TP9-10-10bis-11 : Graphes

2023/2024

1 Théorie des graphes

1.1 Degré

Exercice 1. Suite des degrées

La **suite des degrées** d'un graphe non-orienté est la suite décroissante des degrés d'un graphe.

1. Montrer que deux graphes isomorphe ont la même suite des degrés.
2. Montrer que deux graphes avec la même suite des degrés ne sont pas nécessairement isomorphes.

Exercice 2. Chemin eulérien

Un *chemin eulérien* est un chemin dans un graphe qui passe exactement une fois par toutes les arêtes.
Montrer qu'un graphe non-orienté connexe admet un chemin eulérien si et seulement si le nombre de ses nœuds de degré impaires est exactement 0 ou 2.

Exercice 3. Sommets de même degrés

Soit $G = (S, A)$ un graphe non orienté avec $|S| > 2$. Montrer qu'il existe deux sommets de même degré.

1.2 Connexité

Exercice 4. Connexité unilatérale

On dit qu'un graphe orienté $G = (S, A)$ est **unilatéralement connexe** si pour toute paire de sommet $s, s' \in S$ il existe un chemin de s à s', ou de s' à s.

1. Montrer que la forte connexité implique la connexité unilatérale.
2. Montrer que la connexité unilatérale implique la connexité faible.
3. Donner un contre-exemple pour chacune des implications réciproques.

Exercice 5. Transformation entre graphes orientés et non orientés

Soit $G = (S, A)$ un graphe orienté.

On note $\mathcal{E}(G)$ le graphe non-orienté S, A' où pour tout $s \neq s' \in S$, on a $\{s, s'\} \in A' \Leftrightarrow (s, s') \in A \vee (s', s) \in A$.

On note $\mathcal{B}(G)$ le graphe non-orienté S, A' où pour tout $s \neq s' \in S$, on a $\{s, s'\} \in A' \Leftrightarrow (s, s') \in A \wedge (s', s) \in A$.

Ainsi, $\mathcal{E}(G)$ est le graphe non-orienté où il y a une arête entre deux sommets s'il y a une arête entre les sommets dans un sens ou dans l'autre, tandis que $\mathcal{B}(G)$ est le graphe non-orienté où il y a une arêtre entre deux sommets s'il y a une arête entre les deux sommets dans un sens et dans l'autre.

1. Montrer que G est faiblement connexe si et seulement si $\mathcal{E}(G)$ est connexe.
2. Montrer que si $\mathcal{B}(G)$ est connexe, alors G est fortement connexe.
3. Montrer que la réciproque est fausse.

Exercice 6. Composantes connexes et relations d'équivalence

Soit $G = (S, A)$ un graphe non orienté. On défini la relation R sur l'ensemble des sommets S de sorte à ce que, pour tous sommets u et v, uRv si et seulement si u et v appartiennent à la même composante connexe.

Montrer que R est une relation d'équivalence.

1.3 Graphes bipartis

Exercice 7. Algo pour déterminer si un graphe est biparti

Un *graphe biparti* est un graphe $G = (S, A)$ tel qu'il existe une partition des sommets $S = E \sqcup F$ tel qu'il n'existe aucune arête entre deux sommets de E, ni entre deux sommets de F.

Proposer un algorithme qui détermine si un graphe est biparti.

Exercice 8. Graphe biparti

Un *graphe biparti* est un graphe $G = (S, A)$ tel qu'il existe une partition des sommets $S = E \sqcup F$ tel qu'il n'existe aucune arête entre deux sommets de E, ni entre deux sommets de F.

Montrer qu'un graphe non-orienté est biparti si et seulement si il n'existe aucun cycle de longueur impaire.

1.4 Un peu de dénombrement

Exercice 9. Nombres de sommets de graphes isomorphes

Montrer que deux graphes isomorphes ont nécessairement le même nombre de sommets et d'arêtes.

Montrer que ce n'est pas une condition suffisante.

Exercice 10. Nombre maximum d'arêtes

Quel est le nombre maximum d'arêtes dans un graphes non-orienté à n sommets.

Exercice 11. Nombre de graphes non-orientés

Quel est le nombre de graphes à n sommets non-orientés à isomorphisme près ?

Exercice 12. Nombre maximum de cycles

Quel est le nombre maximum de cycles dans un graphe non-orienté à n sommets ?

1.5 Arbres et graphes

Exercice 13. Graphes et forêts

1. Proposer une définition des forêts dans la théorique des graphes.
2. Montrer qu'un graphe est une forêt si et seulement si il existe au plus un chemin entre chaque paire de sommets de ce graphe.

Exercice 14. Relation entre les nombres d'arêtes, de sommets et de composantes connexes

Soit $G = (S, A)$ un graphe non-orienté. On note k le nombre de ses composantes connexes.

1. Montrer que $|S| - k \leqslant |A|$
2. Montrer que le cas d'égalité correspond exactement aux forêts.

2 Représentation des graphes

2.1 Propriétés sur les matrices et listes d'adjacences d'adjacences.

Exercice 15. Matrice d'adjacence et graphe transposé

Soit $G = (S, A)$ un graphe orienté. On nomme *graphe transposé* de G le graphe $G' = (S, A')$ tel que $A' = \{(v, u) | (u, v) \in A\}$.

Montrer que la matrice d'adjacence d'un graphe transposé est la transposé de la matrice d'adjacence du graphe.

Exercice 16. Matrice d'adjacence et exponentiation

Soit G un graphe dont la matrice d'adjacence est M.

Montrer que pour tout $k > 0$, M^k représente la matrice dont l'élément d'indice (i, j) correspond au nombre de chemins de longueur exactement k qui vont de i à j.

Exercice 17. Demi-anneau et exponentiation de matrice d'adjacence de booléens

Un demi-anneau est une structure algébrique $(E, +, \times, 0, 1)$ qui a les propriétés suivantes :
- $(E, +, 0)$ est un monoïde commutatif ;
- $(E, \times, 1)$ est un monoïde ;
- $\times$ est distributif par rapport à $+$;
- 0 est absorbant pour le produit.

Il s'agit, dans les faits, d'un anneau dans lequel on n'a pas nécessairement l'inverse par rapport à l'addition.

1. Montrer que $(\{\top, \bot\}, \vee, \wedge, \bot, \top)$, avec $\top$ et $\bot$ respectivement le vrai et le faux, est un demi-anneau.
2. Soit $G = (S, A)$ un graphe orienté, on considère la matrice d'adjacence booléene M de G définie par
$$m_{i,j} = \begin{cases} \top & \text{si } \{i, j\} \in A \\ \bot & \text{sinon} \end{cases}$$
On considère le produit de matrice sur les matrices booléens en utilisant le semi-anneau défini à la question précédente. Montrer que pour tout k, $M^k_{i,j}$ est égal à $\top$ si et seulement s'il existe un chemin de longueur exactement k de i à j.
3. Comment modifier M de sorte à considérer les chemins de longueur au plus k ?
4. Dans ce cas, que peut-on dire de la suite $(M^k)_{k \geqslant 1}$ et des composantes fortement connexes de G ?

2.2 Implémentation des graphes

Exercice 18. Graphes par matrices d'adjacence en OCaml

On se donne le type des matrices d'adjacence suivante :

```
type matrice = int array array
```

1. En utilisant Array.make_matrix proposer une fonction de type int -> matricequi crée un graphe de taille n donné en entrée avec aucune arêtes.
2. En utilisant Array.make, proposer une fonction de type int -> matricequi crée un graphe complet de taille n donné en entrée.
3. Proposer une fonction de type matrice -> intrenvoie le nombre de sommets d'un graphe.
4. Proposer une fonction de type matrice -> intqui renvoie le nombre d'arêtes d'un graphe orienté.
5. Proposer une fonction de type matrice -> intqui renvoie le nombre d'arêtes d'un graphe non-orienté.
6. Proposer une fonction de type matrice -> int -> int -> unit qui ajoute une arête dans une matrice d'adjacence d'un graphe orienté dont les indices des extrémités sont donnés en entrée.
7. Proposer une fonction de type matrice -> int qui renvoie le degré maximum d'un graphe non-orienté.
8. Proposer une fonction de type matrice -> intqui renvoie l'indice du sommet qui a le plus d'arêtes sortante d'un graphe orienté.
9. Proposer une fonction qui renvoie si un graphe est symmétrique.

Exercice 19. Graphes pondérés par matrice d'adjacence en OCaml

On se donne le type des matrices d'adjacence des graphes pondérés suivante :

```
type matrice_ponderee = float array array
```

1. Proposer une fonction de type int -> float -> matrice_pondere qui crée une matrice d'adjacence pondérée de taille n donnée en entrée, et dont tous les coefficients sont égaux à x donné en entrée.
2. Proposer une fonction de type int list -> matrice_pondere -> float qui renvoie le poids du sous-graphe induit par une liste d'indices de nœuds donnés en entrée.

Exercice 20. Graphes par listes d'adjacence en OCaml

On se donne le type suivant pour les listes d'adjacence d'un graphe :

```
type listes_adjacence = int list array
```

1. Proposer une fonction de type `int -> listes_adjacence` qui crée un graphe de taille n donné en entrée avec aucune arêtes.
2. Proposer une fonction de type `int -> listes_adjacence` qui crée un graphe complet de taille n donné en entrée.
3. Proposer une fonction de type `int -> listes_adjacence` qui renvoie le graphe orienté cyclique à n sommets.
4. Proposer une fonction de type `int -> listes_adjacence` qui renvoie le graphe non-orienté cyclique à n sommets.
5. Proposer une fonction de type `listes_adjacence -> int` qui renvoie le nombre de sommets d'un graphe.
6. Proposer une fonction de type `listes_adjacence -> int` qui renvoie le nombre d'arêtes d'un graphe orienté.
7. Proposer une fonction de type `listes_adjacence -> int`qui renvoie le nombre d'arêtes d'un graphe non-orienté.
8. Proposer une fonction de type `listes_adjacence -> int -> int -> unit` qui ajoute une arête dans des listes d'adjacence d'un graphe orienté dont les indices des extrémités sont donnés en entrée.
9. Proposer une fonction de type `listes_adjacence -> int` qui renvoie le degré maximum d'un graphe non-orienté.
10. Proposer une fonction de type `listes_adjacence -> int`qui renvoie l'indice du sommet qui a le plus d'arêtes sortante d'un graphe orienté.
11. Proposer une fonction de type `listes_adjacence -> int`qui renvoie l'indice du sommet qui a le plus d'arêtes entrante d'un graphe orienté.
12. Proposer une fonction qui renvoie si un graphe est symmétrique.

Exercice 21. Graphes par listes d'adjacence en C avec taille dans la première case

On considère un graphe représenté par ses listes d'adjacence en C. Le nombre de voisins vers qui il est possible d'aller est stocké dans le première élément de la liste d'adjacence.

Pour chacune de sfonctinos suivantes, on précisera la complexité en fonction de $|S|$ et de $|A|$ pour un graphe $G = (S, A)$.

1. Proposer une fonction de prototype `int nombre_aretes(int ** graphe, int taille )` qui renvoie le nombre d'arêtes dans un graphe supposé non-orienté.
2. Proposer une fonction de prototype `bool est_voisin(int ** graphe, int taille , int i, int j)` qui renvoie s'il existe une arête de i à j dans le graphe en entrée.
3. Proposer une fonction de prototype `void ajouter_arete(int ** graphe, int taille , int i, int j)` qui rajoute une arête de i à j dans un graphe orienté. On supposera que les listes d'adjacence sont allouées sur le tas, et on pensera à libérer les éventuels éléments retirés.

   On pourra supposer que l'arête n'est pas déjà présente dans le graphe.
4. Proposer une fonction de prototype `int ** copie(int** graphe, int taille )` qui renvoie une copie d'un graphe allouée sur le tas.
5. Proposer une fonction de prototype `void detruire(int ** graphe, int taille )` qui libère la méroire associé à un graphe en supposant que chacune des listes d'adjacence est allouée sur le tas, ainsi que le tableau total.

Correction :

1. On fait la somme des éléments dans les premières cases, puis on divise par deux.

```c
int nombre_aretes(int **graphe, int taille){
    int res = 0;
    for (int i = 0; i<taille; i++){
        res+=graphe[i][0];
    }
    return res/2;
}
```

Ici, on n'a pas besoin de parcourir l'intégralité des listes d'adjacence, mais seulement un traitement en temps constant pour chaque sommet, et on obtient une complexité meilleure que le résultat attendu : $O(|S|)$.

2. On parcours la liste d'adjacence d'indice i.

```c
bool est_voisin(int * graphe, int taille, int i, int j){
    for (int k = 1; k<=graphe[i][0];k++){
        if (graphe[i][k]==j)
            return true;
    }
    return false;
}
```

On doit parcourir dans le pire des cas tous les sommets présents dans la liste d'adjacence qui contient tous les sommets, et on fait un traitement en temps constant : $O(|S|)$.

3. Étant donné qu'on doit modifier la taille de la table, on doit créer un nouveau tableau.

```c
void ajouter_arete(int ** graphe, int taille, int i, int j){
    int ancienne_taille = graphe[i][0]+1;
    int * nouvelle_liste = (int *) malloc ((ancienne_taille +1) * sizeof(int));
    if (nouvelle_liste ==NULL)
        abort();
    nouvelle_liste[0]=ancienne_taille;
    nouvelle_liste[ancienne_taille]=j;
    for (int k=1; k<ancienne_taille; k++)
        nouvelle_liste[k]=graphe[i][k]
    free(graphe[i]);
    graphe[i]=nouvelle_liste;
}
```

Dans le pire des cas, on doit recopier presque tous les sommets, cela nous donne un traitement en $O(|S|)$.

4. Il faut allouer autant de tableau que nécessaire, et remplir les bons éléments.

```c
int ** copie(int ** graphe, int taille){
    int ** nouveau = (int**) malloc(sizeof(int*) * taille);
    for (int i = 0; i<taille; i++){
        int * nouvelle_ligne = (int *) malloc(sizeof(int) * (graphe[i][0]+1));
        for (int j = 0; j<=graphe[i][0]; j++){
            nouvelle_ligne[j]=graphe[i][j];
        }
        nouveau[i]=nouvelle_ligne;
    }
    return nouveau;
}
```

Il faut parcourir tous les nœuds et toutes les arêtes avec un traitement en temps constant, la complexité temporelle est donc en $O(|S|+|A|)$.

5. On doit libérer le tableau principal, ainsi que tous les sous-tableaux.

```c
void detruire(int ** graphe, int taille){
    for (int i = 0; i<taille;i++){
        free(graphe[i]);
    }
    free(graphe);
}
```

Exercice 22. Graphe listes d'adjacence en C avec sentinelle nulle

Une autre manière de procéder pour représenter les listes d'adjacence en C est d'utiliser un procédé qui est similaire aux chaîne de caractère : on peut utiliser une sentinelle qui vaut 0 pour décrire la fin du tableaux, mais il faut alors faire attention à décaler les indices dans le graphe pour commencer à 1.

Mêmes questions qu'à l'exercice précédent.

Exercice 23. Graphes par matrice et listes d'adjacence en OCaml

On se donne les types suivants pour respectivement les matrices et les listes d'adjacence :

```
type matrice_adjacence = int array array
type listes_adjacence = int list array
```

1. Proposer une fonction de type matrice_adjacence -> listes_adjacence qui convertit une matrice d'adjacence en les listes d'adjacence associées.
2. Proposer une fonction de type listes_adjacence qui fait l'inverse.

Exercice 24. Accès rapide par liste d'adjacence

On considère un arbre non-orienté représenté par ses listes d'adjacence.

Proposer une méthode pour tester si (s, s') est dans le graphe en $O(\min(d(s), d(s')))$.

3 Recherche de chemin dans un graphe

3.1 Accessibilité, chemin de poids minimal

Exercice 25. Chemin de poids minimal dans un graphe pondéré

1. Montrer que la distance dans un graphe pondéré par des poids positifs est atteinte par un chemin de poids minimal .
2. Que se passe-t-il dans le cas où il y a un cycle de poids négatif ?

Exercice 26. Propriétés sur le rayon et le diamètre

Soit $G_n = (S_n, A_n)$ un graphe non-orienté, et $G_o = (S_o, A_o)$ un graphe orienté.
Montrer les propositions suivantes, et donner un contre-exemple dans le cas des implications simples.

1. G_o est fortemment connexe si et seulement si $\text{diam}(G_o) < +\infty$.
2. G_n est connexe si et seulement si $\text{diam}(G_n) < +\infty$.
3. G_n est connexe si et seulement si $r(G_n) < +\infty$.
4. Si G_o est unilatéralement connexe, alors $r(G_o) < +\infty$.
5. Si $r(G_o) < +\infty$, alors G_o est faiblement connexe.
6. Si G_o est fortement connexe, alors $r(G_o) < +\infty$.

Exercice 27. Distance et plus court chemin

Une *distance* sur un ensemble E est une fonction d de $E \times E \mapsto \mathbb{R}^+$ telle que :
— $\forall x, y \in E, d(x, y) = 0 \Leftrightarrow x = y$
— $\forall x, y \in E, d(x, y) = d(y, x)$
— $\forall x, y, z \in E, d(x, y) + d(y, z) \geqslant d(x, z)$.

1. Soit $G = (S, A)$ un graphe non-orienté non pondéré. Montrer que la distance sur le graphe G (au sens défini par la théorie des graphes) est une distance sur S (au sens mathématique).
2. Soit $G = (S, A, p)$ un graphe non-orienté pondéré par des poids strictement positifs. Montrer que la distance sur le graphe G est une distance sur S.
3. Montrer que la distance sur les graphes orientés n'est pas nécessairement une distance sur l'ensemble des sommets.

3.2 Recherche de chemin

Exercice 28. Chemin dans un graphe par matrice d'adjacence

On se donne un graphe orienté non pondéré représenté par sa matrice d'adjacence au type `type matrice = int array array`.

1. Proposer une fonction de type `matrice -> int -> int -> int` qui renvoie la distance d'un sommet à un autre. On renverra -1 si la distance est infinie.
2. Proposer une fonction de type `matrice -> int -> int -> int list` qui renvoie un plus court chemin d'un sommet à un autre. On renverra une liste vide s'il n'en existe pas.
3. Même chose avec des listes d'adjacence au type `type listes_adjacence = int list array`.

Exercice 29. Reconstruction de l'arbre avec Dijkstra

1. Proposer une implémentation de l'algorithme de Dijkstra en OCaml. La fonction sera de type `(float * int) list array -> int -> float array * int array` et renverra d'une part le tableau des distances depuis le sommet source précisé en argument, et d'autre part le tableau des prédecesseurs.
2. Soit le type suivant :

```
type arbre = Vide | Node of int * arbre list
```

On cherche à représenter l'arbre des plus court chemin depuis un nœud. Le nœud dans la racine correspond à l'indice du nœud source, et pour chaque nœud, ses enfants sont les nœuds qui doivent être visités après ce nœud dans un plus court chemin depuis la racine.

Proposer une fonction de type `int array -> arbre` qui construit l'arbre des plus court chemin depuis le nœud source à l'aide du tableau des prédecsseurs.

Exercice 30. Reconstruction de l'arbre avec Floyd-Warshall

1. Proposer une fonction de type `'a array array -> 'a array array` qui recopie une matrice en entrée et dont les valeurs sont les mêmes que les valeurs de la matrice en entrée.
2. Proposer une implémentation de l'algorithme de Floyd-Warshall en OCaml. La fonction sera de type `float array array -> float array array` et construira une nouvelle matrice qui contient les distances entre les sommets.
3. Modifier cette implémentation pour renvoyer à la place la matrice des prédecesseurs, c'est-à-dire la matrice P telle que $p_{i,j}$ contienne l'indice du prédécesseur de j dans un plus court chemin de i à j.

   La fonction sera de type `float array array -> int array array`.
4. Proposer une fonction de type `int array array -> int -> int -> int list` qui à l'aide d'une matrice des prédecesseurs, ainsi que i et j deux sommets, renvoie un plus court chemin de i à j sous la forme d'une liste
5. Quelle est la complexité temporelle de la question précédente ?
6. Soit le type suivant :

```
type arbre = Vide | Node of int * arbre list
```

   On cherche à représenter l'arbre des plus court chemin depuis un nœud. Le nœud dans la racine correspond à l'indice du nœud source, et pour chaque nœud, ses enfants sont les nœuds qui doivent être visités après ce nœud dans un plus court chemin depuis la racine.

   Proposer une fonction de type `int array array -> int -> arbre` qui construit l'arbre des plus court chemin depuis un nœud en entrée à l'aide de la matrice des prédecsseurs.
7. Quelle est la complexité temporelle de la question précédente ?

Exercice 31. Complexité de l'implémentation de Dijkstra avec la matrice d'adjacence

Quelle est la complexité temporelle de l'algorithme de Dijkstra si on utilise une matrice d'adjacence à la place des listes d'adjacence ?

Exercice 32. Complexité de l'implémentation de l'algorithme de Floyd-Warshall avec les listes d'adjacence

1. Quelle est la complexité temporelle de l'algorithme de Floyd-Warshall en utilisant la représentation par listes d'adjacence pour construire la matrice ?
2. Quelle est la complexité temporelle de l'algorithme de Floyd-Warshall si on stocke les valeurs de la matrice dans des listes d'adjacence ?

Exercice 33. Tas avec modification de clef en temps logarithmique en OCaml

On cherche à réaliser un tas contenant des paires clef-valeur en OCaml et dont les clefs figurent parmi les nombres de 0 à $n-1$ pour un certain nombre n.

On veut pouvoir faire les opérations habituelles sur les tas d'ajout d'une paire quelconque et du retrait de la racine, mais on veut par ailleurs pouvoir modifier pour chaque clef la valeur qui lui est associée en temps logarithmique.

Pour se faire, on utilise un tableaux `valeurs` qui contient les valeurs et qui a une structure de tas, mais aussi deux tableaux `vers_clefs` et `vers_valeurs` tels que, pour tout i inférieur au nombre d'éléments dans le tas `vers_clefs.(i)` contiennent la clef de la valeur `valeurs.(i)`, et telle que pour tout i parmi les clefs dans la racine `vers_valeurs.(i)` contiennent l'indice de la valeurs associée à la clef `i`.

Les indices contenus dans les tableaux `vers_clefs` et `vers_valeurs` qui ne sont pas définis au dessus devront tout de même respecter la propriété que `vers_clefs.(vers_valeurs.(i))` est égal à i, et pareil pour `vers_valeurs.(vers_clefs(i))` est égal à i.

Pour les indices qui ne sont pas utilisés dans le tableau `valeurs`, on mettra la valeur $+\infty$.

On utilisera donc le type suivant en OCaml :

```
type tas_maj = {
    mutable nb_elements : int;
    valeurs : float array ;
    vers_clefs : int array ;
    vers_valeurs : int array
}
```

On a rajouté un nombre `nb_elements` pour avoir le nombre d'éléments déjà ajoutés dans le tas.

1. Proposer une fonction de signature `int -> tas_maj` qui construit un tas avec une taille maximum pour les tableau d'au plus n passé en entrée. Initialement, le tas doit contenir toutes les clefs de 0 à $n-1$ avec des valeurs égales à $+\infty$.

   On pourra utiliser `Float.infinity` pour avoir la valeur de $+\infty$.
2. Proposer une fonction de signature `tas_maj -> int -> float option` qui avec une fonction qui avec un tas et une clef renvoie `Some x` où `x` est la valeur si elle est présente, et `None` sinon.
3. Proposer une fonction `echanger_un` de signature `'a array -> int -> int -> unit` qui échange les valeurs dans un tableau en entrée à deux indices donnés.
4. Proposer une fonction `echanger` de signature `tas_maj -> int -> int -> unit` qui avec un tas et deux indices réalise l'échange dans le tableau entre ces deux indices.

   On prendra soin d'échanger les valeurs dans les trois tableaux : les indices en entrées sont ceux des tableaux de `valeurs` et `vers_clefs` mais pas nécessairement les bon indices dans `vers_valeurs`.
5. Proposer une fonction `remonter` de signature `tas_maj -> int -> unit` qui réalise la remontée dans un tas à partir de l'indice de sorte à rectifier la structure de tas.
6. Expliquer comment implémenter la fonction `descendre` qui fait l'inverse.
7. Expliquer comment implémenter les fonctions d'ajout et de retrait dans le tableau.
8. Proposer une fonction `modifier_valeur` de signature `tas_maj -> int -> float -> unit` qui modifie une valeur d'indice donnée pour devenir une valeur donnée. On mettra à jour le tas pour conserver les contraintes sur la structure.

   Si la clef n'est pas présente dans le tas, la fonction n'aura aucun effet.
9. Quelle est la complexité de cette fonction ?
10. Proposer une fonction de signature `tas_maj -> int -> float -> bool` qui modifie une valeur à une clef donnée si cette clef existe et si la nouvelle valeur est strictement inférieure à la précédente. Si la modification a lieu, la fonction renvoie `true` et `false` sinon.

Exercice 34. Dijkstra avec la bonne complexité

On souhaite implémenter l'algorithme de Dijkstra avec la structure précédente.
L'idée est de n'avoir pour chaque sommet que la valeur la plus faible pour cette clef là.
On suppose qu'on dispose d'un graphe pondéré donné par le type suivant :

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

Il s'agit d'une liste d'adjacence où, pour chaque arête, on dispose d'un couple du poids puis de l'indice du sommet de destination.

1. Proposer une fonction de signature `graphe -> int -> float array` qui renvoie le tableau des distances à partir d'un sommet dans un graphe passés en argument.
2. Comment modifier la fonction précédente de sorte à pouvoir renvoyer le tableau des prédecesseur en plus ?
   La nouvelle fonction aura une signature `graphe -> int -> (float array * int array)`
3. Montrer que l'implémentation que l'on s'est donné a une complexité spatiale en $O(|S|)$.

4 Graphes particuliers

4.1 Graphes orientés acycliques

Exercice 35. Relation d'ordre dans un graphe orienté acyclique

Soit $G = (S, A)$ un graphe orienté acyclique. On définie la relation R sur S de la manière suivante : pour tout $u, v \in S$, uRv si et seulement s'il existe un chemin de u à v dans G ou si $u = v$.
Montrer que R est une relation d'ordre.

Exercice 36. Transposé d'un graphe orienté acyclique

Soit $G = (S, A)$ un graphe. Son graphe transposé $G^T = (S, A')$ est le graphe de sorte à ce que $(u, v) \in A \Leftrightarrow (v, u) \in A'$.
Montrer que G est acyclique si et seulement si G^T est acyclique.

Exercice 37. Chemin dans un graphe orienté acyclique

Proposer un algorithme de complexité temporelle en $O(|S| + |A|)$ pour trouver le plus court chemin d'un sommet à un autre dans un graphe orienté acyclique.

Exercice 38. Cloture transitive d'un graphe orienté acyclique

La clôture transitive d'un graphe $G = (S, A)$ orienté est un graphe orienté $G' = (S, A')$ de sorte à ce que A' soit minimal au sens de l'inclusion pour les propriétés suivantes :
— $A \subset A'$
— Pour toute paire de sommet u, v, s'il existe un chemin de u à v, alors il existe une arête de u à v.

1. Montrer que pour toute paire de sommet u et v, il y a un chemin de u à v dans G si et seulement s'il y a un chemin de u à v dans G'.
2. En déduire que la clôture transitive d'un graphe orienté acyclique est un graphe orienté acyclique.

Exercice 39. Parcours en profondeur en C

On se donne un graphe sous la forme de ses matrices d'adjacence `int t[ taille ][ taille ]`, en supposant qu'on a préalablement défini taille de la manière suivante `const int taille = 4;`.

1. Proposer une fonction récursive de prototype `void explorer(int t[ taille ][ taille ], int s, bool deja_vu [taille])` qui affiche les sommets dans l'ordre préfixe dans l'exploration en profondeur depuis le sommet s.

   `deja_vu` contient le tableau des sommets que l'on a déjà vu lors du parcours et on mettra à jour ce tableau.
2. Proposer une fonction de prototype `void afficher_ordre_prefixe(int t[ taille ][ taille ])` qui affiche les sommets dans un ordre préfixe d'un parcours en profondeur.
3. Comment modifier la fonction de sorte à pouvoir renvoyer l'ordre préfixe sous la forme d'un tableau d'entiers ?

Exercice 40. Ordre topologique à base des matrices en C

On se donne un graphe représenté par sa matrice d'adjacence. Propoposer une fonction de prototype `int * tri_topologique(int t[ taille ][ taille ])` qui renvoie un tableau alloué sur le tas dont les valeurs forment un tri topologique de t.

Exercice 41. Ordre topologique à base des listes d'adjacence en C

Même question que l'exercice précédent en utilisant des listes d'adjacence en C.
Le premier élément de chaque tableau contient le degré sortant.

Correction : On commence par réaliser une matrice qui réalise l'exploration à partir d'un nœud. Elle prend en argument le tableau des éléments déjà vu, le tableau à modifier, et un pointeur vers l'indice de l'élément à parcourir.

```
void explorer(int **graphe, int source, bool deja_vu[], int * compteur_ptr, int
    ordre[]){
    if (deja_vu[source]){
        return;
    }
    deja_vu[source]=true;

    for (int i = 1; i<=graphe[source][0]; i++){
        explorer(graphe, graphe[source][i], deja_vu, compteur_ptr, ordre);
    }

    ordre[*compteur_ptr]=source;
    (*compteur_ptr)++;
}
```

Il nous faudra une fonction d'inversion d'un tableau. Il est possible de s'en passer, mais nous préférons l'implémenter ici :

```
void inverser(int tableau[], int taille){
    for (int i = 0; i<taille/2; i++){
        int c = tableau[i];
        tableau[i] = tableau[taille - i - 1];
        tableau[taille - i - 1] = c;
    }
}
```

Il faut maintenant la fonction qui fait le parcours depuis tous les éléments.

```c
int * tri_topologique(int ** graphe, int taille){
    bool * deja_vu = (bool *) malloc (sizeof(bool) * taille);
    int * ordre = (int *) malloc (sizeof(int) * taille);
    int compteur = 0;
    for (int i = 0; i<taille; i++){
        deja_vu[i]=false;
    }
    for (int i = 0; i<taille; i++){
        explorer(graphe, taille, i, deja_vu, &compteur, ordre);
    }
    free(deja_vu);
    inverser(ordre, taille);
    return ordre;
}
```

4.2 Graphes de flux de contrôle

Exercice 42. Sommets non accessibles et code mort

Que peut-on dire d'un sommet qui n'est pas accessible depuis le point d'entrée dans un graphe de flux de contrôle ?

Que dire d'un sommet depuis lequel aucun point d'arrivée n'est accessible ?

Exercice 43. Code mort qui n'est pas visible sur le graphe

Proposer un programme de sorte à ce dans son graphe de flux de contrôle,un sommet soit accessible au sens de la théorie des graphes, mais que la ligne du programme ne puisse pas être exécutée.

4.3 Graphes planaires

Exercice 44. Formule d'Euler

Un graphe est dit planaire quand il est possible de le représenter dans le plan sans que les arêtes s'intersectent.

Par exemple, le graphe du cube est planaire, et le graphe complet à 5 sommets n'est pas planaire.

On note f le nombre de faces d'un graphe, c'est-à-dire le nombre de zones délimitées par les arêtes (en comptant la zone extérieur) dans une représentation où il n'y a pas d'intersection entre les arêtes.

1. Montrer que pour un graphe connexe $G = (S, A)$ planaire, on a $|S| - |A| + f = 2$.
2. Montrer que pour un graphe connexe $G = (S, A)$ planaire, on a $3f \leqslant 2|A|$.
3. En déduire que pour un graphe connexe $G = (S, A)$ planaire avec au moins 3 sommets, on a $|A| \leqslant 3|S| - 6$.