ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2025

JEUDI 17 AVRIL 2025
08h00 - 12h00

FILIERE MPI - Epreuve n° 7

INFORMATIQUE C (XULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Couplages maximaux et parfaits

Le sujet comporte 18 pages, numérotées de 1 à 18.

Début de l'épreuve.

Complexité. Par la complexité d'un algorithme, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de celui-ci dans le pire cas. Lorsque la complexité dépend d'un ou plusieurs paramètres $k_0, \ldots, k_r$, on dit que l'algorithme a une complexité en $O(f(k_0, \ldots, k_r))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_0, \ldots, k_r$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0, \ldots, k_r$ la complexité est au plus $C \times f(k_0, \ldots, k_r)$.

Fonctions utiles en OCaml. Dans les questions de programmation en OCaml, il est interdit d'utiliser des fonctions de la bibliothèque qui ne sont pas incluses dans la liste suivante.

- `Option.get: 'a option -> 'a` renvoie `v` si l'argument de type `'a option` est égal à `Some v`, et lève une exception sinon. La complexité est en $O(1)$.
- `List.length: 'a list -> int` renvoie la taille de la liste donnée en argument. La complexité est en $O(n)$, où n est la longueur de la liste donnée en argument.
- `List.hd: 'a list -> 'a` renvoie le premier élément de la liste donnée en argument si celle-ci n'est pas vide, et lève une exception sinon. La complexité est en $O(1)$.
- `List.tl: 'a list -> 'a list` renvoie la liste donnée en argument privée de son premier élément si la liste donnée en argument n'est pas vide, et lève une exception sinon. La complexité est en $O(1)$.
- `List.rev: 'a list -> 'a list` renvoie la liste donnée en argument dans l'ordre inverse. La complexité est en $O(n)$, où n est la longueur de la liste donnée en argument.
- `List.map: ('a -> 'b) -> 'a list -> 'b list` renvoie la liste de type `'b list` obtenue en appliquant la fonction de type `'a -> 'b` donnée en premier argument à chaque élément de la liste de type `'a list` donnée en second argument. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.map f lst` est en $O(n)$, où n est la longueur de `lst`.
- `List.mem: 'a -> 'a list -> bool` renvoie `true` si la valeur donnée en premier argument est un élément de la liste donnée en second argument, et renvoie `false` sinon. La complexité est en $O(n)$, où n est la longueur de la liste donnée en second argument.
- `List.assoc: 'a -> ('a * 'b) list -> 'b` : cette fonction spécifique aux listes d'association est décrite en page 4.

Fonctions utiles en C. Dans les questions de programmation en C, on pourra supposer que les en-têtes `<stdbool.h>`, `<stdio.h>`, `<stdlib.h>`, et `<math.h>` ont été inclus. En particulier, on pourra utiliser la fonction `double log2(double x)` définie par l'entête `<math.h>`, qui calcule le logarithme en base 2.

Présentation du sujet. Ce sujet concerne les couplages dans les graphes non orientés. La partie I, à composer dans le langage OCaml, concerne la recherche d'un couplage maximal dans un graphe. On étudiera et implémentera un algorithme dû à Edmonds. Les parties II et III, à composer dans le langage C, concernent le problème de l'existence d'un couplage parfait dans un graphe. On y implémente des méthodes algébriques et probabilistes basées sur des calculs de déterminants. La partie II concerne l'implémentation d'un algorithme pour calculer le déterminant d'une matrice carrée dû à Mahajan et Vinay. La partie III exploite cet algorithme pour implémenter un algorithme probabiliste testant l'existence d'un couplage parfait dans un graphe non orienté.

Les parties I et II sont indépendantes. La partie III dépend de la partie II et de notions introduites en partie I.

Graphes non orientés. On travaillera dans les parties I et III sur des graphes non orientés et sans boucles. Un tel graphe G est défini par une paire d'ensembles finis (V, E), où V est l'ensemble des sommets du graphe et E l'ensemble de ses arêtes. Une arête est une paire non-ordonnée $\{u, v\}$ d'exactement deux sommets $u, v \in V$. Ainsi, les arêtes sont non orientées ($\{u, v\} = \{v, u\}$) et il n'y a pas de boucles ($\{v\} = \{v, v\}$ n'est jamais une arête de G). Une arête $\{u, v\}$ est dite **incidente** aux sommets u et v. On supposera toujours que V est non vide.

Étant donné un graphe $G = (V, E)$:

— Un **chemin** est une suite finie de sommets $v_0, v_1, \ldots, v_n$ avec $n \geqslant 0$ et telle que pour tout $i = 0, \ldots, n-1$, on a $\{v_i, v_{i+1}\} \in E$. La **longueur** du chemin $v_0, v_1, \ldots, v_n$ est n.

— Un chemin est **élémentaire** si ses sommets sont deux-à-deux distincts.

— Un **cycle** est un chemin $v_0, v_1, \ldots, v_n$ de longueur $\geqslant 3$, avec $v_0 = v_n$ et tel que le chemin $v_1, \ldots, v_n$ est élémentaire. Autrement dit, le seul sommet autorisé à apparaître plusieurs fois dans un cylce $v_0, v_1, \ldots, v_n$ est le sommet $v_0 = v_n$, qui apparaît exactement deux fois (une fois au début et une fois à la fin).

Partie I : Algorithme d'Edmonds

Cette partie concerne l'algorithme d'Edmonds pour la recherche de couplage maximal.

Question 1. *Écrire une fonction* `del: 'a list -> 'a list -> 'a list` *qui prend en entrée deux listes* `lst1` *et* `lst2` *et renvoie la liste des éléments de* `lst1` *n'apparaissant pas dans* `lst2`. *On attend une complexité en* $O(m_1 m_2)$ *où* m_1 *est la longueur de* `lst1` *et* m_2 *est la longueur de* `lst2`, *sans la justifier.*

On suppose définie une fonction `join: 'a list -> 'a list -> 'a list` telle que `join lst1 lst2` est une liste sans répétitions contenant l'ensemble de tous les éléments apparaissant dans `lst1` et de tous les éléments apparaissant dans `lst2`.

Nous utilisons des listes d'association pour représenter les graphes. Les listes d'association implémentent une structure de dictionnaire ; il s'agit de listes de paires `(a,b)`, dont le premier élément `a` représente la clef et le second élément `b` représente la valeur associée à cette clef.

Il existe des fonctions OCaml permettant de travailler avec les listes d'association. Dans la suite, nous utiliserons seulement la fonction `List.assoc: 'a -> ('a * 'b) list -> 'b` qui prend en arguments une clef `x` et une liste d'association `lst`, et renvoie la valeur `b` associée à cette clef, c'est-à-dire l'élément `b` du premier couple `(x,b)` appartenant à `lst`, s'il en existe. La fonction lève une exception si `x` n'est pas une clef de la liste `lst`. La complexité est en $O(n)$, où n est la longueur de `lst`.

Question 2. *Écrire une fonction* `keys: ('a * 'b) list -> 'a list` *qui prend en entrée une liste d'association* `lst` *et qui renvoie la liste de ses clefs.*
On attend une complexité en $O(m)$ *où* m *est la longueur de* `lst`, *sans la justifier.*

Nous représentons les graphes en utilisant le type OCaml suivant :

```
type graphe = (int * (int list)) list
```

Un graphe est représenté par une liste de paires `(a,lst)` où `a` est un sommet du graphe et `lst` est la liste d'adjacence de ce sommet. Dans la suite, on suppose toujours que :

— les sommets du graphe sont exactement les clefs de la liste d'association ;

— chaque clé de la liste d'association est unique.

Par exemple, le graphe

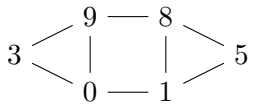
$$2 \text{ — } 0 \text{ — } 4$$

peut être représenté par la liste d'association `[(0,[2;4]);(4,[0]);(2,[0])]`.

Couplages

Un **couplage** C dans un graphe G est un ensemble (possiblement vide) d'arêtes de G sans intersections : un sommet de G ne peut appartenir à plus d'une arête dans C.

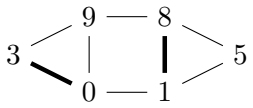
Considérons par exemple le graphe :

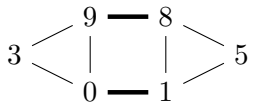


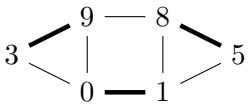
Les ensembles d'arêtes suivants sont des couplages dans ce graphe :

$$\{\{0,3\},\{1,8\}\} \qquad \{\{0,1\},\{8,9\}\} \qquad \{\{0,1\},\{5,8\},\{3,9\}\}$$

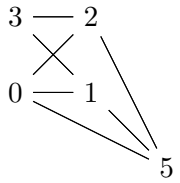
Les arêtes d'un couplage sont représentées par des traits plus épais. Par exemple, les trois couplages ci-dessus sont représentés comme suit :







Question 3. *On considère le graphe suivant :*



Indiquer lesquels des ensembles suivants sont des couplages dans le graphe ci-dessus :

(1) $\{\{1,2\},\{0,5\}\}$,

(2) $\{\{1,3\},\{0,5\}\}$,

(3) $\{\{0,1\},\{2,3\},\{0,5\}\}$.

Un couplage est représenté par une liste de paires de sommets :

```
type couplage = (int * int) list
```

Dans toute la suite, une arête $\{$`a`, `b`$\}$ sera représentée par la paire `(min a b, max a b)`.

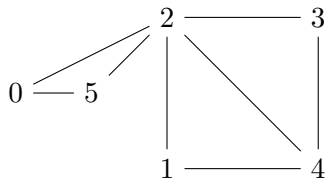
Étant donné un couplage C dans un graphe G, on dit qu'un sommet v de G est **couvert** par C s'il existe une arête dans le couplage C qui est incidente à v.

Question 4. *Écrire une fonction* `couverts: couplage -> int list` *qui prend en entrée un couplage* `cpl` *et qui renvoie la liste des sommets couverts par* `cpl`.
On attend une complexité en $O(m)$ où m est la longueur de la liste `cpl`*, sans la justifier.*

Couplages maximaux

Nous allons maintenant nous intéresser aux couplages maximaux, c'est-à-dire aux couplages ayant un nombre maximal d'arêtes. On remarque qu'un couplage dans un graphe à n sommets ne peut contenir plus de $\lfloor n/2 \rfloor$ arêtes.

Considérons le graphe suivant :



(1)

Le couplage $\{\{0,5\},\{2,3\},\{1,4\}\}$ est maximal. Mais le couplage $\{\{0,5\},\{2,4\}\}$ n'est pas maximal bien qu'il ne soit pas possible de lui adjoindre d'arête supplémentaire.

Question 5. *Donner un autre couplage maximal dans le graphe ci-dessus.*

Chemins d'augmentation

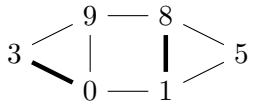
L'algorithme d'Edmonds est basé sur la recherche de chemins d'augmentation. Étant donné un couplage C dans un graphe G, un **chemin d'augmentation** pour C (dans G) est un chemin élémentaire $v_0, \ldots, v_n$ dans le graphe G tel que $n > 0$ et :

— v_0 et v_n ne sont pas couverts par C ;

— c'est un chemin **alternant** : pour tout $i = 0, \ldots, n-2$, on a $\{v_i, v_{i+1}\} \in C$ si et seulement si $\{v_{i+1}, v_{i+2}\} \notin C$.

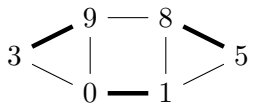
Un chemin d'augmentation P pour C permet de construire un nouveau couplage $C(P)$ constitué des arêtes de C qui ne sont pas dans P et des arêtes de P qui ne sont pas dans C.

Par exemple, dans le graphe



le chemin suivant est un chemin d'augmentation :

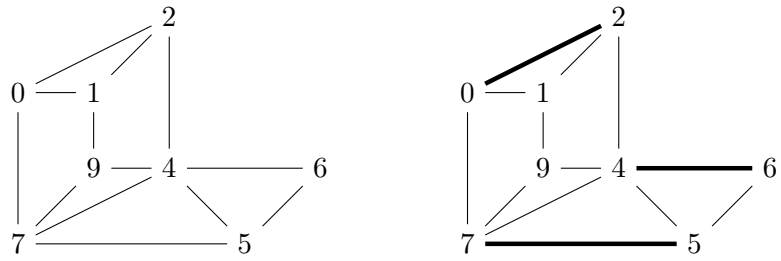
Le nouveau couplage obtenu à partir de ce chemin d'augmentation est le suivant :



Question 6. *Considérons le graphe en* (1) *ci-dessus. Donner un chemin d'augmentation P*

pour le couplage $C = \{\{0,5\},\{2,4\}\}$ *de manière à ce que le couplage* $C(P)$ *soit maximal.*

Question 7. *On considère le graphe et le couplage suivants :*



Donner un chemin d'augmentation et le couplage augmenté correspondant.

Question 8. *Soit* P *un chemin d'augmentation pour un couplage* C *dans un graphe* G*. Montrer que* $C(P)$ *est bien un couplage, et qu'il contient strictement plus d'arêtes que* C*.*

En OCaml, un chemin est représenté par une liste de sommets, c'est-à-dire par une valeur de type `int list`.

Question 9. *Écrire une fonction*

```
separer : int list -> ((int * int) list) * ((int * int) list)
```

qui prend en entrée un chemin d'augmentation `chm` *et renvoie deux listes* `lstin` *et* `lstout` *où* `lstin` *contient les arêtes composant le chemin* `chm` *qui appartiennent au couplage, et où* `lstout` *contient les arêtes composant le chemin* `chm` *qui n'appartiennent pas au couplage.*
On attend une complexité en $O(m)$*, où* m *est la longueur de la liste* `chm`*, sans la justifier.*

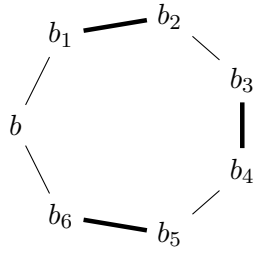
Question 10. *Écrire une fonction* `augmente: couplage -> int list -> couplage` *qui prend en entrée un couplage* `cpl` *et un chemin d'augmentation* `chm`*, et qui renvoie le couplage* `cpl(chm)`*.*

L'algorithme d'Edmonds, que nous allons voir ci-dessous, repose essentiellement sur le lemme de Berge. Ce résultat, que nous admettrons, énonce qu'un couplage C dans un graphe G est maximal si et seulement s'il n'existe pas de chemin d'augmentation pour C dans G.

Bourgeons

L'algorithme d'Edmonds va de plus effectuer des opérations de **contraction de bourgeons**. Soit C un couplage dans un graphe G. Un bourgeon est un cycle dans G de longueur impaire $2m+1$ dont exactement m arêtes sont dans C. La **base** b d'un bourgeon B est le seul sommet

de B dont les deux arêtes adjacentes dans B ne sont pas dans C :



La contraction d'un bourgeon consiste à identifier tous les sommets de celui-ci. On représente un bourgeon B par une valeur de type `int list` qui contient exactement les sommets de B, dans un ordre quelconque, mais sans répétition. Par exemple, le bourgeon illustré ci-dessus peut être représenté par la liste `[`b_1`;`b_6`;`b_2`;`b_5`;`b_3`;`b`;`b_4`]`.

Étant donné un graphe $G = (V, E)$ et une liste de sommets $L = \ell_1, \ldots, \ell_k$, le **graphe contracté** G/L est défini comme suit :

— L'ensemble des sommets du graphe G/L est $(V \setminus L) \cup \{w\}$ où w, **le nouveau sommet de** G/L, est un entier qui n'est pas un sommet de G.

— Les arêtes de G/L sont les arêtes de G entre sommets n'étant pas dans L, ainsi que les arêtes de la forme $\{u, w\}$ où u n'appartient pas à L mais est adjacent dans G à un sommet de L. L'ensemble des arêtes de G/L est donc défini comme

$$\{\{u, v\} \in E \mid u \notin L \text{ et } v \notin L\} \cup \{\{u, w\} \mid u \notin L \text{ et } u \text{ adjacent dans } G \text{ à un sommet de } L\}$$

On suppose donnée une fonction `renomme: int list -> int list -> int -> int list` telle que `renomme lst rnm w` renvoie une liste obtenue en remplaçant dans `lst` tous les éléments de `rnm` par `w`, et en supprimant tous les doublons.

Question 11. *Écrire une fonction* `contracteG: graphe -> int list -> int -> graphe` *qui prend en entrée un graphe* `grph`*, une liste de sommets* `lst`*, et le nom* `w` *du nouveau sommet, et qui renvoie le graphe contracté* `grph/lst`*.*

Soit C un couplage dans un graphe G, et soit B un bourgeon. On définit un couplage C/B dans le graphe contracté G/B. En notant w le nouveau sommet de G/B, l'ensemble C/B a pour éléments :

— les arêtes $\{a, b\} \in C$ telles que $a \notin B$ et $b \notin B$;

— les arêtes $\{a, w\}$ telles que $a \notin B$ et telles qu'il existe un $b \in B$ avec $\{a, b\} \in C$.

On remarquera que C/B ne dépend que de C, B, et du nom w du nouveau sommet.

Question 12. *Montrer que C/B est un couplage dans G/B.*

Question 13. *Écrire une fonction*

`contracteC : couplage -> int list -> int -> couplage`

qui prend en entrée un couplage `cpl`*, un bourgeon* `brg`*, et le nom* `w` *du nouveau sommet, et qui renvoie le couplage* `cpl/brg`*.*

On admettra dans la suite qu'il existe un chemin d'augmentation pour le couplage C/B dans G/B si et seulement s'il existe un chemin d'augmentation pour le couplage C dans G.

Recherche de chemins d'augmentation

La recherche de chemins d'augmentation va utiliser des forêts, c'est-à-dire des listes d'arbres. Les types ci-dessous représentent des arbres et des forêts dont les nœuds sont étiquetés par des entiers. Ces entiers seront par la suite des sommets d'un graphe.

```
type arbre = N of int * foret
and foret = arbre list
```

Question 14. *Écrire une fonction* `find: foret -> int -> (int list) option` *telle que* `find foret v` *renvoie* `None` *si l'entier* `v` *n'étiquette aucun des nœuds de* `foret`*, et telle que* `find foret v` *renvoie* `Some c` *sinon, où* `c` *est la liste des étiquettes le long d'un chemin allant d'une racine à un nœud d'étiquette* `v` *dans* `foret`*.*
On attend une complexité en $O(n)$, où n est le nombre de nœuds de `foret`*, sans la justifier.*

Question 15. *Écrire une fonction* `extend: foret -> int -> int -> foret`*, qui prend en arguments une forêt* `foret` *et deux entiers* `u` *et* `v`*, et qui renvoie une copie de* `foret` *dans laquelle pour chaque nœud étiqueté par* `u`*, on a créé un nouvel enfant étiqueté par* `v`*.*
On attend une complexité en $O(n)$, où n est le nombre de nœuds de `foret`*, sans la justifier.*

La **profondeur** d'un nœud dans un arbre est définie inductivement comme suit : la racine est à profondeur 0, et pour $h \geqslant 0$, les nœuds à profondeur $h+1$ sont les enfants des nœuds à profondeur h.

L'algorithme de recherche de chemins d'augmentation prend en entrée un graphe G et un couplage C dans G. Il manipule un ensemble de sommets T et peut être décrit comme suit.

(1) On construit une forêt F dont les nœuds consistent exactement en une racine u pour chaque sommet u de G non couvert par C.

(2) On fixe $T = \emptyset$.

(3) Tant qu'il existe un sommet $u \notin T$ tel que u apparaît à profondeur paire dans F :

On ajoute u à l'ensemble T.

Pour chaque voisin v de u dans G tel que $v \notin T$:

(a) Si v n'est pas dans la forêt F, il est couvert par une arête $\{v, z\}$ de C, et on ajoute les arêtes $\{u, v\}$ et $\{v, z\}$ à F.

(b) Si v est dans la forêt F :

(i) Si u, v apparaissent à profondeurs paires dans des arbres distincts de racines respectives r_u, r_v, alors on renvoie la suite des sommets de G vus dans F en allant de r_u à u (r_u et u inclus), puis en allant de v à r_v (v et r_v inclus).

(ii) Si u et v apparaissent dans le même arbre à profondeurs paires, alors on construit le bourgeon B correspondant, et on cherche un chemin d'augmentation pour le couplage C/B dans le graphe G/B. Si on en trouve un, alors on renvoie un chemin d'augmentation pour C dans G.

Question 16. *Montrer que la suite de sommets renvoyée par l'algorithme dans le cas (3)(b)(i) est un chemin d'augmentation pour C dans G.*

On admet dans toute la suite que l'algorithme ci-dessus renvoie un chemin d'augmentation pour C si et seulement s'il en existe un.

Nous allons étudier une implémentation OCaml de cet algorithme. On suppose données les fonctions suivantes.

— La fonction `frais: int list -> int` renvoie un entier n'appartenant pas à la liste donnée en argument.

— La fonction `prochain: foret -> int list -> int option` prend en arguments une forêt `f` et une liste de sommets `lst`. Un appel `prochain f lst` renvoie `Some d` où `d` est un entier apparaissant dans `f` à profondeur paire et n'appartenant pas à `lst` si un tel entier `d` existe. Sinon, `prochain f lst` revoie `None`.

— La fonction `apparie: couplage -> int -> int` prend en arguments un couplage `cpl` et un sommet `u` couvert par le couplage, et renvoie le sommet apparié, c'est-à-dire l'unique `v` tel que `(min u v, max u v)` appartient à `cpl`.

— La fonction `gonfle: graphe -> int list -> int list -> int -> int list` prend en arguments un graphe `grph`, un bourgeon `brg`, un chemin d'augmentation dans le graphe contracté `grph/brg`, et le nom du nouveau sommet `nv`, et renvoie le chemin d'augmentation correspondant dans `grph`.

La figure 1 page 11 représente une implémentation *incomplète* d'une fonction

```
recherche : graphe -> couplage -> (int list) option
```

Les questions ci-dessous demandent de donner le code de la ligne 15, des lignes 17–18 et des lignes 24–26, ainsi que d'implémenter la fonction `extrait` (ligne 20).

La fonction `recherche` prend en arguments un graphe et un couplage. Elle doit renvoyer `Some p` s'il existe un chemin d'augmentation `p`, et renvoyer `None` sinon. Certaines fonctions appelées dans le code de `recherche` peuvent lever des exceptions, mais on admettra que ces exceptions ne sont jamais levées si les données d'entrée sont bien formées.

Question 17. *Donner le code devant apparaître à la ligne 15 de la figure 1.*

Question 18. *Donner le code devant apparaître dans le bloc aux lignes 17–18 de la figure 1.*

Question 19. *Écrire une fonction* `extrait: int list -> int list -> int list` *telle que l'appel* `extrait cu cv` *ligne 20 renvoie le bourgeon associé aux chemins* `cu` *et* `cv`*.*

Question 20. *Donner le code devant apparaître dans le bloc aux lignes 24–26 de la figure 1.*

Question 21. *Écrire une fonction* `edmonds: graphe -> couplage` *qui prend en entrée un graphe et qui renvoie un couplage maximal.*

```
let rec recherche graphe couplage =
    let nc = del (keys graphe) (couverts couplage) in
    let foret = ref (List.map (fun x -> N(x,[])) nc) in
    let rec aux_voisins u liste_voisins =
        let cu = Option.get (find !foret u) in
        match liste_voisins with
        | [] -> None
        | v :: tl ->
                  match find !foret v with
                  | None -> let z = apparie couplage v in
                            foret := extend (extend !foret u v) v z;
                            aux_voisins u tl
                  | Some cv when (List.length cv) mod 2 = 0
                  (* nombre de sommets pair = profondeur impaire *)
                         -> ...
                  | Some cv when (List.hd cu) <> (List.hd cv)
                         -> ...
                            ...
                  | Some cv
                         -> let bourgeon = extrait cu cv in
                            let nv = frais (keys graphe) in
                            let ng = contracteG graphe bourgeon nv in
                            let nc = contracteC couplage bourgeon nv in
                            ...
                            ...
                            ...
    in
    let rec aux_sommets traites =
        match prochain !foret traites with
        | None -> None
        | Some u -> let liste_voisins = del (List.assoc u graphe) traites in
                    match aux_voisins u liste_voisins with
                    | None -> aux_sommets (u::traites)
                    | Some ch -> Some ch
    in
    aux_sommets []
```

FIGURE 1 – Fonction `recherche`.

Partie II : Calculs de déterminants

Cette partie concerne l'algorithme de Mahajan-Vinay pour le calcul de déterminants de matrices carrées. Elle est à composer dans le langage C.

Préliminaires : matrices linéarisées

On travaille avec des matrices carrées de taille arbitraire indexées à partir de 0. On utilise une représentation linéarisée de ces matrices. Une matrice A de dimension $\mathtt{n} \times \mathtt{n}$ (c'est-à-dire avec `n` lignes et `n` colonnes) est représentée par un tableau `A` avec $\mathtt{n}^2$ éléments de manière à ce que le coefficient $A_{\mathtt{i},\mathtt{j}}$ de la matrice (indice de ligne `i` et indice de colonne `j`) corresponde à l'élément `A[i*n+j]` du tableau.

On suppose dans la suite que les opérations arithmétiques ont une complexité en $O(1)$.

Question 22. *Écrire des fonctions*

```
int read_sqmatrix (int n, int *A, int i, int j)
void write_sqmatrix (int n, int *A, int i, int j, int val)
```

telles que

- `read_sqmatrix(n,A,i,j)` *renvoie la valeur du coefficient* $A_{\mathtt{i},\mathtt{j}}$ *où* A *est la matrice* $\mathtt{n} \times \mathtt{n}$ *représentée par* `A` *;*
- `write_sqmatrix(n,A,i,j,val)` *modifie la valeur du coefficient* $A_{\mathtt{i},\mathtt{j}}$ *pour qu'il soit égal à* `val`*.*

On supposera que `A` *est un tableau avec* $\mathtt{n}^2$ *éléments et que* $\mathtt{i}, \mathtt{j} \in \{0, \ldots, \mathtt{n}-1\}$.
On attend une complexité en $O(1)$ *pour ces fonctions, sans la justifier.*

Algorithme de Mahajan-Vinay

L'algorithme de Mahajan-Vinay est basé sur une formulation du déterminant d'une matrice A en termes de **suites de marches fermées** dans un graphe associé à A.

Soit A une matrice carrée de dimension $n \times n$. On associe à A son **graphe d'adjacence** $G(A)$. Le graphe $G(A)$ est **orienté et pondéré**. Il est défini comme suit :

- les sommets de $G(A)$ sont les entiers de 0 à $n-1$,
- les arcs sont les paires ordonnées $e = (i, j)$ telles que $A_{i,j} \neq 0$,
- le poids $\omega(e)$ d'un arc $e = (i, j)$ est égal à $A_{i,j}$.

Les sommets de $G(A)$ sont donc des entiers. L'ordre usuel sur ces entiers est utilisé dans la suite de cette partie.

Question 23. *Dessiner le graphe d'adjacence de la matrice suivante :*

$$\begin{pmatrix} 2 & 3 \\ -2 & 1 \end{pmatrix}$$

Une **marche fermée** dans le graphe $G(A)$ est une suite de sommets $C = v_0 \dots v_m$ avec $m > 0$ et qui satisfait les conditions suivantes :

— pour tout $i = 0, \dots, m-1$, il existe un arc (v_i, v_{i+1}) dans $G(A)$;

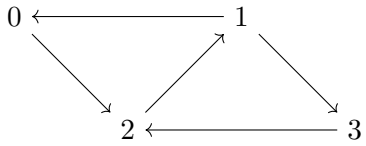
— le sommet $v_0 = v_m$ est le plus petit entier de la suite, appelé la **tête** de C, notée $t(C)$;

— le sommet v_0 n'apparaît qu'au début et à la fin de la suite : pour tout $i = 1, \dots, m-1$, on a $v_i \neq v_0$.

La **longueur** $|C|$ de la marche fermée C est le nombre d'arcs qui la composent. La longueur de $C = v_0 \dots v_m$ est donc égale à m. Le **poids** $\omega(C)$ d'une marche fermée C est le produit des poids des arcs (avec multiplicités) qui la composent :

$$\omega(C) = \prod_{i=0}^{m-1} \omega(v_i, v_{i+1}).$$

Exemple. Considérons le graphe suivant, dont les arcs ont tous poids 1 :



Dans ce graphe, les suites de sommets suivantes sont des marches fermées :

$$0 \to 2 \to 1 \to 0 \qquad \text{et} \qquad 0 \to 2 \to 1 \to 3 \to 2 \to 1 \to 0.$$

On remarque qu'il est possible de passer plusieurs fois par le même sommet. Cependant la suite

$$0 \to 2 \to 1 \to 0 \to 2 \to 1 \to 0$$

n'est pas une marche fermée car la tête ne doit apparaître qu'au début et à la fin. La suite

$$2 \to 1 \to 0 \to 2$$

n'est pas non plus une marche fermée car la tête n'est pas le plus petit sommet.

Question 24. *Donner une marche fermée de longueur* 3 *et de tête* 1 *dans le graphe ci-dessus.*

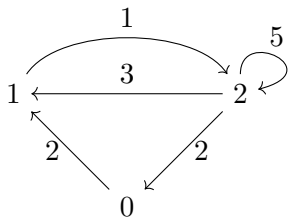
Une **suite de marches fermées** est une suite S de marches fermées $C_1, \dots, C_k$ telle que :

— les têtes sont croissantes : $t(C_1) < t(C_2) < \dots < t(C_k)$;

— le nombre total d'arcs (en comptant les multiplicités) est égal au nombre de sommets n de $G(A)$: $\sum_{i=1}^{k} |C_i| = n$.

Le **poids** d'une suite de marches fermées est le produit des poids des marches fermées qui la composent : $\omega(S) = \prod_{i=1}^{k} \omega(C_i)$. La **tête** de S, notée $t(S)$, est la tête de la première marche fermée de S : $t(S) = t(C_1)$.

Question 25. *On considère le graphe suivant :*



(1) Donner les marches fermées de longueur au plus 3 *et leurs poids.*

(2) Donner les suites de marches fermées et leurs poids.

On définit aussi le **signe** d'une suite de marches fermées. Si $S = C_1 \dots C_k$, alors on pose :

$$\mathrm{sgn}(S) = (-1)^{n+k}.$$

On dit que S est une suite de marches fermées **positive** lorsque $\mathrm{sgn}(S) = 1$, et que S est une suite de marches fermées **négative** lorsque $\mathrm{sgn}(S) = -1$.

Le résultat fondamental de Mahajan et Vinay est que le déterminant d'une matrice carrée A peut être exprimé comme suit :

$$\det(A) = \sum_{\substack{S \text{ suite de marches fermées} \\ \text{dans } G(A)}} \mathrm{sgn}(S)\omega(S).$$

Soit A une matrice avec n lignes et n colonnes. Pour implémenter le calcul du déterminant de A selon la formule ci-dessus, Mahajan et Vinay introduisent un nouveau graphe orienté et pondéré H_A, que nous allons maintenant décrire.

Le graphe H_A possède trois sommets distingués : s, t_0, et t_1. La construction de H_A permet d'assurer que les chemins de s à t_0 (respectivement t_1) dans H_A correspondent aux suites de marches fermées positives (respectivement négatives) dans $G(A)$. Les autres sommets sont des quadruplets $\langle p, h, u, i\rangle$ avec $p \in \{0, 1\}$, $0 \leqslant h \leqslant u \leqslant n-1$, et $i \in \{0, \dots, n-1\}$. Ceux-ci représentent des étapes de tentatives de construction de suites de marches fermées. Le sommet $\langle p, h, u, i\rangle$ représente le cas où des marches fermées $C_1, \dots, C_k$ ont été construites, et où on cherche à construire une marche fermée commençant par $C_{k+1} = v_0, v_1, \dots, v_m$, où

— le signe de la suite de marches fermées $C_1, \dots, C_k$ est $(-1)^p$;

— h est la tête de la marche fermée en cours de construction, c'est-à-dire $h = v_0$;

— u est le sommet courant, c'est-à-dire $u = v_m$;

— i est le nombre d'arcs parcourus jusque là, c'est-à-dire $i = m + \sum_{j=1}^{k} |C_j|$.

Le graphe H_A a les arcs suivants.

(1) Un arc de poids 1 de s vers chaque sommet de la forme $\langle n \bmod 2, h, h, 0\rangle$. Ces arcs permettent d'initialiser les suites de marches fermées.

(2) Pour chaque coefficient $A_{u,v} \neq 0$, chaque $h < v$ et chaque $i < n-1$, un arc de poids $A_{u,v}$ de $\langle p, h, u, i\rangle$ vers $\langle p, h, v, i+1\rangle$. Ces arcs correspondent à l'extension de la marche fermée en cours de construction (la marche C_{k+1} dans l'explication ci-dessus).

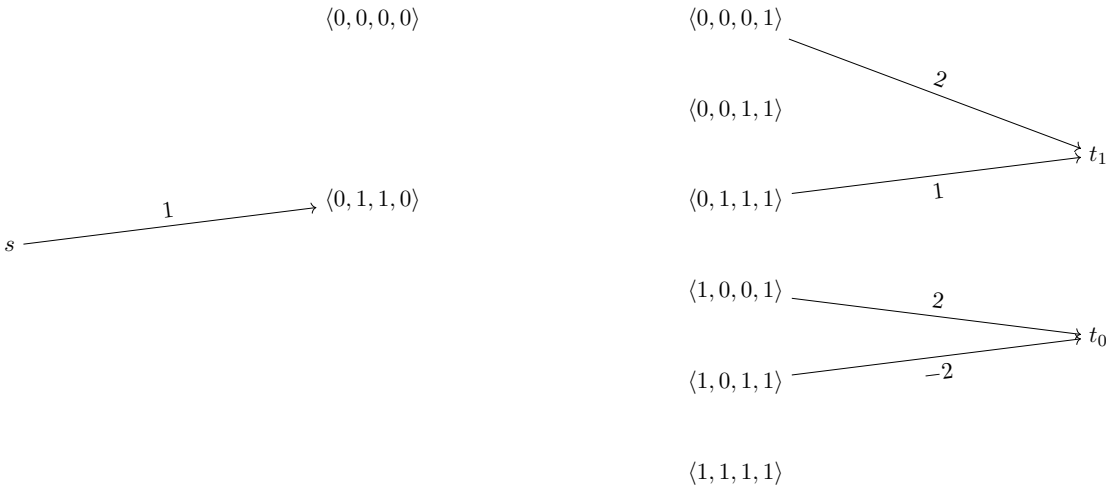
(3) Pour chaque coefficient $A_{u,h} \neq 0$, chaque $h' \in \{h+1, \dots, n-1\}$, et chaque $i < n-1$, un arc de poids $A_{u,h}$ de $\langle p, h, u, i\rangle$ vers $\langle 1-p, h', h', i+1\rangle$. Ces arcs correspondent à la fermeture de la marche fermée en cours de construction, et à l'initialisation d'une nouvelle marche fermée de tête h'.

(4) Pour chaque coefficient $A_{u,h} \neq 0$ et chaque $p \in \{0, 1\}$, un arc de poids $A_{u,h}$ de $\langle p, h, u, n-1\rangle$ vers t_{1-p}. Ces arcs correspondent à la fermeture de la dernière marche fermée de la suite.

Question 26. *On considère la matrice A suivante :*

$$\begin{pmatrix} 2 & 3 \\ -2 & 1 \end{pmatrix}$$

La figure ci-dessous représente certains sommets du graphe H_A correspondant, ainsi que certains arcs. Indiquer les arcs manquants entre les sommets représentés, ainsi que leurs poids.



Question 27. *Soit A une matrice carrée avec n lignes et n colonnes. Montrer que si $C = h, u_1, \ldots, u_m, h$ est une marche fermée dans $G(A)$, alors*

$$\langle p, h, h, i\rangle \to \langle p, h, u_1, i+1\rangle \to \cdots \to \langle p, h, u_m, i+m\rangle$$

est un chemin dans H_A pour tout $p \in \{0, 1\}$ et tout $i \in \{0, \ldots, n-1-m\}$.
En déduire qu'à chaque suite de marches fermées positive dans $G(A)$ correspond un chemin de s à t_0 dans H_A.

On peut démontrer que les chemins de s à t_0 dans H_A sont en fait en bijection avec les suites de marches fermées positives dans $G(A)$. Dans toute la suite, on admet ce résultat ainsi que le résultat analogue pour les chemins de s à t_1 dans H_A et les suites de marches fermées négatives dans $G(A)$.

Nous ne construirons pas le graphe H_A mais travaillerons de manière implicite avec celui-ci. Afin de calculer le déterminant de la matrice A en utilisant la formule de Mahajan-Vinay, nous allons calculer la somme des poids des chemins de s vers t_0 et t_1 dans H_A. Soit F_A la fonction qui à chaque sommet x de H_A associe la somme des poids des chemins de s à x dans H_A.

Nous allons commencer par calculer la valeur de $F_A(x)$ pour l'ensemble des sommets de la forme $x = \langle p, h, v, i\rangle$. On remarque que les sommets du graphe H_A peuvent être organisés en couches selon la valeur du dernier entier. Formellement, la **couche** i de H_A est l'ensemble des sommets de H_A de la forme $\langle p, h, v, i\rangle$.

Question 28. *Exprimer la valeur de $F_A(\langle p, h, v, i\rangle)$ en fonction des valeurs de la couche $i-1$, c'est-à-dire des $F_A(\langle p', h', v', i-1\rangle)$.*

Implémentation

On veut représenter $F_A(x)$ pour chaque x de la forme $\langle p, h, v, i\rangle$. Pour cela, on suppose donnés les types et fonctions C suivants.

— Un type `Fourtable`.

— Une fonction `Fourtable *create (int n)`. Un appel `create(n)` alloue dans le tas une valeur de type `Fourtable` et renvoie un pointeur `t` vers celle-ci. La valeur `*t` ainsi créée a la taille nécessaire pour enregistrer les valeurs de F_A lorsque A est une matrice $\mathtt{n} \times \mathtt{n}$. Les valeurs contenues dans `*t` sont toutes initialisées à 0. On supposera que `free(t)` a une complexité en $O(1)$.

— Une fonction `void write(Fourtable *t, int val, int p, int h, int v, int i)` qui enregistre dans `*t` l'entier `val` comme valeur correspondant au sommet $\langle \mathtt{p,h,v,i} \rangle$.

— Une fonction `int read(Fourtable *t, int p, int h, int v, int i)` qui renvoie la valeur correspondant au sommet $\langle \mathtt{p,h,v,i} \rangle$ dans `*t`.

On suppose de plus que les fonctions `read` et `write` ont une complexité en $O(1)$, et que la fonction `create` a une complexité en $O(\mathtt{n}^3)$, où `n` est la valeur donnée en argument.

Notons que $F_A(\langle p, h, u, 0 \rangle)$ ne dépend que de p, h, u et de la dimension de la matrice A.

Question 29. *Écrire une fonction* `Fourtable *initialise (int n)` *telle que* `initialise(n)` *renvoie un pointeur vers un* `Fourtable` *contenant les valeurs de F_A pour tous les sommets de la couche 0 de H_A, où A est une matrice $\mathtt{n} \times \mathtt{n}$.*
On attend une complexité en $O(\mathtt{n}^3)$ pour cette fonction, sans la justifier.

Question 30. *Écrire une fonction* `Fourtable *remplit (int n, int *A)` *qui prend en entrée la représentation linéarisée* `A` *d'une matrice A de dimension $\mathtt{n} \times \mathtt{n}$, et renvoie un pointeur vers un* `Fourtable` *contenant les valeurs de F_A pour tous les sommets de la forme $\langle p, h, v, i \rangle$ avec $p \in \{0, 1\}$, $0 \leqslant h \leqslant v \leqslant \mathtt{n} - 1$, et $0 \leqslant i \leqslant \mathtt{n} - 1$.*
On attend une complexité $O(\mathtt{n}^4)$ pour cette fonction, ***qu'il faudra justifier****.*

Question 31. *Écrire une fonction* `int determinant (int n, int *A)` *qui prend en entrée la représentation linéarisée* `A` *d'une matrice A de dimension $\mathtt{n} \times \mathtt{n}$ et renvoie son déterminant.*
On veillera à libérer correctement la mémoire allouée dans le tas.
On attend une complexité en $O(\mathtt{n}^4)$ pour cette fonction, ***qu'il faudra justifier****.*

Partie III : Méthode algébrique et probabiliste pour les couplages parfaits

Cette partie concerne un algorithme qui teste si un graphe possède un couplage parfait, c'est-à-dire un couplage qui couvre tous ses sommets. Elle est à composer dans le langage C. On pourra réutiliser le matériel de la partie précédente.

On ne considère que des graphes **non orientés** (voir page 3). Un tel graphe G d'ensemble de sommets $\{0, \ldots, n-1\}$ est représenté par sa **matrice d'adjacence** A, de dimension $n \times n$, et dont les coefficients sont donnés par

$$A_{i,j} = \begin{cases} 1 & \text{si } \{i,j\} \text{ est une arête de } G, \\ 0 & \text{si } \{i,j\} \text{ n'est pas une arête de } G. \end{cases}$$

Nous allons utiliser des méthodes algébriques, basées sur le déterminant de la matrice de Tutte. La matrice de Tutte d'un graphe est une matrice symbolique, et son déterminant est un polynôme réel à plusieurs variables. Un polynôme réel à m variables $X_1, \ldots, X_m$ est une somme finie de monômes, c'est-à-dire de termes de la forme $aX_1^{d_1} \cdots X_m^{d_m}$ avec $a \in \mathbb{R}$ et $d_1, \ldots, d_m \in \mathbb{N}$. Lorsque $d_i = 0$, la variable X_i est omise dans $aX_1^{d_1} \cdots X_m^{d_m}$. Par exemple, le monôme $aX_1^0 \cdots X_m^0$ est noté a.

Considérons un graphe non orienté G dont l'ensemble de sommets est $\{0, \ldots, n-1\}$. La **matrice de Tutte** $T(G)$ de G est une matrice $n \times n$ dont les éléments sont des monômes sur l'ensemble de variables $\{X_{i,j} \mid 0 \leqslant i < j \leqslant n-1\}$. Les coefficients de $T(G)$ sont donnés par

$$T(G)_{i,j} = \begin{cases} 0 & \text{si } i = j \text{ ou si } \{i,j\} \text{ n'est pas une arête de } G, \\ X_{i,j} & \text{si } i < j \text{ et } \{i,j\} \text{ est une arête de } G, \\ -X_{j,i} & \text{si } i > j \text{ et } \{i,j\} \text{ est une arête de } G. \end{cases}$$

Un théorème de Tutte énonce que G possède un couplage parfait si, et seulement si, le déterminant de $T(G)$ est différent du polynôme nul (c'est-à-dire du polynôme dont toutes les valeurs sont 0).

Une partie du problème consiste donc à décider si un polynôme est nul ou non. Nous allons implémenter un algorithme **probabiliste** qui teste si un polynôme s'annule en un certain nombre de points choisis aléatoirement.

Dans la suite, on suppose donnée une fonction `int rand_range(int n)` qui prend en entrée un entier `n` et renvoie un entier tiré aléatoirement entre `0` et `n` (inclus) selon la distribution uniforme. On suppose que cette fonction s'exécute en temps constant.

On suppose que l'instruction suivante, qui crée un tableau d'entiers `t` de longueur `m`, s'exécute en temps O(`m`) :

```
int t [m];
```

Question 32. *Écrire une fonction* `int tirage_tutte(int n, int *A)` *qui prend en entrée la représentation linéarisée* `A` *d'une matrice d'adjacence A de dimension* `n` $\times$ `n`*, et renvoie le*

déterminant d'une instantiation de la matrice de Tutte du graphe représenté par A obtenue en tirant aléatoirement les valeurs des variables $X_{i,j}$ dans l'ensemble $\{0, 1, 2, \ldots, \texttt{n*n} - 1, \texttt{n*n}\}$. On attend une complexité en $O(\texttt{n}^4)$ pour cette fonction, sans la justifier.

On va utiliser la fonction définie ci-dessus pour décider si un polynôme réel à plusieurs variables est nul ou non. On se propose d'appliquer le lemme de Schwartz-Zippel. Ce résultat permet, pour un polynôme non nul P, de borner le nombre de zéros de P (dans un ensemble fini donné) en fonction du degré total de P (et de la taille de cet ensemble). Le degré total d'un monôme $aX_1^{d_1} \ldots X_n^{d_n}$ est la somme $d_1 + \cdots + d_n$. Le **degré total** d'un polynôme est le degré total maximal des monômes qui le composent.

Soit P un polynôme réel non nul à m variables et de degré total $d \geqslant 0$, et soit S un sous-ensemble fini non vide de $\mathbb{R}$. On note $\Pr[P(r_1, r_2, \ldots, r_m) = 0]$ la probabilité pour que P s'annule en $r_1, r_2, \ldots, r_m$, où $r_1, r_2, \ldots, r_m$ sont des réels tirés aléatoirement dans S, de manière uniforme et indépendante. Le lemme de Schwartz-Zippel énonce que

$$\Pr[P(r_1, r_2, \ldots, r_m) = 0] \leqslant \frac{d}{|S|}.$$

On souhaite obtenir un algorithme probabiliste décidant si un graphe possède un couplage parfait. En s'aidant du lemme de Schwartz-Zippel, on pourra assurer la correction de cet algorithme et en borner l'erreur. On pourra utiliser la fonction `log2` définie lorsque l'entête `<math.h>` est incluse.

Question 33. *Écrire une fonction*

```
bool parfait (int n, int *A, int p)
```

qui prend en entrée la matrice d'adjacence linéarisée `A` *d'un graphe G ayant* `n` *sommets, et décide si G possède un couplage parfait avec une probabilité d'erreur inférieure à $2^{-\texttt{p}}$,* ***qu'il faudra justifier****.*
On attend une complexité en $O\left(\texttt{p}\frac{\texttt{n}^4}{\log_2(\texttt{n})}\right)$, ***qu'il faudra justifier****.*

Fin du sujet.