Graphes #1

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Graphe

Un *graphe* est un ensemble fini de *sommets* (ou *nœuds* ; en anglais : *vertex*/*vertices*) reliés par des *arcs* (généralement orientés) ou *arêtes* (généralement non orientées) (en anglais : *edges*)

Notation générale

Soit G un graphe, on note S (V) l'ensemble de ses sommets (généralement représentés par des entiers naturels) ; $A \subseteq S \times S$ (E) l'ensemble de ses arcs

Graphes : orientation

Graphe non-orienté

Un graphe est *non-orienté* si ses arcs ne sont pas orientés : on ne distingue pas un arc $u \to v$ d'un arc $v \to u$; un arc $u \leftrightarrow v$ apparaît comme (u, v) **et** (v, u) dans A

Graphe orienté

Un graphe est *orienté* si ses arcs sont orientés : on distingue un arc $u \to v$ d'un arc $v \to u$, et la présence de l'un n'implique pas celle de l'autre

Adjacence

Deux sommets reliés par un arc sont dits *adjacents* ou *voisins*

Exemples

Représentation graphique

On représente les sommets par des ronds, les arcs (orientés) par des traits (flèches) entre les sommets

Vocabulaire de base #1

Boucle

Une *boucle* est un arc $u \leftrightarrow u$ d'un sommet vers lui-même

- Sauf mention du contraire on considérera des graphes sans boucle
  - En particulier dans le cas non-orienté
- Mais l'on pourra aussi parfois supposer implicitement leur présence

Degré

Le *degré* d'un sommet est le nombre d'arcs impliquant ce sommet ; les boucles comptent double

- Graphe non-orienté : $d(u) := \#\{a \in A \mid a = (u, \_)\} + [(u, u) \in a]$
- Graphe orienté : on distingue degré *sortant* : $d_+(u) := \#\{a \in A \mid a = (u, \_)\}$ et degré *entrant* : $d_-(u) := \#\{a \in A \mid a = (\_, u)\}$; on a $d(u) := d_+(u) + d_-(u)$

On peut aussi définir le degré **max**, **min** d'un graphe (de façon évidente)

Vocabulaire de base #2 : chemins

Chemin

Un *chemin* de longueur ℓ entre deux sommets u, v est une suite finie d'arcs $a_1, \ldots, a_\ell$ telle que : $a_1 = (u, \_)$; $a_\ell = (\_, v)$; $\forall i \in [\![1, \ell - 1]\!]$, $a_i[1] = a_{i+1}[0]$

Notation *ad hoc* : $u \rightsquigarrow v = u \to w_1 \to w_2 \to \cdots \to w_{\ell-1} \to v$

- Chaîne : chemin dont les sommets sont deux à deux distincts
- Chemin *élémentaire* : chemin dont les sommets sont deux à deux distincts, sauf éventuellement le premier et le dernier
  - *fermé* : si le premier et le dernier sommet sont **égaux**
- Chemin *simple* : chemin dont les **arcs** sont deux à deux distincts

Cycle

Un *cycle* est un chemin simple élémentaire fermé non vide, qui n'est pas une boucle

On part d'un sommet et y revient en passant par d'autres sommets tous distincts, sans jamais prendre deux fois le même arc

Vocabulaire de base #3 : sous-graphes

Sous-graphe

Un *sous-graphe* d'un graphe $G = (S, A)$ est un graphe $G' = (S', A')$ avec $S' \subseteq S$, $A' \subseteq A$.

Sous-graphe induit

Le *sous-graphe* de $G = (S, A)$ induit par $S' \subset S$ est le graphe $G' = (S', A')$ avec $A' = (S' \times S') \cap A$

On garde un sous-ensemble de sommets de G, et uniquement les arcs impliquant ces sommets

Vocabulaire de base #4 : connexité

Connexité

Un graphe $G = (S, A)$ est *connexe* si pour tout $u, v \in S$ il existe un chemin $u \rightsquigarrow v$ ou $v \rightsquigarrow u$

- Dans un graphe non orienté, les deux conditions sont équivalentes
- Notion pas forcément très utile pour un graphe orienté

Forte connexité

Un graphe **orienté** est dit **fortement** *connexe* si pour tout $u, v \in S$, il existe un chemin $u \rightsquigarrow v$ **et** un chemin $v \rightsquigarrow u$

Faible connexité

Un graphe **orienté** est dit *faiblement connexe* si le graphe non-orienté obtenu en « oubliant » les orientations est connexe

- Notion pas forcément très utile

Composantes connexes

Les *composantes connexes* de $G = (S, A)$ sont ses sous-graphes induits connexes **maximaux pour l'inclusion** ; elles forment une partition de G

Illustrations

Quelques graphes particuliers

Graphe complet

« Le » *graphe complet* à n sommets K_n est un graphe non-orienté tel que toute paire de sommets sont adjacents

Graphe cycle

« Le » *graphe cycle* à $n \geq 3$ sommets C_n est un graphe non-orienté constitué d'un unique cycle de longueur n

Graphe biparti

Les sommets sont l'union disjointe de L et R, avec aucun arc entre deux sommets de L ou deux sommets de R

Arbre

Un graphe non-orienté connexe sans cycle est un *arbre* (au sens des graphes)

Forêt

Un graphe non-orienté sans cycle est une *forêt* (pareil)

Ajout d'information

On peut enrichir les graphes d'information supplémentaire, généralement à des fins de modélisation

Pondération

Un graphe *pondéré* est un graphe $G = (S, A)$ où chaque arc $a \in A$ est pondéré par un *poids* w_a (pour nous : généralement $\in \mathbb{N}$)

- Représente souvent un coût à minimiser (ou plus généralement, une contrainte à optimiser)

Étiquettes

On peut ajouter des *étiquettes* aux sommets ou arcs d'un graphe pour indiquer ce qu'ils représentent (*cf.* exemples de modélisation plus bas)

Table des matières

Représentation #1

Soit $G = (S, A)$, on veut *représenter* S, A & d'éventuelles informations ajoutées avec les types de données habituels, pour pouvoir algorithmiquement manipuler des graphes

Sommets

La plupart du temps : $S \hookrightarrow [\![\#S]\!]$

- Avec éventuellement des étiquettes en plus

Arcs : deux grandes approches classiques (mais pas les seules) :

Listes d'adjacence

- Pour **chaque sommet** u, on liste les sommets v tels qu'il existe un arc (u, v)
  - généralement dans un ordre quelconque
  - avec les éventuelles informations ajoutées (poids, étiquette…)
- On collecte les listes pour chaque sommet, par ex. dans une liste, un tableau, un tableau associatif…

Représentation #2

Matrice d'adjacence

Pour $S \hookrightarrow [\![\#S]\!]$, on représente A par une matrice booléenne carrée a de dimension $\#S$ t.q. `a[i][j]` vaut *vrai* (resp. *faux*) s'il y a (resp. n'y a pas) un arc (i, j) dans A

- On peut remplacer les valeurs booléennes par des poids, des étiquettes…

Quelques comparaisons des représentations

- Taille (avec poids etc. de taille $O(1)$) :
  - Liste de listes d'adjacence : $O(\#S + \#A)$
  - Matrice d'adjacence : $O(\#S^2)$
- Coût pour déterminer si $(u, v) \in A$
  - LA : $O(d(u)) = O(\#S)$
  - MA : $O(1)$
- Coût pour visiter les voisins de u
  - LA : $O(d(u))$
  - MA : $O(\#S)$

Matrice d'adjacence : bien adapté aux graphes *denses* (où $\#A \approx \#S^2$)

Représentation #3 : exemple OCaml

Tableau de listes d'adjacence

```
type graph = int list array
let adj g u v = List.mem v g.(u)
let visit_adj g f u = List.iter f g.(u)
```

Matrice d'adjacence

```
type graph = bool array array
let adj g u v = g.(u).(v)
let visit_adj g f u =
    let f' i x = if x then f i in
    Array.iteri f' g.(u)
```

Exercice / exemple

Table des matières

Modélisation

Les graphes modélisent « naturellement » beaucoup de systèmes. Par exemple :

- Des réseaux (télécom, électrique, de transport de personnes…)
  - Exemple de poids : longueur d'une route
- Des relations entre entités (souhaits d'intégration v. propositions d'admission)
  - Exemple de poids : ordre de préférence
- Des circuits/programmes sans branchement
  - Exemple d'étiquette : opération ; littéral
- Les états d'un *automate*, d'un puzzle à résoudre

On peut alors formuler des problèmes sur de tels systèmes en terme des graphes qui les modélisent, pour les résoudre algorithmiquement

- Beaucoup de problèmes *d'optimisation* (« quelle est la meilleure façon de… »), mais aussi de *décision* (« est-ce qu'il est vrai que… ») ou de recherche (« trouver quelque chose qui… »)

Représentation #4 : représentations implicites

- Il peut arriver qu'un graphe ne soit pas représenté explicitement, notamment quand il est très grand
- Une représentation *implicite* se base à la place sur une description de S et une fonction décidant/listant l'adjacence

Exemples

- Graphe des parties d'un jeu (d'échecs, go…)
- Graphe fonctionnel d'une fonction
- Graphe d'un grand réseau

Exemples de problèmes de graphes #1

Quelques problèmes « structurels »

Ignorent les éventuelles informations auxiliaires (poids etc.)

Soit un graphe $G = (S, A)$:

- Soit $u, v \in S$, existe-t'il un chemin entre deux sommets ?
- Quelles sont les composantes (fortement) connexes de G ?
  - De plus si G orienté, quelle est sa *fermeture transitive* ?
- G possède-t'il des cycles ?
- G possède-t'il un chemin (ou cycle) « hamiltonien » passant exactement une fois par chaque sommet ?
  - Coûte cher
- — « eulérien » — arc ?
  - Ne coûte pas cher
- Quel est un *coloriage* optimal de G ?
- Si G est biparti, quel est un *couplage de cardinal maximum* ?
- Soit un autre graphe G', est-il isomorphe à G (« identique à renommage près »)

Exemples de problèmes de graphes #2

Problèmes pas uniquement structurels

Font intervenir la structure du graphe, mais aussi les éventuelles informations auxiliaires
Soit un graphe $G = (S, A)$:

- Soit $u, v \in S$ reliés par un chemin, quel est un *plus court chemin* entre eux ; un chemin de *capacité maximum* ?
- G possède-t'il un cycle de poids négatif ?
- Pour G connexe non-orienté, quel est un *arbre couvrant* de poids minimum ?
- Si G possède un cycle hamiltonien, quel en est un de poids minimum ?

Table des matières

Parcours : définition

Un parcours de graphe depuis un sommet u est un algorithme *visitant* exactement les sommets v t.qu'il existe un chemin $u \rightsquigarrow v$, *uniquement en suivant les arcs du graphe*

- *visiter* : appliquer un traitement (par ex. ajouter à une liste des sommets visités)
- Même chose qu'un parcours d'arbre, mais pour un graphe…

Contre-exemple

- Un algorithme qui pour un graphe non-orienté garanti connexe tire un ordre aléatoire sur les sommets et les visite dans cet ordre

Parcours : utilité

- Permet de parcourir (…) un graphe même quand il est représenté de façon implicite, et de calculer la relation d'accessibilité entre sommets
  - Un parcours depuis un sommet visite exactement le sous-graphe accessible depuis lui-même (sa composante connexe), mais on peut évt. recommencer
- Respecte la « topologie » du graphe
  - Permet d'apprendre des choses sur la connexité, les cycles, les distances…
- Définit un *ordre de visite* (généralement non unique)
- Définit une *arborescence* de parcours (pareil)

Parcours : algorithme générique (pour listes d'adjacences)

Principe générique

On part d'un sommet, on l'ajoute dans une structure de donnée (pile, file, …) de sommets à traiter, puis tant qu'il y a des sommets à traiter on en choisit un (disons u) ; on découvre ses voisins et ajoute ceux pas encore traités aux sommets à traiter ; on marque u comme ayant été visité

- En fonction de la structure utilisée, on obtient différents *types* de parcours
- Attention : l'ordre des opérations à un impact ; ici : le marquage des sommets est **tardif** (quand on les *extrait* de la structure, pas quand on les insère)
- On peut appliquer un traitement aux sommets au moment du marquage (et dans certains cas à d'éventuels autres moments, *cf.* plus tard)

Exemple

Parcours générique en OCaml

```
let wfs make push pop_opt f g v =
    let vis = Array.make (Array.length g) false in
    let bag = make () in
    let push_cond v = if not vis.(v) then push bag v in
    let rec loop
        = function
        | None -> ()
        | Some nxt -> if not vis.(nxt) then (
                          f nxt ; vis.(nxt) <- true ;
                          List.iter push_cond g.(nxt)) ;
                      loop (pop_opt bag)
    in loop (Some v)
```

À éventuellement répéter pour chaque sommet non encore visité…

Esquisse de preuve

On veut montrer qu'un sommet w est visité ss.'il existe un chemin $v \rightsquigarrow w$

On a l'invariant que le `bag` contient uniquement des sommets accessibles depuis v, ce qui donne $\Leftarrow$ (initialisation okay à condition de modifier légèrement l'appel initial ; conservation okay par `List.iter` et la structure de `g`)

Pour $\Rightarrow$, on peut procéder par récurrence sur la *distance* des sommets (la longueur (le nombre d'arcs) d'un plus court chemin au point de départ).

- Le cas de base est l'appel initial à `loop` qui considère l'unique sommet v à distance zéro de lui-même (c'est à dire lui-même)
- Préservation : HR : les sommets à distance n sont visités. Par déf. un sommet x à distance $n+1$ (s'il y en a un) est à distance un (arc) d'un sommet w à distance n, et est donc mis dans le `bag` (au plus tard) quand w est visité ; il en est également sorti et donc visité

Algorithme générique *bis*

What kind of bag?

- Pile $\rightsquigarrow$ parcours en profondeur

```
let dfs f g v =
    wfs Stack.create (Fun.flip Stack.push) Stack.pop_opt
        f g v
```

- File $\rightsquigarrow$ parcours en largeur

```
let bfs f g v =
    wfs Queue.create (Fun.flip Queue.push) (Queue.take_opt)
        f g v
```

- (File de priorité, pour le calcul de plus-court chemins pondérés)

Parcours : ordre de visite, arborescence

Il est parfois utile dans un parcours de déterminer :

- l'ordre de visite (de traitement) des sommets (parfois *les* ordres)
- les arcs utilisés pour (effectivement) découvrir les sommets

Il est facile d'adapter l'algorithme générique pour obtenir ces informations, par exemple grâce à une horloge virtuelle incrémentée à chaque visite et en stockant les arcs plutôt que les sommets dans le `bag`

Parcours générique avec informations supplémentaires

```
let wfs' make push pop_opt g v =
    let vis = Array.make (Array.length g) None in
    let bag = make () in
    let push_cond p v = if vis.(v) = None then push bag (p, v) in
    let clock = ref 0 in
    let rec loop
        = function
        | None -> vis
        | Some (p, v) -> if vis.(v) = None then (
                            vis.(v) <- Some (!clock, (p, v)) ;
                            incr clock ; List.iter (push_cond v) g.(v)) ;
                            loop (pop_opt bag)
    in loop (Some (v, v))
```

Arborescence

L'*arborescence* d'un parcours est le graphe formé des sommets visités et des arcs utilisés par le parcours pour les visiter

- Dans l'algorithme générique, les arcs inclus correspondent à $p \to v$
  - Un sommet peut être *potentiellement* découvert depuis plusieurs prédécesseurs, mais on ne garde que l'arc « qui a servi » à la visite

Arborescence : arbres & forêts

L'arborescence d'un parcours est un arbre (enraciné)

- Chaque sommet a au plus un prédécesseur : un parent
  - Exactement un sauf pour l'origine du parcours : la racine naturelle
- L'origine du parcours est reliée à tout sommet de l'arborescence par un chemin (*cf.* dans un instant)

Forêt couvrante pour un graphe non-orienté

- Connexe : l'arborescence est un *arbre couvrant*
- Non-connexe : les arborescences (si parcours répétés) donnent une forêt couvrante des composantes connexes

Exemples

Reconstruction de chemin

Les informations supplémentaires calculées par un parcours permettent de construire un chemin (simple, élémentaire) entre l'origine du parcours et chaque sommet visité

- Il suffit de remonter les arcs jusqu'à la racine

Exemple

```
let rec build_path g vis v
    = match vis.(v) with
    | None -> assert false
    | Some (_, (p, _)) -> if p = v then [] else
                          v::(build_path g vis p)
```

Parcours générique : coût

Coût

Pour un parcours **depuis un sommet** v dans un graphe de S sommets (S_v accessibles depuis v), avec A_v arcs accessibles depuis v et $\beta(n)$ le coût des opérations sur un `bag` de taille n :

- Basique : $O(A_v \times \beta(A_v) + S)$ temps et espace
  - $O(S)$ pour l'initialisation de la structure de visite, $O(A_v \times \beta(A_v))$ car on n'ajoute et n'enlève un arc dans le `bag` qu'au plus une fois
- Basique + structure efficace pour visite : $O(A_v \times \beta(A_v) + S_v \times \tau(S_v))$ temps, $O(A_v)$ espace ($\tau(n)$: coût des opérations sur une structure efficace de taille n)
- Basique + structure efficace pour visite + test `bag` (si possible) : $O(A_v \times \beta(A_v) + S_v \times \tau(S_v))$ temps, $O(S_v)$ espace

Pour un parcours basique $\beta(n) = O(1)$; dans ce cas le coût est linéaire en la taille de l'entrée (« efficace ») quand elle est représentée explicitement

Caractérisation d'un parcours

Arborescence & ordre de visite (et évt. plus) permettent de caractériser le type de parcours effectué

Exemples

- Parcours en largeur
  - L'arborescence donne des plus-court-chemins non-pondérés à l'origine
- Parcours en profondeur
  - Les temps de *pré-visite* et *post-visite* des sommets sont bien parenthésés
- Algorithme dit de Dijkstra
  - L'ordre de visite se fait par distance pondérée à l'origine croissante
- Algorithme dit de Prim 🙈
  - L'arborescence est un arbre couvrant de poids minimal

Table des matières

Parcours en largeur : caractérisation

Caractérisation

Un parcours en largeur depuis un sommet u est tel que les sommets sont visités par distance (non pondérée) croissante à l'origine

- La profondeur de tout sommet v dans l'arborescence du parcours enracinée à u est égale à la distance non-pondérée de u à v

Proposition

Un parcours en largeur peut s'implémenter en utilisant une file comme `bag` d'un parcours générique

Idée de preuve

Par l'invariant de boucle que la file contient des sommets à distance à l'origine $[i, i, \ldots, i]$ ou $[i, i, \ldots, i+1, i+1]$

Parcours en largeur : implémentation directe sans file

On maintient et alterne deux listes des sommets à distance i et $i+1$

```
let bfs' g v =
    let vis = Array.make (Array.length g) None in
    let rec push_cond p x y = match x with
        | [] -> y
        | v::xs -> if vis.(v) = None then push_cond p xs ((p, v)::y)
                   else push_cond p xs y in
    let clock = ref 0 in
    let rec loop = function
        | [], [] -> vis
        | [], nxtip1 -> loop (nxtip1, [])
        | (p, v)::nxti, nxtip1 -> if vis.(v) = None then (
                    vis.(v) <- Some (!clock, (p, v)) ; incr clock ;
                    loop (nxti, (push_cond v g.(v) nxtip1))) else
                    loop (nxti, nxtip1)
    in loop ([(v, v)], [])
```

Parcours en largeur : coût & applications

Coût

Pour un parcours depuis un sommet v dans un graphe de S sommets (S_v accessibles depuis v), avec A_v arcs accessibles depuis v :

- Basique : $O(A_v + S)$ temps et espace
- Basique + structure efficace pour visite : $O(A_v)$ temps et espace (simplifié)
- Basique + structure efficace pour visite + test file : $O(A_v)$ temps, $O(S_v)$ espace (simplifié)

Applications

- Donne directement un algorithme de plus-court-chemins non pondérés
- Base utile pour des recherches de distance plus sophistiquées, par ex. :
  - Plus-court-chemins pondérés
  - Recherche de cycle de longueur (non-pondérée) minimale

Table des matières

Temps de pré/post-visite

On peut utiliser l'horloge virtuelle t maintenue dans un parcours pour définir pour tout sommet visité un :

- temps de pré-visite : la valeur de t juste après (ou avant) son marquage
  - notation pour un sommet v : v_a (a : *ante*)
- temps de post-visite : la valeur de t juste après que **tous** les sommets adjacents pas encore pré-visités ont été **post**-visités (si besoin récursivement)
  - (la valeur de t juste après que tous les descendants dans l'arborescence du parcours ont été post-visités)
  - notation pour un sommet v : v_p (p : *post*)

On incrémente l'horloge de 1 à chaque pré/post-visite ; les temps de pré/post-visite sont distincts deux à deux et consécutifs

Proposition

- v est une feuille dans l'arborescence du parcours ssi. $v_p = v_a + 1$

Parcours en profondeur : caractérisation

Caractérisation

Un parcours depuis un sommet v est en profondeur ssi. pour toute paire de sommets visités les temps de pré/post-visites sont *bien parenthésés*, et que pour tout sommet w de descendants $w^{(1)}, \ldots, w^{(n)}$ dans l'arborescence l'on a (sans perte de généralité)
$w_a^{(1)} = w_a + 1$; $w_a^{(i+1)} = w_p^{(i)} + 1$ pour tout $1 \leq i < n$; $w_p = w_p^{(n)} + 1$

Le temps de pré-visite correspond à l'ordre de traitement pour un parcours *préfixe*, et celui de post-visite à celui d'un parcours *postfixe*

Bon parenthésage

Soit v, w deux sommets visités et v_a, v_p, w_a, w_p leurs temps de pré/post-visite, les seuls cas possibles pour un parcours en profondeur sont :

- $v_a < w_a < w_p < v_p$ (et symétrique)
- $v_a < v_p < w_a < w_p$ (et symétrique)

(Contre-)exemples

Existence

Un parcours en profondeur peut s'implémenter en utilisant une pile comme bag d'un parcours générique **avec marquage tardif**, ou récursivement

Parcours en profondeur : exemple d'implémentation récursive

```
let dfs' g v =
    let vis = Array.make (Array.length g) None in
    let clock = ref 0 in
    let rec _dfs p v =
        if vis.(v) = None then (
        incr clock ;
        let pre = !clock in
        vis.(v) <- Some ((pre, pre), (p, v)) ;
        List.iter (_dfs v) g.(v) ;
        incr clock ;
        vis.(v) <- Some ((pre, !clock), (p, v)))
    in
    _dfs v v ; vis
```

Parcours en profondeur : correction

Proposition

`dfs'` satisfait la caractérisation d'un parcours en profondeur

Preuve (esquisse)

C'est bien un algorithme de parcours ; on considère deux sommets v et w t.q. $w_a > v_a$ et distingue les deux cas possibles :

- l'appel à `_dfs w _` se fait quand l'appel à `_dfs v _` n'a pas encore terminé : le premier terminera avant le second, donc $w_p < v_p$
- l'appel à `_dfs w _` se fait quand l'appel à `_dfs v _` a déjà terminé : $v_p < w_a$

De plus lorsqu'un sommet est pré-visité, tous ses sommets adjacents non encore visités sont visités en séquence (il n'y a pas d'autre sommet pré-visité antérieurement qui déclenchera une visite d'un sommet lui étant adjacent)

Remarque

Ceci montre aussi l'*existence* d'un parcours en profondeur (pas *a priori* évidente)

Parcours en profondeur : autre exemple d'implémentation récursive

```
let dfs'' f g v =
    let vis = Array.make (Array.length g) false in
    let rec _dfs
        = function
        | [] -> ()
        | v::vs -> if not vis.(v) then (
                    f v ; vis.(v) <- true ; _dfs g.(v)) ;
                    _dfs vs
    in
    _dfs [v]
```

Parcours en profondeur : coût

Coût

Pour un parcours depuis un sommet v dans un graphe de S sommets (S_v accessibles depuis v), A_v arcs accessibles depuis v, pour une implémentation itérative :

- Basique : $O(A_v + S)$ temps et espace
- Basique + structure efficace pour visite : $O(A_v)$ temps et espace (simplifié)
- Basique + structure efficace pour visite + pile à accès direct : $O(A_v)$ temps, $O(S_v)$ espace (simplifié)

Ordre de visite en profondeur : classification des arcs

On peut utiliser les temps de pré/post-visite et l'arborescence d'**un** parcours en profondeur pour classifier les arcs du (sous-)graphe (relativement à **ce** parcours)

Pour tout arc $v \to w$ (dans le cas d'un graphe non orienté, sans distinguer avec $w \to v$), si :

- $v \to w$ est inclus dans l'arborescence, c'est un *arc de l'arborescence*, ou *arc direct* ; on a $v_a < w_a < w_p < v_p$
- $v_a < w_a < w_p < v_p$ mais $v \to w$ ne fait pas partie de l'arborescence, c'est un *arc avant*
- $w_p > v_p > v_a > w_a$, $v \to w$ est un *arc arrière*
- $v_a > w_p$, $v \to w$ est un *arc transverse*
- $v_p < w_a$ est impossible (car tout sommet adjacent à v est pré-visité avant sa post-visite)

Exercice

Dessinez un graphe et une arborescence de parcours en profondeur illustrant tous les cas (possibles) ci-dessus