Graphes

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

12 juin 2024

1 Introduction

Au chapitre 8, nous avons étudié les arbres, qui sont des structures ***hiérarchiques***. Cela signifie que les arbres sont adaptés pour représenter et modéliser des situations comme des organigrammes d'entreprise, des arbres syntaxiques (pour les expressions arithmétiques ou le code HTML par exemple), où chaque élément considéré a un "parent", ou un "prédécesseur", unique. Les graphes vont permettre de représenter des situations plus larges, où les différents éléments peuvent présenter des relations plus complexes. On parlera de structure ***relationnelle***.

Quelques exemples de situations que l'on pourra modéliser avec des graphes :

— Un réseau social type Facebook, où deux personnes peuvent être amies (relation symétrique)

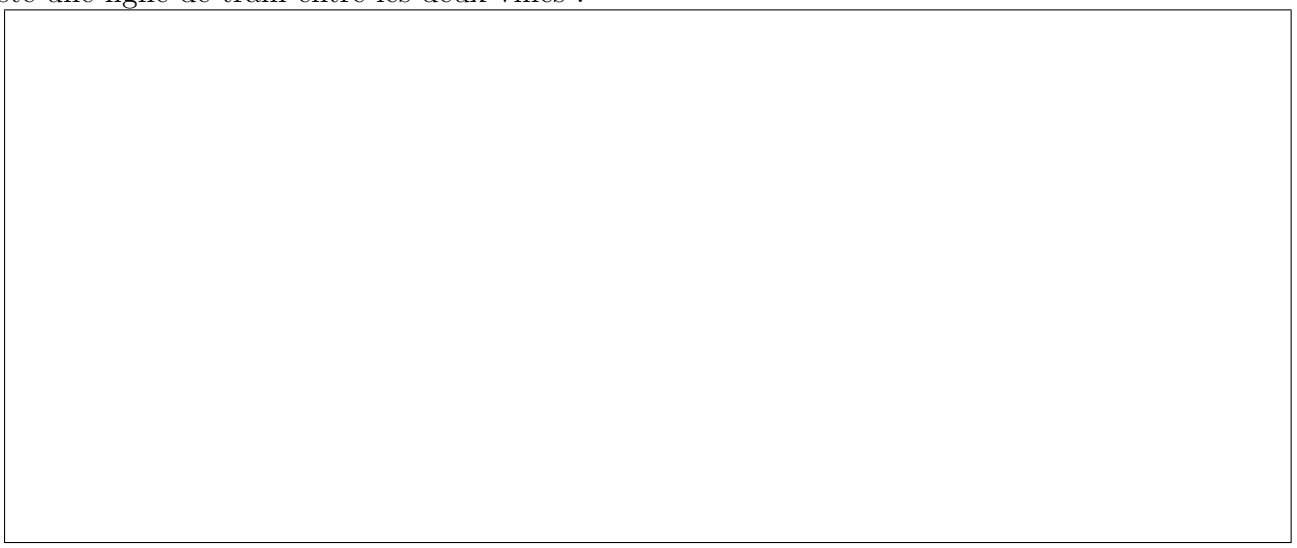
— Un réseau social type Twitter, où une personne peut en suivre une autre (relation asymétrique)

— Le World Wide Web : des pages web pointant les unes vers les autres.

— Un réseau routier : des routes de longueurs différentes, reliant des villes, des intersections, des ronds-points, etc...

A Premières définitions

Schématiquement, un graphe est un ensemble de points reliés par des segments. Les points s'appellent des ***sommets***, et les segments des ***arêtes***. Par exemple, voici un graphe dont les sommets sont des villes de france, et dans lequel deux sommets sont reliés par une arête s'il existe une ligne de train entre les deux villes :



Définition 1. Un graphe non-orienté est un couple $G = (S, A)$ où $A \subseteq \{\{x, y\} \mid x, y \in S\}$. S est appelé l'ensemble des **sommets**, et A l'ensemble des **arêtes**.

Pour $x \in S$, si $\{x\} \in A$ est une arête reliant le sommet x à lui-même, on dit que c'est une **boucle**.

On dit que $x, y \in S$ sont **voisins** s'il existe une arête entre les deux, i.e. si $\{x, y\} \in A$.

Pour $x \in S$, on appelle **voisinage** de x dans G l'ensemble des voisins de x. On le note $\mathcal{V}(x)$:

$$\mathcal{V}(x) = \{y \in S \mid \{x, y\} \in A\}$$

On appelle **degré** de x dans G le nombre de voisins de x, on le note $\mathbf{deg}(x)$:

$$\mathbf{deg}(x) = |\mathcal{V}(x)|$$

Exemple 1. Le graphe donné graphiquement à l'exemple précédent est donc le graphe $G = (S, A)$ non-orienté suivant :

— $S =$ { Toulouse, Strasbourg, Montpellier, Lyon, Marseille, St-Étienne }

— $A =$ {(Toulouse, Montpellier), (Toulouse, Lyon), (Toulouse, Marseille), (Montpellier, Lyon), (Montpellier, Marseille), (Lyon, Lyon), (Lyon, Marseille), (Lyon, St-Étienne)}

Le voisinage de Toulouse est l'ensemble {Montpellier, Lyon, Marseille}, son degré est de 3. Le graphe comporte une boucle, au niveau du sommet Lyon.

Un graphe non-orienté est donc exactement la donnée d'un ensemble S et d'une relation binaire symétrique A sur S. Par opposition, un graphe orienté permettra de représenter des relations pas nécessairement symétriques. Dans un graphe orienté, on relie les sommets non pas par des segments mais par des flèches. Par exemple, le graphe suivant représente quelques types du jeu Pokémon, et l'on ajoute une flèche entre deux types si le premier est efficace contre le deuxième :

Définition 2. Un graphe orienté est une paire $G = (S, A)$ où $A \subseteq S^2$. S est appelé l'ensemble des **sommets**, et A l'ensemble des **arcs**.

Pour $x \in S$, si $(x, x) \in A$ est un arc reliant le sommet x à lui-même, on dit que c'est une **boucle**.

Si $(x, y) \in A$, on dit que x est prédecesseur de y, et que y est successeur de x.

Pour $x \in S$, on notera $\mathcal{V}^-(x)$ l'ensemble de ses prédécesseurs, que l'on appellera **voisinage entrant**, et $\mathcal{V}^+(x)$ l'ensemble de ses successeurs, que l'on appellera **voisinage sortant**. On note $\mathbf{deg}^-(x) = |\mathcal{V}^-(x)|$, on l'appelle **degré entrant** : c'est le nombre de flèches qui entrent dans x. On note de même $\mathbf{deg}^+(x) = |\mathcal{V}^+(x)|$, on l'appelle **degré sortant** : c'est le nombre de flèches qui sortent de x.

Exemple 2. Le graphe dessiné ci-dessus est donc le graphe $G = (S, A)$ orienté suivant :

— $S =$ { Feu, Eau, Plante, Vol, Glace, Dragon }

— $A =$ { (Feu, Plante), (Feu, Glace), (Eau, Feu), (Plante, Eau), (Vol, Plante), (Glace, Plante), (Glace, Vol), (Glace, Dragon), (Dragon, Dragon) }

Le type Plante a un degré entrant de 3, et un degré sortant de 1. Le graphe comporte une boucle, car il y a un arc entre le type Dragon et lui-même.

Proposition 1. Soit $G = (S, A)$ un graphe non-orienté, sans boucle. On note $n = |S|, m = |A|$. Alors :

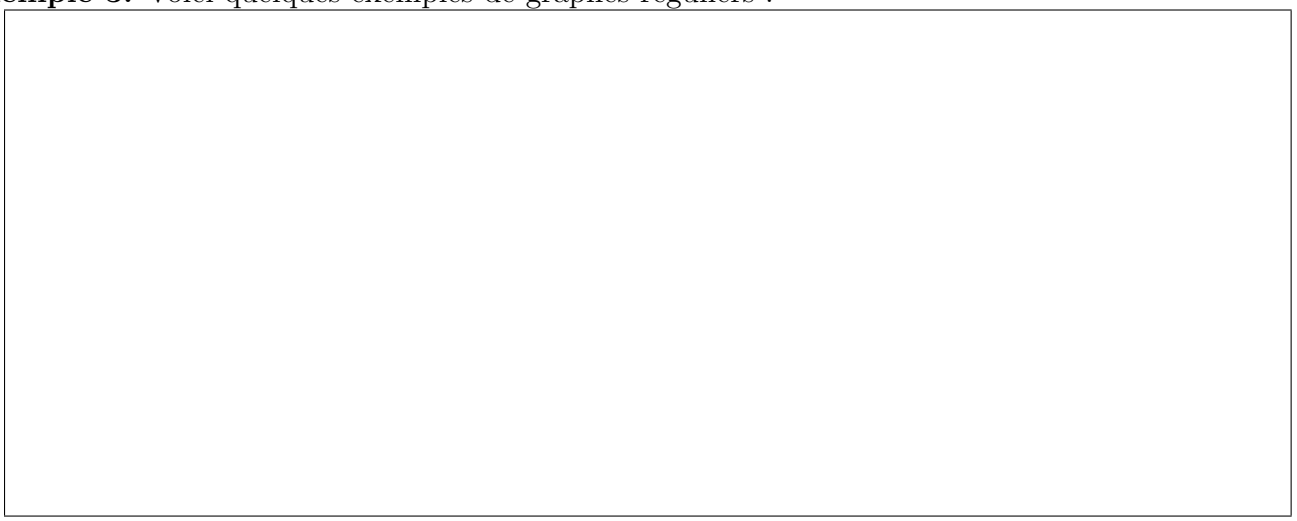
$$\sum_{x \in S} \mathbf{deg}(x) = 2m$$

Proposition 2. Soit $G = (S, A)$ un graphe orienté, sans boucle. On note $n = |S|, m = |A|$. Alors :

$$\sum_{x \in S} \mathbf{deg}^-(x) = \sum_{x \in S} \mathbf{deg}^+(x) = m$$

Définition 3. Un graphe est dit régulier si tous ses sommets sont de même degré.

Exemple 3. Voici quelques exemples de graphes réguliers :



Exercice 1.

Question 1. Soit G un graphe régulier, de degré d, avec n sommets et m arêtes. Donner un lien entre n, m, d.

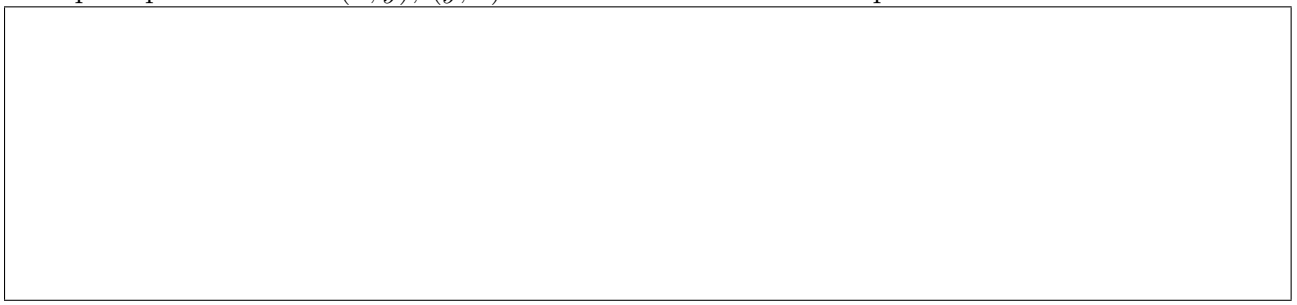
Question 2. Décrire les graphes réguliers de degré 2.

On dit qu'un graphe non-orienté est **cubique** s'il est régulier, de degré 3.

Question 3. Dessiner des graphes cubiques à 4, 6 et 8 sommets.

Question 4. Montrer qu'un graphe cubique a un nombre pair de sommets.

Graphe non-orienté Parfois, le terme "graphe non-orienté" désigne les graphes orientés dans lesquels pour tout arc (x, y), (y, x) est aussi un arc. Par exemple :



Dans la suite, on utilisera les notations des graphes orientés pour les graphes pour les arêtes : on écrira (x, y) et pas $\{x, y\}$.

2 Représentation en mémoire

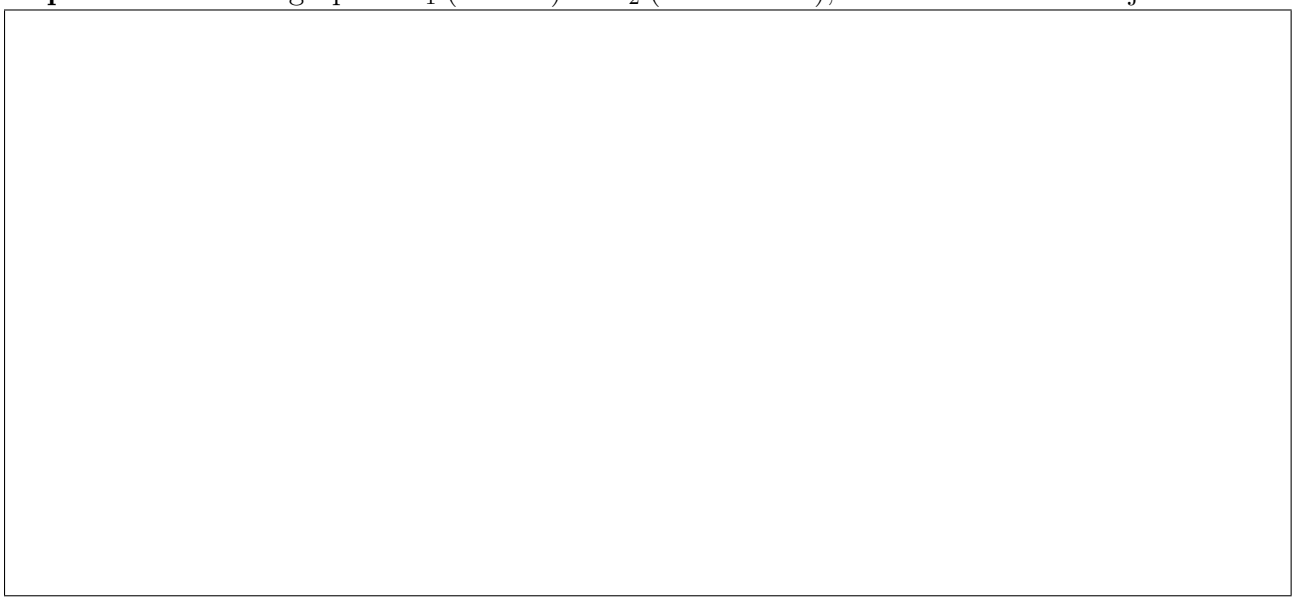
Dans cette partie, on considère que nos graphes ont comme ensembles de sommets des intervalles de la forme $[\![0, n-1]\!]$ avec $n \in \mathbb{N}$. Cela permettra de simplifier la description des structures utilisées, qui utiliseront alors de simples tableaux. On verra en TP comment généraliser ces représentations, en utilisant des dictionnaires.

Les deux manières les plus communes de représenter les graphes dans un programme sont les **matrices d'adjacences** et les **listes d'adjacence**.

A Matrice d'adjacence

On considère un graphe orienté $G = (S, A)$, avec $S = \{0, \dots, n-1\}$. La matrice d'adjacence de G est $M = (m_{ij})_{0 \leq i,j < n}$ avec $m_{ij} = 1$ si i et j forment une arête, 0 sinon.

Exemple 4. Voici deux graphes G_1 (orienté) et G_2 (non-orienté), et leurs matrices d'adjacence :



Remarque 1. La matrice d'adjacence d'un graphe non-orienté est symétrique.

On peut donc représenter un graphe $G = (S, A)$ avec $|S| = n$ et $|A| = m$ par un tableau 2D stockant les coefficients de la matrice d'adjacence. La taille nécessaire est $\Theta(n^2)$. Intéressons nous à la complexité d'opérations simples :

— Étant donné $s, t \in S$, déterminer si (s, t) est une arête : $\mathcal{O}(1)$. Il suffit de regarder la case correspondante de la matrice d'adjacence.

— Étant donné $s \in S$, déterminer la liste des voisins de s : $\mathcal{O}(n)$. On parcourt la liste des sommets de G, en regardant pour chaque sommet $t \neq s$ si (s, t) est une arête.

Implémentation en C Comme on considère des graphes où les sommets sont des entiers consécutifs entre 0 et $n-1$ (avec n la taille du graphe), les listes de sommets seront des listes d'entiers positifs. En s'inspirant du fonctionnement des strings, nous allons marquer la fin des listes d'entier avec des -1. On pourra alors manipuler les listes sans connaître leur taille au préalable.

```c
typedef struct graph {
  int n; // taille du graphe
  bool** m_adj; // matrice d'adjacence: m_adj[i][j] indique une arête i -> j
} graph_t

bool est_arete(graph_t* g, int i, int j){
  return g->m_adj[i][j];
}

// renvoie une liste d'entiers positifs contenant les voisins
// de i dans g. La valeur sentinelle -1 indique la fin de la liste.
int* liste_voisins(graph_t* g, int i){
  // compter les voisins
  int cnt = 0;
  for (int j = 0; j < g->n; j++){
    cnt += g->m_adj[i][j];
  }
  int* res = malloc((cnt+1)*sizeof(int));
  int curseur = 0; // prochaine case de res à remplir
  for (int j = 0; j < g->n; j++){
    if (g->m_adj[i][j]){
      res[curseur] = j;
      curseur++;
    }
  }
  res[curseur] = -1;
  return res;
}
```

Dans `liste_voisins`, on aurait pu se passer de la première étape comptant le nombre de voisins, en utilisant un système de tableaux redimensionnables.

B Listes d'adjacence

On considère un graphe non orienté $G = (S, A)$, avec $S = \{0, \ldots, n-1\}$. Notons $m = |A|$. La représentation de G par listes d'adjacence de G est la donnée, pour chaque sommet s, de la liste des voisins de i, ce que l'on appelle sa liste d'adjacence. On représente donc un graphe par un tableau de listes.

Exemple 5. Voici deux graphes G_1 (orienté) et G_2 (non-orienté), et leurs représentations par listes d'adjacence :

Cette représentation demande un espace proportionnel à $\sum_{i=1}^{n}(1+\mathbf{deg}(s_i)) = n+m$, car il faut stocker la liste d'adjacence (éventuellement vide) de chaque sommet. Étudions les complexités des deux opérations simples étudiées plus haut :

— Étant donné $s, t \in S$, déterminer si (s, t) est une arête : on parcourt la liste d'adjacence de s pour y chercher t, en temps $\mathcal{O}(\mathbf{deg}(s))$.. Si le graphe est non-orienté, on peut chercher également dans la liste d'adjacence de t pour y chercher s, et donc en choisissant la liste la plus courte, on obtient une complexité en $\mathcal{O}(1 + \min(\mathbf{deg}(s), \mathbf{deg}(t)))$.

— Étant donné $s \in S$, déterminer la liste des voisins de s : $\mathcal{O}(\mathbf{deg}(s))$. On copie la liste d'adjacence de s.

Implémentation en C :

```c
typedef struct graph {
  int n;
  int** voisins; // voisins[i] contient les voisins de i
                 // (la fin est marquée par -1)
} graph_t;

bool est_arete(graph_t* g, int i, int j){
  int k = 0;
  while (g->voisins[i][k] != -1 && g->voisins[i][k] != j){
    k++;
  }
  // soit voisins[i][k] = -1, auquel cas on a parcouru toute la liste sans
  // trouver j, soit voisins[i][k] = j, auquel cas i -> j est une arête
  return g->voisins[i][k] == j;
}

int* liste_voisins(graph_t* g, int i){
  // compter les voisins
  int cnt = 0
  while (g->voisins[i][cnt] != -1){
    cnt++;
  }
  int* res = malloc((cnt+1)*sizeof(int));
  for (int k = 0; k <= cnt; k++){
    res[k] = g->voisins[i][k];
  }
  return res;
}
```

Comparaison des performances Pour déterminer quand utiliser quelle représentation, il faut donc réfléchir aux opérations que l'on veut effectuer sur le graphe, ainsi qu'à la structure du graphe. S'il est très dense, c'est à dire s'il contient beaucoup d'arêtes, alors m est de l'ordre de n^2, donc les deux représentations prennent la même place en mémoire. Cependant, si le graphe est creux, c'est à dire si m est très faible devant n^2, alors la représentation par listes d'adjacence est bien plus légère. Dans de nombreux graphes réels (les réseaux sociaux, les sites internets...), il y a un nombre immense de sommets (plusieurs millions), mais chaque sommet n'a que quelques voisins, autrement dit le degré moyen des sommets du graphe est faible devant n. La représentation par listes d'adjacence devient alors bien plus efficace. De plus, cette représentation permet d'accéder facilement à la liste des voisins d'un sommet, ce qui permet d'explorer le graphe de manière efficace, comme nous allons le voir plus tard.

3 Accessibilité

Les questions d'accessibilité dans un graphe portent sur la manière dont les différentes parties du graphe sont connectées : est-il possible de passer d'un sommet à un autre en suivant des arêtes / arcs, est-il possible de passer de tout sommet à tout autre sommet ainsi, etc...

Définition 4. Soit $G = (S, A)$ un graphe orienté, et $s, t \in S$. Un **chemin** entre s et t dans G est une suite de sommets $s_0 s_1 \dots s_k$ tels que :

- $s_0 = s$
- $s_k = t$
- $\forall i \in [\![1, k]\!], (s_{i-1}, s_i) \in A$

La longueur d'un chemin est le nombre d'arcs qu'il comporte. Avec les notations précédentes, c'est k.

Pour $0 \leq i \leq j \leq k$, le chemin $s_i \dots s_j$ est un **sous-chemin** de $s_0 \dots s_k$.

On étend ces définitions de manière analogue aux graphes non-orientés. Parfois, pour les graphes non-orientés, on utilise le terme "**chaîne**" plutôt que "chemin".

Définition 5. Soit $G = (S, A)$ un graphe, et $s \in S$. Un chemin entre s et s est appelé un **circuit**. Parfois, pour les graphes non-orientés, on utilise le terme "**cycle**".

On dit qu'un chemin est **élémentaire** s'il ne contient aucun circuit, autrement dit si aucun de ses sous-chemuns n'est un circuit.

Exemple 6. Dans le graphe G suivant, il existe un circuit de longueur 4 : 2-4-3-5-2. Il existe donc une infinité de chemins entre 1 et 6 : 1-2-6 qui est de longueur 2, 1-2-4-3-5-2-6 qui est de longueur 6, etc...

A Composantes connexes

On considère pour l'instant des graphes non-orientés.

Remarque 2. Soit $G = (S, A)$ un graphe. A est une relation symétrique. Donc, sa clôture transitive réflexive est une relation d'équivalence. Notons la $\leftrightarrow$. Cette relation correspond en fait exactement à l'existence d'un chemin entre deux sommets :

$$\forall x, y \in S, x \leftrightarrow y \iff \text{il existe un chemin entre } x \text{ et } y$$

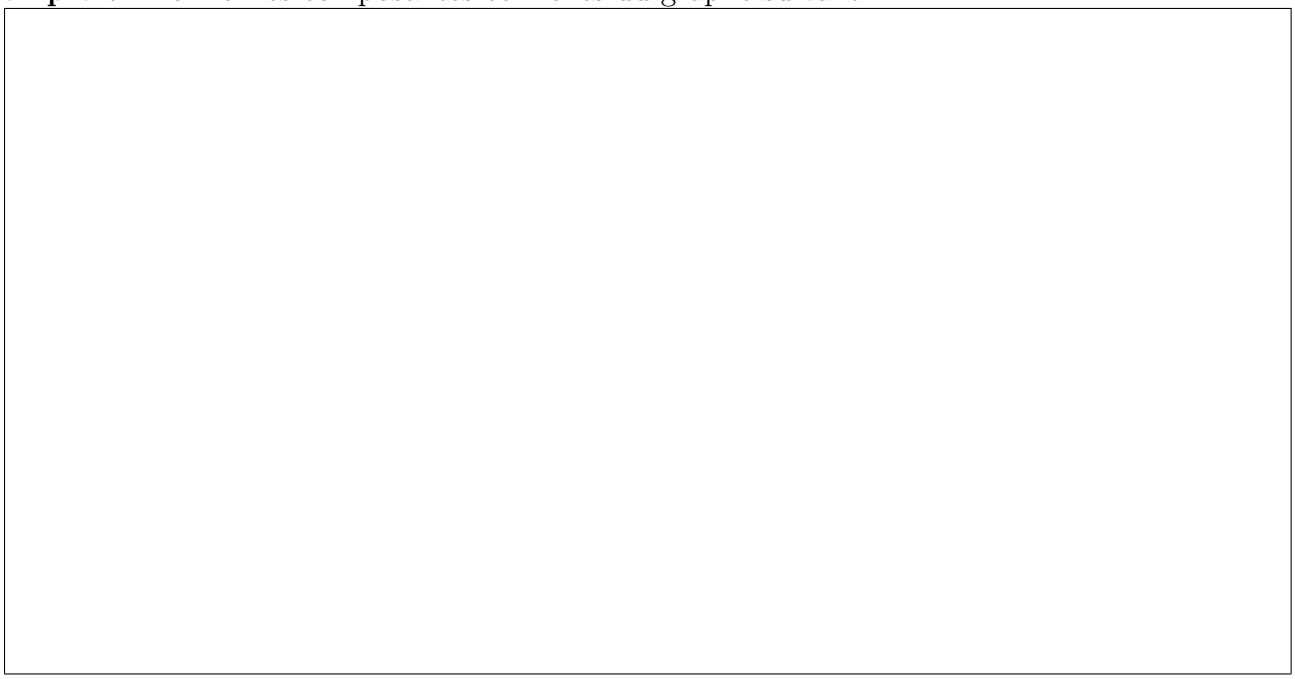
On peut donc regrouper les sommets selon les classes d'équivalences de cette relation :

Définition 6. Une composante connexe d'un graphe $G = (S, A)$ non-orienté est une classe d'équivalence de la relation $\leftrightarrow$. Autrement dit, c'est un ensemble $C \subseteq S$ tel que :

— $\forall x, y \in C$, il existe un chemin entre x et y ;

— $\forall x \in C, \forall y \notin C$, il n'existe pas de chemin entre x et y.

Exemple 7. Donner les composantes connexes du graphe suivant :



Remarque 3. En tant que classes d'une relation d'équivalence, les composantes connexes d'un graphe $G = (S, A)$ forment une partition de l'ensemble des sommets S.

Définition 7. On dit qu'un graphe $G = (S, A)$ est connexe s'il ne possède qu'une seule composante connexe, i.e. si pour tout couple de sommets $(x, y) \in S^2$, il existe un chemin entre x et y.

B Composantes fortement connexes

Remarque 4. Soit $G = (S, A)$ un graphe orienté. On considère la clôture transitive réflexive de A, que l'on note $\to$. Cette relation correspond à l'existence d'un chemin entre deux sommets :

$$\forall x, y \in S, x \to y \iff \text{il existe un chemin de } x \text{ à } y$$

Notons que $\to$ n'est pas nécessairement une relation d'équivalence, car elle n'est pas a priori symétrique. On introduit la relation $\leftrightarrow$ définie comme suit :

$$\forall x, y \in S, x \leftrightarrow y \iff x \to y \text{ et } y \to x$$

Proposition 3. $\leftrightarrow$ est une relation d'équivalence.

On peut donc regrouper les sommets selon les classes d'équivalences de cette relation :

Définition 8. Une **composante fortement connexe** d'un graphe $G = (S, A)$ est une composante fortement connexe pour la relation $\leftrightarrow$, i.e. un ensemble $C \subseteq S$ tel que :

— $\forall x, y \in C, x \to y$ et $y \to x$;

— $\forall x \in C, \forall y \notin C$, on n'a pas à la fois $x \to y$ et $y \to x$.

Exemple 8. Donner les composantes fortement connexes du graphe suivant :

Remarque 5. En tant que classes d'équivalences d'une relation d'équivalence, les composantes connexes d'un graphe $G = (S, A)$ forment une partition de l'ensemble des sommets S.

Exercice 2.

Question 1. On considère un graphe non-orienté $G = (S, A)$. Quel est le lien entre les composantes connexes et les composantes fortement connexes de G ?

Question 2. On considère un graphe $G = (S, A)$ orienté. On considère le graphe non-orienté $G^* = (S, A^*)$ obtenu en enlevant l'orientation des arcs, i.e. avec $A^* = \{\{i, j\} \mid (i, j) \in A\}$. Est-il vrai que les composantes connexes de G^* sont les composantes fortements connexes de G ?

C Parcours de graphe non orienté

Lors du chapitre 8, nous avons vu différentes manières de parcourir un arbre, c'est à dire de visiter chacun de ses noeuds. Les parcours de graphe ont un rôle analogue, et sont utilisés lorsque l'on veut visiter chaque sommet d'un graphe. De nombreux algorithmes de graphes prennent la forme d'un parcours, par exemple le calcul des composantes connexes d'un graphe.

Rappel : Parcours d'arbre. Nous avons vu deux types de parcours d'arbre : les parcours en profondeur et les parcours en largeur. Les parcours en profondeur s'expriment de manière assez naturelle récursivement, par exemple en OCaml :

```
let rec parcours (a: arbre) = match a with
  | Feuille(x) -> (* opération sur x *)
  | Noeud(x, g, d) -> (* opération sur x *); parcours g; parcours d
```

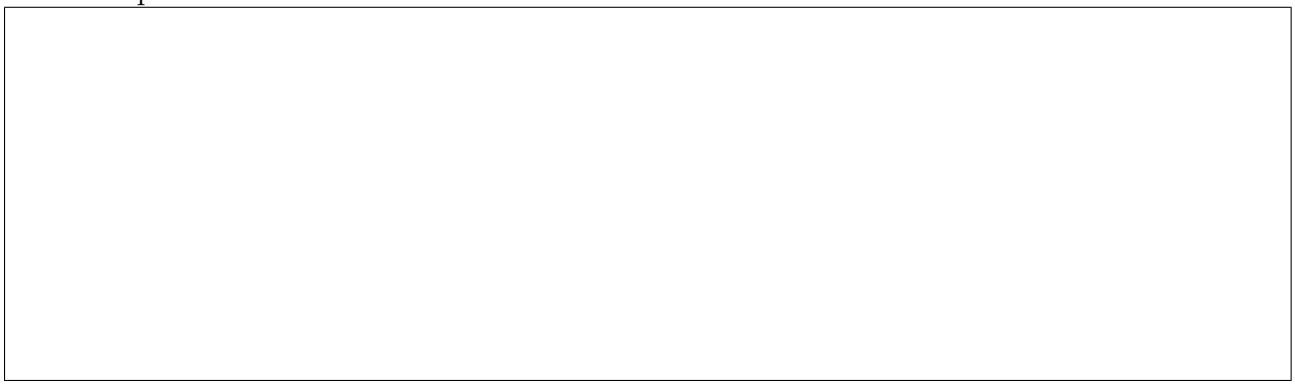
On peut également exprimer ce parcours sans récursivité, en utilisant une boucle et une pile pour stocker les sommets à visiter. Par exemple en C :

```
typedef struct tree {
  struct tree* g; // enfant gauche
  struct tree* d; // enfant droit
  int et; // etiquette
} tree_t;

void parcours(tree_t* a){
  pile_t* p = pile_vide();
  p->empiler(a);
  while (p->taille > 0){
    tree_t* t = p->depiler();
    printf("Traitement de l'étiquette %d\n", t->et);
    if (t->g != NULL){
      p->empiler(g);
    }
    if (t->d != NULL){
      p->empiler(d);
    }
  }
  liberer_pile(p);
}
```

Si l'on remplace la pile par une file, on obtient un parcours en **largeur**.

Exercice 3. Appliquer l'algorithme de parcours d'arbre sur le graphe G suivant, à partir du sommet 1, et donner les premières étapes de l'exécution en représentant l'état de la pile, jusqu'à atteindre un problème.



On se rend donc vite compte que contrairement aux arbres, les graphes peuvent contenir des cycles, et donc en appliquant naïvement le même algorithme de parcours, certains sommets sont

visités plusieurs fois. Pour un graphe non-orienté c'est même pire : les deux premiers sommets visités vont se rajouter l'un l'autre sur la pile à l'infini.

Pour gérer ce nouveau problème, il suffit de garder en mémoire au long du parcours quels sommets ont déjà été visités. Pour cela, on peut utiliser une structure d'ensemble, afin de pouvoir déterminer rapidement si un sommet a déjà été visité.

Algorithme 1 : Parcours_profondeur_source

```
   Entrée(s) : G = (S, A) graphe, s ∈ S sommet de départ
 1 P ← pile_vide();
 2 V ← ∅ // ensemble des sommets visités
 3 V.ajouter(s);
 4 P.empiler(s);
 5 tant que P non vide faire
 6 |   u ← P.depiler();
 7 |   Traiter u;
 8 |   pour v voisin de u faire
 9 |   |   si v n'est pas dans V alors
10 |   |   |   V.ajouter(s);
11 |   |   |   P.empiler(v);
```

Complexité On suppose que l'on a implémenté les ensembles de façon à avoir des tests d'appartenance en temps constant.
Un sommet ne peut être ajouté dans P que s'il n'a pas été visité, auquel cas on le marque comme visité. Ainsi, **chaque sommet n'est empilé, et donc dépilé, qu'une seule fois**. De plus, pour chaque sommet, lorsqu'on le dépile, on doit parcourir la liste de ses voisins. En utilisant une matrice d'adjacence, cela prendrait $\mathcal{O}(n)$ par sommet, soit un total de $\mathcal{O}(n^2)$. En utilisant une liste d'adjacence, lorsqu'on dépile un sommet $u \in S$, le dépiler prend un temps $\mathcal{O}(1)$, et parcourir la liste des voisins de u prend un temps $\mathcal{O}(\mathbf{deg}(u))$. Donc, la complexité totale est en :

$$\mathcal{O}(\sum_{u \in S}(1 + \mathbf{deg}(u))) = \mathcal{O}(n + m)$$

Notons qu'en remplaçant la pile par une file, on obtient un autre type de parcours, appelé **parcours en largeur**.

Exemple 9. Sur le graphe suivant, faisons un parcours en profondeur puis un parcours en largeur en partant du même sommet source, et notons les deux ordres de visite obtenus :

Sur les graphes non-orientés, les parcours sont fortement liés aux composantes connexes. En effet, un parcours ne peut pas sauter d'une composante connexe à une autre. En fait, un parcours permet de détecter intégralement une composante connexe.

Exemple 10. Écrivons un algorithme qui permet de classifier les sommets d'un graphe non-orienté selon leur composante connexe. Commençons par adapter l'algorithme de parcours précédent pour qu'il renvoie l'ensemble des sommets visités. Cela nous donnera un algorithme permettant de calculer l'ensemble des sommets contenus dans la composante connexe d'un sommet donné :

Algorithme 2 : Composante_connexe_source

Entrée(s) : $G = (S, A)$ graphe non-orienté, $s \in S$ sommet de départ
Sortie(s) : Liste des sommets dans la composante connexe de s

```
1  P ← pile_vide();
2  V ← {} // ensemble des sommets déjà vus lors du parcours
3  Ajouter s à V;
4  P.empiler(s);
5  tant que P non vide faire
6      u ← P.depiler();
7      pour v voisin de u faire
8          si v ∉ V alors
9              Ajouter v à V;
10             P.empiler(v);
11 retourner V
```

Ensuite, il suffit de lancer **Composante_connexe_source** depuis chaque sommet du graphe, en gardant en mémoire tous les sommets déjà visités, afin de ne pas relancer le parcours depuis un sommet dont on a déjà exploré la composante connexe :

Algorithme 3 : Composantes_connexes

Entrée(s) : $G = (S, A)$ graphe non-orienté
Sortie(s) : CC liste d'ensembles représentant les composantes connexes de G

```
1 V ← ∅ // ensemble des sommets déjà vus lors du parcours
2 CC ← [] // liste des composantes connexes
3 pour u ∈ S faire
4 |   si u ∈ V alors
5 |   |   Passer au sommet suivant;
6 |   C ← Liste_Composante_Connexe(G, u);
7 |   CC.ajouter(C);
8 |   V = V ∪ C;
9 retourner CC
```

Correction . Montrons que la routine Liste_Composante_Connexe renvoie bien la composante connexe du sommet donné en argument.

Soit $s \in S$. Montrons que pour tout $u \in S$, alors u et s sont dans la même composante connexe si et seulement si Liste_Composante_Connexe(G, u) contient s.

— Sens indirect : Montrons que "V et P ne contiennent que des sommets de la composante connexe de s" est un invariant de la boucle while de LCC. Avant le premier passage de boucle, V est vide et P ne contient que s : la propriété est vraie.

Supposons la propriété vraie au début d'un passage de boucle. Tous les sommets rajoutés à V et P lors du passage sont des voisins du sommet dépilé de P, qui est dans la composante connexe de s. Donc, la propriété reste vraie à la fin du passage.

En particulier, l'ensemble renvoyé par $LCC(G, s)$ ne contient que des éléments de la composante connexe de s.

— Sens direct : On suppose que u et s sont dans la même composante connexe. Il existe alors un chemin $s = u_0, u_1, \ldots, u_p = u$ dans G. Supposons par l'absurde que u_p n'est pas dans $LCC(s)$, on peut donc considérer $i \in [\![0, p]\!]$ minimal tel que u_i n'est pas dans $LCC(G, s)$. $s \in LCC(G, s)$, donc $i > 0$, et donc $u_{i-1} \in LCC(s)$. Donc, lors de l'exécution de l'algorithme, on a empilé u_{i-1}, et on l'a donc également dépilé. Or, (u_{i-1}, u_i) est une arête. Donc, au passage de boucle ou l'on a dépilé u_{i-1}, on a testé si u_i était déjà dans V, et si ce n'était pas le cas, on l'y a rajouté. Dans tous les cas, apres ce passage de boucle, u_i était dans V. Comme on n'enlève aucun élément de V, u_i fait partie de l'ensemble renvoyé par l'algorithme, i.e. de $LCC(G, s)$: c'est absurde.

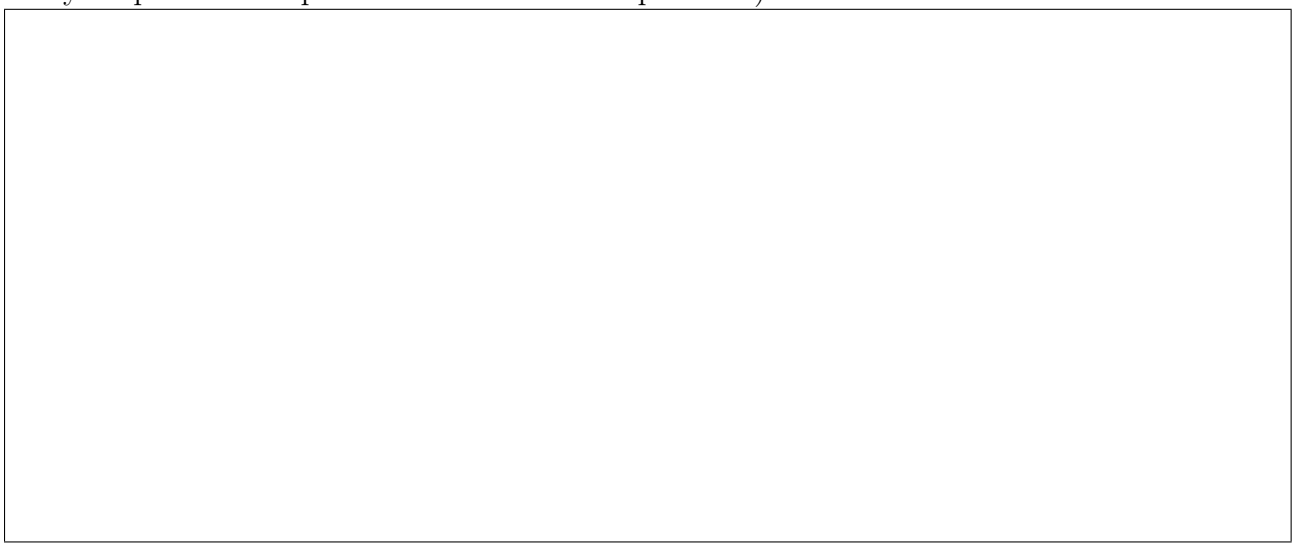
D'où l'équivalence : $LCC(G, s)$ est **exactement** la composante connexe contenant s.

Cet exemple nous montre un schéma récurrent pour construire des algorithmes de parcours de graphe :

— Une fonction auxiliaire faisant un parcours à partir d'un sommet source donné, et traitant tous les sommets accessibles à partir de cette source, en marquant les sommets visités au fur et à mesure ;

— La fonction principale, lançant la fonction auxiliaire sur chaque sommet n'ayant pas déjà été exploré, et combinant éventuellement les informations renvoyées à chaque appel.

Remarquons qu'un parcours de graphe construit en réalité des arbres. Par exemple, si l'on effectue un parcours en profondeur du graphe suivant, en marquant les arêtes utilisées (i.e. les arêtes ayant permis d'empiler un sommet lors du parcours) :



L'ensemble des arbres construit à partir d'un parcours s'appelle **l'arborescence** de ce parcours. On peut modifier le schéma de parcours vu précédemment pour calculer cette arborescence. Plus précisément, nous allons construire un dictionnaire **Pred** tel que pour tout sommet $u \in S$ exploré, **Pred**$[u]$ est le sommet à partir duquel u a été exploré, i.e. son parent dans l'arborescence. Les sommets sources, i.e. les sommets successifs choisis pour démarrer les parcours, n'auront pas de parent.

Algorithme 4 : Arborescence de parcours

Entrée(s) : $G = (S, A)$ graphe non-orienté
Sortie(s) : Pred tableau des prédécesseurs du parcours en profondeur de G

```
1  Pred ← dictionnaire vide // table des prédécesseurs
2  pour u ∈ S faire
3      si u n'est pas une clé de Pred alors
           // Parcours depuis la source u
4          P ← pile_vide();
5          P.empiler(s);
6          tant que P non vide faire
7              u ← P.depiler();
8              pour v voisin de u faire
9                  si v ∉ V alors
10                     Pred[v] = u;
11                     P.empiler(v);
12 retourner Pred
```

Définition 9. Les racines des arbres construits ainsi, autrement dit les sommets sources choisis successivement, s'appellent les **points de régénération** du parcours.

Exercice 4.

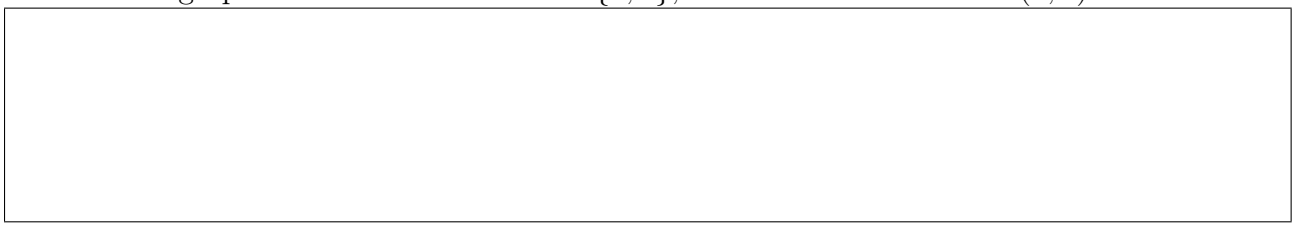
Question 1. Appliquer cet algorithme sur le graphe suivant, et dessiner l'arborescence du parcours :

Question 2. Sur le même graphe, appliquer une variante de l'algorithme précédent en faisant un parcours en largeur, et noter à nouveau l'arborescence du parcours.

On peut déjà remarquer que le parcours en largeur semble construire une arborescence donnant lieu à des chemins les plus courts possible entre la source et les autres sommets. Nous allons voir dans la partie suivante que c'est effectivement le cas.

D Parcours de graphe orienté

Les schémas de parcours en profondeur et en largeur présentés dans la partie précédente s'appliquent presque directement sur les graphes orientés. Commençons par faire un parcours en profondeur en suivant le même algorithme que précédemment, en choisissant les points de régénération dans l'ordre croissant.
On considère le graphe dont les sommets sont $\{1, 2\}$, avec comme seule arête $(2, 1)$:



On remarque donc qu'à cause de l'orientation des arcs, il devient possible de trouver, au cours d'un parcours à partir d'une source, un sommet ayant déjà été visité. Il suffit donc de fournir en entrée de nos algorithmes de parcours l'ensemble des sommets ayant déjà été visités, et de remplir cet ensemble à chaque parcours depuis un point de régénération.

Algorithme 5 : parcours_profondeur_source

Entrée(s) : $G = (S, A)$ graphe orienté, $s \in S$ sommet de départ, V ensemble des sommets déjà parcourus
Sortie(s) : Ajoute à V les sommets vus en parcourant le graphe depuis s

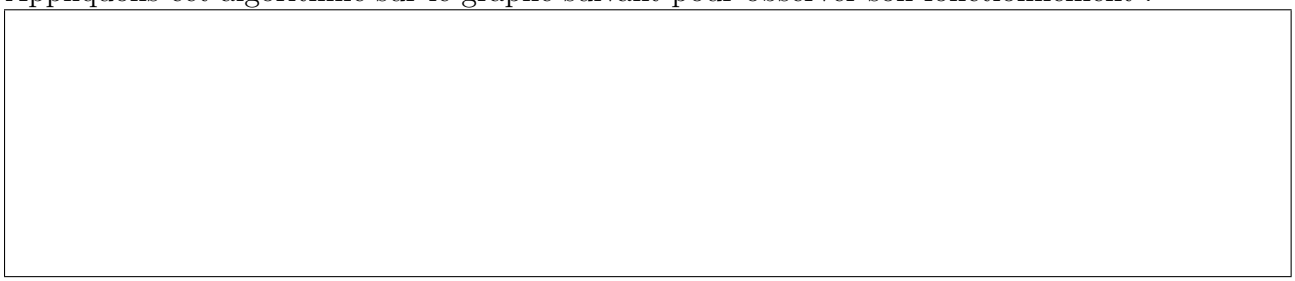
```
1  P ← pile_vide();
2  P.empiler(s);
3  Ajouter s à V;
4  tant que P non vide faire
5      u ← P.depiler();
6      Traiter u;
7      pour v voisin de u faire
8          si v ∉ V alors
9              Ajouter v à V;
10             P.empiler(v);
```

Algorithme 6 : parcours_profondeur

Entrée(s) : $G = (S, A)$ graphe orienté

```
1  V ← ∅ // ensemble des sommets déjà vus lors du parcours
2  pour u ∈ S faire
3      si u ∉ V alors
4          parcours_profondeur_source(G, u, V);
```

Appliquons cet algorithme sur le graphe suivant pour observer son fonctionnement :



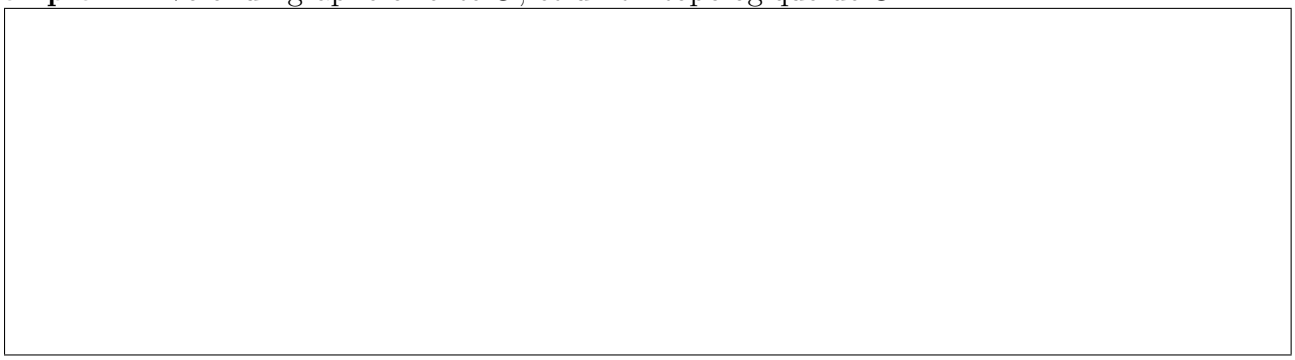
E Plus court chemin

Étant donné un graphe $G = (S, A)$ orienté, et $s, t \in S$, un plus court chemin de s à t dans G est un chemin de s à t de longueur minimale. La recherche de plus court chemin est un problème fondamental permettant de modéliser de nombreux problèmes de la vie réelle : itinéraire GPS, résolution d'un puzzle en le moins de coups possible, etc...
Les parcours en largeur permettent directement de calculer des plus courts chemins dans un graphe orienté (voir TD).

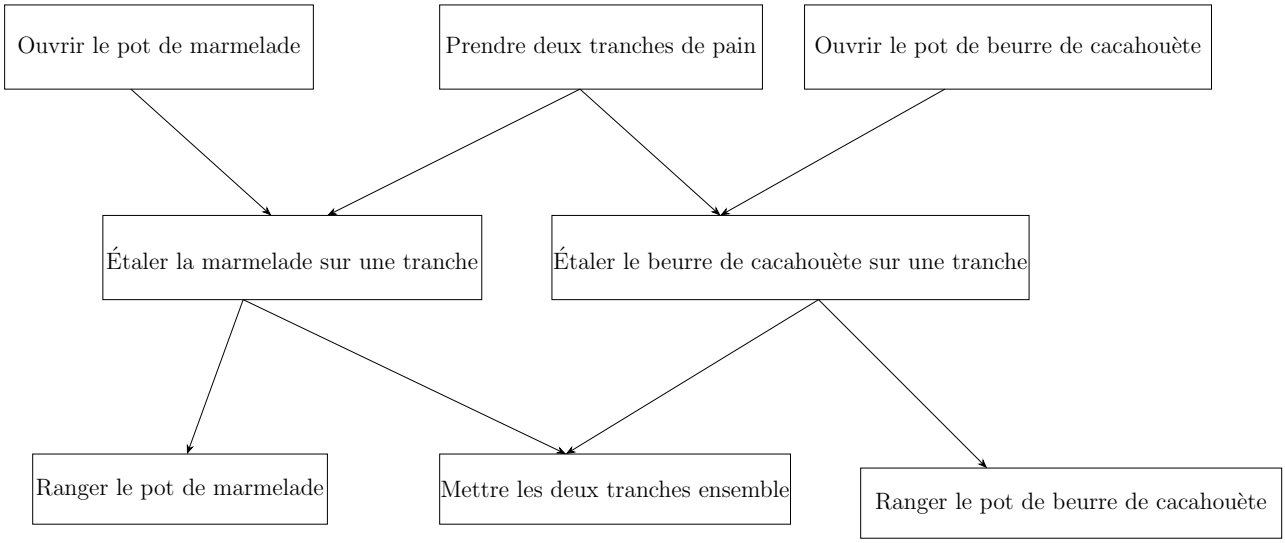
F Tri topologique

Définition 10. Soit $G = (S, A)$ un graphe orienté, notons $n = |S|$. Un tri topologique de G est une liste L contenant une et une seule fois chaque sommet de G, telle que pour tout arc $(u, v) \in A$, u apparaît avant v dans L.

Exemple 11. Voici un graphe orienté G, et un tri topologique de G :



Les graphes orientés acyliques apparaissent naturellement en informatique, et dans la vie de tous les jours. Ils permettent notamment de représenter un ensemble de tâches à réaliser, avec certaines tâches qui ne peuvent être commencées avant d'en avoir finies certaines. Par exemple, le graphe suivant modélise les différentes étapes pour fabriquer un sandwich marmelade et beurre de cacahouète (ce que les américains appellent un PBJ) :



Sur un tel diagramme, trouver un tri topologique donne un ordre dans lequel effectuer les taches tout en respectant les différentes dépendances. Un exemple plus conséquent est le logiciel Make, qui détermine l'ordre dans lequel effectuer les différentes étapes de compilation, ce qui peut être non-trivial pour des gros projets avec des centaines ou des milliers de fichiers source.

Au vu de l'importance de cette famille de graphes, on leur a donné un nom : les **DAG** (Directed Acyclic Graph). On a en fait le lien suivant très fort entre les DAG et les tris topologiques :

Proposition 4. Un graphe orienté admet un tri topologique si et seulement si il est acyclique.

Démonstration. Par double équivalence.

Sens direct Par l'absurde, soit $G = (S, A)$ un graphe contenant un cycle et admettant un tri topologique L. Il existe alors $u_0, u_1, \dots u_p \in S$ tels que $(u_i, u_i + 1) \in A$ pour $i \in [\![0, p-1]\!]$, et $u_0 = u_p$.

Pour $i \in [\![0, p]\!]$, notons k_i la position de u_i dans L. On a donc $k_0 < k_1 < \cdots < k_p < k_0$: c'est absurde.

Sens indirect On considère un graphe $G = (S, A)$ acyclique. Nous allons exhiber un algorithme permettant de calculer un tri topologique valide.

Le principe est de chercher à chaque étape un sommet sans prédécesseur (ce que l'on appellera une **source**), à l'ajouter à la liste, à le retirer du graphe, puis à recommencer sur les sommets restants :

Algorithme 7 : TT (Tri topologique)

Entrée(s) : $G = (S, A)$ DAG
Sortie(s) : L tri topologique de G

```
1 si G ne contient aucun sommet alors
2     retourner []
3 s ← sommet de G de degré entrant nul;
4 G' ← sous graphe de G sur S \ {s};
5 retourner s :: TT(G')
```

Pour montrer la correction de cet algorithme, il faut montrer :

Lemme 1.

Soit $G = (S, A)$ un DAG non vide. G admet une source.

Lemme 2.

Soit G est un DAG et s une source de G. En notant G' le sous graphe de G sur $S \setminus \{s\}$ et L' un tri topologique sur G', alors $[s] + L'$ est un tri topologique sur G

Preuve du Lemme 1 Supposons par l'absurde que G n'admet pas de source. Alors, tout sommet de G admet au moins un prédécesseur. Soit $u_0 \in S$ quelconque. Soit u_1 un prédécesseur de u_0. On construit ainsi une suite $(u_i)_{i \in \mathbb{N}}$ de sommets tels que (u_{i+1}, u_i) est une arête pour tout $i \in \mathbb{N}$. En notant $n = |S|$, et en considérant $u_0, \dots, u_n$, par principe des tiroirs à chaussettes, il existe $0 \leq i < j \leq n$ avec $u_i = u_j$. Donc, $u_j \to u_{j-1} \to \cdots \to u_i$ est un cycle dans G, ce qui est absurde.

Preuve du Lemme 2 Montrons que pour tout arc $(u, v) \in A$, u apparaît avant v dans $L = [s] + L'$.

— Si $u = s$, u est le premier élément de L, donc apparaît avant v

— v ne peut pas valoir s car s n'a pas de prédécesseur

— Si $u \neq s$ et $v \neq s$, alors (u, v) est une arête de G', et donc u apparaît avant v dans L', et donc dans L.

Ceci permet de conclure le sens indirect. □

Cependant, l'algorithme précédent n'est pas efficace, car il peut être très long de trouver un sommet de degré entrant nul, et encore plus long de construire le sous-graphe G'. Plus précisément, chaque appel récursif va prendre de l'ordre de $\mathcal{O}(n + m)$ pour reconstruire le graphe G. La complexité totale sera donc en $\mathcal{O}(n(n + m))$.
On peut trouver un algorithme plus efficace, en s'inspirant des parcours.

Nous allons construire et utiliser les structures suivantes :

— Une pile P dans laquelle on empile les sommets qui n'ont plus aucun prédécesseur, et qui peuvent donc être traités et ajoutés au tri

— Un dictionnaire d^- marquant le nombre de prédécesseurs restants pour chaque sommet

— Une liste L contenant le tri topologique en cours de construction. On y ajoute les sommets dans l'ordre.

L'algorithme maintiendra les invariants suivants :

1. Pour tout u dans S, $d^-[u]$ est le cardinal de $\{v \in S \mid (v, u) \in A$ et $v \notin L$
2. Pour tout u dans S, $d^-[u] = 0$ si et seulement si $u \in P$ ou $u \in L$

Algorithme 8 : Tri topologique

Entrée(s) : $G = (S, A)$ graphe orienté
Sortie(s) : L tri topologique de G

```
1  n ← |S|;
2  L ← liste vide;
3  P ← pile vide;
4  d⁻ ← dictionnaire vide ;
   // Initialiser d⁻
5  pour u sommet de G faire
6      d⁻[u] ← 0;
7  pour u sommet de G faire
8      pour v successeur de u faire
9          d⁻[v] = d⁻[v] + 1;
   // Empiler les sources
10 pour u sommet de G faire
11     si d⁻[u] = 0 alors
12         P.empiler(u);
   // Parcours
13 tant que P non vide faire
14     u ← P.depiler();
15     Ajouter u à la fin de L;
16     pour v voisin de u faire
17         d⁻[v] = d⁻[v] − 1;
18         si d⁻[v] = 0 alors
19             P.empiler(v);
20 retourner L
```

Terminaison Un sommet u ne peut être empilé sur P que si son degré entrant initial est 0, ou si au cours de l'exploration, une arête (v, u) permet de baisser $d^-[u]$ à 0. Ainsi, un sommet u est empilé au plus une fois sur P : soit avant la boucle while, soit lorsque l'algorithme emprunte la dernière arête vers ce sommet.

Complexité Notons que l'on ajoute les sommets à droite de la liste L

Donc, la boucle while finit, car on empile au plus n éléments sur P, et donc on peut passer au plus n fois dans la boucle. L'algorithme termine bien. De plus, à chaque passage, en notant u le sommet dépilé, on doit effectuer $\mathcal{O}(\mathbf{deg}(u))$ opérations (même argument que pour les parcours vus dans les parties d'avant), ce qui garantit une complexité de $\mathcal{O}(n+m)$ pour la boucle while. Les étapes d'initialisation de d^- prennent aussi $\mathcal{O}(n+m)$ car on parcourt chacune des listes d'adjacence.

Correction La correction peut se montrer à partir des invariants proposés plus haut (Voir TD).

4 Types de graphes

La théorie des graphes fait partie des domaines les plus actifs de la recherche en informatique. En conséquence, il y a de très nombreuses catégories de graphes, et tout autant de propriétés et problèmes à étudier sur les graphes. On présente ici quelques concepts récurrents sur les graphes.

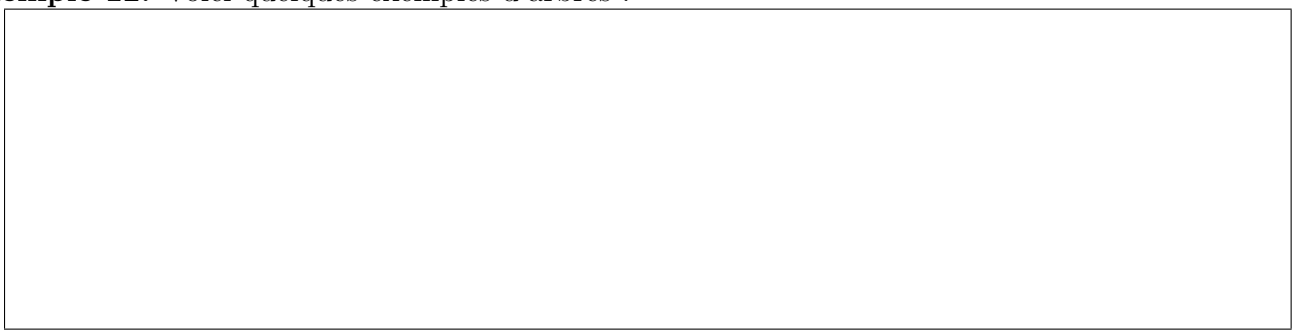
A Arbres

Définition 11. Un graphe est **acyclique** s'il ne contient aucun cycle. Un graphe non-orienté est un **arbre** s'il est connexe et acyclique.

Les graphes acycliques sont parfois appelés des forêts, car chacune de leurs composantes connexes est un arbre.

Cette notion d'arbre n'est pas exactement la même que celle des structures arborescentes étudiées plus tôt dans l'année. En effet, les arbres en tant que graphes ne sont pas des structures récursives, car il n'y a pas de sommet privilégié qui sert de racine à l'arbre.

Exemple 12. Voici quelques exemples d'arbres :



Proposition 5. Soit $G = (S, A)$ un graphe, avec $n = |S|$ et $m = |A|$. Alors :

1. Si G est connexe, alors $m \geq n - 1$.
2. Si G est acyclique, alors $m \leq n - 1$.

Les arbres sont donc des graphes connexes minimaux, et des graphes acycliques maximaux :

Proposition 6. Soit $G = (S, A)$ un graphe, avec $n = |S|$ et $m = |A|$. Les trois propriétés suivantes sont équivalentes :

(i) G est un arbre

(ii) G est connexe et $m = n - 1$.

(iii) G est acyclique et $m = n - 1$.

Exercice 5. Soit F une forêt à n sommets et m arêtes. Donner un lien entre n, m et p le nombre de composantes connexes de F.

B Graphe complet

Définition 12. Un graphe non-orienté $G = (S, A)$ est complet si $A = \{(s, t) \mid s, t \in S, s \neq t\}$, autrement dit si toutes les paires de sommets sont connectées.

On note généralement K_n le graphe complet d'ordre n, à n sommets. Voici K_3, K_4, K_5 :

Proposition 7. K_n possède $\frac{n(n-1)}{2}$ arêtes.

Définition 13. Soit $G = (S, A)$ un graphe non-orienté. Une clique de G est un ensemble de sommets $K \subseteq S$ tel que le sous-graphe de G induit par K est complet. Autrement dit, c'est un ensemble de sommets de G étant tous deux à deux reliés.

Exemple 13. Voici un graphe G, avec deux cliques entourées, une de taille 3, une de taille 5 :

Ainsi, un graphe complet est un graphe dont l'unique composante connexe est une clique.

Une clique représente une zone du graphe qui est particulièrement connectée. Un problème classique d'informatique est la recherche de cliques d'une taille donnée :

Problème 9 : CLIQUE

Entrée(s) : G un graphe, $k \in \mathbb{N}$
Sortie(s) : Existe-t'il une clique de taille k dans G ?

Ce problème est **NP**-complet, on ne connaît donc pas encore d'algorithme polynomial pour le résoudre.

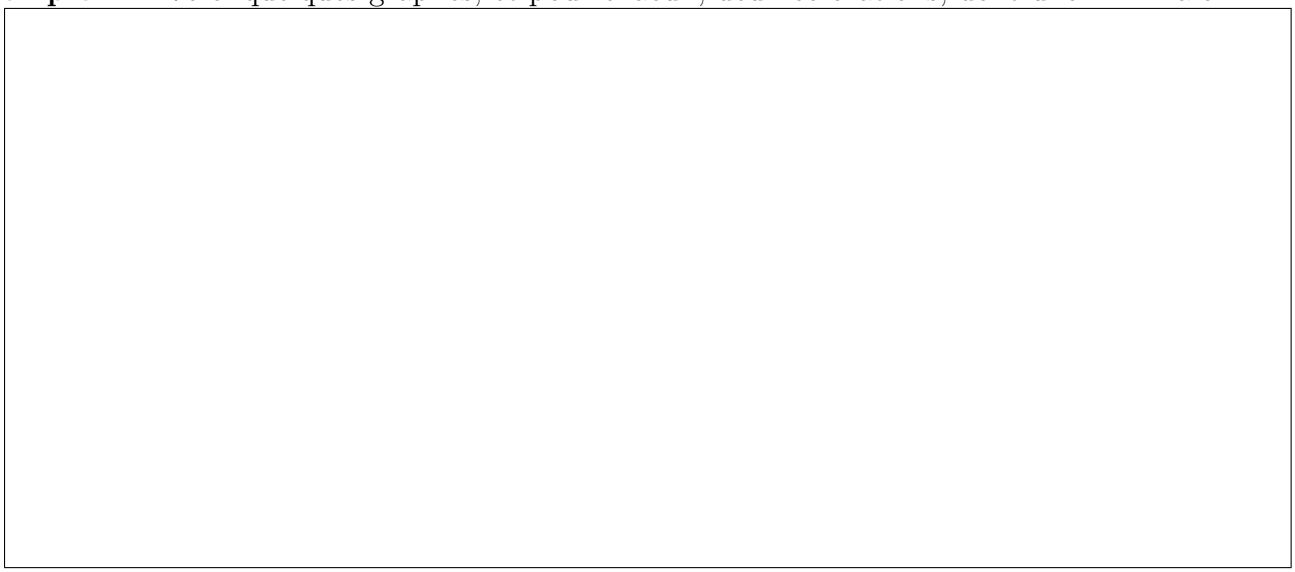
C Coloration de graphe

La coloration de graphe est une généralisation de la notion de graphe biparti. C'est une notion centrale en théorie des graphes, qui permet de modéliser de nombreux problèmes.

Définition 14. Soit $G = (S, A)$ un graphe non-orienté. Pour $k \in \mathbb{N}$, une k-coloration de G est une fonction $c : S \to [\![1, k]\!]$ tel que pour toute arête $(x, y) \in A$, $c(x) \neq c(y)$. Autrement dit, c'est une manière de colorer les sommets de G en utilisant k couleurs, de telle sorte que deux sommets adjacents ne sont jamais de la même couleur.

On dit que G est k-colorable s'il existe une k-coloration de G. On appelle **nombre chromatique** de G le plus petit entier k tel que G est k-coloriable. On le note $\chi(G)$.

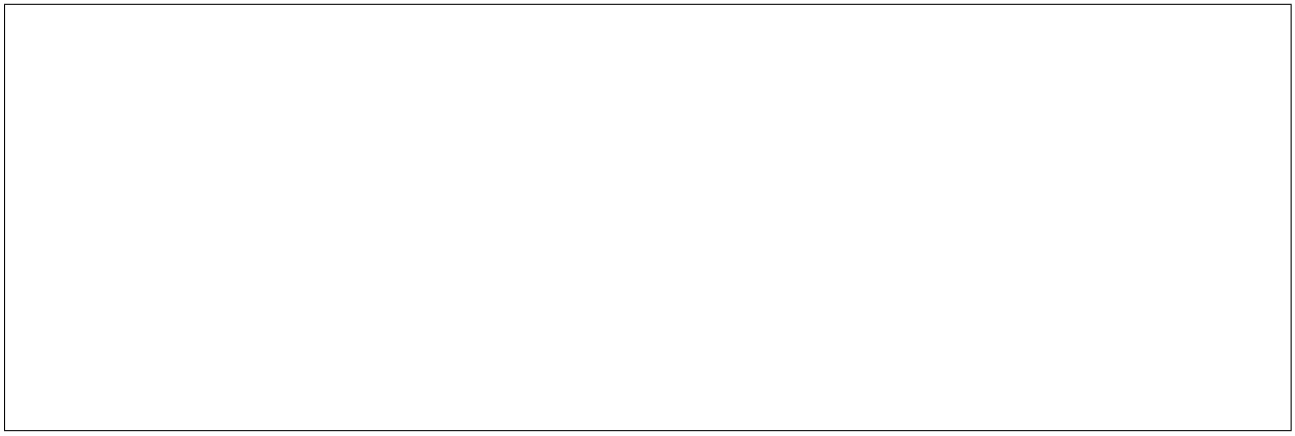
Exemple 14. Voici quelques graphes, et pour chacun, deux colorations, dont une minimale.



La coloration de graphe est une généralisation de la notion suivante :

Définition 15. Un graphe $G = (S, A)$ est **biparti** s'il existe une partition de S en deux ensembles $S = X \coprod Y$ tels que $A \subseteq X \times Y \cup Y \times X$. Autrement dit, G est biparti si l'on peut classifier les sommets en deux ensembles X et Y tels qu'aucune arête ne relie deux sommets de X ou deux sommets de Y.

Exercice 6. Déterminer lesquels des graphes suivants sont bipartis :



Proposition 8. Un graphe G est biparti si et seulement si il est 2-coloriable.

On dispose également d'un critère simple pour savoir si un graphe est biparti :

Proposition 9. Soit G un graphe non-orienté. G est biparti si et seulement si il ne contient pas de cycle impair, i.e. de cycle contenant un nombre impair de sommets.

Démonstration. Notons que G est biparti si et seulement si chacune de ses composantes connexes l'est, et que G ne contient pas de cycle impair si et seulement si aucune de ses composantes connexes ne contient de cycle impair. Il suffit donc de montrer le résultat annoncé sur les graphes connexes. On suppose donc dans la suite que G est connexe.
Commençons par le sens direct, que l'on montre par contraposée :

Pour le sens indirect, on exhibe un algorithme qui permet de construire une 2-coloration d'un graphe, et on montre qu'il est correct lorsque le graphe ne contient pas de cycle impair. L'algorithme consiste à marquer un sommet arbitraire avec 1, puis à marquer ses voisins avec 2, puis les voisins des voisins avec 1, etc... Autrement dit, on effectue un parcours de graphe, en marquant les sommets rencontrés par 1 et 2, en alternant. L'algorithme s'arrête lorsqu'il a visité tous les sommets, ou bien lorsqu'il a rencontré une contradiction.

□

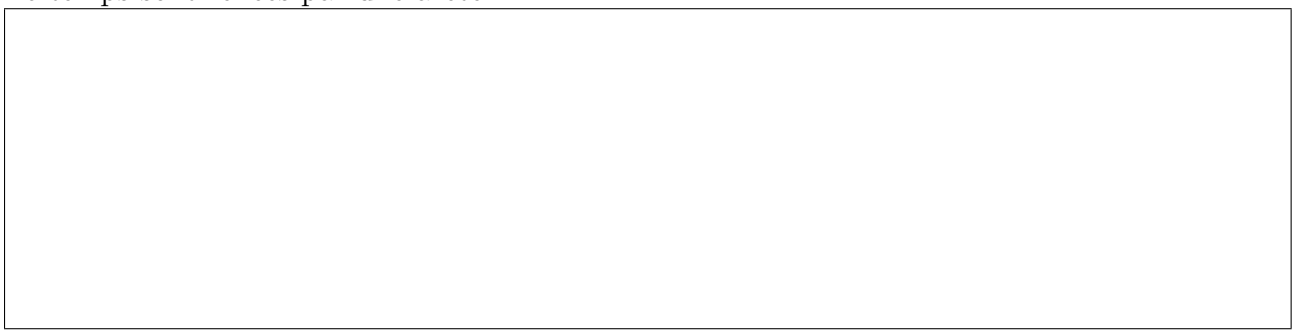
Application Intuitivement, la coloration de graphe correspond à une gestion de ressource. Les couleurs correspondent à des ressources à utiliser sur les sommets, et les arêtes du graphe représentent des contraintes, empêchant d'utiliser les mêmes ressources sur certains sommets.

Voyons un exemple. On considère un programme C. Pour compiler ce programme en langage machine, le compilateur doit déterminer où chaque variable doit être stockée en mémoire. Le but étant d'utiliser le moins d'emplacements mémoires possible. On appellera les emplacements $R_1, R_2, \ldots$

Par exemple, sur le programme suivant :

```
1 int main(){
2   int x = 3;
3   int y = x + 1;
4   int z = x + y;
5   int u = 7;
6   int v = y + 1;
7   u = u + y;
8   z = v + u;
9 }
```

Il suffit de 5 emplacements mémoires pour stocker les 5 variables : on stocke x, y, z, u, v dans les emplacements R_1, R_2, R_3, R_4, R_5 respectivement. On peut faire mieux : on peut stocker x et v dans le même emplacement mémoire car au moment où v apparaît pour la première fois, x ne sert plus à rien. On remarque donc que toutes les variables ne sont pas "vivantes" à tout instant du programme. Par exemple, la variable u n'est pas utilisée pour les trois premières instructions, et la variable x n'est pas utilisée pour les 4 dernières instructions. On peut donc assigner à chaque variable une date de naissance et une date de mort. On considère ensuite le graphe G dont les sommets sont les variables x, y, z, u, v, et où deux variables étant vivantes en même temps sont reliées par une arête :

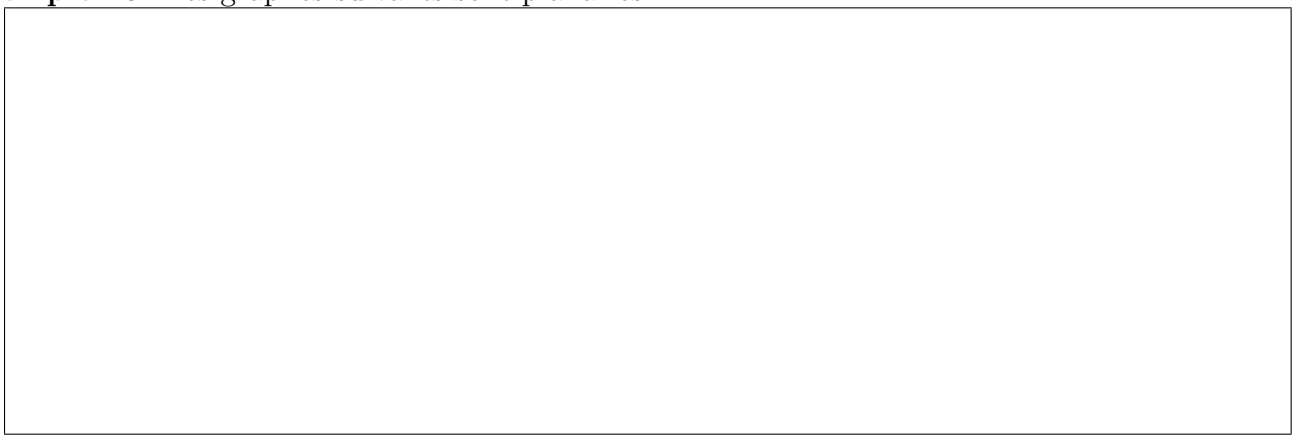


Sur ce graphe, une k coloration correspond précisément à assigner à chaque variable un emplacement mémoire. En effet, deux variables reliées dans le graphe sont vivantes en même temps, et donc ne peuvent pas être assignées au même emplacement. On remarque que le graphe précédent peut être coloré en utilisant 4 couleurs :

D Graphes planaires

Définition 16. Un graphe est **planaire** si l'on peut le représenter dans le plan sans que les arêtes se croisent.

Exemple 15. Les graphes suivants sont planaires :



En revanche, le graphe complet K_5 n'est pas planaire : il est **impossible** de le représenter dans le plan.

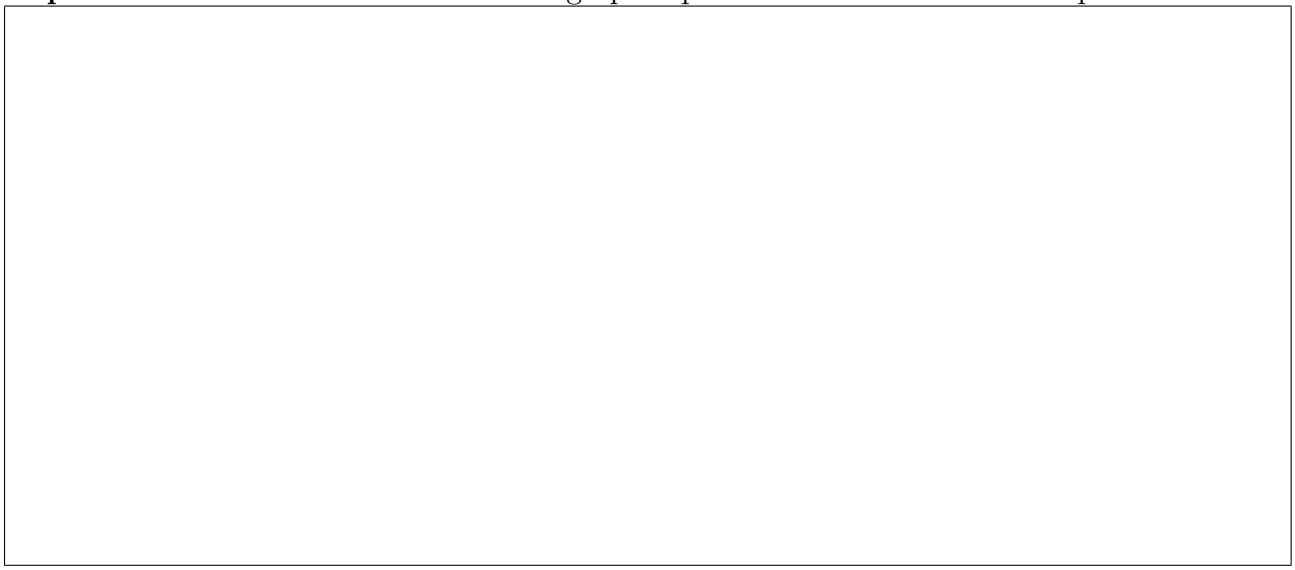
Exercice 7. Montrer que les graphes suivants sont planaires :

1. Le cube 3D
2. Le graphe complet K_4
3. Le graphe dont la matrice d'adjacence est $\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$

Remarque 6. Si l'on rajoute une arête entre les sommets 5 et 6 dans le dernier graphe de l'exercice précédent, on obtient un graphe qui n'est pas planaire : impossible de le dessiner dans le plan sans croisement.

Lorsque l'on dessine un graphe planaire sous une forme sans intersection, le graphe délimite plusieurs zones de l'espace appelées **faces**, y compris une face extérieure.

Exemple 16. Voici les différentes faces des graphes planaires donnés à l'exercice précédent :



Proposition 10 (Formule d'Euler)**.** Soit G un graphe planaire. Notons n son nombre de sommets, m son nombre d'arêtes et f son nombre de faces. Alors :

$$n - m + f = 2$$

Historiquement, les graphes planaires ont joué un rôle majeur en informatique, car ils ont été l'objet d'un des premiers théorèmes majeurs prouvé par ordinateur : le **théorème des quatre couleurs**.

Théorème 1. (*Hors programme*) Tout graphe planaire est 4-colorable.

Ce théorème implique par exemple que la carte du monde peut être coloriée en n'utilisant que 4 couleurs, sans que deux pays frontaliers n'aient la même couleur.

Dans l'article original prouvant ce théorème, les auteurs ont trouvé 1834 graphes "minimaux" et montré que tout graphe représentant un contre-exemple au théorème devait forcément pouvoir se réduire à un de ces 1834 graphes. Ils ont ensuite montré qu'aucun contre-exemple ne pouvait se réduire à un de ces graphes, montrant qu'aucun contre-exemple ne peut exister. Les 1834 graphes ont du être vérifiés, et les auteurs ont utilisé un programme informatique pour automatiser certaines parties de la vérification.

5 Plus court chemin

Dans la partie précédente, nous avons étudié des questions d'accessibilité, et on se demandait donc s'il existait un chemin entre deux sommets. Dans cette partie, on s'intéresse à la recherche de chemins les plus courts possibles. Ce problème a de nombreuses applications : calcul d'itinéraire dans Google Maps, résolution de jeux/puzzles, routage de paquets réseau...

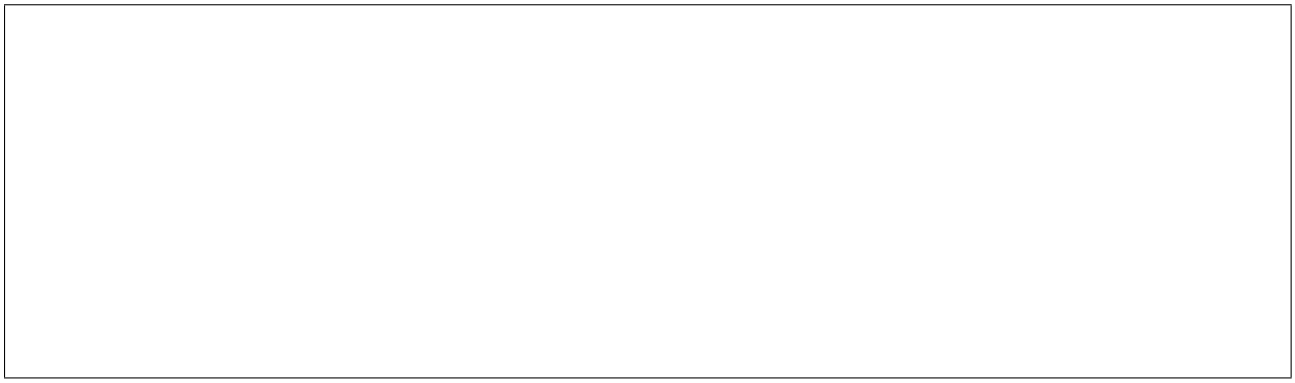
A Graphes pondérés

On étend notre définition des graphes pour permettre d'étiqueter les arêtes :

Définition 17. Un graphe pondéré est un triplet $G = (S, A, w)$ avec (S, A) un graphe et $w : A \to \mathbb{R}$ que l'on appelle pondération, ou fonction de poids.

Ce nouveau type de graphe, qui peut être orienté ou non, permet de représenter certaines situations plus fidèlement que les graphes, car chaque liaison entre deux sommets est pondérée.

Exemple 17. Un exemple de graphe pondéré :



Concrètement, une pondération peut correspondre :

— à un coût : $w(u, v)$ est le coût de passer de u à v

— à une capacité : $w(u, v)$ représente le débit maximal entre u et v

— à une mesure de similarité : $w(u, v)$ quantifie le lien entre u et v

— à tout autre type d'information imaginable...

Cette année, on considère le cas simple où la pondération correspond à un coût, c'est à dire à une distance. Par convention, lorsque u et v sont des sommets mais (u, v) n'est pas un arc, on posera $w(u, v) = +\infty$.

Représentation mémoire Pour représenter en mémoire un graphe pondéré, on peut adapter les matrices et les listes d'adjacences vues plus tôt. Pour les matrices d'adjacence, on peut considérer w comme une matrice, et stocker ses coefficients dans un tableau 2D. Pour les listes d'adjacence, plutôt que de stocker, pour un sommet u donné, la liste de ses voisins / successeurs, on stocke des couples (v, d), où v est un voisin et d le poids de l'arête (u, v).

Exemple 18. Voici le graphe précédent sous forme de matrice d'adjacence, puis de liste d'adjacence.

Définition 18. Soit $G = (S, A, w)$ un graphe pondéré et C un chemin dans G. Le poids de C est la somme des poids de ses arêtes.

Un **plus court chemin** entre deux sommets u et v de G est un chemin de poids minimal.

Il n'existe pas toujours de plus court chemin entre deux sommets. En effet, si le graphe contient des arêtes de poids négatifs, alors il se peut que le graphe contienne un cycle $u_0u_1 \dots u_{k-1}u_0$ de poids négatif. Alors, tout chemin qui passe par un sommet u_i de ce cycle peut être étendu comme suit : si on a comme chemin $v_0v_1 \dots v_{j-1}v_jv_{j+1} \dots v_l$ avec $v_j = u_i$, alors on peut considérer le chemin $v_0v_1 \dots v_{j-1}u_i \dots u_{k-1}u_0 \dots u_iv_{j+1} \dots v_l$ qui est plus long, mais de poids inférieur. En rajoutant à nouveau une itération du cycle, on peut encore réduire le poids du chemin, et ainsi de suite.

Les différents algorithmes de recherche de plus court chemin ne se comportent pas tous pareil en présence de poids négatifs. Certains algorithmes permettent de détecter les cycles négatifs (ex : l'algorithme de Bellman-Ford), mais certains ne sont même plus correct s'il y a une **arête** négative dans le graphe (ex : l'algorithme de Dijkstra).

B Première approche de programmation dynamique

On considère $G = (S, A, w)$ un graphe pondéré. On pose $S = \{s_1, \dots s_n\}$. On considère $C^t(i, j) =$ le poids du plus court chemin entre i et j utilisant au plus t arêtes (et $+\infty$ si aucun tel chemin n'existe). Alors, on peut trouver une formule de récurrence sur $C^t(i, j)$, comme suit :

— $C^0(i, j) = 0$ si $i = j$

— $C^0(i, j) = +\infty$ sinon

— $C^{t+1}(i, j) = \min_{k=1}^{n}(C^t(i, k) + w(k, j))$

en effet, il suffit de considérer, pour un plus court chemin entre i et j de longueur au plus $t + 1$, le dernier sommet k qui a pu être emprunté. Le meilleur moyen d'atteindre k depuis i et de suivre un plus court chemin de longueur au plus t, de poids $C(i, k, t)$, puis d'emprunter la dernière arête (k, j).

Remarque 7. En voyant w et les C^t comme des matrices, C^{t+1} est presque le produit matriciel de C^t et w. Il suffit de changer le min par un $\sum$ et le $+$ par un $\times$!

Cette formule sur les $C^t(i, j)$ est compatible avec la programmation dynamique : on peut utiliser des tableaux T^t pour stocker les $C^t(i, j)$. Il reste à déterminer pour quel indice t maximal il faut calculer C^t.

Proposition 11. Si G ne contient aucun cycle négatif, alors tout plus court chemin entre deux sommets u et v emprunte au plus $n - 1$ arêtes.

En effet, si un chemin emprunte n ou plus arêtes, alors il passe deux fois par le même sommet, et contient donc un cycle : en enlevant ce cycle on obtiendrait un chemin strictement plus court.

On peut donc calculer les valeurs successives de $T^0, T^1, \dots, T^{n-1}$. A la fin, T^{n-1} contient la longueur des plus courts chemins entre tout couple de sommets. Pour l'ordre de calcul, il suffit de calculer tout T^t, dans n'importe quel ordre, avant de calculer T^{t+1}. En effet, chaque case $T^{t+1}[i, j]$ se calcule uniquement à partir de valeurs dans T^t.

Voici le pseudo-code correspondant : l'algorithme renverra une matrice D telle que D_{ij} est la distance entre s_i et s_j

Algorithme 10 : PCC : Prog. Dyn.

Entrée(s) : $G = (S, A, w)$ graphe pondéré
Sortie(s) : D matrice des distances entre sommets

$T^0, T^1 \dots T^n \leftarrow$ tableaux de taille $n \times n$;
// Cas de base
pour $i = 1$ **à** n **faire**
  $T^0[i, i] \leftarrow 0$;
  pour $j \neq i$ **faire**
    $T^0[i, j] \leftarrow +\infty$;
// Remplissage des tableaux
pour $t = 1$ **à** $n - 1$ **faire**
  pour $i = 1$ **à** n **faire**
    pour $j = 1$ **à** n **faire**
      $T^t[i, j] \leftarrow \min_{k=1}^{n}(T^t[i, k] + w(k, j))$;
retourner T^{n-1}

La complexité temporelle de cet algorithme est $\mathcal{O}(n^4)$: on a trois boucles for imbriquées de taille n ou $n - 1$, et chaque passage demande de calculer un minimum sur n éléments.

On peut également s'intéresser à l'espace de stockage nécessaire pour exécuter l'algorithme. Ici, il faut un espace $\mathcal{O}(n^3)$ pour stocker les n tableaux de taille $n \times n$. En fait, on peut s'en sortir avec un espace seulement en $\mathcal{O}(n^2)$. En effet, une fois que l'on a calculé T^t, il ne sert plus à rien de garder en mémoire T^{t-1}. On peut donc utiliser deux tableaux et utiliser l'un pour stocker les valeurs actuelles de T^t, l'autre comme copie de T^{t-1}.

Algorithme 11 : PCC : Prog. Dyn.

Entrée(s) : $G = (S, A, w)$ graphe pondéré
Sortie(s) : D matrice des distances entre sommets

$T, T_c \leftarrow$ tableaux de taille $n \times n$// T_c servira de copie temporaire
// Cas de base
pour $i = 1$ **à** n **faire**
  $T[i, i] \leftarrow 0$;
  pour $j \neq i$ **faire**
    $T[i, j] \leftarrow +\infty$;
// Remplissage des tableaux
pour $t = 1$ **à** $n - 1$ **faire**
  $T_c \leftarrow$ copie de T;
  pour $i = 1$ **à** n **faire**
    pour $j = 1$ **à** n **faire**
      $T[i, j] \leftarrow \min_{k=1}^{n}(T_c[i, k] + w(k, j))$;
retourner T

Remarque 8. Une fois que l'on a calculé T^{n-1}, si le graphe ne contient pas de cycle négatif, alors on a calculé la matrice exacte des distances des plus courts chemins, et ainsi toutes les matrices $T^{n-1}, T^n, T^{n+1}, \dots$ seront identiques. Cependant, si le graphe contient un cycle négatif, alors T^n contiendra une case strictement inférieure à T^{n-1}. On peut donc modifier l'algorithme et calculer un rang de plus de la matrice, ce qui permet de **détecter les cycles négatifs** du graphe en entrée.

La complexité de cet algorithme de programmation dynamique, en $\mathcal{O}(n^4)$, n'est pas satisfaisante. Nous allons voir étudier deux algorithmes classiques de calcul de plus court chemin :

— L'algorithme de Floyd-Warshall permet de calculer en $O(n^3)$ le tableau des distances donné par l'algorithme précédent. Il peut aussi détecter les cycles négatifs. Il fonctionne selon un principe de programmation dynamique très proche de l'algorithme précédent, et est donc également adapté aux matrices d'adjacence.

— L'algorithme de Dijkstra permet de calculer la **distance d'un sommet source à tous les autres sommets** du graphe. Il **nécessite que les arêtes soient positives**, et fonctionne selon un principe proche du parcours en largeur. Cet algorithme utilise une **file de priorité**. Sa complexité dépend donc de l'implémentation utilisée. Avec une implémentation naïve, la complexité atteinte est $\mathcal{O}(m + n^2) = \mathcal{O}(n^2)$. La meilleure implémentation atteint une complexité $\mathcal{O}(m + n \log n)$.

C Algorithme de Floyd-Warshall

Soit $G = (S, A, w)$ un graphe orienté pondéré, avec $S = \{s_0, \dots, s_{n-1}\}$. **On suppose que G ne contient pas de cycle strictement négatif**. L'algorithme de Floyd-Warshall consiste à calculer $C^k(i, j)$ la plus courte distance entre s_i et s_j en ne passant que par les sommets intermédiaires $\{s_0, s_1, \dots, s_{k-1}\}$ (les sommets s_i et s_j ne sont pas comptés comme des sommets intermédiaires). En effet, pour calculer $C^{k+1}(i, j)$, il y a deux cas à considérer :

— Le meilleur chemin entre s_i et s_j ne passant que par les sommets s_0 à s_k n'utilise pas s_k. Alors, il est de longueur $C^k(i, j)$

— Le meilleur chemin entre s_i et s_j ne passant que par les sommets s_0 à s_k passe par s_k. Alors, ce chemin va de s_i à s_k puis de s_k à s_j, et chacun des sous-chemins ne passe que par des sommets intermédiaires parmi $s_0, \dots, s_{k-1}$. Donc, la longueur de ce chemin est $C^k(i, k) + C^k(k, j)$

Ainsi, on a la relation de récurrence suivante sur C :

— Pour $i, j \in [\![0, n-1]\!], C^0(i, j) = w(i, j)$

— Pour $i, j, k \in [\![0, n-1]\!]$, $C^{k+1}(i, j) = \min(C^k(i, j), C^k(i, k) + C^k(k, j))$

De plus, $C^n(i, j)$ donne la longueur du plus court chemin entre i et j pouvant emprunter des sommets parmi $\{s_0, \dots s_{n-1}\}$, i.e. parmi S tout entier : c'est la longueur du plus court chemin de i à j.

On stocke les valeurs de $C^k(i, j)$ dans $n + 1$ tableaux $T^0, \dots T^n$. Contrairement à la formule utilisée dans le premier algorithme de programmation dynamique, le calcul d'une case du tableau prend maintenant un temps $\mathcal{O}(1)$:

Algorithme 12 : PCC : Floyd-Warshall

Entrée(s) : $G = (S, A, w)$ graphe pondéré
Sortie(s) : D matrice des distances entre sommets

```
1 T^0, T^1 ... T^n ← tableaux de taille n × n;
  // Cas de base
2 pour i = 0 à n − 1 faire
3     pour j = 0 à n − 1 faire
4         T^0[i, j] ← w(i, j);
  // Remplissage des tableaux
5 pour k = 0 à n − 1 faire
6     pour i = 0 à n − 1 faire
7         pour j = 0 à n − 1 faire
8             T^{k+1}[i, j] ← min(T^k[i, j], T^k[i, k] + T^k[k, j]);
9 retourner T^k
```

La complexité temporelle de cet algorithme est en $\mathcal{O}(n^3)$, et la complexité spatiale aussi. Comme pour la programmation dynamique précédente, on peut réduire la complexité spatiale à du $\mathcal{O}(n^2)$ en ne gardant pas en mémoire tous les T^k mais seulement le dernier calculé.

Exercice 8. Expliquer comment modifier l'algorithme de Floyd-Warshall pour pouvoir reconstruire les plus courts chemins entre les différentes paires de sommets.

D Algorithme de Dijkstra

L'algorithme de Dijkstra est une modification du parcours en largeur, permettant de prendre en compte que les arêtes sont pondérées. Reprenons le principe du parcours en largeur. On considère un graphe $G = (S, A)$ et $s \in S$ un sommet de départ du parcours. On considère s et ses voisins. Un voisin u de s est à distance exactement 1 de s. Une fois que l'on a visité tous les voisins de s, les prochains sommets à explorer sont les voisins des voisins de s, qui sont exactement à distance 2 de s, et ainsi de suite.

Considérons maintenant un graphe $G = (S, A, w)$ pondéré, avec w à valeurs dans R^+, et $s \in S$ un sommet de G. Considérons les voisins de s. Parmis eux, on considère le sommet u tel que $w(s, u)$ est minimal. Alors, on est certain que le chemin $s \to u$ est un plus court chemin. En effet, tout autre chemin doit emprunter un autre voisin de s, et est donc au moins plus long. De plus, u est le sommet le plus proche de s du graphe, à part s lui-même.

Le principe de l'algorithme de Dijkstra est d'identifier un à un les sommets les plus proches de s, comme dans un parcours en largeur. Pour cela, on maintient les structures suivantes :

- Un dictionnaire d associant à chaque sommet la distance du plus court chemin de s à d trouvé pour l'instant.
- Un dictionnaire **Pred** associant à chaque sommet son prédécesseur dans le plus court chemin trouvé pour l'instant.
- Une structure Q stockant tous les sommets pour lesquels un plus court chemin n'a pas encore été trouvé.

Pour $u \in S$, notons $\delta(u)$ la distance de s à u. Le code maintiendra les invariants suivants :

(i) $\forall u \in S \setminus \{s\}$, si u est une clé de **Pred** alors **Pred**$[u]$ est le prédécesseur de u sur un chemin de s à u de longueur $d[u]$. En particulier, $d[u] \geq \delta(u)$.

(ii) $\forall u \in S \setminus Q$, $d[u] = \delta(u)$

(iii) $\forall u \in Q$, $d[u]$ est la longueur minimale d'un chemin de s à u n'utilisant que des arêtes partant de sommets de $S \setminus Q$, ou bien $+\infty$ si aucun tel chemin n'existe.

Regardons le pseudo-code de l'algorithme.

Algorithme 13 : PCC : Dijkstra

Entrée(s) : $G = (S, A, w)$ graphe pondéré à n sommets, $s \in S$
Sortie(s) : **d** tableau des distances depuis s, et **Pred** tableau des prédecesseurs

```
1  d ← dictionnaire avec S comme clés, et ∞ pour toutes les valeurs;
2  Pred ← dictionnaire vide;
3  d[s] = 0;
4  Q ← ensemble contenant chaque élément de S;
5  tant que Q non vide faire
6      u ← extraire sommet de Q avec d[u] minimal;
7      pour v voisin de u faire
8          si d[u] + w(u, v) < d[v] alors
9              d[v] ← d[u] + w(u, v);
10             Pred[v] ← u;
11 retourner d, Pred
```

Terminaison La boucle tant que de l'algorithme s'exécute au plus n fois, car à chaque passage on extrait un élément de Q, qui contient chaque sommet au début de l'exécution. Autrement dit, $|Q|$ est un variant de boucle assurant la terminaison.

Complexité Q est en réalité une file de priorité, où la priorité d'un élément u est $\mathbf{d}[u]$ (plus cette quantité est faible, plus l'élément est prioritaire). On utilise trois opérations pour cette file de priorité : création d'une file avec n éléments, extraction de l'élément minimal, et modification de la priorité. notons respectivement $A(p), E(p)$ et $M(p)$ le coût de ces opérations sur une file à p éléments.

Alors, le coût total est $\mathcal{O}(nA(n)+nE(n)+mM(n))$. Le terme $nA(n)$ correspond à la création de la file de priorité, le terme $nE(n)$ à l'extraction du min pour chaque étape de l'algorithme, et $mM(n)$ à la modification de la case $d[v]$ ligne 9 de l'algorithme, qui change a priori l'ordre de priorité dans Q. Cette ligne ne s'exécute que m fois car pour chaque sommet, on n'utilise qu'une seule fois chaque arête qui en sort au cours du parcours.

Une implémentation naïve de cette file de priorité utilise un tableau de booléen, indiquant quels sommets font partie de la file. Pour extraire l'élément le plus prioritaire, il faut parcourir l'ensemble des sommets, et chercher celui de distance minimale parmi ceux encore présents dans la file. Le coût d'extraction est donc linéaire : $E(n) = \mathcal{O}(n)$. La construction initiale de la structure coûte aussi $\mathcal{O}(n)$ au total, et la modification de priorité consiste simplement à modifier la valeur du tableau d, sans toucher à Q, donc $M(n) = \mathcal{O}(1)$. La complexité totale est donc $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$.

Si l'on utilise un tas binaire (Cf. chapitre sur les arbres), alors $A(n)$, $M(n)$ et $E(n)$ sont en $\mathcal{O}(\log n)$. On obtient donc une complexité $\mathcal{O}(n \log n + m \log n) = \mathcal{O}((n + m) \log n)$.

L'implémentation des files de priorité par tas de Fibonacci est assez complexe, mais permet les complexités amorties suivantes : $E(n) = \mathcal{O}(\log n)$, $A(n) = \mathcal{O}(1) = M(n)$. On trouve donc une complexité $\mathcal{O}(n \log n + m)$.

En pratique, les tas binaires fonctionnent assez bien, car la constante du $\mathcal{O}$ de l'implémentation par tas de Fibonacci est assez large, et ne compense pas le facteur $\log n$ gagné pour des valeurs raisonnables de n.

Correction : La correction de cet algorithme vient des trois invariants donnés plus haut, ainsi que du lemme suivant :

Lemme 3. Soit $G = (S, A, w)$ un graphe et $s \in S$. Si $E \subseteq S$ est un ensemble ne contenant pas s, que $u \in E$ et $C : s \to u$ est un chemin, alors C emprunte forcément une arête traversant la frontière de E, autrement dit il existe deux sommets v_1, v_2 tels que $v_1 \in S \setminus E, v_2 \in E$ et tels que C contient l'arête (v_1, v_2).

Démonstration. On note $(s = u_0 u_1 \dots u_k = u) = C$ le chemin considéré. Alors, l'ensemble $\{i \in [\![0, k]\!] \mid u_i \in E\}$ n'est pas vide car il contient k. On peut donc considérer son minimum i_0, et remarquer que $i_0 \neq 0$ car $u_0 = s \notin E$. Alors, (u_{i_0-1}, u_{i_0}) est une arête convenable car $u_{i_0-1} \notin E$ et $u_{i_0} \in E$. □