with
    |[] -> acc
    |(a,b,p)::q -> if rep2 cc a <> rep2 cc b then
                          (fusion2 cc a b;construction q ((a,b,p)::acc))
                   else construction q acc

  in
  construction g [];;
```

13) Arbre couvrant aléatoire : On considère un graphe $G = (V, E)$ supposé connexe. Pour tout sommet x de V, on note V_x les voisins de x. On veut construire un arbre couvrant (voir exercice 12) aléatoire. On va utiliser l'algorithme d'Aldous-Broder (1990) qui consiste à générer une marche aléatoire dans le graphe.

Algorithme 1 : Algorithme d'Aldous - Broder : recherche d'arbre couvrant aléatoire

Entrée : Graphe $G = (V, E)$
$x \leftarrow$ sommet au hasard du graphe
$E' \leftarrow \emptyset$
$V' \leftarrow \{x\}$
tant que $V' \neq V$ **faire**
  $y \leftarrow$ sommet au hasard dans V_x
  si $y \notin V'$ **alors**
    $V' \leftarrow S' \cup \{y\}$
    $E' \leftarrow E' \cup \{(x, y)\}$
  fin
  $x \leftarrow y$
fin

On renvoie alors $T = (V, E')$.

a) Pour chaque arête ajoutée, on ajoute un sommet dans V', donc à la fin, il y a bien $|V|-1$ arête dans E'. De plus, le graphe construit est connexe car pour chaque sommet ajouté, on ajoute une arête qui relie ce sommet à un sommet du graphe construit jusque là. Par l'exercice 8, on peut donc conclure qu'à la fin, le graphe construit étant connexe avec $|V|-1$ arête, c'est bien un arbre. Cet arbre couvrant tous les sommets de G, le graphe construit est un arbre couvrant.

b)
```
let aldous g =
  let n = Array.length g in
  let x = ref (Random.int n) in
  let y = ref 0 in
  let res = Array.make n [] in
  let tailleVx = Array.make n 0 in
  for i = 0 to (n-1) do
    tailleVx.(i) <- List.length(g.(i))
  done;
  let visit = Array.make n 0 in
  visit.(!x) <- 1 ;
  let nbv = ref 1 in
  while !nbv < n do
    y := List.nth g.(!x) (Random.int tailleVx.(!x));
    if visit.(!y) = 0 then
```

```
      (visit.(!y) <- 1;
       nbv := !nbv + 1;
       res.(!x)<- !y::res.(!x);
       res.(!y)<- !x::res.(!y));
    x := !y
done;
res;;
```