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)

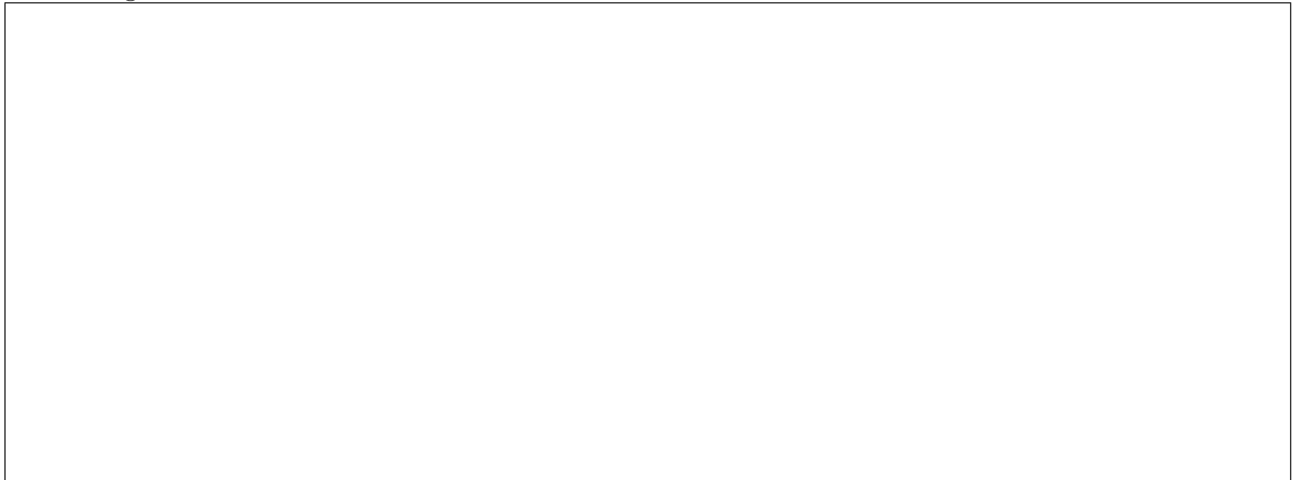
— Un réseau social type Instagram, où une personne peut en suivre une autre (relation asymétrique)

— 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**.
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 note $\mathbf{deg}(x) = |\mathcal{V}(x)|$ le nombre de voisins de x, on l'appelle **degré** de x.

Une arête $\{x\}$ reliant un sommet $x \in S$ à lui-même est appelée une **boucle**.

Exercice 1

En notant $G = (S, A)$ le graphe non-orienté des villes donné plus haut, on a :

— $S = \{$ 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)$\}$

Expliciter le voisinage de Toulouse, et donner son degré. Le graphe comporte-t-il une boucle ?

Là où les graphes non-orientés représentent des relations binaires symétriques, les graphes orientés correspondront aux relations binaires quelconques, pas nécessairement symétriques. Dans un graphe orienté, on relie les sommets non pas par des segments mais par des flèches indiquant le sens de la relation. Par exemple, le graphe suivant représente une généralisation du pierre-feuille-ciseau, issue des jeux Pokémon : on ajoute une flèche entre deux éléments pour indiquer que le premier bat 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.

Exercice 2

Le graphe des efficacités donné plus haut est donc le graphe $G = (S, A)$ orienté suivant :

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

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

Donner le degré entrant et sortant de l'élément Plante.

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. On note $n = |S|, m = |A|$. Alors :

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

Dans la suite, on suppose que les graphes sont sans-boucles, sauf si précisé autrement.

Définition 3

Soit $d \in \mathbb{N}$. Un graphe est dit d-régulier si tous ses sommets sont de même degré d.

Exercice 3

Q1. Pour chaque couple d, n suivant, dessiner un graphe d-régulier de taille n :

a) $d = 2, n = 5$ b) $d = 2, n = 4$ c) $d = 3, n = 4$ d) $d = 3, n = 6$
e) $d = 3, n = 8$ f) $d = 4, n = 5$ g) $d = 4$, $n \geq 6$ au choix

Q2. Soit G un graphe régulier, de degré d, avec n sommets et m arêtes. Donner un lien entre n, m, d, et le démontrer.

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 : Pour $G = (S, A)$ un graphe non-orienté, 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.

Une première approche peut être d'imiter la façon dont on manipule les graphes pour l'instant : en spécifiant l'ensemble des sommets et l'ensemble des arêtes/arcs. Pour $G = (S, A)$ un graphe orienté, avec $S = [\![0, n-1]\!]$, on peut représenter un graphe par une **liste de couples** donnant ses arêtes. Par exemple :

Lorsque l'on étudie une manière de représenter les graphes, on s'intéresse d'une part au coût mémoire de stockage, et d'autre part au coût temporel de différentes opérations. On peut considérer deux opérations élémentaires :

— `est_arete`(u, v) : savoir si deux sommets $u, v \in S$ sont voisins, i.e. savoir si $(u, v) \in A$;

— `liste_voisins`(u) : calculer la liste des successeurs (ou voisins en non-orienté) d'un sommet $u \in S$.

Avec la représentation naïve de liste d'arêtes, ces deux opérations seront en $\mathcal{O}(m)$, avec $m = |A|$, car elles nécessitent de parcourir toute la liste des arêtes.

Exercice 4

Quelle structure de données pourrait-on utiliser pour améliorer la complexité de `est_arete` sans impacter négativement `liste_voisins` ?

Les deux manières efficaces 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 1

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.

```c
typedef struct graph {
  int n; // taille du graphe
  bool** m_adj; // matrice d'adjacence: m_adj[i][j] = 1 si i -> j
} graph_t

/* Renvoie true si (i, j) est une arete de g */
bool est_arete(graph_t* g, int i, int j){
  return g->m_adj[i][j];
}

/* Renvoie un tableau d'entiers positifs contenant les voisins
   de i dans g. Stocke dans *cnt la taille du résultat */
int* liste_voisins(graph_t* g, int i, int* cnt){
  // compter les voisins
  *cnt = 0;
  for (int j = 0; j < g->n; j++){
    *cnt += g->m_adj[i][j];
  }
  int* res = malloc((*cnt)*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++;
    }
  }
  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 pour les compter au fur et à mesure qu'on les traite.

B Listes d'adjacence

On considère un graphe non orienté $G = (S, A)$, avec $S = \{0, \dots, n-1\}$. 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 2

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* degres; // degres[i] contient le degré du sommet i
  int** voisins; // voisins[i] contient les voisins de i
} graph_t;

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

int* liste_voisins(graph_t* g, int i, int* cnt){
  *cnt = g->degres[i]; // nombre de voisins
  int* res = malloc((*cnt)*sizeof(int));
  for (int k = 0; k < cnt; k++){
    res[k] = g->voisins[i][k];
  }
  return res;
}
```

Notons que dans cette version de `liste_voisins`, on pourrait aussi renvoyer directement `g->voisins[i]`, en $\mathcal{O}(1)$, mais on obtiendrait un pointeur vers une liste interne à `g`, que l'on a pas le droit de modifier.

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 internet...), 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 négligeable devant n. La représentation par listes d'adjacence devient alors bien plus efficace. La plupart des algorithmes que nous allons voir en cours utilisent les listes d'adjacences pour explorer le graphe de proche en proche.

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 n'emprunte pas deux fois le même arc (ou la même arête en non-orienté).

Exercice 5

Un chemin élémentaire peut-il contenir un cycle ?

Exemple 3

Considérons le graphe suivant :

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... Parmi eux, seul le chemin 1-2-6 est élémentaire.

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 4

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 :

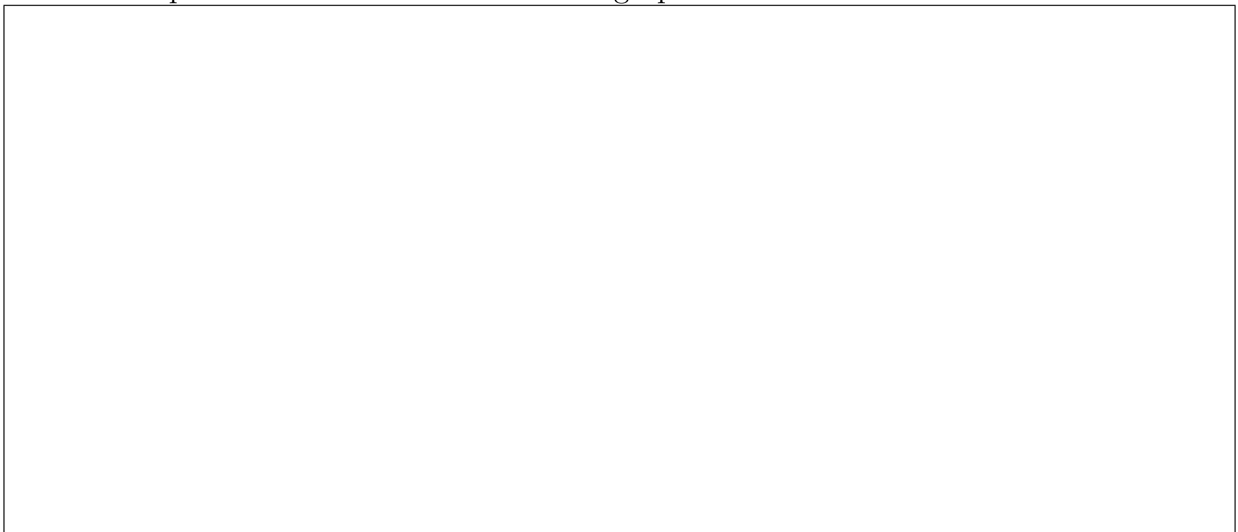
Définition 8

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 5

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 6

Q1. 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 fortement connexes de G ?

Exercice 7

Soit $G = (S, A)$ un graphe orienté. On définit le graphe des composantes de G, noté $C(G)$ comme suit :

— L'ensemble des sommets de $C(G)$ est l'ensemble des composantes connexes de G

— Il y a une arête entre deux sommets C_1, C_2 de $C(G)$ s'il existe deux sommets s_1, s_2 de G tels que $s_1 \in C_1, s_2 \in C_2$ et $s_1 \to s_2$.

Montrer que $C(G)$ ne contient pas de cycle.

C Parcours de graphe non orienté

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

```
1 let rec parcours (a: arbre) = match a with
2   | F(x) -> (* opération sur x *)
3   | N(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 :

```
1  typedef struct tree {
2    struct tree* g; // enfant gauche
3    struct tree* d; // enfant droit
4    int et; // etiquette
5  } tree_t;
6
7  void parcours(tree_t* a){
8    pile_t* p = pile_vide();
9    p->empiler(a);
10   while (p->taille > 0){
11     tree_t* t = p->depiler();
12     printf("Traitement de l'étiquette %d\n", t->et);
13     if (t->g != NULL){
14       p->empiler(g);
15     }
16     if (t->d != NULL){
17       p->empiler(d);
18     }
19   }
20   liberer_pile(p);
21 }
```

En remplaçant la pile par une file, on obtient un parcours en **largeur**.

Si l'on essaie d'appliquer tels quels ces algorithmes sur les graphes, on se vite compte qu'à cause des potentiels circuits/cycles, les parcours vont visiter plusieurs fois certains sommets, tourner à l'infini, etc...

On rajoute donc à nos parcours une structure d'**ensemble**, pour garder en mémoire tout au long du parcours quels sommets ont déjà été visités. Voici la version itérative du parcours de graphe à partir d'un sommet de départ :

Algorithme 1 : Parcours_profondeur_source

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

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

Implémentation On considère des graphes dans lesquels l'ensemble des sommets est un intervalle $[\![0, n-1]\!]$. On peut implémenter l'ensemble des sommets vus par un simple tableau de booléens `vu: bool array` : la case `vu.(i)` indique si le sommet i a déjà été visité. De plus, pour les piles, nous allons utiliser le module `Stack` d'OCaml, qui implémente des piles mutables :

```
1  type 'a Stack.t (* pile contenant des 'a *)
2
3  (* Stack.create () renvoie une pile vide *)
4  Stack.create : unit -> Stack.t
5
6  (* Stack.is_empty p renvoie true si p est vide, false sinon *)
7  Stack.is_empty : 'a Stack.t -> bool
8
9  (* Stack.push x p empile x sur p *)
10 Stack.push : 'a -> 'a Stack.t -> unit
11
12 (* Stack.pop p dépile p et renvoie l'élément supprimé.
13    Précondition: p est non-vide. *)
14 Stack.pop : 'a Stack.t -> 'a
```

D'où, en OCaml :

```
(* affiche les sommets de g dans un parcours en profondeur depuis s *)
let parcours_profondeur_source (g: int list array) (s: int) : unit =
  let n = Array.length g in
  let vu = Array.make n false in
  let p = Stack.create () in
  vu.(s) <- true;
  Stack.push s p;
  while not (Stack.is_empty p) do
    let u = Stack.pop p in
    print_int u; print_newline();
    (* empile tous les sommets non-vus de l sur p *)
    let rec empile_liste (l: int list) : unit =
      match l with
      | [] -> ()
      | v :: q -> if not vu.(v) then begin
          vu.(v) <- true;
          Stack.push v p
        end;
        empile_liste q
    in
    ajoute_liste g.(u)
  done
```

Exercice 8

Implémenter une version **récursive** du parcours en profondeur, en ajoutant le mécanisme de tableau des sommets vus au parcours d'arbre :

```
let parcours_profondeur (g: int list array) (s: int) : unit =
  let n = Array.length g in
  let vus = Array.make n false in
  let rec visiter (u: int) : unit =
    ...
  in
  visiter s
```

Notons qu'en remplaçant la pile par une file, on obtient un autre type de parcours, appelé **parcours en largeur**. Comme pour les arbres, ce parcours énumère les sommets par ordre de distance au sommet source, ce qui permet de construire efficacement des plus courts chemins (voir partie suivante).

Un parcours (en profondeur ou en largeur) ne peut pas sortir de la composante connexe de son sommet de départ. Ainsi, pour $G = (S, A)$ un graphe non-orienté et $s \in S$, en notant L la liste des sommets visités par un parcours en profondeur de G depuis s, et C la composante connexe de s, on a forcément $L \subseteq C$. En fait, il y a même égalité : un parcours de graphe non-orienté visite **exactement** les sommets dans la composante connexe de la source. On peut en déduire un algorithme efficace de **calcul des composantes connexes**, qui fonctionne en deux parties : une routine qui calcule la composante d'un sommet donné, et une autre qui itère la première sur tous les sommets un par un.

Algorithme 2 : `composante_source`(G, s)

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
```

Algorithme 3 : `liste_composantes`(G)

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 ← composante_source(G, u);
7      CC.ajouter(C);
8      V = V ∪ C;
9  retourner CC
```

Exercice 9

On s'intéresse à la correction de la routine `composante_source` (notée CS dans la suite). On considère $G = (S, A)$ un graphe, $s \in S$. On note C la composante connexe de s : l'objectif est de montrer que $CS(G, s) = C$.

Q1. Montrer que "$V \subseteq C$" est un invariant de la boucle tant que.

Q2. Montrer que tout sommet de C est empilé dans P une fois au cours de l'algorithme. On pourra raisonner par l'absurde et considérer un sommet $u \in C$ jamais empilé, ainsi qu'un chemin $x_0 x_1 \ldots x_k$ de s à u.

Q3. Conclure.

La complexité de `composante_source` est $\mathcal{O}(n+m)$, donc on peut dire que `liste_composantes` est en $\mathcal{O}(n(n+m))$. On peut même être plus précis : chaque sommet $u \in S$ n'est traité qu'une seule et unique fois au cours de tous les appels à `composante_source`. Donc, la complexité de `liste_composante` est une somme de :

— $\mathcal{O}(n)$ pour la boucle principale

— $\sum_{u\in S}(1 + \mathbf{deg}(u))$ pour le coût combiné de tous les appels à `composante_source`

Soit $\mathcal{O}(n + m)$ au total.

L'algorithme précédent montre un schéma assez général 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 ;

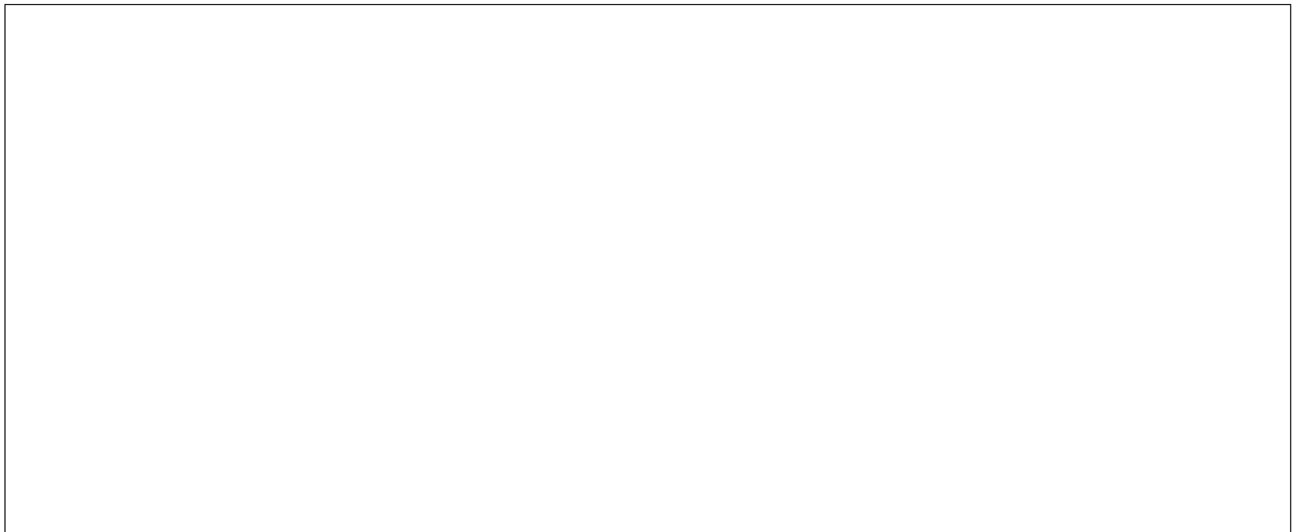
— La fonction principale, lançant la fonction auxiliaire sur chaque sommet n'ayant pas déjà été exploré, et combinant éventuellement les informations renvoyées à chaque appel.

Ceci permet de traiter les graphes composante connexe par composante connexe.

A retenir : La complexité d'un parcours à partir d'un sommet s est $\mathcal{O}(n_s + m_s)$, où n_s, m_s sont le nombre de sommets et d'arêtes de la composante connexe de s, et la complexité d'un parcours de graphe total est $\boxed{\mathcal{O}(n + m)}$.

D Arborescence de parcours

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, $\mathbf{Pred}[u]$ 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 10

Q1. Appliquer cet algorithme sur le graphe suivant, et dessiner l'arborescence du parcours :

Q2. 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.

E 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`(G, s, V)

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. $P \leftarrow$ **pile_vide**();
2. P.**empiler**(s);
3. Ajouter s à V;
4. **tant que** P ***non vide*** **faire**
5. $u \leftarrow P$.**depiler**();
6.   Traiter u;
7.   **pour** v ***voisin de*** u **faire**
8.   **si** $v \notin V$ **alors**
9.   Ajouter v à V;
10. P.**empiler**(v);

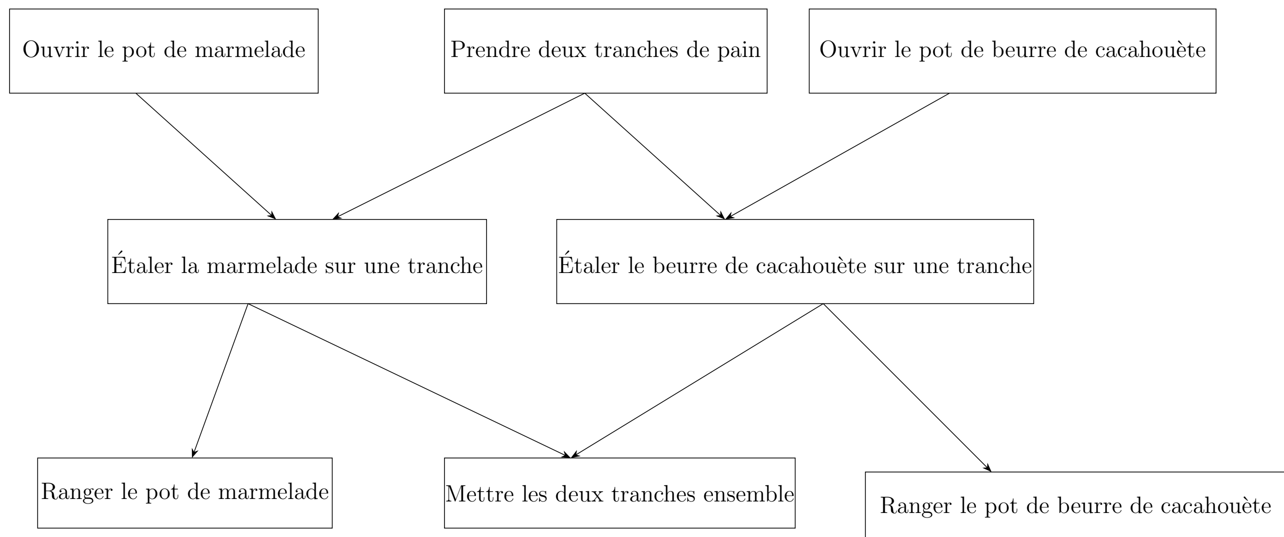
Algorithme 6 : `parcours_profondeur`(G)

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

1. $V \leftarrow \emptyset$ `// ensemble des sommets déjà vus lors du parcours`
2. **pour** $u \in S$ **faire**
3.   **si** $u \notin V$ **alors**
4.   `parcours_profondeur_source`(G, u, V);

F Tri topologique

On considère un graphe $G = (S, A)$ orienté. On peut voir les sommets comme des tâches à accomplir, et un arc (x, y) signifie "la tâche x doit être accomplie avant la tâche y". On appelle un telle graphe un **graphe de dépendances**. Par exemple, lorsque l'on prépare un sandwich beurre de cacahouète-marmelade, on doit procéder dans un certain ordre, résumé par le graphe suivant :



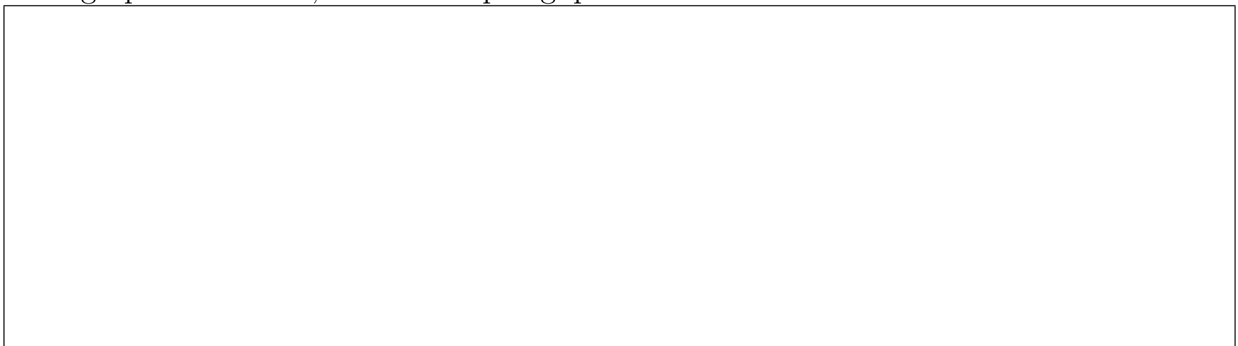
Les graphes de dépendances apparaissent naturellement lors de projets, où les nombreuses tâches doivent être prévues à l'avance.
Dans un graphe de dépendances, un problème classique est de trouver un **ordre** dans lequel exécuter les tâches qui respecte les dépendances. On appelle un tel ordre un **tri 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 6

Voici un graphe orienté G, et un tri topologique valide :



On peut voir un tri topologique comme une manière d'ordonner les sommets dans l'espace de façon à ce que tous les arcs aillent de gauche à droite.
Notons qu'un graphe orienté contenant un circuit ne peut pas admettre de tri topologique. C'est même une condition nécessaire et suffisante.

Définition 11

Un **DAG** (Directed Acyclic Graph) est un graphe orienté sans circuit.

Proposition 4

Un graphe orienté admet un tri topologique si et seulement si c'est un DAG.

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 $k_0 < k_1 < \dots < k_p < k_0$: c'est absurde.

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

Le principe est de chercher à chaque étape une **source**, c'est à dire un sommet sans prédécesseur. On ajoute ce sommet à la liste, on le retire du graphe, et on recommence l'opération.

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 $s \leftarrow$ sommet de G de degré entrant nul;
4 $G' \leftarrow$ sous graphe de G sur $S \setminus \{s\}$;
5 **retourner** $s :: TT(G')$

Si l'on arrive à trouver une source, la preuve par récurrence de la correction est presque immédiate : si $TT(G')$ est un tri topologique valide, puisque s n'a aucun prédecesseur, $s :: TT(G')$ est aussi un tri topologique valide : aucune arête ne va à contre-sens.
Il reste donc à montrer :

Lemme 1

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

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 par récurrence une suite $(u_i)_{i \in \mathbb{N}}$ de sommets tels que (u_{i+1}, u_i) est une arête pour tout $i \in \mathbb{N}$. Prenons $n = |S|$, et considérons $u_n, \dots, u_0$, qui est un chemin valide de G. Il contient $n + 1$ sommets, donc par principe des tiroirs à chaussettes, il existe $0 \leq i < j \leq n$ avec $u_i = u_j$. Donc, $u_j, \dots, u_i$ est un cycle dans G, ce qui est absurde.

Ceci permet de conclure le sens indirect. □

L'algorithme proposé dans la preuve est simple à écrire et à démontrer, mais il n'est pas efficace, car il peut être très long de trouver une source, et encore plus long de construire le sous-graphe G'. Chaque appel récursif va prendre de l'ordre de $\mathcal{O}(n+m)$, pour une complexité totale en $\mathcal{O}(n(n+m))$.

Afin d'améliorer cet algorithme, nous allons mettre en place une structure qui permet de ne pas reconstruire G', mais plutôt de se souvenir des sommets supprimés. Précisément, on stocke dans un dictionnaire le **degré** de chaque sommet. Ainsi, lorsqu'un sommet devient de degré 0, on note que l'on peut l'utiliser comme source.

L'algorithme va donc utiliser trois structures :

- une pile P dans laquelle on empile les sources successives ;
- un dictionnaire d^- marquant le nombre de prédécesseurs restants pour chaque sommet ;
- une liste L contenant le tri topologique en cours de construction.

L'algorithme maintiendra les invariants suivants :

1. Pour tout u dans S, $d^-[u]$ est le cardinal de $\{v \in S \mid (v,u) \in A \text{ et } v \notin L\}$.
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 le dernier arc vers ce sommet. Donc, on ne peut dépiler P qu'au plus n fois lors de l'algorithme : la boucle while termine.

Complexité Pour les mêmes raisons que dans les algorithmes de parcours vus précédemment, un passage de boucle while qui dépile un sommet u, effectue $\mathcal{O}(1+\mathbf{deg}(u))$ opérations, la boucle while est donc en $\mathcal{O}(n+m)$ au total. Les étapes d'initialisation de d^- prennent aussi $\mathcal{O}(n+m)$ car on parcourt chacune des listes d'adjacence une fois. Finalement, tout l'algorithme est en $\mathcal{O}(n+m)$.

Correction Supposons que l'on a montré les deux invariants. Alors, lorsque l'on dépile un sommet u et qu'on l'ajoute un sommet à L, tous ses prédécesseurs sont dans L. Donc, toutes les arêtes entrantes de u respectent l'ordre de la liste renvoyée, et ce pour tout sommet u.
Preuve des invariants : voir TD.

4 Types de graphes

Définition 12

Soit $G = (S, A)$ un graphe, et $S' \subseteq S$. Le sous-graphe de G induit par S' est le graphe $G' = (S', A')$ avec $A' = A \cap S'^2$. Autrement dit, c'est le graphe obtenu en supprimant tous les sommets hors de S', et toutes les arêtes dont les deux extrémités ne sont pas dans S'.

A Arbres

Définition 13

Un graphe non-orienté est dit **acyclique** s'il ne contient aucun cycle. on dit que c'est un **arbre** s'il est connexe et acyclique.

Dans un graphe non-orienté acyclique, chaque composante connexe est un arbre : on appelle parfois les graphes acycliques non-orientés des **forêts**.

Notons que cette notion d'arbre ce correspond pas exactement à celle de structure arborescente des chapitres précédents : il n'y a pas de sommet “racine” privilégié.

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 $m \leq n - 1$.

En particulier, un arbre doit vérifier $m = n - 1$. 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 11

Soit F une forêt à n sommets, m arêtes, et p composantes connexes. Donner un lien entre n, m et p.

B Graphe complet

Définition 14

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

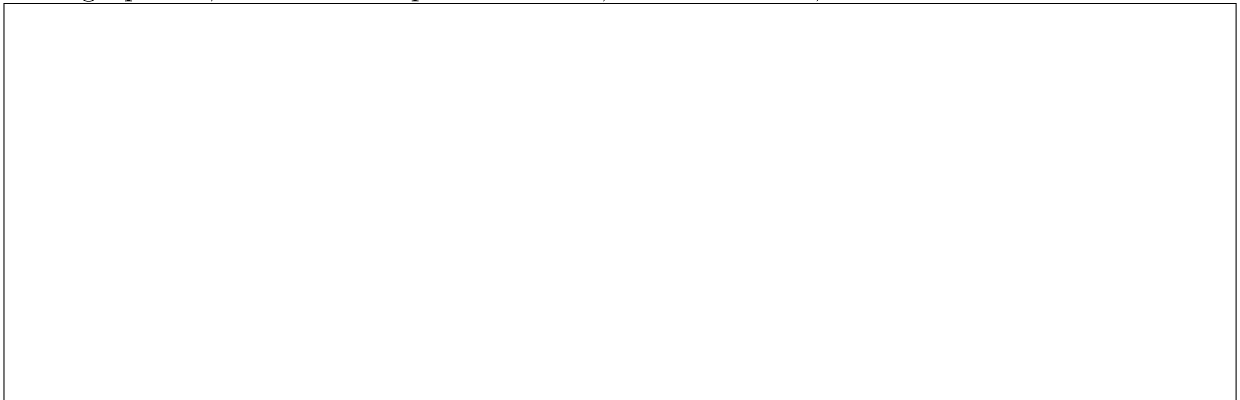
K_n possède $\frac{n(n-1)}{2}$ arêtes.

Définition 15

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 7

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



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 au moins k dans G ?

On ne connaît pas d'algorithme polynomial pour résoudre ce problème : on peut montrer qu'il est aussi difficile que **SAT**[1].

1. La terminologie précise est que **CLIQUE** et **SAT** sont des problèmes **NP**-complets.

C Coloration de graphe, graphe biparti

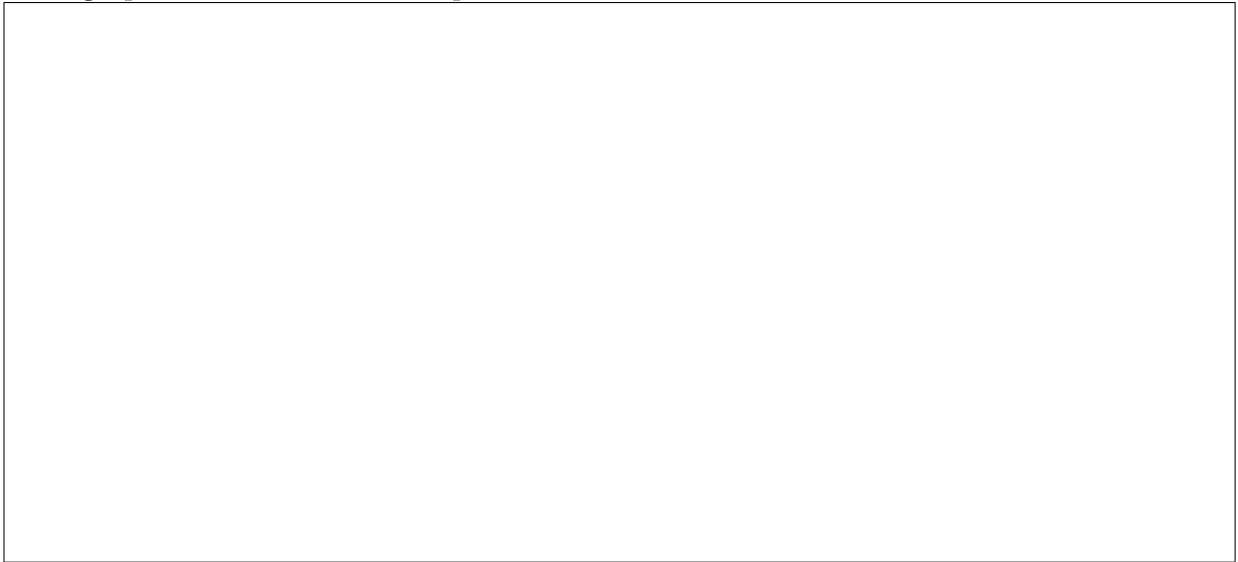
Définition 16

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-colorable. On le note $\chi(G)$.

Exemple 8

Voici un graphe et deux colorations possibles



Définition 17

Un graphe $G = (S, A)$ est **biparti** s'il est 2-colorable.

Si un graphe $G = (S, A)$ est biparti, c'est qu'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, aucune arête ne relie deux sommets de X ou deux sommets de Y.

Savoir si un graphe est k-colorable est très difficile pour k quelconque : c'est un problème aussi dur que SAT pour lequel on ne connaît pas d'algorithme polynomial. En revanche, on dispose d'un critère simple pour savoir si un graphe est biparti :

Proposition 8

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.

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.

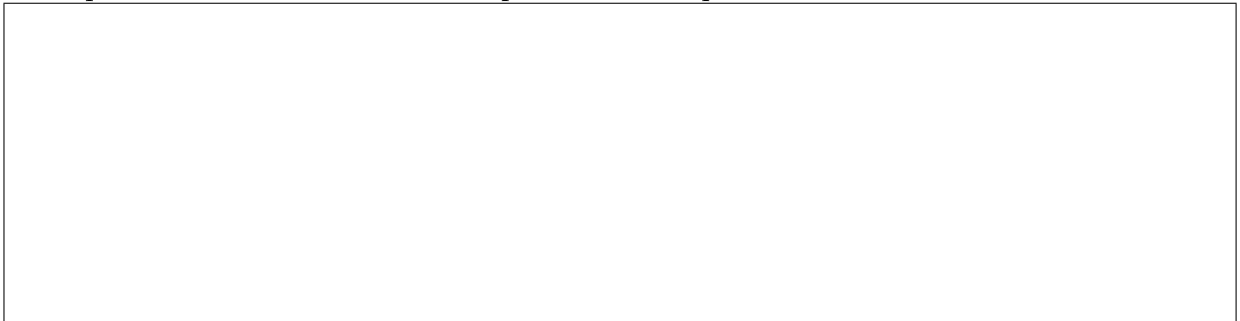
Exemple 9

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 :

```c
int main(){
  int x = 3;
  int y = x + 1;
  int z = x + y;
  int u = 7;
  int v = y + 1;
  u = u + y;
  z = v + u;
}
```

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 qui sont 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 18

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

Exemple 10

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 12

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.

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

Précédemment, 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 de donner une longueur aux arêtes :

Définition 19

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. 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 autorisé 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. 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 successeur et d le poids de l'arc (u, v).

Définition 20

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 $u_0 u_1 \dots u_{k-1} u_0$ 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 $v_0 v_1 \dots v_{j-1} v_j v_{j+1} \dots v_l$ avec $v_j = u_i$, alors on peut considérer le chemin $v_0 v_1 \dots v_{j-1} u_i \dots u_{k-1} u_0 \dots u_i v_{j+1} \dots v_l$ 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.

Il existe de nombreux algorithmes de recherche de plus court chemin. Certains 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 $S = \{1, \dots n\}$. On considère $C^t(i, j)$ = le poids du plus court chemin entre i et j utilisant exactement t arêtes (et $+\infty$ si aucun tel chemin n'existe).
Alors, on peut trouver une formule de récurrence sur $C^t(i, j)$, en raisonnant selon l'avant dernier sommet du chemin :

— $C^0(i, j) = 0$ si $i = j$
— $C^0(i, j) = +\infty$ sinon
— $C^{t+1}(i, j) = \min_{k=1}^{n}(C^t(i, k) + w(k, j))$

La formule exprime que l'on partitionne les chemins de i à j selon le sommet k qui précède j dans le chemin.

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 9

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.

La complexité temporelle de cet algorithme est $\mathcal{O}(n^4)$: n^3 cases à remplir, chacune demandant un calcul de min sur n valeurs.

— 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.
— L'algorithme de Dijkstra permet de calculer la **distance d'un sommet source à tous les autres sommets** du graphe. Il **requiert que les arêtes soient positives**. Avec une implémentation naïve, la complexité atteinte est $\mathcal{O}(m + n^2) = \mathcal{O}(n^2)$, de bonnes structures de données permettent d'atteindre du $\mathcal{O}(m + n \log n)$.

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 10 : 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^n
```

La complexité temporelle de cet algorithme est en $\mathcal{O}(n^3)$, et la complexité spatiale aussi. On peut réduire la complexité spatiale à du $\mathcal{O}(n^2)$, en remarquant que pour construire T^{k+1}, on n'a pas besoin de $T^0, \dots T^{k-1}$ mais uniquement de T^k. On peut donc écraser les tableaux construits au fur et à mesure pour ne garder en mémoire que le dernier calculé. Cependant, pour reconstruire les chemins, il ne faut rien supprimer.

Exercice 13

Quelle information additionnelle pourrait-on stocker dans les T^k afin de pouvoir reconstruire des plus courts chemins efficacement ? Donner le pseudo-code de l'algorithme de reconstruction :

Algorithme 11 : `plus_court_chemin`(T, i, j)

Entrée(s) : $T = (T^0, \dots, T^n)$ liste des tableaux construits par Floyd-Warshall, i, j deux sommets
Sortie(s) : Plus court chemin de i à j.

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 $d[u] \neq +\infty$, alors $\mathbf{Pred}[u]$ 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 12 : 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 est d'utiliser 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 totale $\mathcal{O}(n \log n + m)$.

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 que nous avons déjà montré et utilisé dans de nombreuses preuves :

Lemme 2

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. Pour $u \in S$, notons $\delta(u)$ la distance de s à u, i.e. la longueur d'un PCC de s à u. Montrons les invariants suivants :

(i) $\forall u \in S$, si $d[u] \neq +\infty$, alors il existe un chemin de s à u de longueur $d[u]$, dont la dernière arête est $(P[u], u)$. 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.

— En entrée de boucle :

  (i) Seul s est une clé de d ayant une valeur finie, et $d[s] = 0$, et il existe bien un chemin de s à s de longueur 0, qui n'a aucune arête.

  (ii) Trivialement vrai car $S \setminus Q$ est vide.

  (iii) Le seul chemin n'utilisant que des arêtes partant de $S \setminus Q = \emptyset$ est le chemin vide de s à s, et on a bien $d[s] = 0$ et $d[u] = +\infty$ pour $u \neq s$.

— Supposons les propriétés vraies au début d'un passage de boucle. Notons u le sommet extrait de Q, $v_1, \dots v_k$ les successeurs de u vérifiant la condition de la ligne 8 de l'algorithme. Notons également Q', P', d' les valeurs de Q, P, d à la fin du passage. Montrons que les propriétés restent vraies à la fin du passage. On a :
  — $Q' = Q \setminus \{u\}$, dont $S \setminus Q' = (S \setminus Q) \cup \{u\}$;
  — $P'[v_i] = u$ pour $i = 1, \dots k$;
  — $d'[v_i] = d[u] + w(u, v_i) < d[v_i]$ pour $i = 1, \dots k$;
  — P', d' inchangées sur les autres cases

(i) Soit $v \in S$. Si $d'[v] = d[v]$ alors ni $P'[v] = P[v]$, et donc par HI, il existe un chemin de s à v de longueur $d[v] = d'[v]$ de dernière arête $(P[v], v) = (P'[v], v)$.
Sinon, alors v est l'un des v_i. Alors, $d'[v_i] = d[u] + w(u, v_i)$, et par HI $d[u]$ est la longueur d'un chemin de s à u. En rajoutant l'arête (u, v_i) à ce chemin, on obtient un chemin de s à v_i de longueur $d'[v_i]$ et de dernière arête $(u, v_i) = (P'[v_i], v_i)$.

(ii) Commençons par montrer que $d'[u] = d[u] = \delta(u)$. Par HI, $d[u]$ est la longueur minimale d'un chemin de s à u n'utilisant que des arêtes sortant de $S \setminus Q$. Considérons C un chemin de s à u quelconque, pas forcément restreint à $S \setminus Q$. Soit c le premier sommet de C qui est dans Q. La portion C' de C allant de s à c n'emprunte que des arêtes sortant de $S \setminus Q$, et donc par HI $d[c] = \delta(c)$. Donc C est de longueur au moins $d[c]$. Or, $d[c] \geq d[u]$ car u est extrait lors du passage de boucle. Ainsi, $\delta(u) \geq \delta(c) = d[c] \geq d[u]$, d'où $d[u] = \delta(u)$.
Notons ensuite que tous les sommets modifiés lors du tour (les v_i) sont dans Q. En effet, pour $x \in S \setminus Q$, $d[x] = \delta(x)$, aucun chemin de s à x ne peut donc être strictement plus court que $d[x]$.

(iii) Soit $v \in Q'$. Soit C un PCC de s à u n'utilisant que des arêtes sortant de $S \setminus Q'$.
  — Si C n'emprunte que des arêtes sortant de $S \setminus Q$ (i.e. s'il ne passe pas par u), alors il est de longueur $d[v]$ par HI. De plus, $d'[v] = d[v]$ car sinon, on disposerait d'un chemin de s à v strictement plus court que $d[v]$ n'empruntant que des arêtes sortant de $S \setminus Q'$ (et passant par u).
  — Sinon, alors u est l'avant dernier sommet de C, i.e. le prédécesseur de v. En effet, si u apparaît plus tôt dans C, alors considérons le prédécesseur de v, que l'on note c : $c \in S \setminus Q$ donc il existe un chemin de s à c de longueur $d[c] = \delta(c)$, et ce chemin permet de court-circuiter la portion de C passant par u, ce qui permet de se ramener au cas précédent.
  Alors, $d'[v] = d[u] + w(u, v) = \delta(u) + w(u, v)$ est bien la longueur d'un PCC de s à v n'empruntant que des arêtes sortant de $S \setminus Q'$.

En particulier à la fin de la boucle, $Q = \emptyset$, et les invariants (i) et (ii) permettent immédiatement de conclure. □