1

Graphes

« *Des arêtes, des sommets, des poids.* »

1.1 Fête de Noël sans conflit

On considère une grande famille de n personnes avec beaucoup de gens qui ne s'entendent pas, représentée par une matrice $A = (a_{i,j}) \in \mathscr{M}_n$ telle que $a_{i,j} = a_{j,i} = 1$ si i et j ne peuvent pas se voir, 0 sinon. On a deux maisons de famille et on veut partager les n personnes entre ces deux maisons pour les fêtes de Noël, de façon que deux personnes qui ne s'entendent pas soient toujours dans des maisons différentes.

Question 1

Montrer par un exemple qu'il n'est pas toujours possible de répartir les membres de la famille entre les deux maisons pour éviter tout conflit.

Question 2

On va mettre le résultat dans un tableau `maison` de taille n tel que `maison.(i-1)=1` si la personne i est dans la première maison et `maison.(i-1)=-1` si la personne i est dans l'autre maison. Écrire une fonction `partage (a:int array array) : int array` qui teste s'il est possible de faire un partage sans conflit et propose un partage lorsque cela et possible.

Question 3

Deux personnes i et j sont dites en relation d'influence s'il existe une suite k_1, ..., k_l telle que $a_{i,k_1} = a_{k_1,k_2} = ... = a_{k_{l-1},k_l} = a_{k_l,j} = 1$. Montrer que cette relation est une relation d'équivalence. Soit N le nombre de classes d'équivalence, si l'on pose par convention $a_{i,i} = 1$ pour tout i. Montrer qu'un partage sans conflit est possible si et

seulement si il n'existe pas de suite i_1, ..., i_{2l+1} telle que $a_{i_1,i_2} = ... = a_{i_{2l+1},i_1} = 1$. Montrer que le nombre de partages sans conflits est soit 0, soit $2N$.

Question 4

On suppose maintenant qu'il y a trois maisons de famille. Proposer un algorithme qui fasse un partage sans conflits entre les trois maisons lorsque cela est possible. Qu'en pensez-vous ?

CORRIGÉ

Question 1

Il suffit d'une famille de trois personnes qui sont chacune en conflit avec les deux autres.

Question 2

On met la personne 1 dans la maison 1. On met toutes les personnes avec lesquelles elle est en conflit, s'il en existe, dans la maison -1. Puis on la marque comme « vue ». À chaque étape, on cherche une personne i qui a déjà été mise dans une maison mais n'est pas encore vue.

Premier cas de figure : il existe une telle personne i. On vérifie que i n'est pas en conflit avec des gens déjà placés dans la même maison, et on met tous les gens en conflit avec i encore non placés dans l'autre maison. Puis on marque la personne i comme « vue ».

Deuxième cas de figure : il n'existe pas de tel i. On prend alors une personne quelconque qui n'a pas encore été vue, et on la met dans la maison 1, puis on continue.

En termes plus techniques, on peut dire qu'on définit un graphe à partir de la matrice des conflits, avec un sommet pour chaque personne, une arête reliant deux sommets si les personnes correspondantes sont en conflit. L'algorithme consiste à traiter ce graphe composante connexe par composante connexe. Dans chaque composante, toutes les affectations des personnes dans les maisons sont déterminées dès qu'on a placé la première personne (qui par défaut est mise dans la maison 1).

On peut démontrer facilement l'invariant suivant :

Invariant : si i est vue, alors i est placée dans l'une des deux maisons, et toutes les personnes avec lesquelles i est en conflit se trouvent dans l'autre maison.

Ainsi, l'algorithme, lorsqu'il trouve une solution, trouve toujours une solution correcte.

Inversement, l'algorithme échoue lorsqu'il place i dans une maison où se trouve déjà une personne j en conflit avec i. Donc j est placée dans une maison mais n'a pas encore été vue, et l'algorithme n'a donc utilisé que le premier cas de figure entre le moment où j a été placée et celui où i a été placée. On a donc une chaîne de conflits telle qu'il ne peut exister de partage sans conflit (voir question 3).

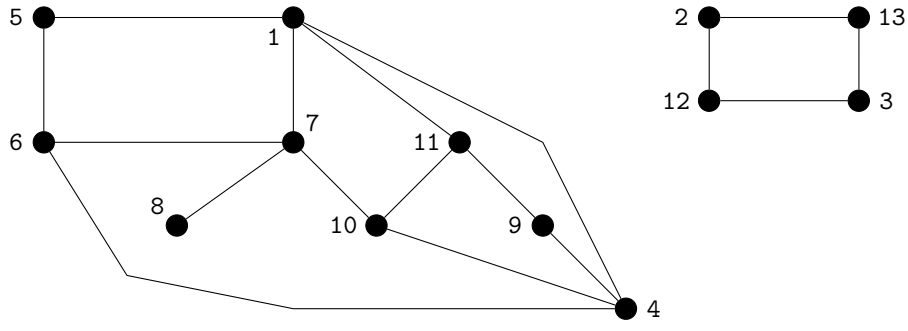
Ceci démontre que l'algorithme résout bien le problème posé.

```
let partage (a:int array array) : int array =
  let n = Array.length a in
```

```
let maison = Array.make n 0 in
let vu = Array.make n false in

for i=0 to n-1 do
  if not vu.(i) then ( (* nouvelle composante connexe *)
    maison.(i) <- 1;
    vu.(i) <- true;
    for j=i+1 to n-1 do (* ajout des personnes en conflit *)
      if a.(i).(j) = 1 then
        maison.(j) <- -1
    done;

    (* recherche d'un personne non vue dans une maison *)
    for j=i+1 to n-1 do
      if maison.(j) <> 0 && not vu.(j) then (
        (* on vérifie que j n'est pas en conflit *)
        for k=i+1 to n-1 do
          if k <> j && maison.(k) = maison.(j) && a.(k).(j) = 1 then
            failwith "partage impossible"
        done;

        (* on met les ennemis de j dans l'autre maison *)
        for k=i+1 to n-1 do
          if k <> j && a.(k).(j) = 1 then
            maison.(k) <- -maison.(j)
        done;

        vu.(j) <- true;
      )
    done
  )
done;
maison
```

Il est possible de regrouper certains tests, au détriment de la lisibilité du programme.

L'exemple suivant peut clarifier le fonctionnement de l'algorithme. Dans la figure suivante, chaque point représente une personne, et deux personnes sont reliées par un trait si elles sont en conflit.



Le tableau ci-dessous donne, pour chaque personne, l'étape à laquelle elle a été placée, ainsi que la maison dans laquelle elle a été placée.

personne	maison 1	maison -1
1	1	
2	7	
3	9	
4		4
5		2
6	3	
7		2
8	6	
9	5	
10	5	
11		2
12		8
13		8

Question 3

Réflexivité : $a_i, i = 1$ pour tout i par convention.
Symétrie : s'il existe une suite de personnes de i à j telle que chaque personne est en conflit avec la suivante et la précédente, la même suite dans l'ordre inverse prouve que $a_{i,j} = 1$.
Transitivité : s'il existe une suite de personnes en conflit de i à j et une autre suite de j à k, la concaténation des deux suites donne une suite de personnes en conflit de i à k.

On a donc une relation d'équivalence. Les classes d'équivalence correspondent à ce que nous avons appelé les « composantes connexes ».

Clairement, s'il existe une suite de i_1 à lui même telle que décrite dans l'énoncé et si i_1 est dans la maison 1, alors i_2 doit être placé dans la maison -1, i_3 dans la maison 1, ..., i_k dans la maison $(-1)^{k-1}$, donc i_{2l+1} dans la maison 1 et $i_1 = i_{2l+2}$ dans la maison -1 : contradiction.

Inversement, si la fonction de la question 2 aboutit à la réponse « partage impossible », c'est parce qu'une personne k dans la composante connexe de i se trouve contrainte d'être placée dans les deux maisons à la fois. Il existe donc deux suites de personnes en conflit allant de i à k, l'une de longueur paire et l'autre de longueur impaire. La concaténation de la première suite avec la deuxième suite en ordre inverse donne une suite de i à i de longueur impaire.

Les différentes classes d'équivalence étant indépendantes les unes des autres, le nombre total de partages sans conflits est le produit du nombre de partages sans conflit pour chaque classe. Ce nombre, pour la classe d'équivalence de i est soit 0 s'il existe une suite impaire de i à i, soit 2 sinon : en effet, le premier élément de la classe d'équivalence peut être placé au choix dans la maison 1 ou -1, et toutes les autres affectations sont ensuite déterminées par la parité de la longueur de la suite reliant une personne à i. Le produit est donc 0 ou 2^N.

Question 4

La première idée consiste à mettre la personne 1 dans la maison 1, son premier ennemi dans la maison 2, et ses ennemis successifs, soit dans la maison 2 s'ils ne sont pas en conflit avec quelqu'un se trouvant déjà dans la maison 2, soit dans la maison 3 sinon. Cela conduirait à une généralisation facile de l'algorithme de la question 2, mais malheureusement cet algorithme ne marche pas : il existe des conflits pour lesquels il ne trouve pas de solution alors qu'il en existe une. Par exemple :

$a_{1,2} = a_{1,3} = 1$;
$a_{2,1} = a_{2,3} = a_{2,4} = a_{2,5} = 1$,
$a_{3,1} = a_{3,2} = a_{3,5} = 1$,
$a_{4,2} = a_{4,5} = 1$,
$a_{5,2} = a_{5,3} = a_{5,4} = 1$.
(Les entrées non définies sont 0). Un algorithme similaire à celui de la question 2 ferait les affectations suivantes :
`maison.(0) <- 1`,
`maison.(1) <- 2`,
`maison.(2) <- 3`,
`maison.(3) <- 1`,
`maison.(4)` « impossible »,
alors que les affectations de 1, 2, 3, 4, 5 à 1, 2, 3, 2, 1 satisfont les contraintes.

De fait, il n'existe pas de modification astucieuse de l'algorithme qui le rende correct : ce problème, connu sous le nom de coloriage d'un graphe par trois couleurs, est bien connu en informatique théorique pour être difficile. Il n'existe actuellement pas d'algorithme efficace pour le résoudre, c'est à dire qui fasse beaucoup mieux que regarder les 3^n affectations possibles des personnes dans les maisons.

Plus généralement, le problème de coloriage d'un graphe avec le nombre minimum de couleurs, de sorte que deux sommets liés par une arête soient toujours de couleurs différentes, est le problème du nombre chromatique. Un problème célèbre, resté longtemps conjecture, est de démontrer que tout graphe planaire (représentable sur un plan sans que ses arêtes se coupent) a un nombre chromatique au plus égal à 4, ou encore que toute carte planaire peut être coloriée avec au plus 4 couleurs de façon que deux pays ayant une frontière commune soient toujours de couleurs différentes : ceci a finalement été démontré en réduisant l'ensemble de tous les graphes planaires possibles à un nombre fini mais élevé de cas de graphes (environ 1900), qui furent ensuite examinés un par un par ordinateur.

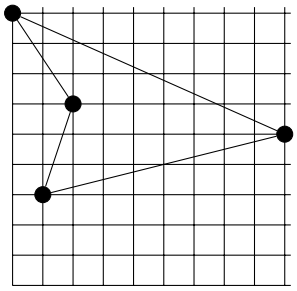
⋆⋆⋆⋆⋆⋆⋆

1.2 La tournée du facteur

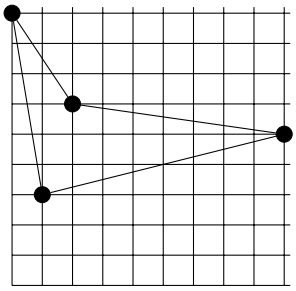
Un facteur doit distribuer du courrier dans n maisons différentes. Les coordonnées de ces maisons (qui sont supposées se trouver dans un village parfaitement plat) sont repérées par deux tableaux de flottants `x` et `y` de taille n.

La i-ième maison est de coordonnées `(x.(i), y.(i))`. On supposera que l'ordre dans lequel apparaissent les maisons est tel que `x.(0)` < `x.(1)` < ... < `x.(n)`. Le

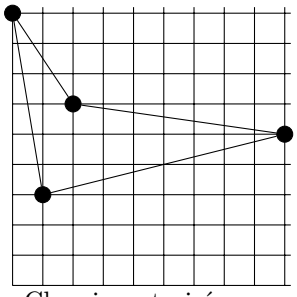
facteur cherche un chemin qui passe par toutes les maisons et qui minimise la distance totale à parcourir. Pour simplifier le problème, les seuls chemins que l'on autorisera seront des chemins qui partent du point d'abscisse minimale `(x.(0), y.(0))`, puis qui vont *toujours dans le sens des abscisses croissantes* vers le point d'abscisse maximale `x.(n-1), y.(n-1)` puis qui repartent *toujours dans le sens des abscisses décroissantes* vers le point de départ `(x.(0), x.(0))`. La figure suivante donne des exemples de chemins autorisés et de chemins interdits.



Chemin non autorisé



Chemin autorisé optimal



Chemin autorisé non optimal

Question 1

Proposer un algorithme qui donne le plus court des chemins autorisés. Justifier votre algorithme. Évaluer (en fonction de n) un ordre de grandeur du temps qu'il nécessite, en supposant qu'une addition ou une comparaison prennent une unité de temps.

Question 2

Si maintenant tous les chemins possibles sont autorisés, votre algorithme de la question précédente donne-t-il toujours un chemin de longueur minimale ?

CORRIGÉ

Question 1

Ce problème est un cas particulier du « problème du voyageur de commerce » dans lequel on recherche un circuit optimal dit « bitonique ». Un algorithme efficace utilise la programmation dynamique.

On note $(p_1, p_2, ..., p_n)$ l'ensemble des points du plan correspondant aux maisons ordonnées par abscisse croissante. On appelle « chemin bitonique » $P_{i,j}$ (où $i \leq j$), un chemin incluant les points p_1, p_2, ..., p_j qui commence à un point p_i, va jusqu'au point p_1 en allant toujours dans le sens des abscisses décroissantes puis va jusqu'au point p_j en allant toujours dans le sens des abscisses croissantes. Notons que p_j est le point le plus à droite dans $P_{i,j}$ et fait partie du sous-chemin de $P_{i,j}$ allant vers la droite. On cherche donc ici le chemin bitonique $P_{n,n}$ de taille minimale.

On note $|p_i, p_j|$ la distance euclidienne entre les points p_i et p_j et on note $b_{i,j}$ (où $i \leq j$) la longueur du chemin bitonique $P_{i,j}$ le plus court. La seule valeur $b_{i,i}$ dont nous avons besoin est $b_{n,n}$, correspondant à la longueur recherchée. On a la formule suivante pour $b_{i,j}$ avec $1 \leq i \leq j \leq n$:

$b_{1,2} = |p_1, p_2|$;
$b_{i,j} = b_{i,j-1} + |p_{j-1}, p_j|$ pour $i < j - 1$;
$b_{j-1,j} = \min_{1 \leq k < j-1} (b_{k,j-1} + |p_k, p_j|)$.

En effet, dans tout chemin bitonique se terminant par p_2, p_2 est le point d'abscisse maximale. Sa longueur est donc $|p_1, p_2|$.

On considère maintenant le chemin bitonique $P_{i,j}$ le plus court. Le point p_{j-1} se trouve quelque part sur ce chemin. S'il se trouve sur le sous-chemin allant vers la droite, alors il précède immédiatement p_j. Sinon, il se trouve sur le sous-chemin allant vers la gauche et il est le point le plus à droite donc $i = j - 1$.
Dans le premier cas, le sous-chemin de p_i à p_{j-1} est forcément le chemin bitonique $P_{i,j-1}$ le plus court, sans quoi on pourrait le remplacer par un chemin $\tilde{P}_{i,j-1}$ plus court et on obtiendrait un chemin plus court que $P_{i,j}$: absurde. La longueur de $P_{i,j}$ est donc donnée par $b_{i,j-1} + |p_{j-1}, p_j|$.
Dans le second cas, p_j a un prédécesseur immédiat p_k (où $k < j - 1$) sur le sous-chemin allant vers la droite. Par le même argument que précédemment, le sous-chemin de p_k à p_{j-1} est forcément $P_{k,j-1}$. La longueur de $P_{i,j}$ est donc donnée par $\min_{1 \leq k < j-1} (b_{k,j-1} + |p_k, p_j|)$.

Dans un chemin bitonique optimal $P_{n,n}$, l'un des points adjacents à p_n est nécessairement p_{n-1}. On a donc : $b_{n,n} = b_{n-1,n} + |p_{n-1}, p_n|$.

Pour reconstruire les points du chemin bitonique optimal $P_{n,n}$, on définit une matrice `b` de taille $(n \times n)$ pour stocker les valeurs $b_{i,i}$ et une matrice `r` telle que `r.(i-1).(j-1)` contient le prédécesseur immédiat de p_j dans le chemin bitonique $P_{i,j}$ le plus court.

L'algorithme comporte une première boucle qui itère sur tous les points puis la recherche du minimum a une complexité en $O(n)$. On a donc une complexité temporelle totale en $O(n^2)$.

Question 2

L'algorithme ne donne pas la longueur minimale dans le cas général (voyageur de commerce sans la contrainte des abscisses toujours croissantes puis toujours décroissantes). Il suffit de voir les exemples de l'énoncé, où le plus court chemin autorisé est de longueur supérieure à celle du chemin non autorisé.

⋆⋆⋆⋆⋆⋆⋆

1.3 Un mariage stable

n garçon et n filles se jugent mutuellement : chaque fille classe les garçons par ordre de préférence et chaque garçon classe les filles par ordre de préférence. Les classements sont représentés par deux matrices `f` et `g` de taille $n \times n$: `f.(i).(j)` est le classement que la fille i donne au garçon j et `g.(i).(j)` est le classement que donne le garçon i à la fille j. Par exemple, si `f.(i).(j)=1`, le garçon j est celui que préfère la fille i.

On appelle *mariage stable* une bijection M de l'ensemble des filles vers l'ensemble des garçons telle que pour tous i_1, i_2, j_1 et j_2 tels que $j_1 = M(i_1)$ et $j_2 = M(i_2)$, soit la fille i_1 préfère le garçon j_1 au garçon j_2, soit le garçon j_2 préfère la fille i_2 à la fille i_1.

Question 1

Montrer qu'un mariage stable est toujours possible et proposer un algorithme permettant de trouver un mariage stable.

CORRIGÉ

Question 1

On va construire petit à petit des couples de fiancés. On crée les deux tableaux suivants : `fiance` et `fiancee` tels que `fiance.(i)` est le numéro du fiancé de la fille i et `fiance.(j)` est le numéro de la fiancée du garçon j. Au début, tous les éléments de `fiance` et `fiancee` sont initialisés à -1. On crée également une fonction `preferee (i:int) (f:int array) : int` telle que `preferee i f` renvoie la fille préférée du garçon i parmi les choix possibles (l'algorithme supprimera des choix au fur et à mesure en plaçant la valeur $n+1$ dans le tableau f ou le tableau g). Il vient donc :

```
let preferee (i:int) (g:int array array) : int =
  let n = Array.length g in
  let ordre = ref (n+1) and p = ref (-1) in
  for j=0 to n-1 do
    if g.(i).(j) < !ordre then (
      ordre := g.(i).(j);
      p := j
    )
  done;
  !p
```

L'algorithme sera le suivant : tant qu'il existe un garçon g non fiancé [1], on cherche quelle est sa fille préférée f. Si cette fille n'est pas fiancée ou si elle ne préfère pas son actuel fiancé à g, on les fiance et on retire f des choix possible de l'ancien éventuel fiancé de f, sinon on retire f des choix de g. Donnons d'abord l'algorithme, on le discutera ensuite.

```
let mariages (f:int array array) (g:int array array) : int array =
  let n = Array.length f in
```

1. On peut bien entendu faire un algorithme équivalent en raisonnant sur les filles, pas de misogynie !

```
let fiance = Array.make n (-1) in
let fiancee = Array.make n (-1) in

let fini = ref false in
while not !fini do
  (* recherche d'un garçon non fiancé *)
  let garcon = ref 0 in
  while !garcon < n && fiancee.(!garcon) <> (-1) do
    incr garcon
  done;
  if !garcon = n then fini := true else (
    let fille = preferee !garcon g in
    let rival = fiance.(fille) in

    if rival <> (-1) then (
      if f.(fille).(!garcon) < f.(fille).(rival) then (
        (* la fille préfère le garçon à son actuel fiancé *)
        (* on rompt les fiançailles et on en célèbre de nouvelles *)
        fiancee.(!garcon) <- fille; fiancee.(fille) <- !garcon;
        fiancee.(rival) <- (-1); g.(rival).(fille) <- n+1
        (* on enlève la fille des choix possibles du rival *)
      ) else (
        (* elle préfère son actuel fiancé : on l'enlève des *)
        (* choix possibles du garçon *)
        g.(!garcon).(fille) <- n+1
      )
    ) else (    (* il n'y a pas de rival *)
      fiancee.(!garcon) <- fille; fiance.(fille) <- !garcon
    )
  )
done; (* à ce stade, tout le monde est fiancé,
         il ne reste plus qu'à célébrer les noces ! *)
fiance
```

Il nous faut maintenant prouver que ce programme termine et que le résultat est un mariage stable (ce qui prouvera l'existence d'un tel mariage).

Terminaison : Il suffit de constater qu'à chaque étape, soit le nombre de choix possibles pour un garçon (le garçon considéré ou son rival) diminue strictement, soit le nombre de garçons non fiancés décroît strictement. Le nombre $\sum$ card $\{$*choix possibles des garçons*$\}$ + card$\{$*garçons non fiancés*$\}$ est une suite d'entiers positifs strictement décroissants, elle est donc finie.

Correction : lors de l'exécution de l'algorithme, on n'enlève une fille des choix possibles d'un garçon que si cette fille préfère un autre garçon auquel elle est fiancée (ou devient fiancée). Par la suite, elle changera peut-être de fiancé, mais uniquement pour un garçon qu'elle préfère. On en déduit que si une fille est enlevée à un moment donné des choix possibles d'un garçon, elle sera finalement mariée à quelqu'un qu'elle préfère à ce garçon. Considérons deux couples (i_1, j_1) et (i_2, j_2) mariés par notre programme. Lorsque l'on a fiancé le garçon j_2 pour la dernière fois, soit la fille i_1 n'était plus dans

la liste des choix possibles de j_2 et dans ce cas, elle se retrouve mariée à un garçon qu'elle préfère à j_2, soit elle était dans la liste des choix possibles mais puisqu'alors on a fiancé j_2 à celle qu'il préférait parmi les choix restants (c'est-à-dire i_2), on en déduit que : soit la fille i_1 préfère le garçon j_1 au garçon j_2, soit le garçon j_2 préfère la fille i_2 à la fille i_1.

D'où le résultat.

⋆⋆⋆⋆⋆⋆⋆⋆

1.4 Coloriage d'un réseau de trains

On désire repeindre les gares d'un réseau de voies ferrées de sorte que deux gares reliées directement [2] ne soient jamais peintes de la même couleur. Dans un premier temps, on supposera que l'on ne dispose que de deux couleurs. Peindre des gares ainsi n'est pas toujours possible.

On suppose qu'il y a n gares. Le réseau est représenté par une matrice de booléens `r` tel que `r.(i).(j)` vaut `true` si la gare i est immédiatement reliée à la gare j et `false` sinon. On a toujours `r.(i).(j) = r.(j).(i)`.

Question 1

On dispose de seulement 2 couleurs, numérotées 1 et 2. Donner une fonction OCaml qui propose une couleur pour chaque gare de sorte que deux gares reliées directement ne soient jamais peintes de la même couleur si cela est possible, et qui indique si cela est impossible.

Question 2

Montrer que si chaque gare est reliée directement à au plus p gares, alors il existe un coloriage possible à l'aide d'au plus $p+1$ couleurs. Proposer un algorithme donnant un coloriage en $p+1$ couleurs dans un tel cas.

CORRIGÉ

Question 1

Le principe de l'algorithme est d'effectuer un parcours en profondeur du graphe des gares en faisant alterner la couleur de coloriage : on choisit une couleur arbitraire pour la première gare, puis on colore les gares voisines de l'autre couleur si elles n'ont pas encore été coloriées, ou on vérifie qu'elles sont de la bonne couleur sinon. Si pour la plupart des graphes, un parcours en largeur permettrait de trouver plus rapidement une éventuelle erreur, un parcours en profondeur est plus simple à mettre en place. On donne la fonction récursive suivante :

2. C'est à dire telles qu'on puisse aller directement en train de l'une à l'autre sans avoir à passer par une autre gare.

```
let coloriage (r:bool array array) : int array =
  let n = Array.length r in
  let couleur = Array.make n 0 in
  let vu = Array.make n false in

  let rec colorier (i:int) (col:int) : unit =
    vu.(i) <- true;
    couleur.(i) <- col;
    for j=0 to n-1 do
      if r.(i).(j) then (* i et j sont voisins *)
        if couleur.(i) = couleur.(j) then
          failwith "coloriage impossible"
        else if not vu.(j) then (
          vu.(j) <- true;
          couleur.(j) <- col;
          let new_col = if col = 1 then 2 else 1 in
          colorier j new_col
        )
    done in

  for i=0 to n-1 do
    if not vu.(i) then
      colorier i 1
  done;
  couleur
```

Question 2

La preuve se fait simplement par récurrence sur le nombre de gares. La propriété est élémentaire pour une gare. Si elle est vraie pour $n-1$ gares, alors considérons un réseau à n gares qui vérifie la propriété P : « chaque gare est connectée à au plus p gares ». Enlevons une gare à ce réseau. Le nouveau réseau vérifie trivialement la propriété P (on n'a fait qu'enlever des connexions). On peut donc le colorier à l'aide de $p+1$ couleurs. La gare qu'on avait enlevée est connectée à au plus p gares : on peut donc trouver pour cette gare au moins une couleur qui n'a pas été utilisée pour les gares adjacentes.

L'algorithme se déduit immédiatement de la récurrence.

⋆⋆⋆⋆⋆⋆⋆⋆

1.5 Graphes de dépendances

On considère l'ensemble $X = \{a, b, c, d\}$. Chaque élément de X représente une action possible d'un système et une relation réflexive et symétrique D incluse dans $X \times X$ décrit les actions ne pouvant pas être exécutées simultanément (D est dite relation de dépendance). Un graphe orienté étiqueté est un triplet $(S, A, etiq)$ où S est l'ensemble fini des sommets, A inclus dans $S \times S$ est l'ensemble fini des arêtes et $etiq$ une application

de S dans X est l'étiquetage du graphe. Un graphe fini orienté étiqueté $G = (S, A, etiq)$ est un graphe de dépendance s'il est sans cycle et si pour toute paire de sommets (s, s') :

$$(etiq(s), etiq(s')) \in D \text{ si et seulement si } [s = s' \text{ ou } (s, s') \in A \text{ ou } (s', s) \in A].$$

Ainsi,, il y a une arête entre s et s' distinct si et seulement si les actions correspondant aux étiquettes de s et s' ne peuvent être exécutées simultanément.

Question 1

Modéliser la notion de graphe orienté étiqueté décrite ci-dessus (pour un tel graphe $G = (S, A, etiq)$, on pourra supposer pour simplifier que les sommets de S sont des entiers et séparer les modélisations de (S, A) d'une part et $etiq$ d'autre part). Écrire un algorithme qui, étant donné un tel graphe, vérifie si c'est ou non un graphe de dépendance.

Question 2

Soit $G = (S, A, etiq)$ un graphe de dépendance. Une bijection C de l'ensemble $\{1, ..., |S|\}$ dans S peut être vue comme un choix successif de sommets distincts : si $C(i) = s$, on a choisi le sommet s au i-ième coup. La bijection C est fiable si, pour tous i, j : $(C(i), C(j)) \in A$ entraîne $i < j$ (i.e s'il y a une arête entre le sommet $C(i)$ et le sommet $C(j)$ alors il faut choisir $C(i)$ avant $C(j)$). Une linéarisation de $G = (S, A, etiq)$ est une suite $(u_1, u_2, ..., u_{|S|})$ d'éléments de X telle qu'il existe une bijection fiable C de $\{1, ..., |S|\}$ dans S vérifiant $u_i = etiq(C(i))$ pour tout i. Donner un algorithme qui, étant donné un graphe de dépendance G, calcule une linéarisation de G.

CORRIGÉ

Question 1

Comme le suggère l'énoncé, on va représenter l'ensemble S des sommets par l'ensemble des entiers de 1 à N pour une certaine constante N fixée. L'ensemble des arêtes sera décrit par un tableau de booléens. Enfin, la fonction d'étiquetage sera donnée par un tableau d'éléments de X.

En ce qui concerne la relation de dépendance D, il suffit de la représenter par une matrice carrée de booléens indexée par les éléments de X.

Pour vérifier qu'un graphe $(S, A, etiq)$ est bien un graphe de dépendance, la seule difficulté est de vérifier qu'il est bien acyclique. Pour cela, on effectue un parcours en profondeur du graphe en marquant en blanc les nœuds non visités, en gris les nœuds en cours de visite (i.e les nœuds se trouvant dans la pile d'appels récursifs) et en noir les nœuds visités. Si au cours du parcours on arrive sur un nœud colorié en gris alors le graphe contient un cycle, sinon il est bien acyclique.

On définit un type couleur :

```
type couleur = Blanc | Gris | Noir
```

```
let est_acyclique (g:bool array array) : bool =
  let n = Array.length g in
  let couleur = Array.make n Blanc in

  let rec parcours i =
    for j=0 to n-1 do
      if g.(i).(j) && couleur.(j) <> Noir then (
        if couleur.(j) = Gris then raise Exit;

        couleur.(j) <- Gris;
        parcours j;
        couleur.(j) <- Noir
      )
    done in

  try
    for i=0 to n-1 do
      if couleur.(i) <> Noir then
        parcours i
    done;
    true
  with Exit -> false
```

On a donc la fonction :

```
let est_graphe_dep (g:bool array array) (d:bool array array)
    (etiq:int array ): bool =
  let n = Array.length g in
  try
    for i=0 to n-1 do
      for j=0 to n-1 do
        if d.(etiq.(i)) = d.(etiq.(j)) then
          if not (i=j || g.(i).(j) || g.(j).(i)) then
            raise Exit;
        if (i=j || g.(i).(j) || g.(j).(i)) then
          if not (d.(etiq.(i)) = d.(etiq.(j))) then
            raise Exit
        done
      done;

      est_acyclique g
  with Exit -> false
```

Question 2

Par définition, une linéarisation commence par l'étiquette d'un sommet s sans prédécesseur, c'est à dire tel qu'il n'existe pas d'arête de la forme (s', s). Pour obtenir une bijection fiable, il faut donc trouver un sommet sans prédécesseur (il en existe puisque le graphe est acyclique), effacer toutes ses arêtes sortantes puis recommencer.

Notons que le graphe privé de s reste acyclique donc il existe toujours un sommet sans prédécesseur jusqu'à ce que tous les sommets aient été considérés.

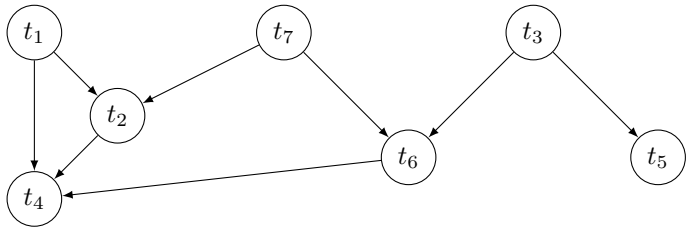
```
(* Retourne un noeud non vu et sans prédécesseur *)
let noeud_sans_pred (g:bool array array) (vus:bool array) : int =
  let n = Array.length g in
  let res = ref 0 in
  try
    for i=0 to n-1 do
      if not vus.(i) then
        let a_pred = ref false in
        for j=0 to n-1 do
          a_pred := !a_pred || g.(j).(i)
        done;
        if not !a_pred then (
          res := i;
          raise Exit
        )
    done; 0 (* impossible *)
  with Exit -> !res

let effacer_aretes_sortantes (g:bool array array)
    (x:int) : unit =
  let n = Array.length g in
  for i=0 to n-1 do
    g.(x).(i) <- false
  done

let lineariser (g:bool array array) (etiq:int array) : int array =
  let n = Array.length g in
  let suite = Array.make n 0 in
  let vus = Array.make n false in

  for i=0 to n-1 do
    let x = noeud_sans_pred g vus in
    effacer_aretes_sortantes g x;
    suite.(i) <- etiq.(x);
    vus.(x) <- true
  done;
  suite
```

⋆⋆⋆⋆⋆⋆⋆⋆

1.6 Ordonnancement

Un système de tâches $(T, <<)$ est un ensemble de n tâches $T = \{t_1, t_2, ..., t_n\}$. Ces tâches sont liées par des contraintes de précédence : on note $t_i << t_j$ pour dire que la tâche t_i doit être terminée avant que la tâche t_j ne puisse commence (on dit que t_j

est un successeur de t_i, et que t_i est un prédécesseur de t_j). Ci-dessous, on représente graphiquement un ensemble de 7 tâches, avec 8 contraintes de précédence ($t_1 << t_2$, $t_1 << t_4$, $t_2 << t_4$, $t_7 << t_2$, ...) matérialisées sous forme de flèches :



Chaque tâche a une durée entière positive $dur(t_i)$. On dispose d'une ou plusieurs machines identiques pour exécuter les tâches. Ordonnancer le système de tâches $(T, <<)$ revient à trouver un date de début d'exécution $deb(t_i)$ pour chaque tâche t_i. Le but est de minimiser le temps total d'exécution tout en respectant les contraintes de dépendance : si $t_i << t_j$, on doit avoir $deb(t_i) + dur(t_i) \leq deb(t_j)$.

Question 1

Proposer une structure de données pour un système de tâches et formaliser le problème. Écrire une fonction OCaml qui pour calcule pour chaque tâche le nombre et les numéros de ses successeurs et de ses prédécesseurs.

Question 2

Quelle condition doivent vérifier les contraintes de précédence pour qu'il soit possible d'ordonnancer le graphe ?

Question 3

On suppose la condition de la question 2 vérifiée. Pour chacun des cas suivants, proposer un algorithme pour ordonnancer le système de tâches :
(a) On dispose d'une seule machine ;
(b) On dispose de n machines ;
(c) On dispose de p machines, $1 \leq p \leq n$.

CORRIGÉ

Question 1

La structure de données la plus simple stocke les contraintes de précédence dans une matrice booléenne `c` telle que `c.(i).(j)` vaut `true` si $t_i << t_j$ et `false` sinon.

La matrice `c` donne les successeurs. Pour trouver les prédécesseurs, il suffit de parcourir la matrice `c` et stocker les résultats dans une matrice `pred` :

```
let pred_succ (c:bool array array) :
    (int array * int array * bool array array) =
  let n = Array.length c in
```

```
  let pred = Array.make_matrix n n false in
  let nb_pred = Array.make n 0 in
  let nb_succ = Array.make n 0 in

  for i=0 to n-1 do
    for j=0 to n-1 do
      if c.(i).(j) then (
        nb_succ.(i) <- nb_succ.(i) + 1;
        nb_pred.(j) <- nb_pred.(j) + 1;
        pred.(j).(i) <- true
      )
    done
  done;
  nb_pred, nb_succ, pred
```

Question 2

Il faut et il suffit que le graphe des tâches soit acyclique (sans cycle). C'est une condition nécessaire : on ne peut exécuter aucune tâche d'un cycle, et suffisante : on exécute d'abord les tâches sans prédécesseurs, puis on refait le graphe sans elles, et on recommence.

Question 3

(a) Avec une seule machine, le temps total d'exécution est la somme des n durées. On exécute successivement les tâches sans prédécesseurs parmi les tâches non traitées.

```
let une_machine (c:bool array array) (dur:int array) : int array =
  let n = Array.length c in
  let deb = Array.make n (-1) in
  let nb_pred, _, _ = pred_succ c in

  let taches_faites = ref 0 and top = ref 0 in
  while !taches_faites < n do
    for i=0 to n-1 do
      (* exécution des tâches sans prédécesseurs *)
      if nb_pred.(i) = 0 && deb.(i) = (-1) then (
        (* exécution de t_i *)
        deb.(i) <- !top;
        top := !top + dur.(i);
        incr taches_faites;

        for j=0 to n-1 do    (* mise à jour du graphe *)
          if c.(i).(j) then
            nb_pred.(j) <- nb_pred.(j) - 1
        done
      )
      done
  done; deb
```

(b) Avec n machines, il n'y a pas de problème de ressource. Par récurrence, on montre que l'optimal est d'exécuter chaque tâche au plus tôt, c'est-à-dire que la tâche i est exécutée lorsque tous ses prédécesseurs ont été exécutés, soit au temps $\max\{deb(j) + dur(j),\ j \text{ précécesseur de } i\}$ si elle a un prédécesseur j et au temps 0 sinon. La fonction ressemble un peu à la précédente : pour chaque tâche sans prédécesseur non déjà traité, on calcule le max, puis on met à jour le tableau des temps de départ.

(c) Divers algorithmes d'approximation sont possibles. Un algorithme de liste (« *list scheduling algorithm* ») est un algorithme glouton qui semble assez naturel. Quant à l'algorithme optimal, le problème problème est NP-difficile donc il n'existe pas d'algorithme polynomial (sauf si P=NP).

⋆⋆⋆⋆⋆⋆⋆⋆