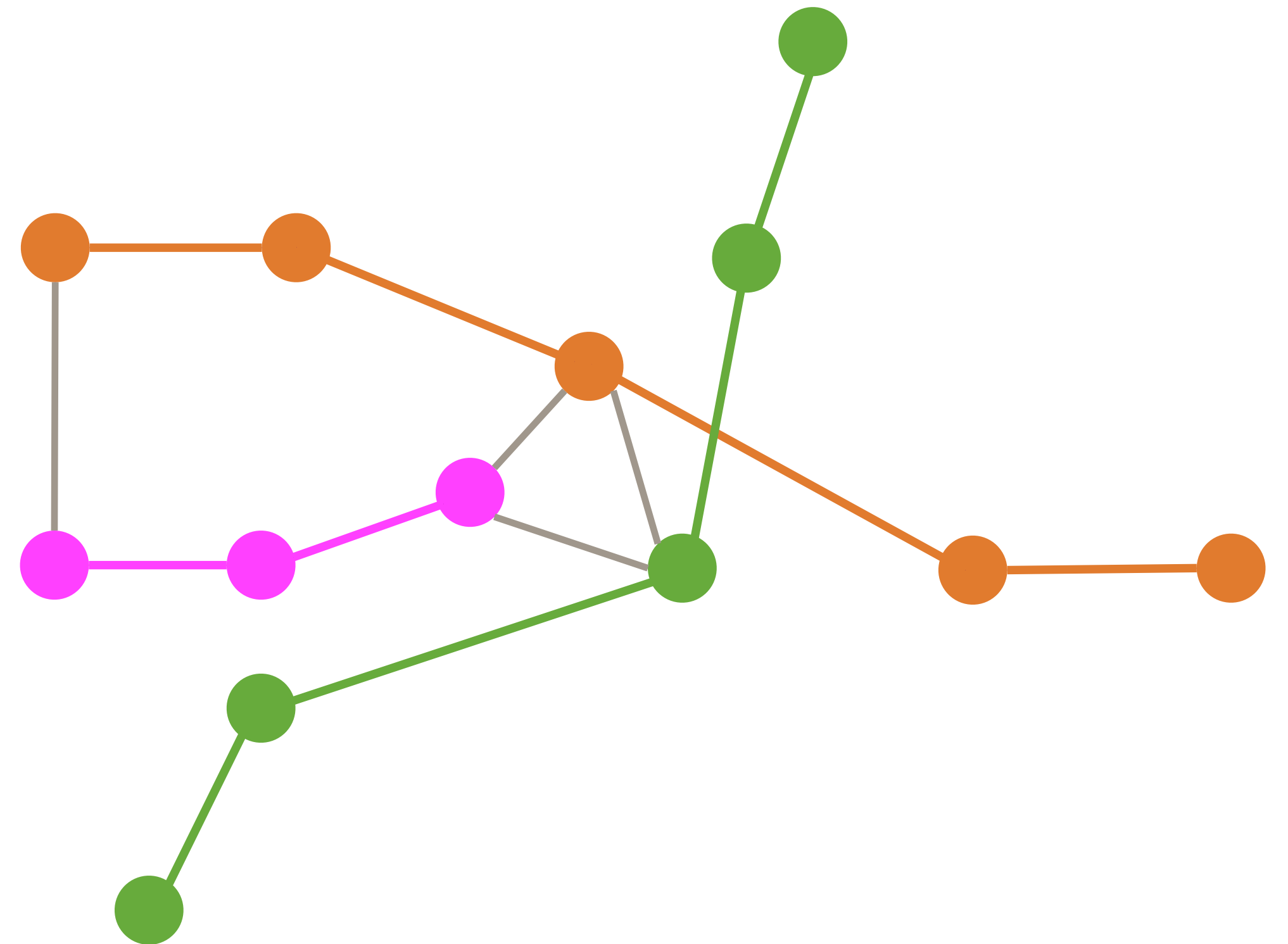
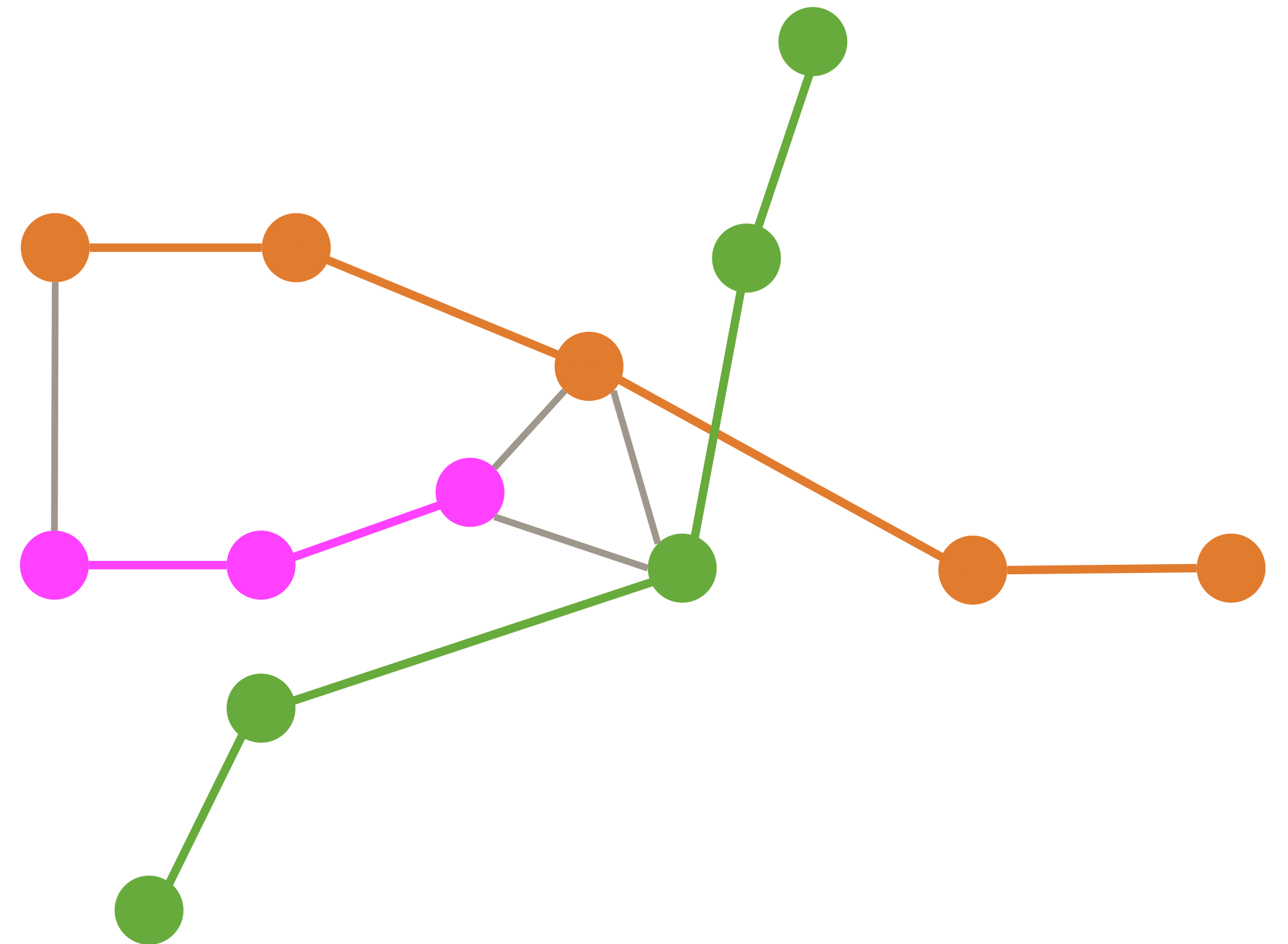


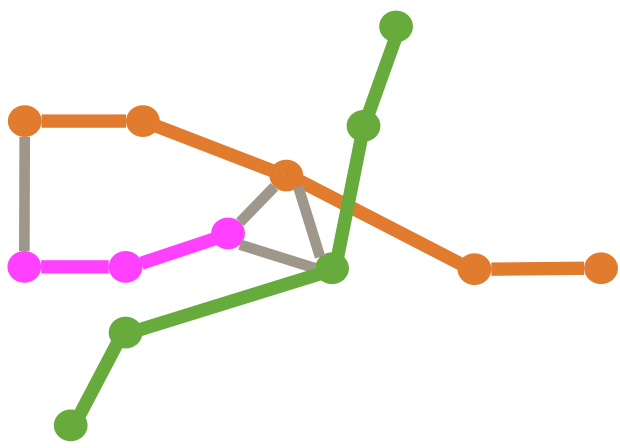
Introduction aux graphes



6.1 Introduction

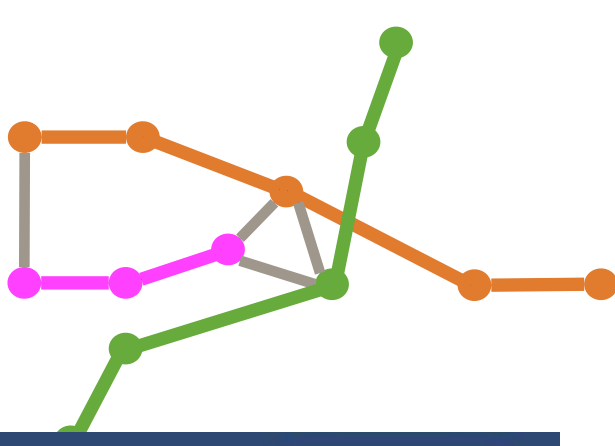


Quelques problèmes pratiques ...

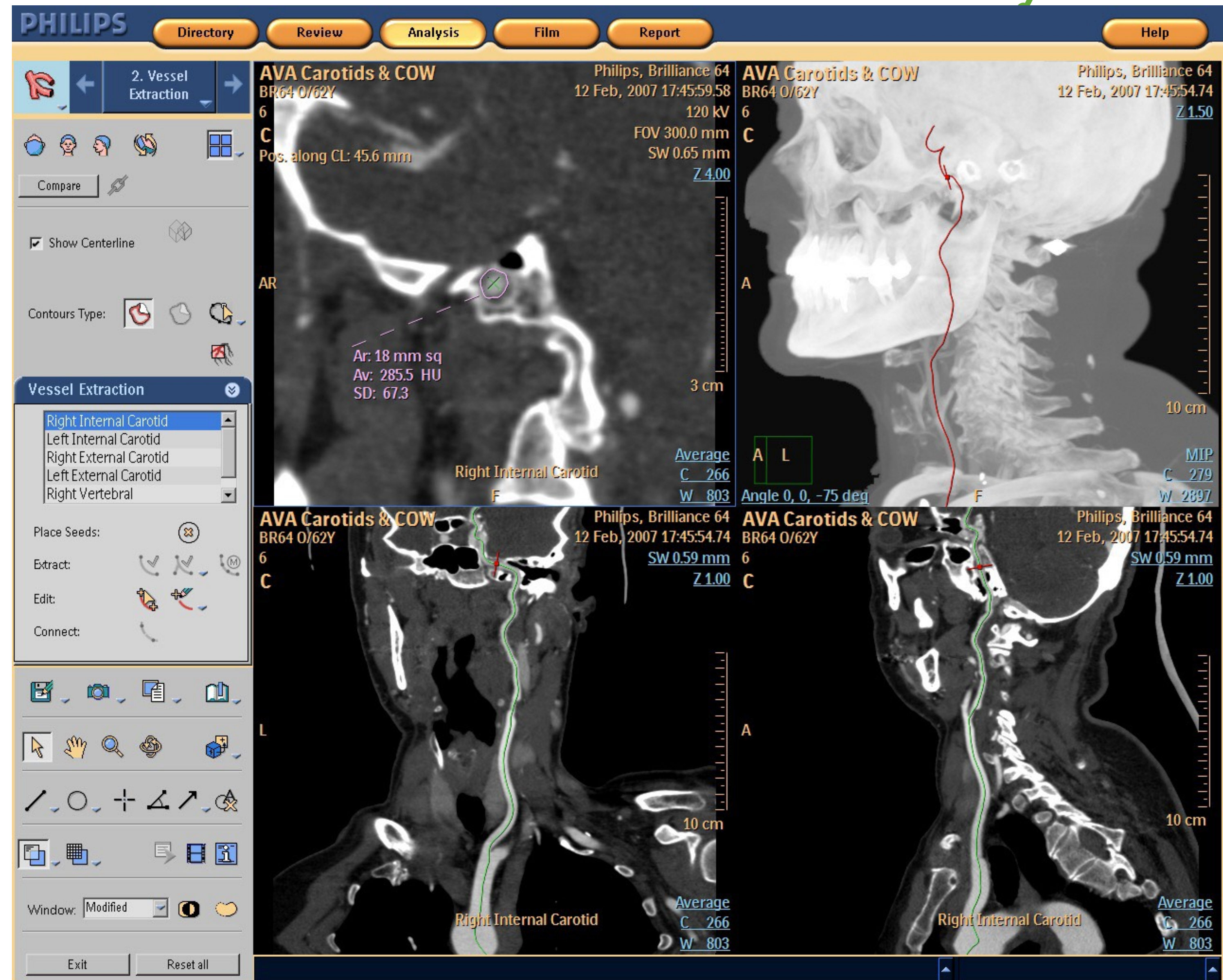


- Détection des carotides dans un scanner CT avec produit de contraste iodé
- Découverte de possibilité d'arbitrage dans un marché de change déséquilibré
- Recherche de meilleur chemin sur la carte du métro parisien

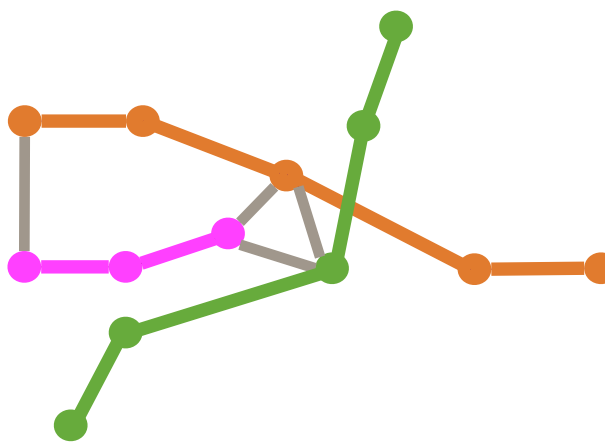
Scanner CT



- Visualisation des artères carotides
- Comment détecter les artères carotides automatiquement ?
- « Meilleurs chemins allant de l'aorte aux artères cérébrales »



Arbitrage d'un marché



- Normalement, pas d'arbitrage possible :

CHF -> EUR -> CHF :

$$1 \times 0.829 \times 1.206 = 0.999$$

	CHF	EUR	USD	GBP
CHF	1	0.829	1.102	0.658
EUR	1.206	1	1.295	0.793
USD	0.907	0.772	1	0.612
GBP	1.519	1.26	1.632	1

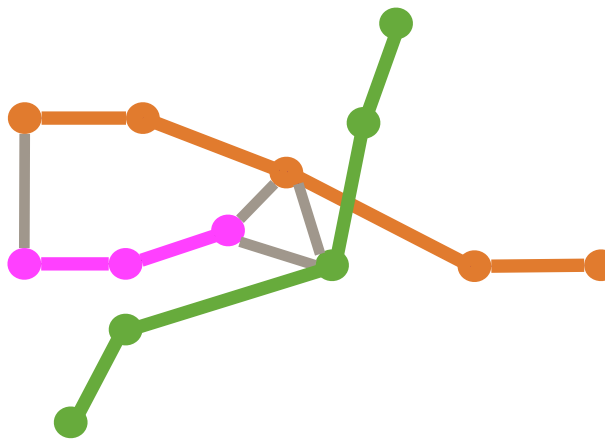
- Ici, les prix sont temporairement déséquilibrés :

CHF -> USD -> EUR -> CHF :

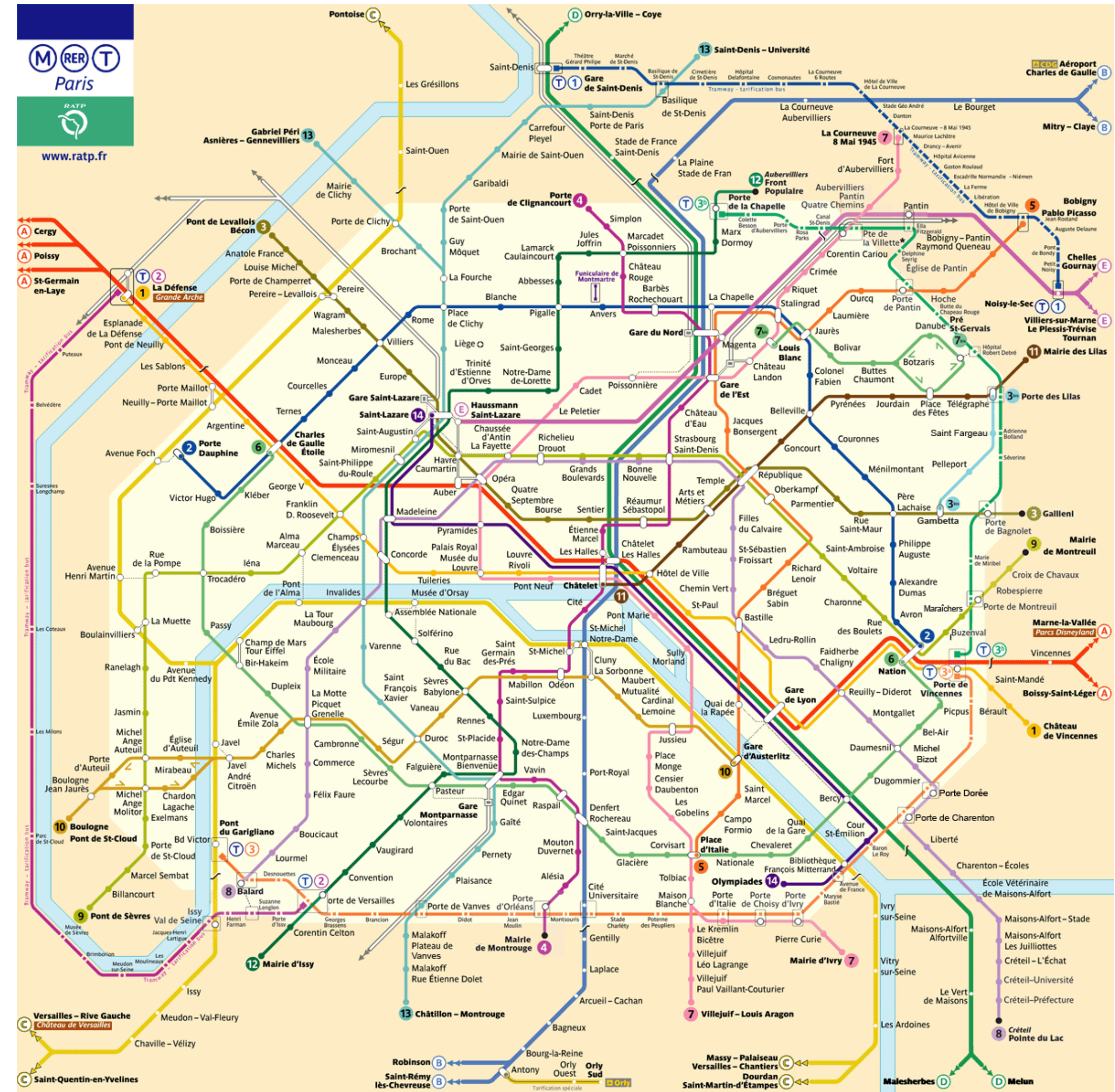
$$1 \times 1.102 \times 0.772 \times 1.206 = 1.026$$

- On cherche un chemin circulaire dont le produit des valeurs est > 1 (la somme des logarithmes est > 0)

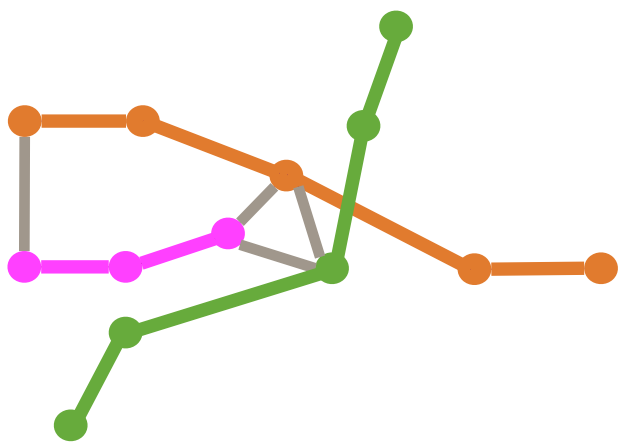
Métro de Paris



- Quel est le meilleur chemin pour aller de Gare de Lyon à Porte de Versailles?
- Le plus rapide ?
- Avec le moins de connexions?
- Avec le moins de marche à pied
- Si les lignes 1 à 9 sont en grève, est il possible de relier ces deux stations?

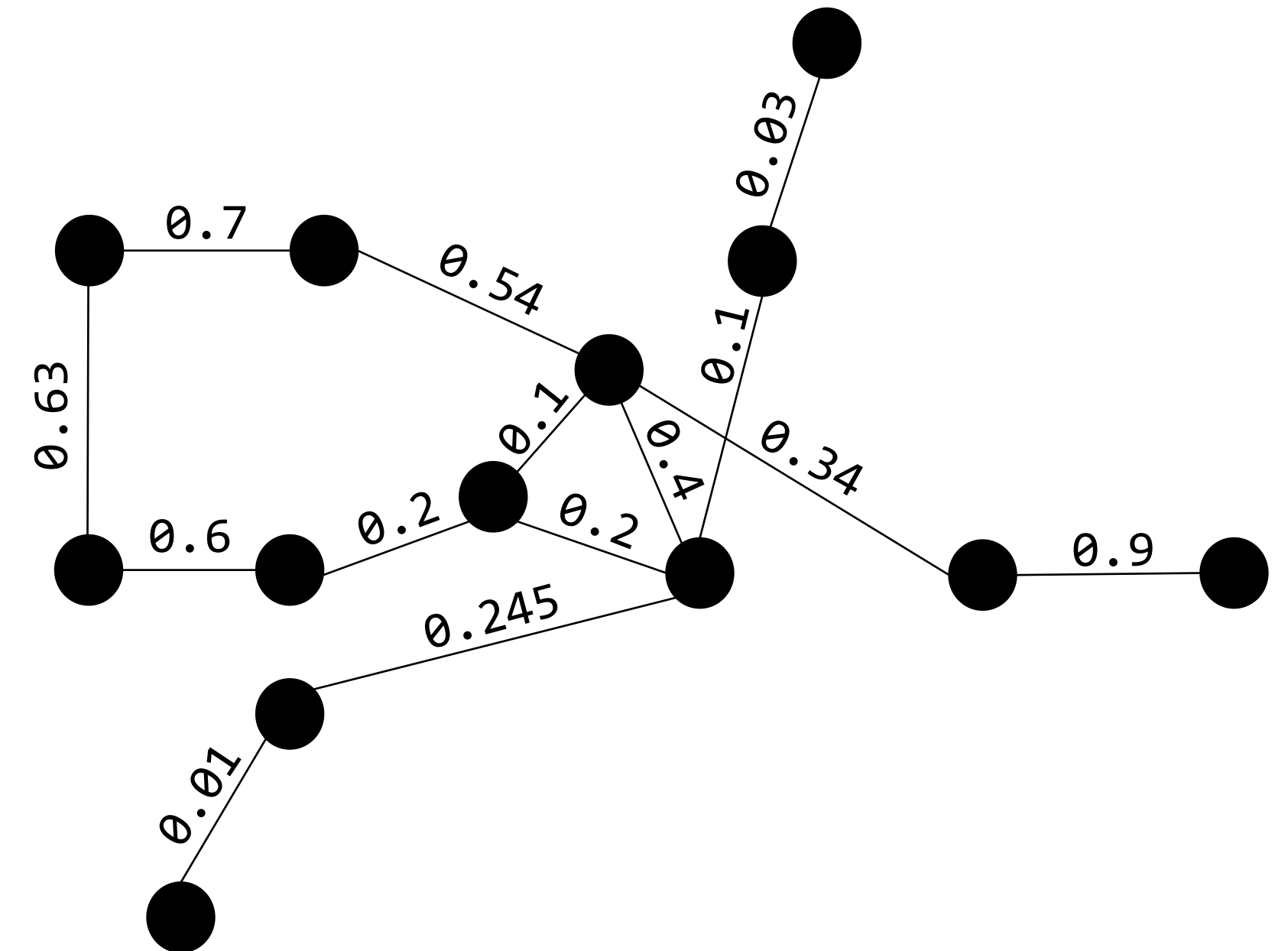


Qu'ont-ils en commun ?

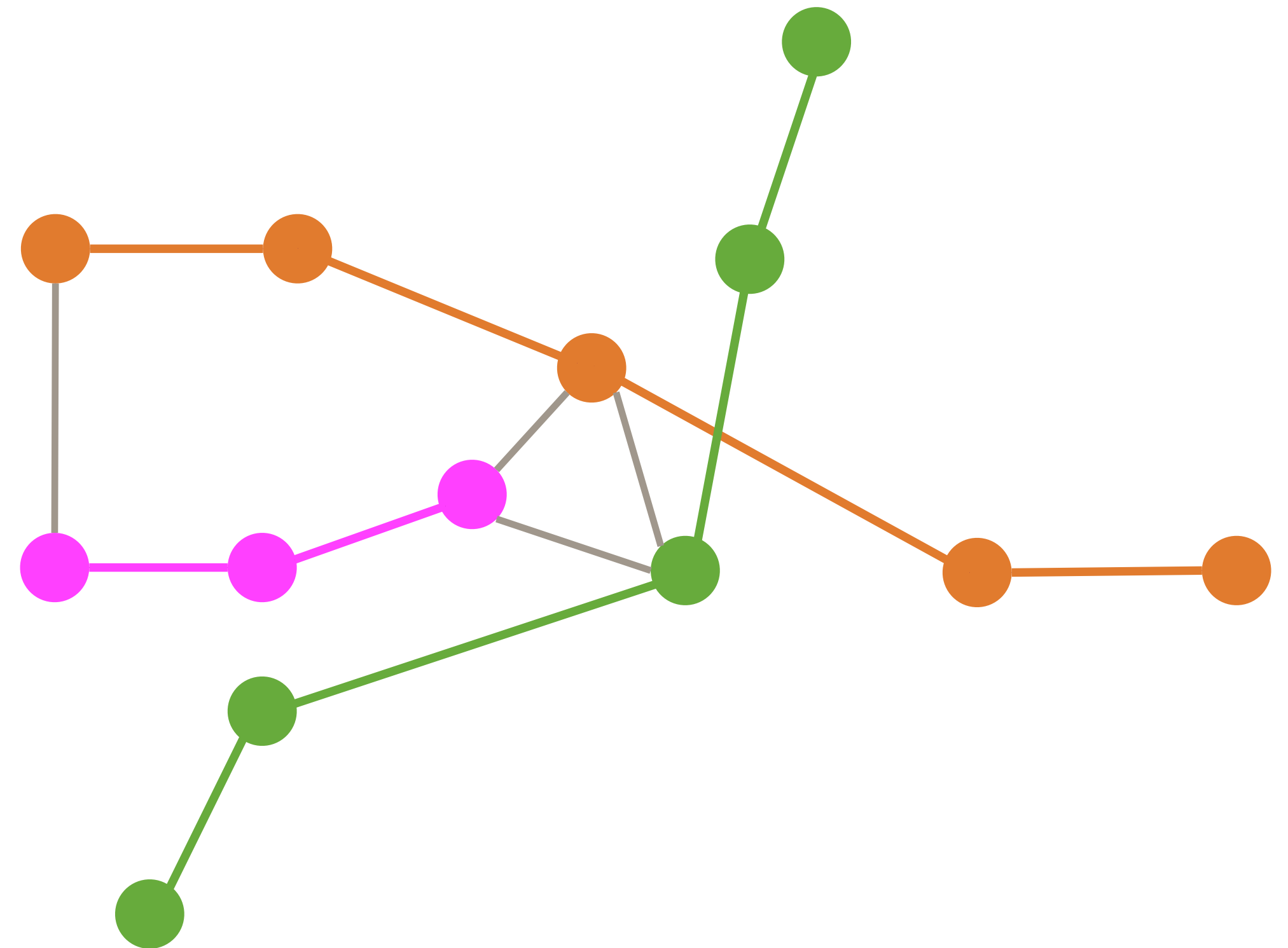


Ce sont tous des problèmes de recherche de chemin dans des graphes

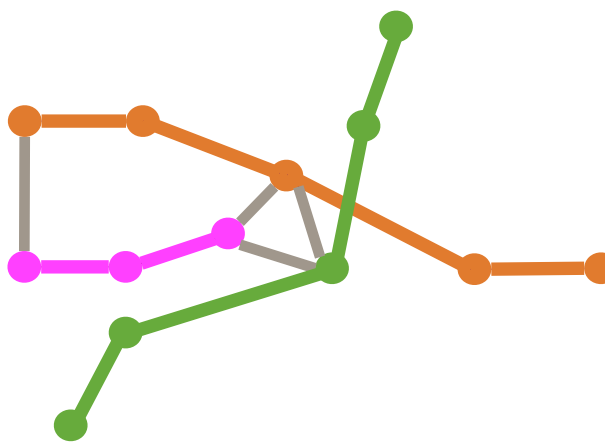
- Un ensemble de **sommets** (voxels / monnaies / stations de métro)
- Connectés par des **arêtes** ou **arcs** deux à deux (voxels voisins, paires de monnaies, lignes de métro)
- **Pondérées** par des poids de connexion (similitude d'intensité, taux de change, longueur / temps de parcours / ...)



6.2 Nomenclature



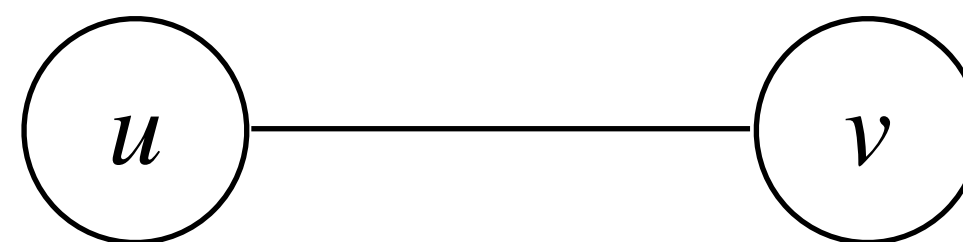
Graphes non-orientés



Un graphe **non-orienté** $G = (V, E)$ est défini par

- Un ensemble V d'éléments appelés **sommets**
- Un ensemble E de paires non-ordonnées de sommets, appelées **arêtes**

Donc, toute arête $e \in E$ a la forme $\{u, v\}$, avec $u, v \in V$.

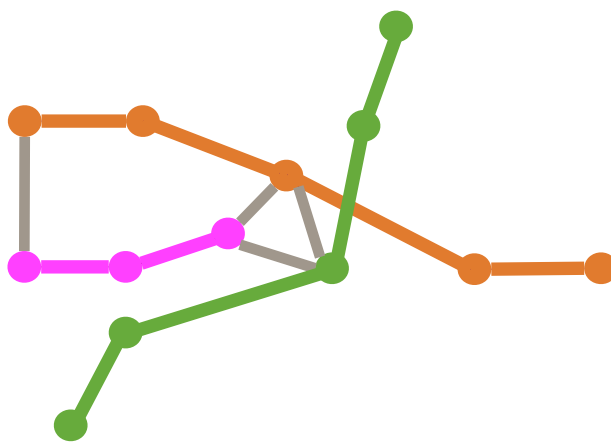


On dit que u et v sont les **extrémités** de e , et que e les **relie**.

On dit alors que u et v sont **adjacents**.

On parle d'**incidence** entre e et la paire u et v .

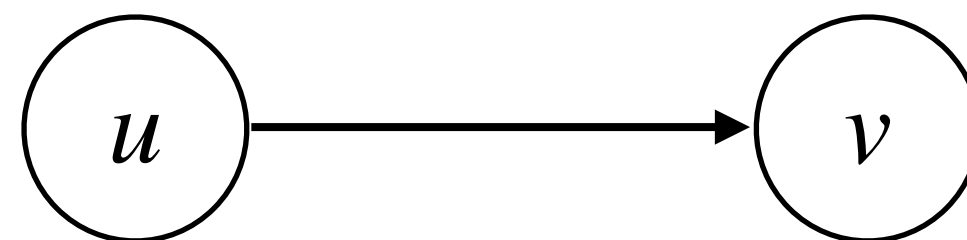
Graphes orientés



Un graphe **orienté** $G = (V, E)$ est défini par

- Un ensemble V d'éléments appelés **sommets**
- Un ensemble E de paires ordonnées de sommets, appelées **arcs**

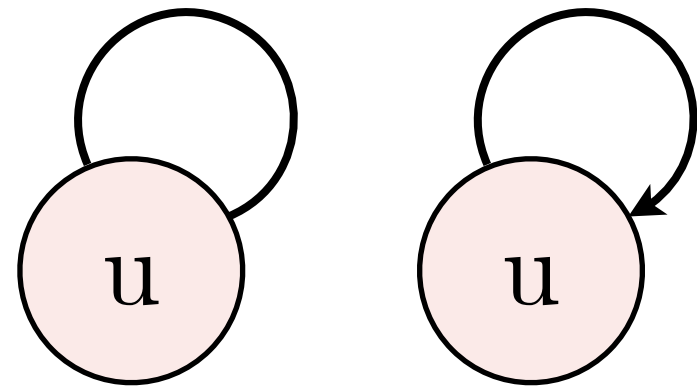
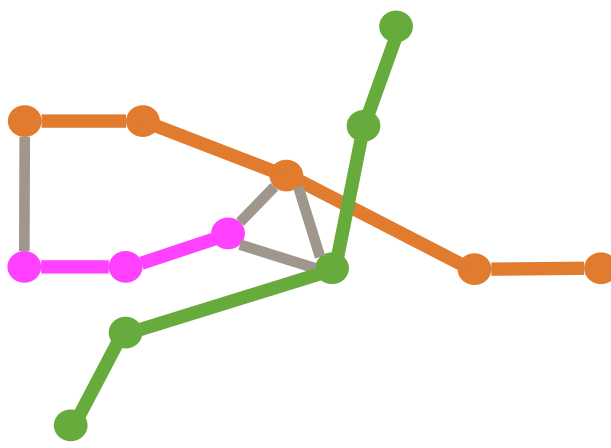
La seule différence avec un graphe non-orienté est donc que tout élément $e \in E$ a la forme (u, v) , c'est à dire une paire *ordonnée* de sommets.



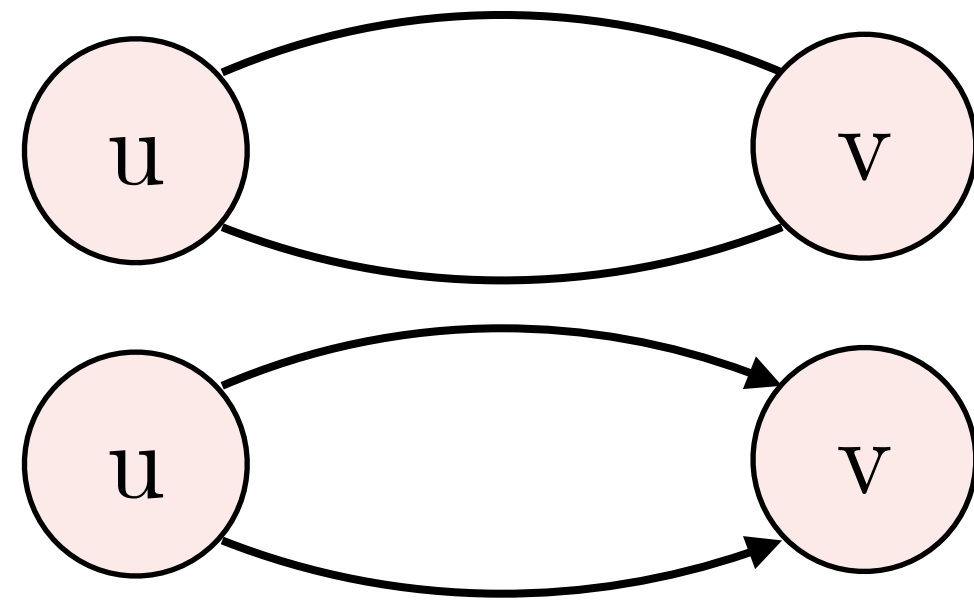
On dit alors que u est l'extrémité **initiale** et v l'extrémité **finale** de e .

Le graphe **sous-jacent** (non-ordonné) de G est obtenu en remplaçant chaque arc par la paire non-ordonnée correspondante.

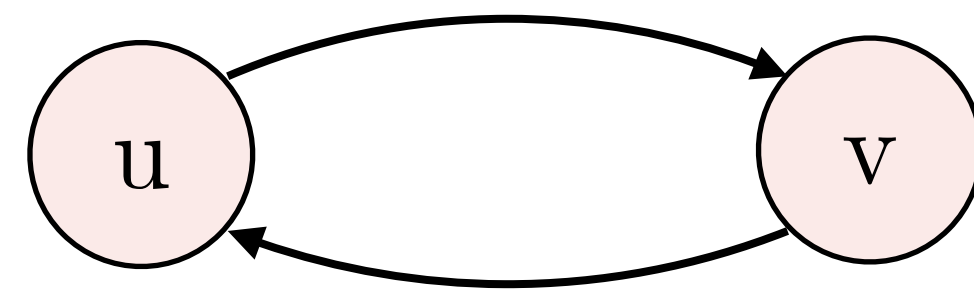
Boucles, arêtes et arcs multiples



Une **boucle** est une arête ou un arc dont les deux extrémités sont le même sommet.



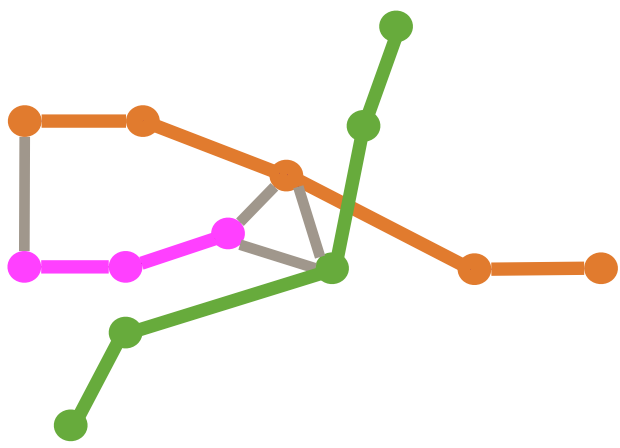
Des arêtes ou arcs sont dits **multiples** lorsqu'ils partagent les mêmes extrémités (initiales/finales).



Des arcs sont dits **opposés** *mais pas multiples* s'ils partagent les mêmes extrémités dans l'ordre inverse.

Un graphe est dit **simple** s'il n'a ni boucles ni arêtes multiples.
Sinon, on parle de **multigraphe**.

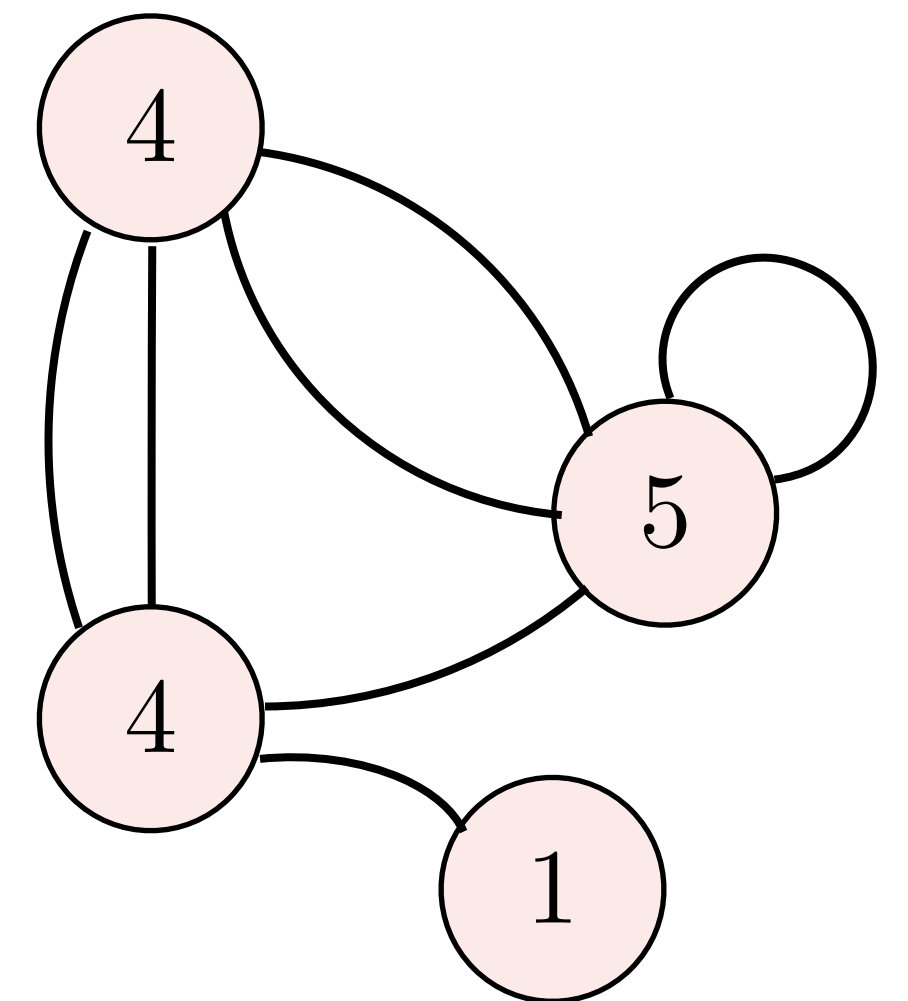
Degrés



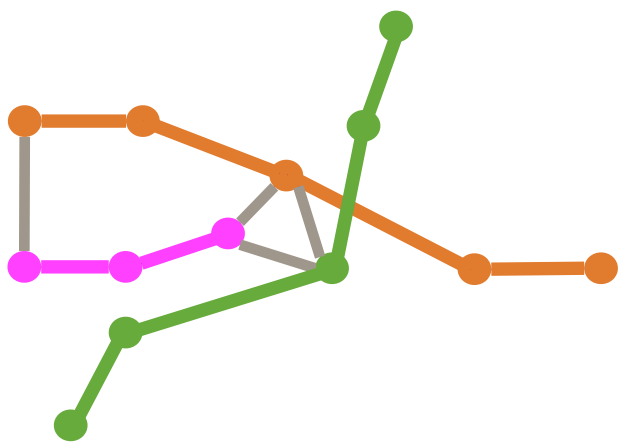
Le **degré** $\deg(v)$ d'un sommet v est le nombre d'arêtes ou arcs incidents à v , les boucles étant comptées en double.

Un sommet est dit **pendant**, s'il a un degré 1.

Remarque : On a donc $\sum_{v \in V} \deg(v) = 2 |E|$



Demi-degrés

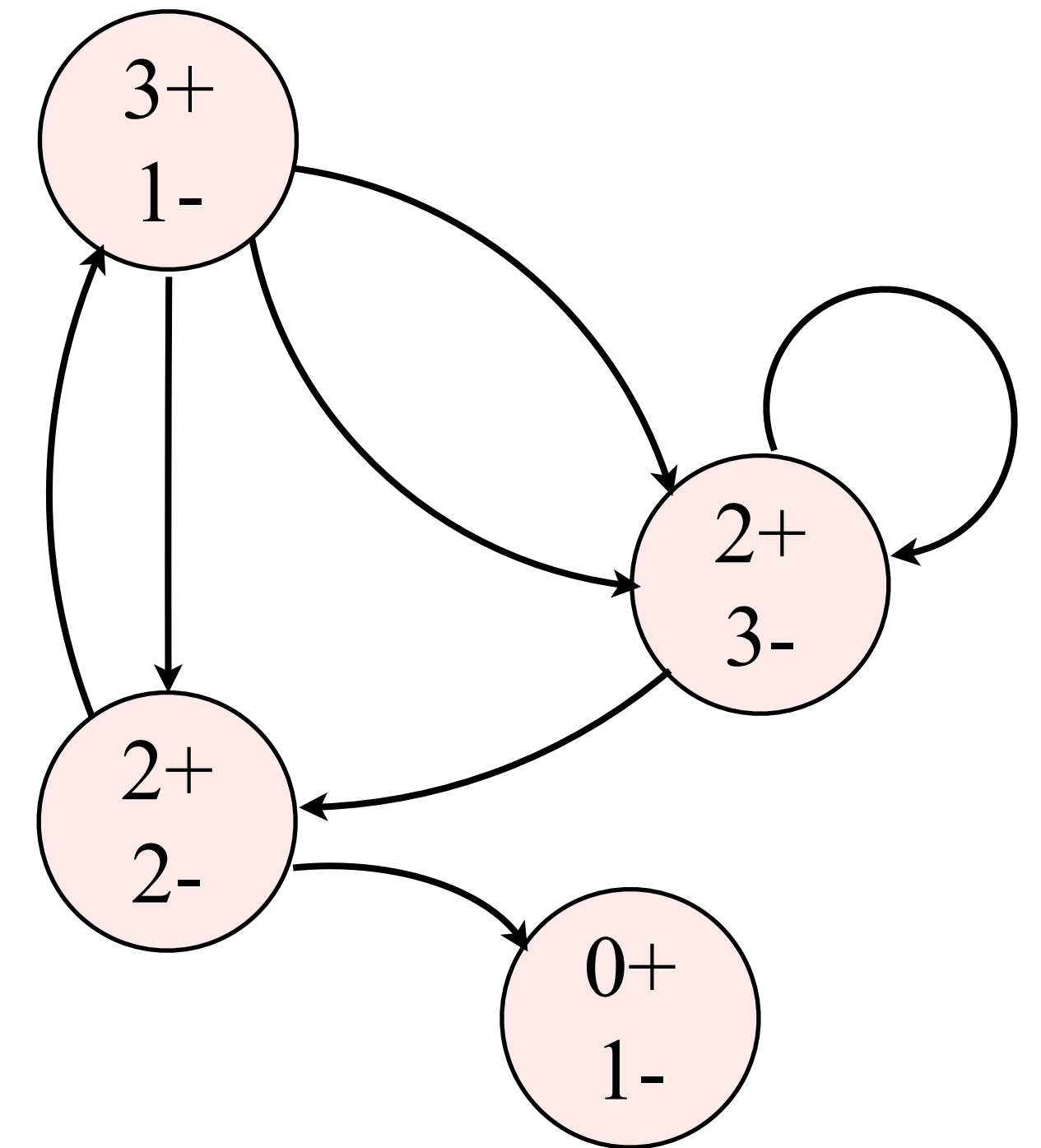


Dans un graphe orienté, on peut aussi définir

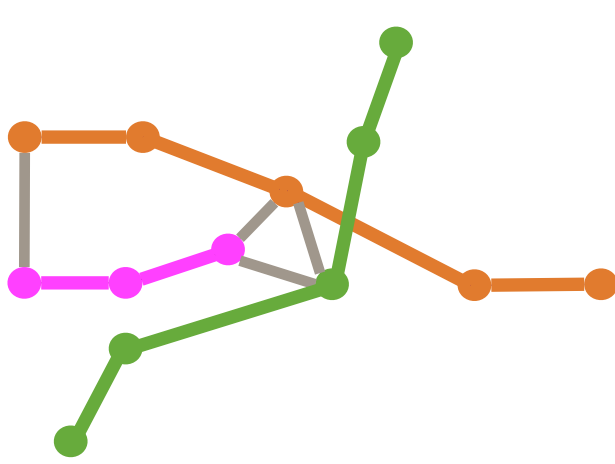
- le **demi-degré sortant** $\deg_+(v)$:
le nombre d'arcs dont v est l'extrémité initiale.
- le **demi-degré entrant** $\deg_-(v)$:
le nombre d'arcs dont v est l'extrémité finale.

Remarque : On a donc $\deg(v) = \deg_+(v) + \deg_-(v)$

$$\text{et } \sum_{v \in V} \deg_+(v) = \sum_{v \in V} \deg_-(v) = |E|$$



Chaîne et cycle



Dans un graphe non-orienté,

Une **chaîne** est une suite alternée de sommets et d'arêtes,

- commençant et terminant par un sommet,
- tel que chaque sommet est incident aux arêtes qui l'entourent.

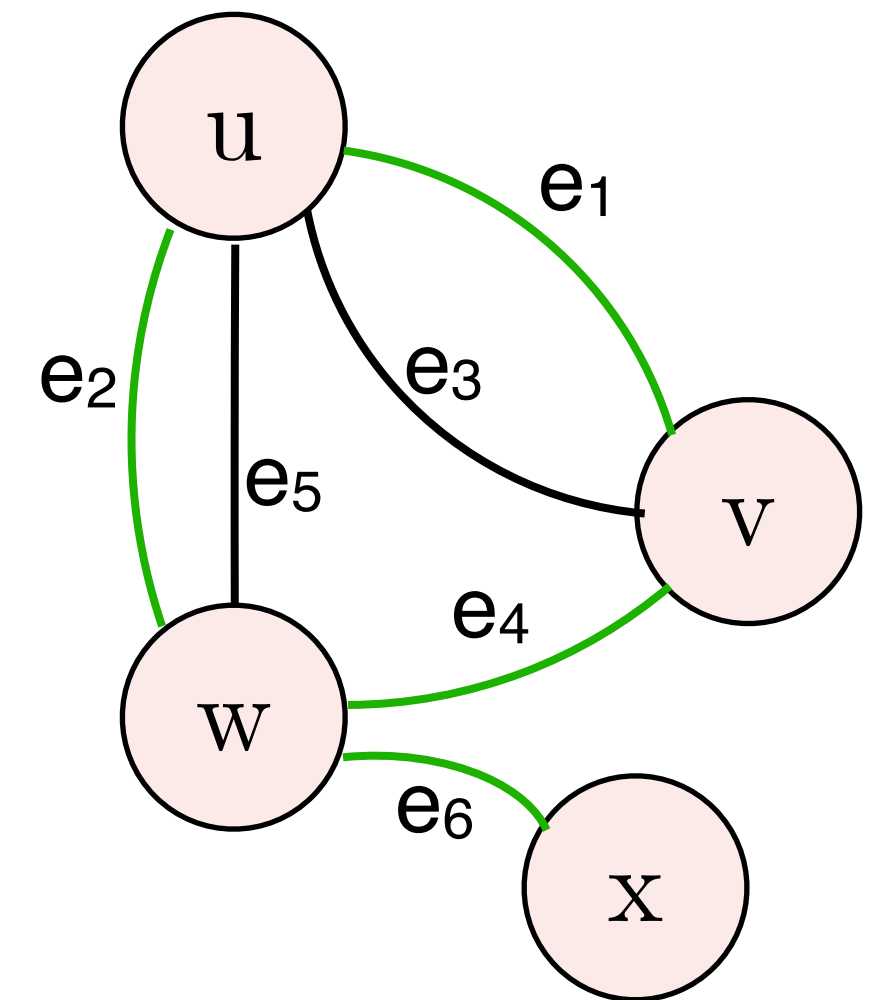
Le premier et dernier sommet sont alors dits **connectés**.

La **longueur** d'une chaîne est son nombre d'arêtes
(en comptant les doublons)

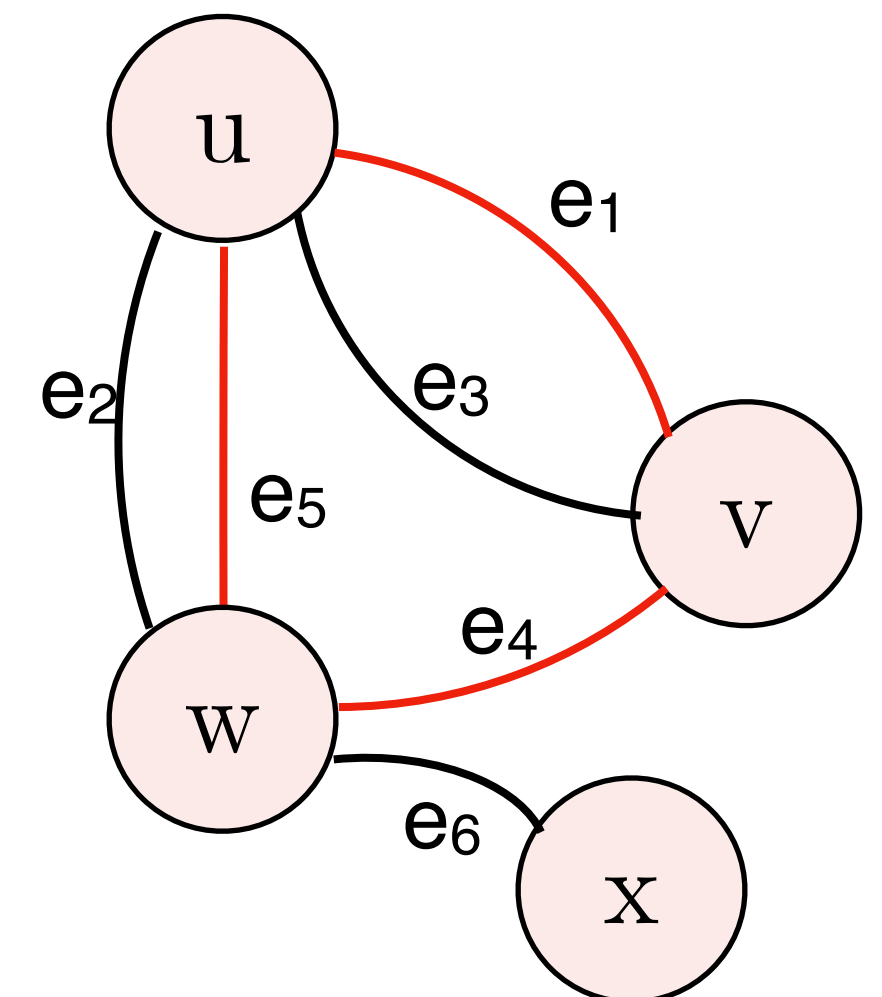
Un **cycle** est une chaîne non-nulle commençant et terminant au même sommet.

Une chaîne est **élémentaire** si aucun sommet n'y est répété, et **simple** si aucune arête n'y est répétée.

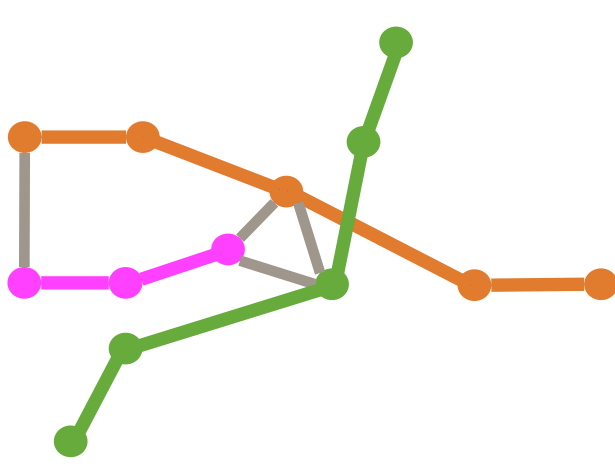
Chaîne $x-e_6-w-e_4-v-e_1-u-e_2-w$



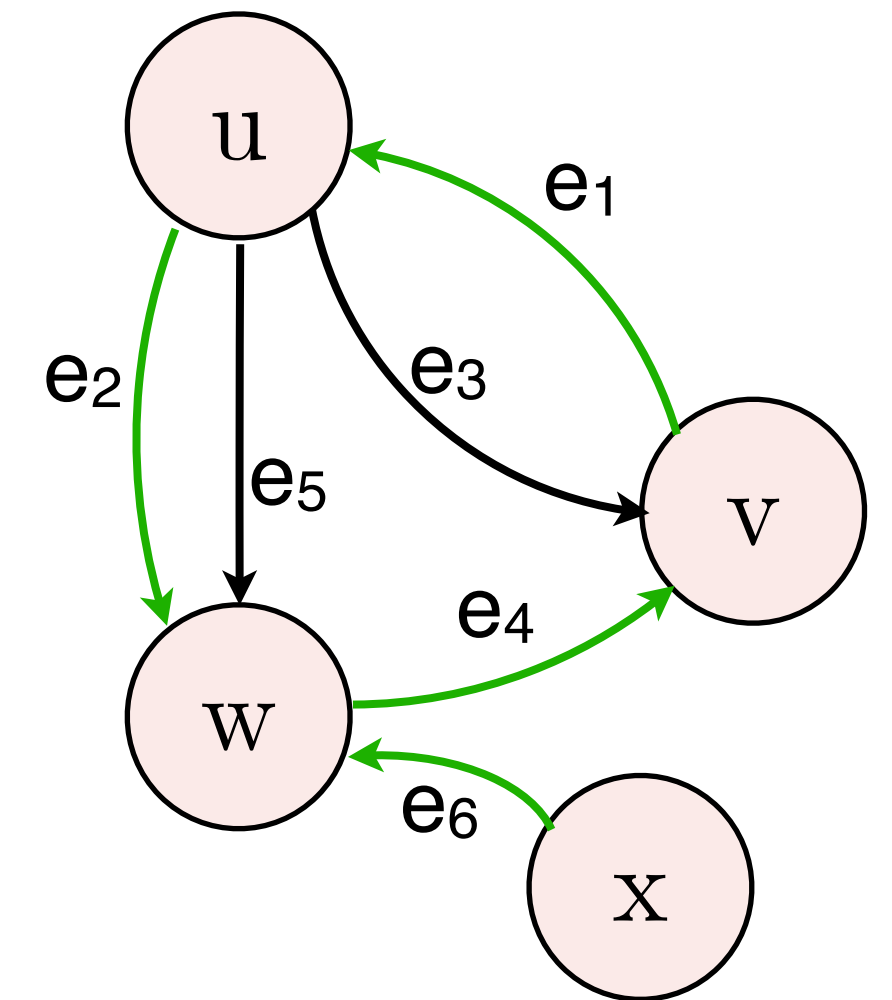
Cycle $w-e_4-v-e_1-u-e_5-w$



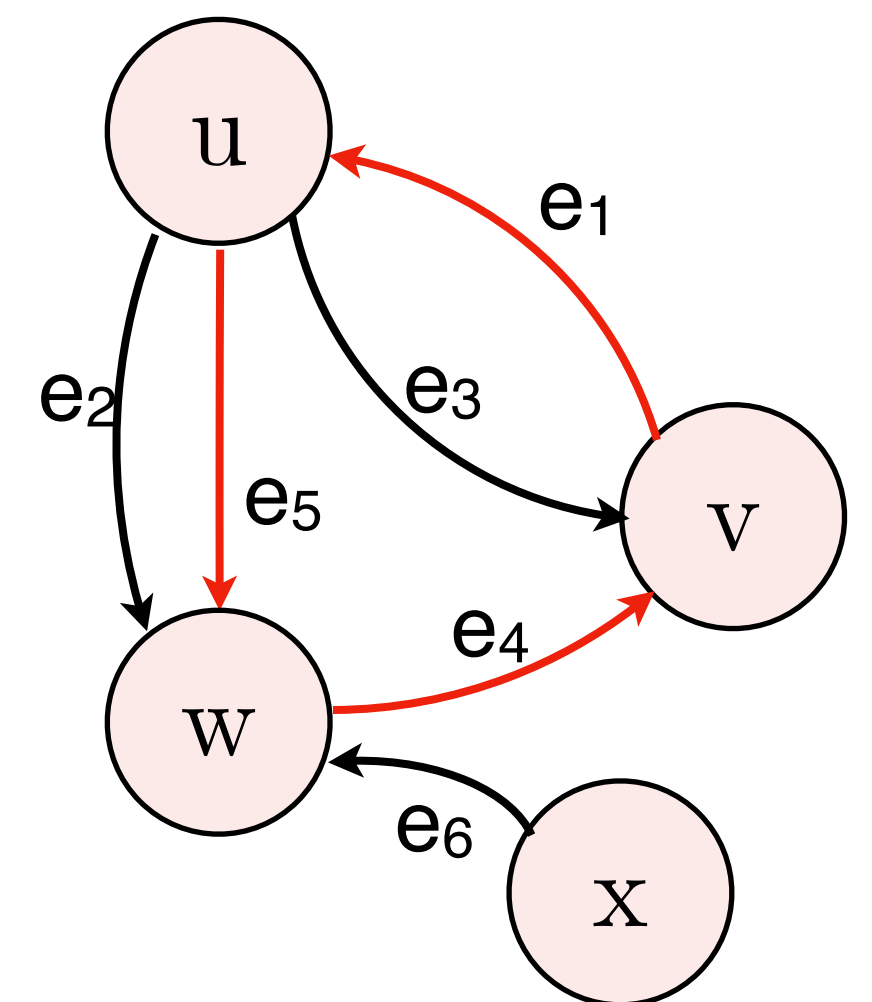
Chemin et circuit



Chemin $x-e_6-w-e_4-v-e_1-u-e_2-w$



Circuit $w-e_4-v-e_1-u-e_5-w$



Dans un graphe orienté,

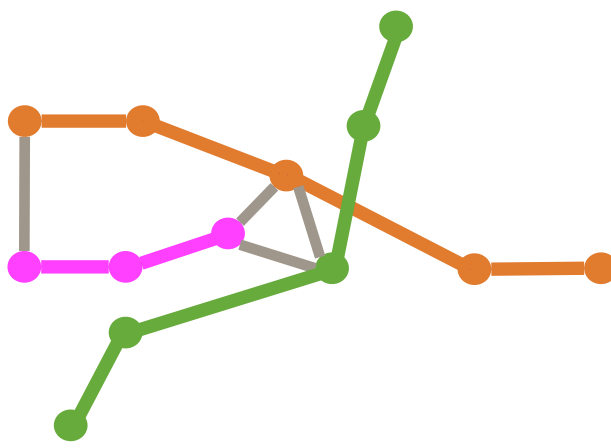
Un **chemin** est une suite alternée de sommets et d'arcs,

- commençant et terminant par un sommet,
- tel que chaque arc est précédé de son extrémité initiale, et suivi de son extrémité finale.

Un **circuit** est un chemin non-nul commençant et terminant au même sommet.

Les concepts de **longueur**, chemin **élémentaire** et **simple**, et sommets **connectés** sont définis similairement au contexte d'un graphe non-orienté.

Arbre et forêt



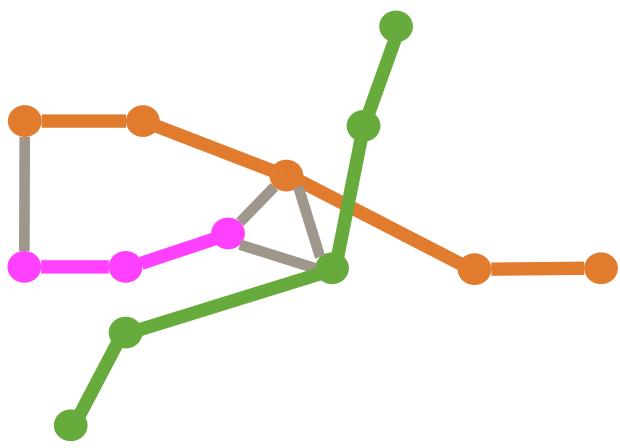
Un graphe est dit **acyclique** s'il ne contient pas de cycle.

C'est alors un **arbre** si tous deux sommets du graphe sont connectés,
une **foret** sinon.

Dans un graphe orienté tel que $\deg_-(v) \leq 1, \forall v \in V$,
on parle d'**arborescence**.

Remarque : les arbres vus au chapitre précédent correspondaient à une arborescence dans laquelle la racine était définie comme le seul sommet r tel que $\deg_-(r) = 0$, et le parent comme le sommet incident à l'unique arc entrant.

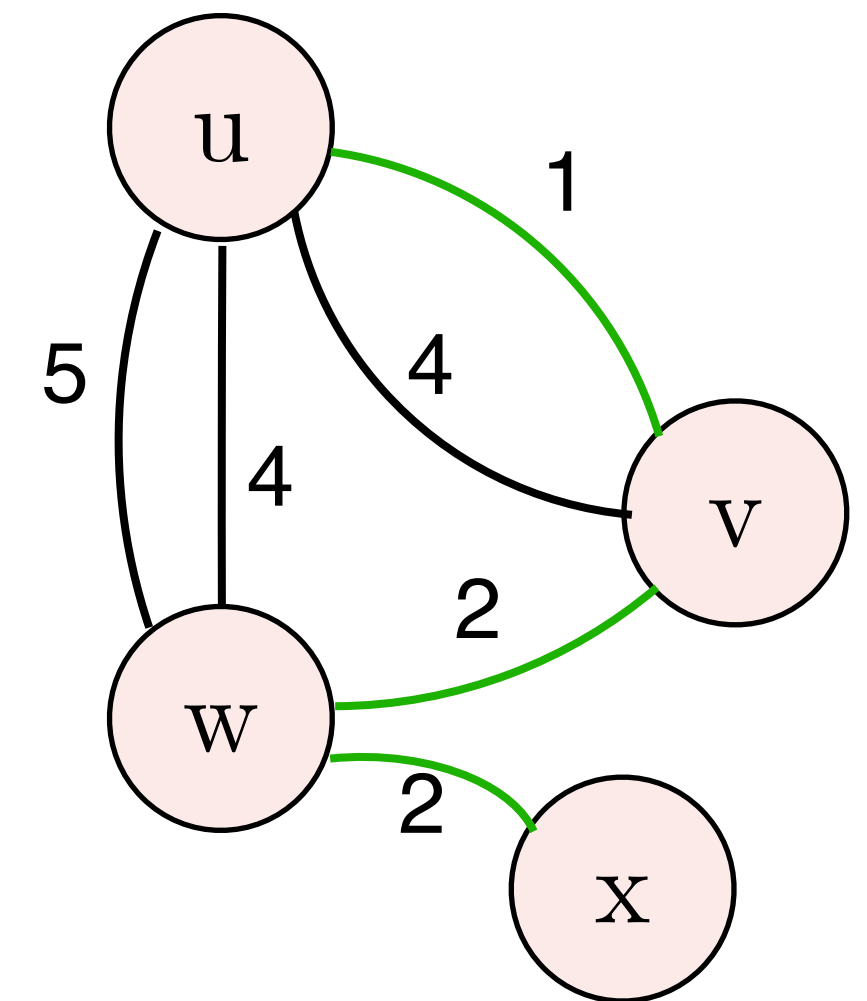
Graphe pondéré



Dans de nombreux problèmes, il est pertinent d'associer un **poids** à chaque arête / arc e du graphe, souvent dénoté $c(e)$.

On s'intéressera à des problèmes tels que

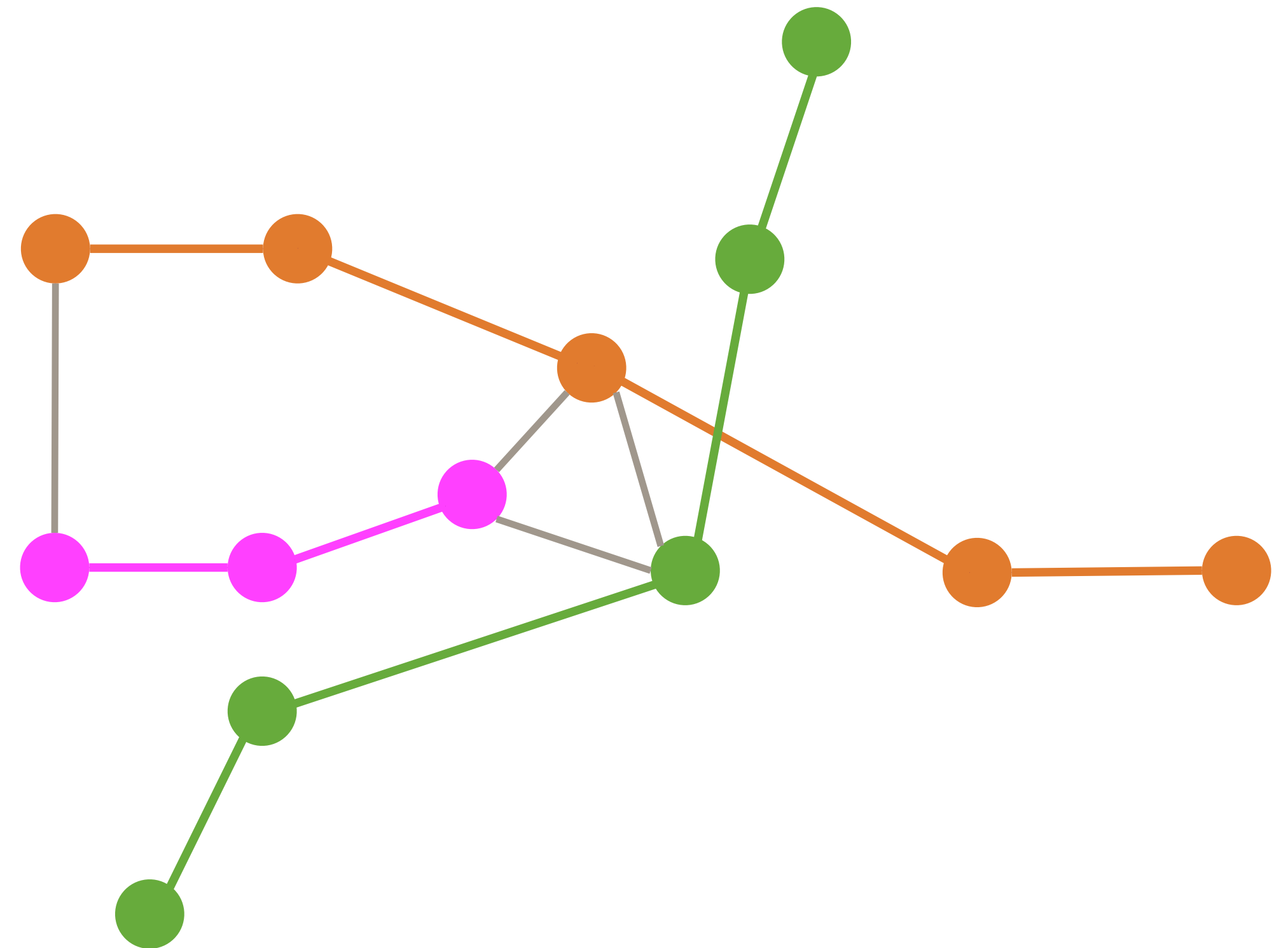
- Trouver la chaîne / le chemin **de poids minimum** entre 2 sommets
- Déterminer s'il existe des cycles / circuits **de poids négatif**
- Trouver l'**arbre couvrant de poids minimum**



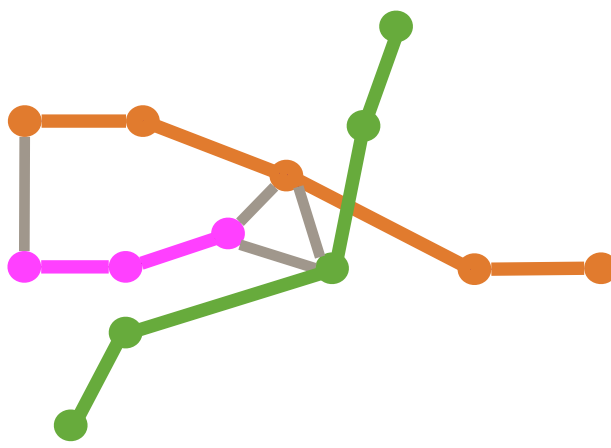
En **vert**, la chaîne de poids minimum reliant u et x .

C'est aussi l'arbre couvrant de poids minimum.

6.3. Représentation des graphes

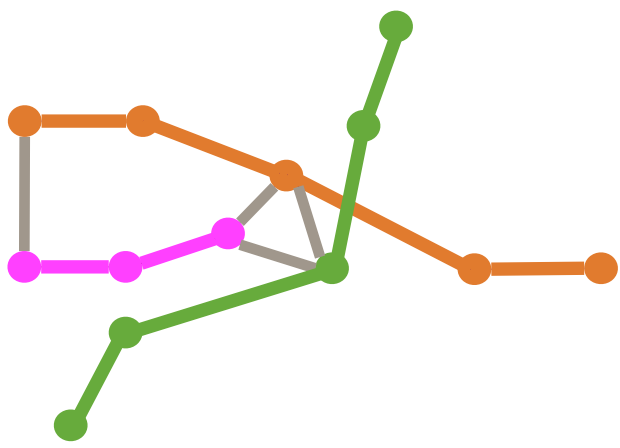


Que stocker ?



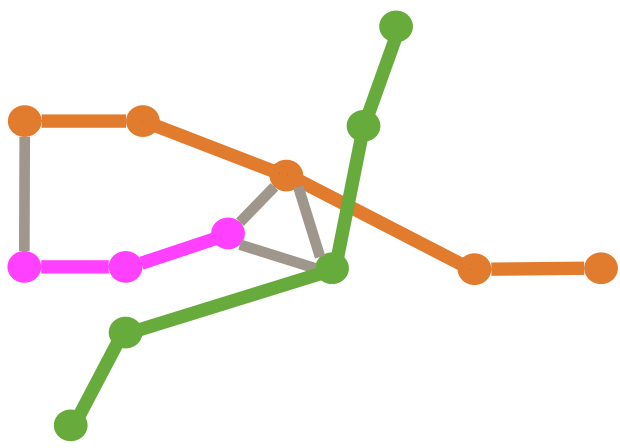
- Pour stocker nos graphes en mémoire, il nous faut une représentation
 - des **sommets**
 - des **arêtes / arcs**
 - éventuellement des **poids** associés à ces arêtes / arcs
- Cette représentation doit
 - Être **compacte**, i.e. ne pas gaspiller de mémoire
 - Permettre de **parcourir la liste** de l'ensemble **des sommets**
 - Permettre d'accéder facilement aux arêtes / arcs **incidents à un sommet**, ou directement à ses **sommets adjacents**
- Certaines représentations sont également utiles pour représenter mathématiquement les graphes et les algorithmes sur ces graphes. Ce n'est pas notre sujet en ASD.

Les sommets



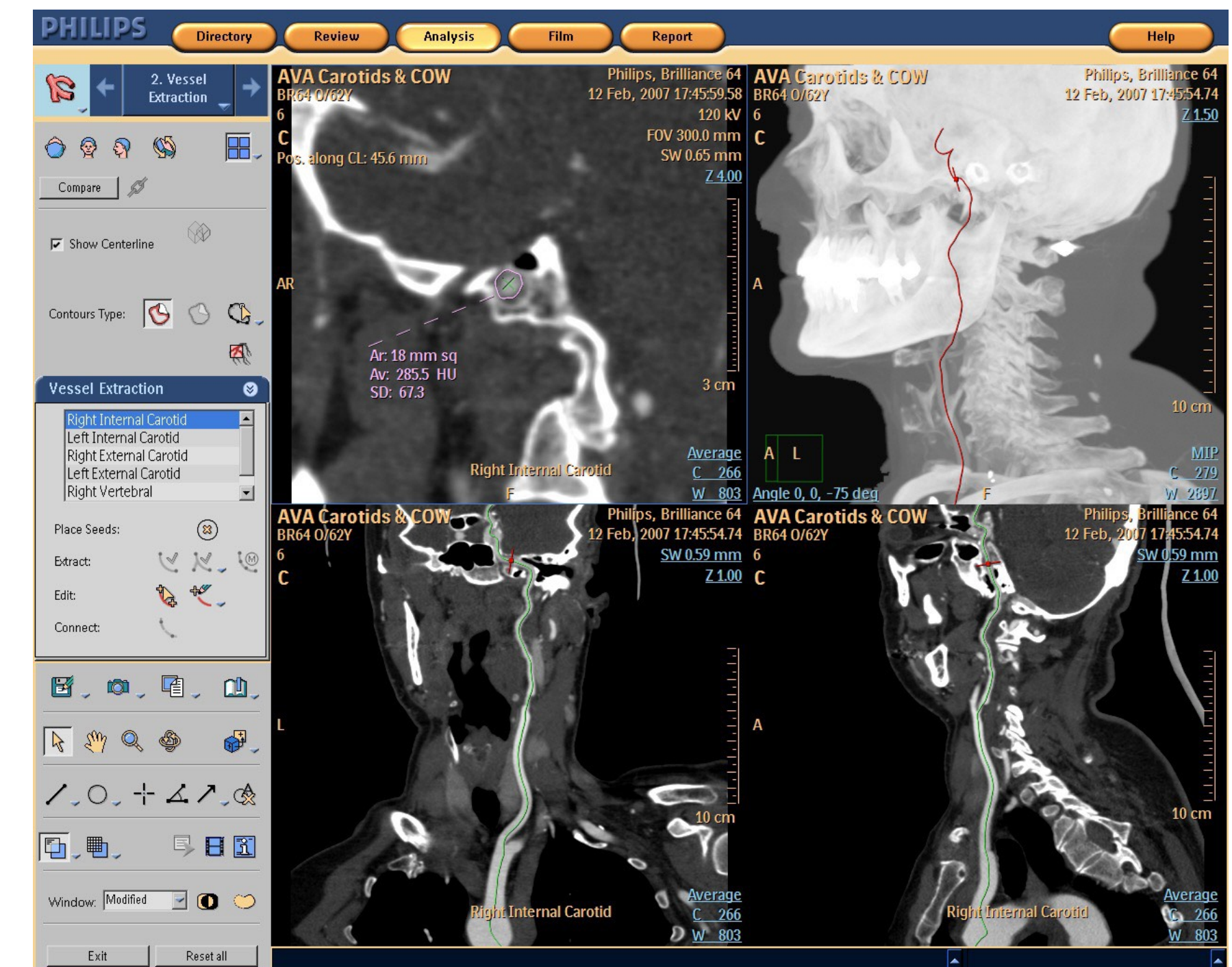
- Si c'est possible, on se contente de **numéroter** les $n = |V|$ sommets de 0 à $n - 1$. Cela permet de
 - ne pas devoir créer une structure explicite pour ces sommets
 - stocker les données associées - y compris incidence et adjacence - dans des **structures linéaires** et d'y accéder via leur indice
- Si ce n'est pas possible (insertion et suppression de sommets autorisées, type non mappable sur des entiers successifs, ...), il faut éventuellement utiliser des types de données alternatifs tels que
 - TDA **ensemble** pour stocker les sommets (i.e. `std::set` en C++)
 - TDA **tableau associatif** pour stocker les données associées (i.e. `std::map` en C++)

Exemples 1 & 2



	CHF	EUR	USD	GBP
CHF	1	0.829	1.102	0.658
EUR	1.206	1	1.295	0.793
USD	0.907	0.772	1	0.612
GBP	1.519	1.26	1.632	1

- Pour le tableau de change, il suffit d'associer un indice à chaque monnaie
- Pour le volume de $w \times h \times s$ voxels du scanner CT, on peut calculer l'indice i du voxel (x, y, z) via $i = x + w \cdot (y + h \cdot z)$

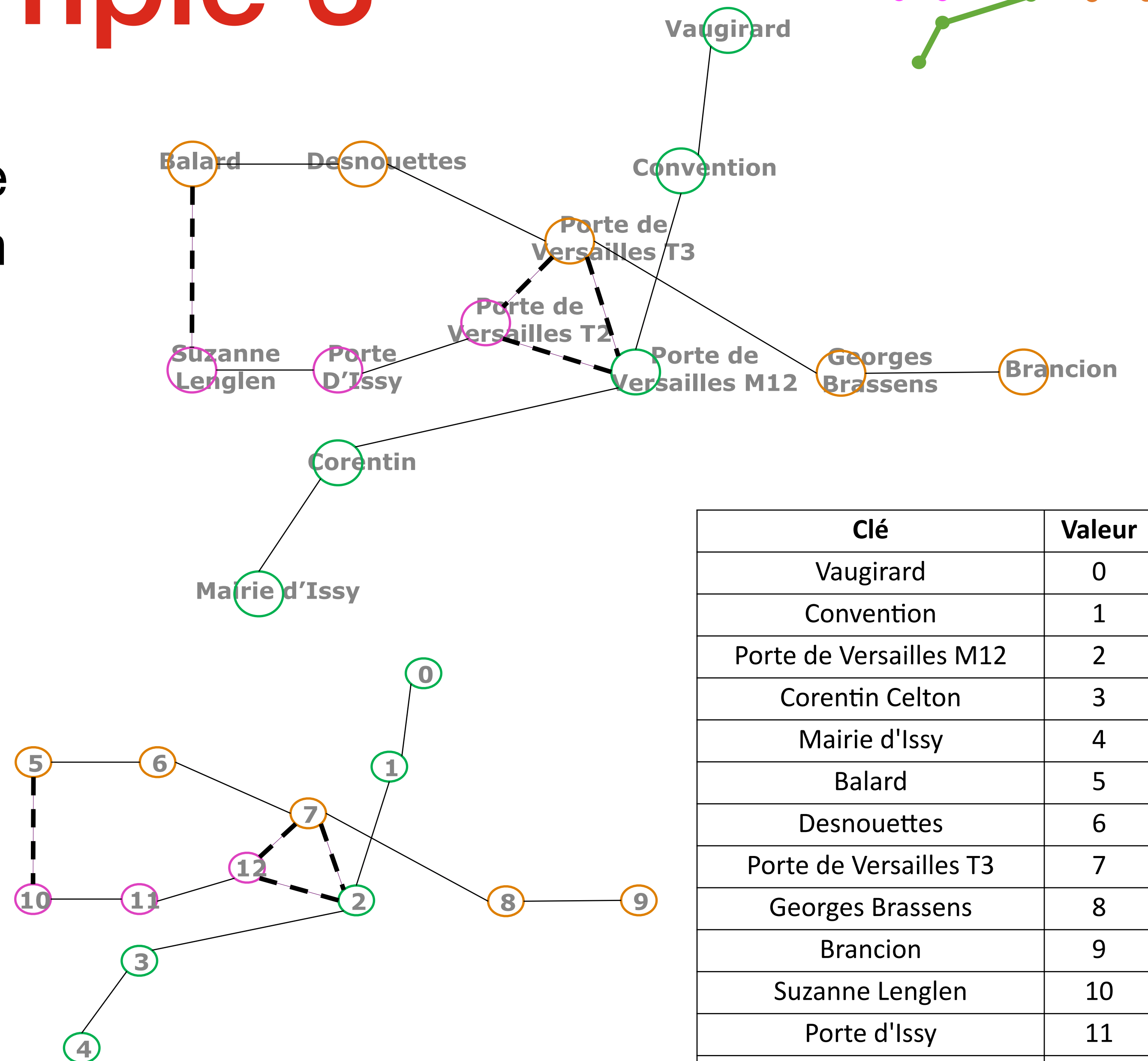


Exemple 3

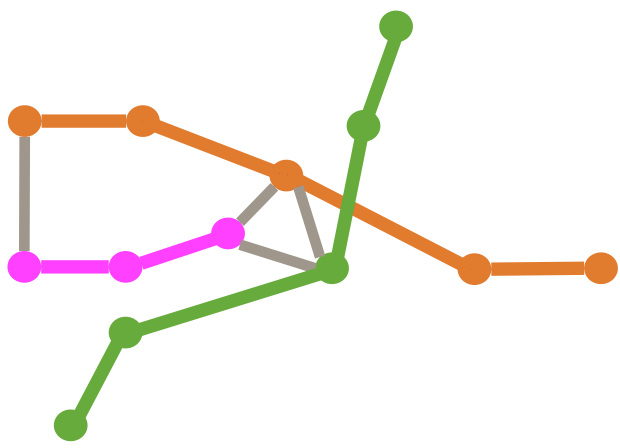
- Pour le métro, on peut par exemple assigner un indice à chaque station de métro, et les retrouver via une table de symboles

```
std::vector<string> noms;
noms[0] = "Vaugirard";
noms[1] = "Convention";
...

std::map<string, int> indices;
for(size_t i = 0; i<noms.size(); ++i) {
    indices[noms[i]] = i;
}
```

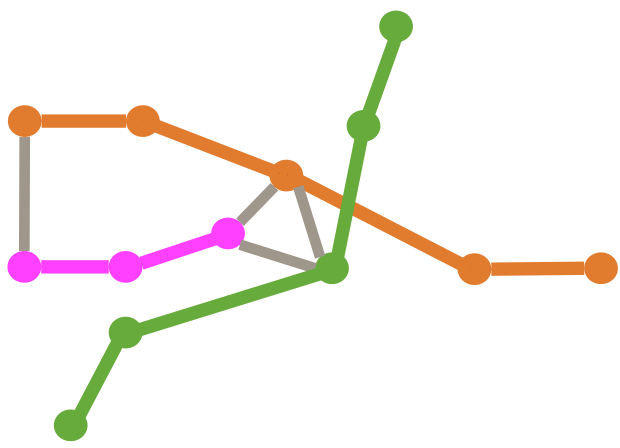


Arêtes / arcs



- Il faut pouvoir efficacement ...
 - accéder à la liste de toutes les arêtes $\{v, w\}$ / arcs (v, w)
 - savoir si deux sommets v et w sont adjacents / s'il y a un arc allant de v à w
 - parcourir la liste des sommets reliés à un sommet v via une arête ou un arc.
- La mise en oeuvre peut être
 - explicite via une **matrice d'adjacence**
 - explicite via des **listes d'adjacence**
 - **implicite** si l'on peut calculer les sommets adjacents de par la nature du problème

Matrice d'adjacence

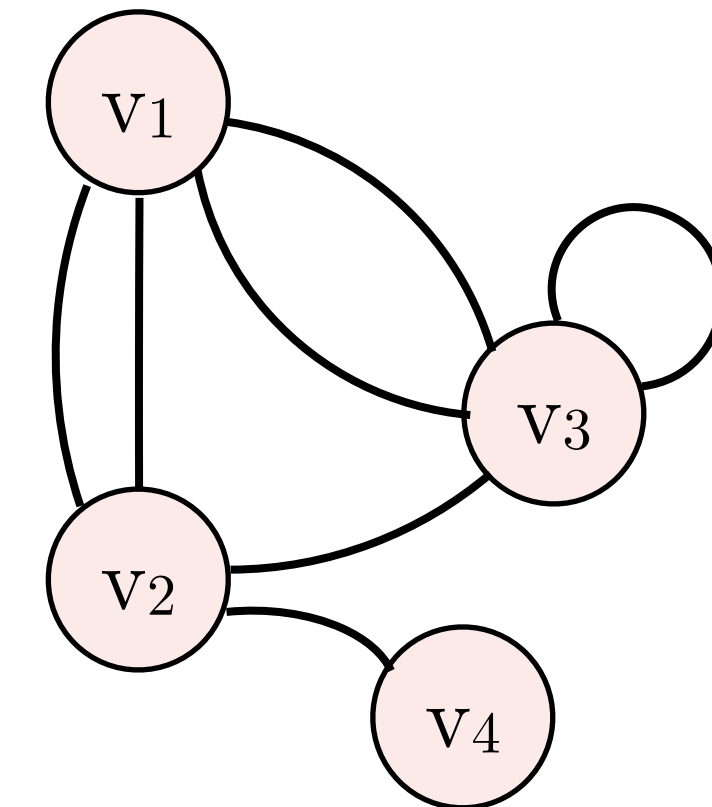


Supposons un graphe $G = (V, E)$ à $n = |V|$ sommets.

La matrice d'adjacence A est $n \times n$.

Graphe non-orienté

a_{ij} stocke le nombre d'arêtes $\{v_i, v_j\}$.

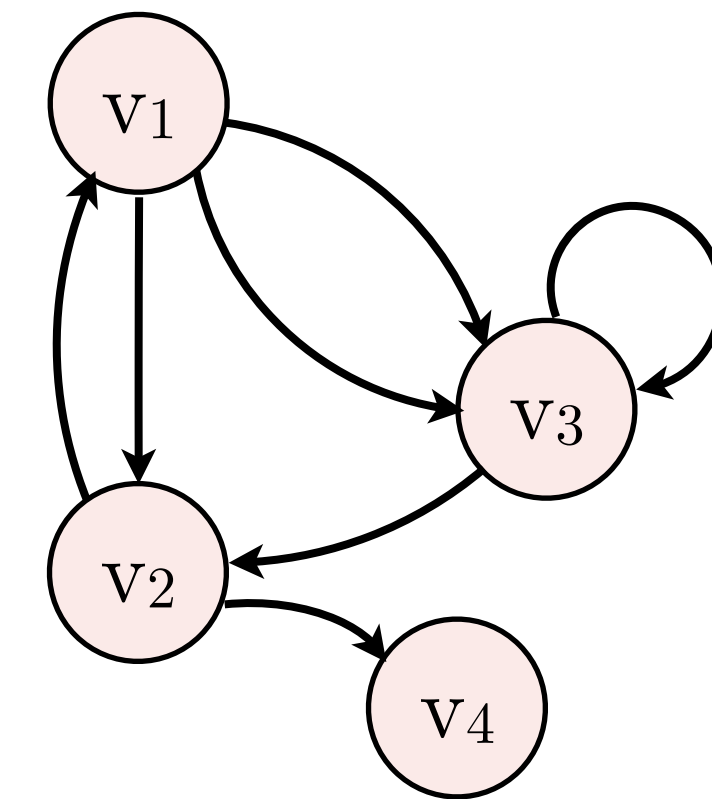


$$A = \begin{bmatrix} 0 & 2 & 2 & 0 \\ 2 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

- Remarques :
- A est symétrique
 - Si G est simple, a_{ij} peut être un booléen.

Graphe orienté

a_{ij} stocke le nombre d'arcs (v_i, v_j) .



$$A = \begin{bmatrix} 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- Remarques :
- A n'est pas forcément symétrique
 - La ligne i liste les sommets atteignables de v_i

```
class Graphe {
    vector<vector<bool>> mAdjacence;

public:
    Graphe(unsigned n) {
        mAdjacence.resize(n, vector<bool>(n, false));
    }

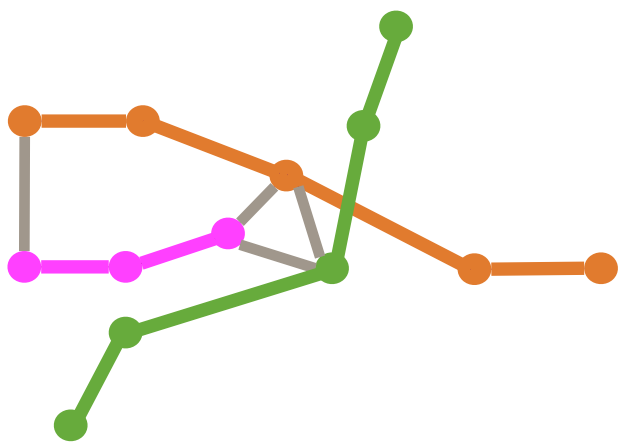
    void ajouterArete( size_t i, size_t j ) {
        mAdjacence[i][j] = mAdjacence[j][i] = true;
    }

    bool sontAdjacents( size_t i, size_t j ) const {
        return mAdjacence[i][j];
    }

    int nSommets() const {
        return mAdjacence.size();
    }

    vector<int> sommetsAdjacents(int i) const {
        vector<int> adj;
        auto const& ligne = mAdjacence[i];
        for (int j = 0; j < nSommets(); ++j)
            if (sontAdjacent(i,j)) adj.push_back(j);
        return adj;
    }
};
```

Graphe non orienté simple en C++



Exemple de code client :

```
int degreSommet(Graphe const& G, int v) {
    int deg = 0;
    for (int w : G.sommetsAdjacents(v))
        ++deg;
    return deg;
}

int degreMaxGraphe(Graphe const& G) {
    int degMax = 0;
    for (int v = 0; v < G.nSommets(); ++v)
        degMax = max(degMax,
                      degreSommet(G, v));
    return dm;
}
```

```
class Graphe {
    vector<vector<unsigned>> mAdjacence;

public:
    Graphe(unsigned n) {
        mAdjacence.resize(n, vector<unsigned>(n, 0));
    }

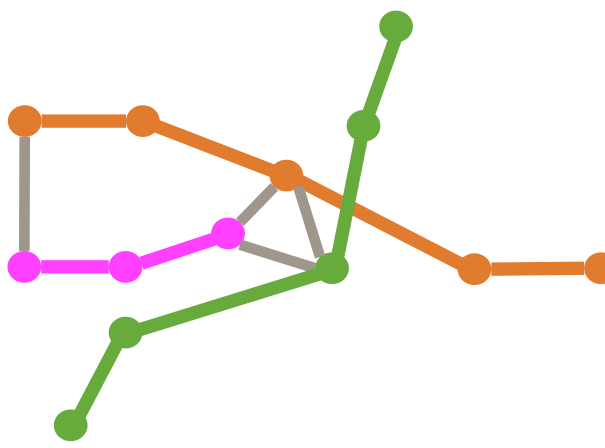
    void ajouterArc( size_t i, size_t j ) {
        ++mAdjacence[i][j];
    }

    unsigned nArcs( size_t i, size_t j ) const {
        return mAdjacence[i][j];
    }

    int nSommets() const {
        return mAdjacence.size();
    }

    vector<int> successeurs(int i) const {
        vector<int> adj;
        for (int j = 0; j < nSommets(); ++j)
            for( int k = 0; k < mAdjacence[i][j]; ++k )
                adj.push_back(j);
        return adj;
    }
};
```

Multigraphe orienté en C++

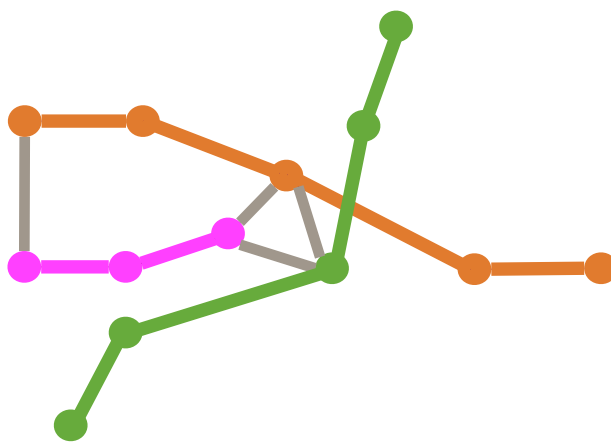


Exemple de code client :

```
int degreSortant(Graphe const& G, int v) {
    int deg = 0;
    for (int w : G.successeurs(v))
        ++deg;
    return deg;
}

int degreEntrant(Graphe const& G, int v) {
    int deg = 0;
    for (int w = 0; w < G.nSommets(); ++w)
        deg += G.nArcs(w, v);
    return deg;
}
```

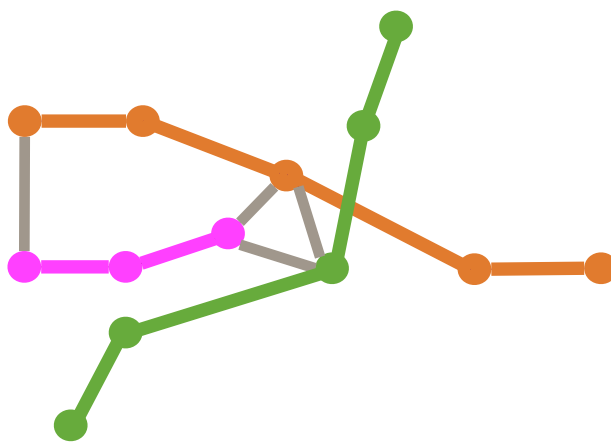
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*), représenté par matrice d'adjacence,

- **Stocker** la matrice a un coût mémoire

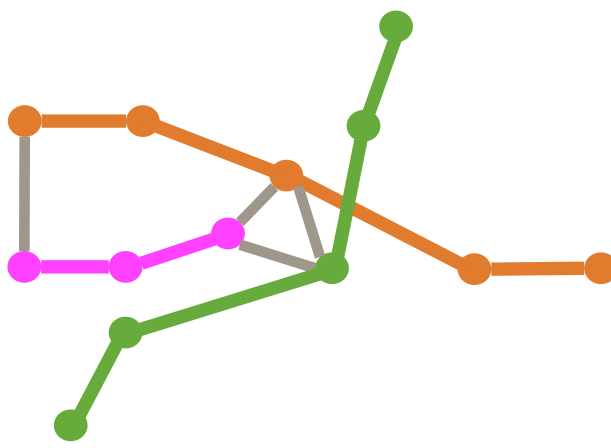
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*), représenté par matrice d'adjacence,

- **Stocker** la matrice a un coût mémoire $O(n^2)$
- **Savoir si** $\{v_i, v_j\}$ est une arête (*ou* (v_i, v_j) est un arc) s'effectue en

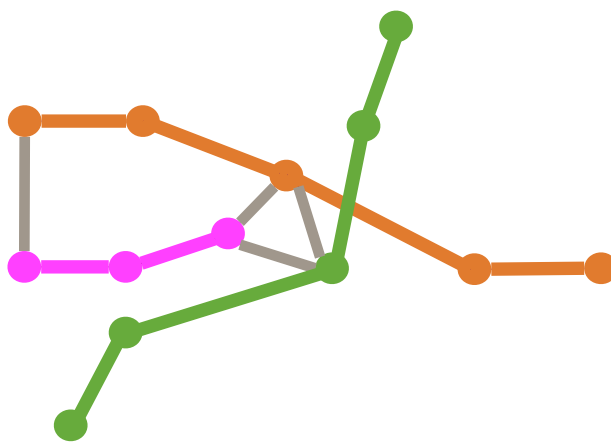
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*), représenté par matrice d'adjacence,

- **Stocker** la matrice a un coût mémoire $O(n^2)$
- **Savoir si** $\{v_i, v_j\}$ est une arête (*ou* (v_i, v_j) est un arc) s'effectue en $O(1)$
- **Parcourir** toutes les arêtes incidentes à v_i (*ou arcs dont* v_i est l'extrémité initiale) s'effectue

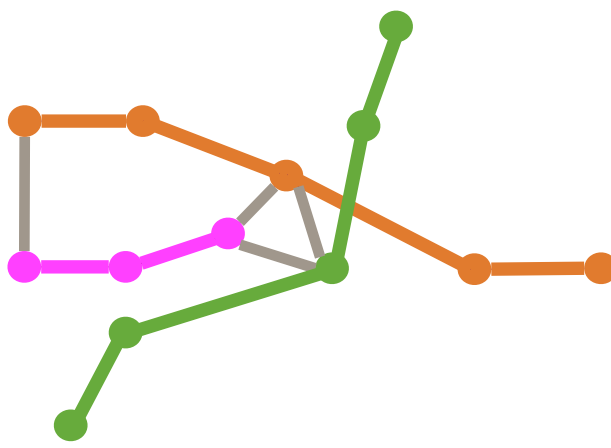
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*), représenté par matrice d'adjacence,

- **Stocker** la matrice a un coût mémoire $O(n^2)$
- **Savoir si** $\{v_i, v_j\}$ est une arête (*ou* (v_i, v_j) est un arc) s'effectue en $O(1)$
- **Parcourir** toutes les arêtes incidentes à v_i (*ou arcs dont* v_i est l'extrémité initiale) s'effectue en parcourant **une ligne de la matrice** en $O(n)$
- **Parcourir** tous les arcs dont v_i est l'extrémité finale s'effectue en

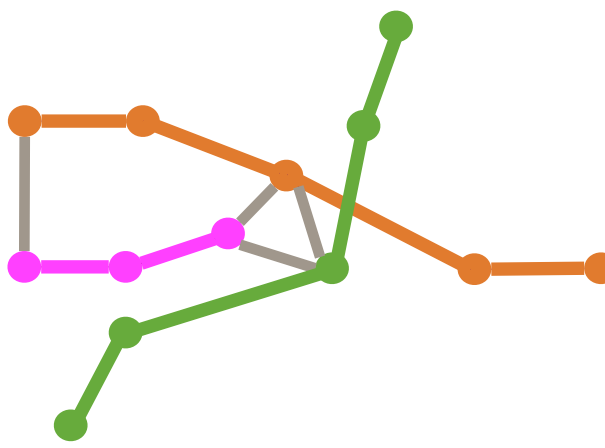
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*), représenté par matrice d'adjacence,

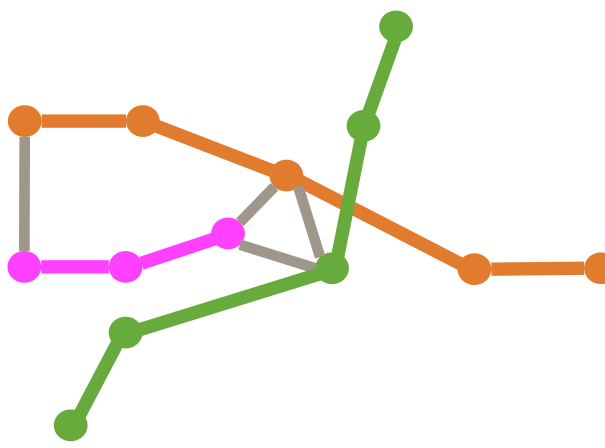
- **Stocker** la matrice a un coût mémoire $O(n^2)$
- **Savoir si** $\{v_i, v_j\}$ est une arête (*ou* (v_i, v_j) est un arc) s'effectue en $O(1)$
- **Parcourir** toutes les arêtes incidentes à v_i (*ou arcs dont* v_i est l'extrémité initiale) s'effectue en parcourant **une ligne de la matrice en** $O(n)$
- **Parcourir** tous les arcs dont v_i est l'extrémité finale s'effectue en parcourant **une colonne de la matrice en** $O(n)$
- **Parcourir toutes** les arêtes (arcs) du graphe s'effectue en

Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*), représenté par matrice d'adjacence,

- **Stocker** la matrice a un coût mémoire $O(n^2)$
- **Savoir si** $\{v_i, v_j\}$ est une arête (*ou* (v_i, v_j) est un arc) s'effectue en $O(1)$
- **Parcourir** toutes les arêtes incidentes à v_i (*ou arcs dont* v_i est l'extrémité initiale) s'effectue en parcourant **une ligne de la matrice en** $O(n)$
- **Parcourir** tous les arcs dont v_i est l'extrémité finale s'effectue en parcourant **une colonne de la matrice en** $O(n)$
- **Parcourir toutes** les arêtes (arcs) du graphe s'effectue en $O(n^2)$



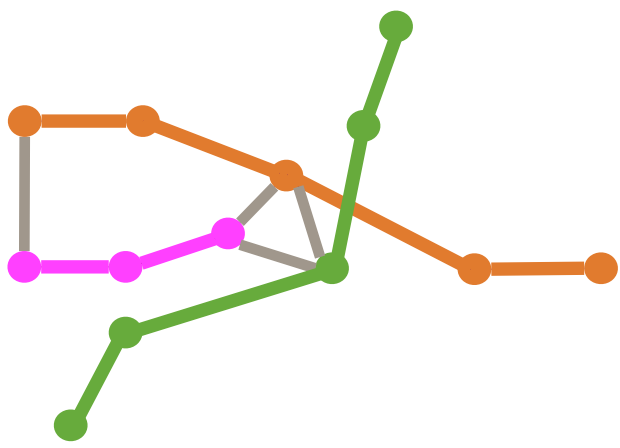
La représentation par matrice d'adjacence n'est efficace que pour des graphes complets ou presque, pour lesquels $|E| \approx |V|^2$

Pour les graphes simples pondérés, on peut aussi directement stocker le poids dans la matrice d'adjacence, en réservant pour les graphes non complets une valeur indiquant l'absence d'arête.

	CHF	EUR	USD	GBP
CHF	1	0.829	1.102	0.658
EUR	1.206	1	1.295	0.793
USD	0.907	0.772	1	0.612
GBP	1.519	1.26	1.632	1

C'est le cas pour notre exemple d'arbitrage des taux de change

Listes d'adjacence / de successeurs



Supposons un graphe $G = (V, E)$ à $n = |V|$ sommets.
 G peut être stocké comme n listes de sommets.

Graphe non-orienté

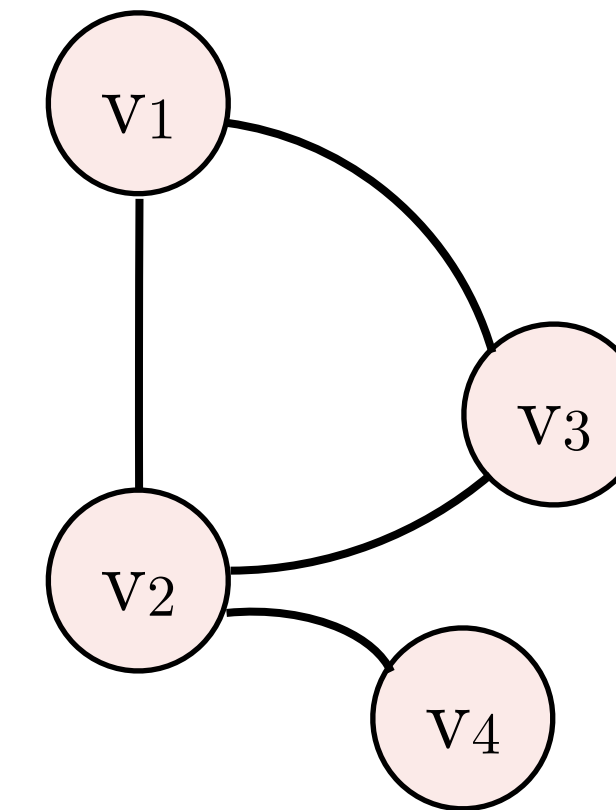
$Adj[i]$ stocke tous les j tels que $\{v_i, v_j\} \in E$.

Graphe orienté

$Succ[i]$ stocke tous les j tels que $(v_i, v_j) \in E$.

Remarque :

- Pour les graphes pondérés, on peut stocker des paires indice/poids.

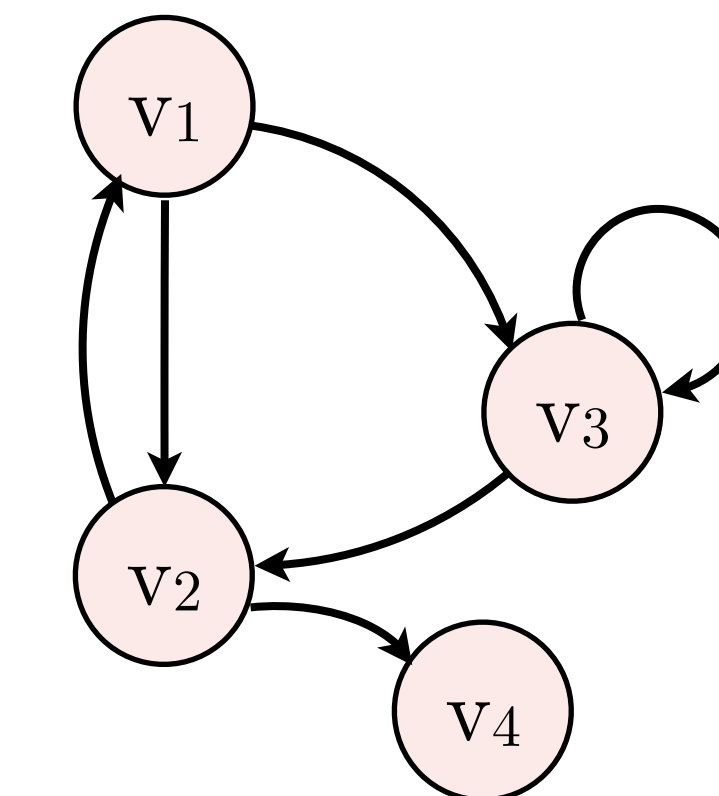


$$Adj[1] = \{2,3\}$$

$$Adj[2] = \{1,3,4\}$$

$$Adj[3] = \{1,2\}$$

$$Adj[4] = \{2\}$$



$$Succ[1] = \{2,3\}$$

$$Succ[2] = \{1,4\}$$

$$Succ[3] = \{2,3\}$$

$$Succ[4] = \{\}$$

```
class Graphe {
    using ListeAdj = vector<unsigned>;
    vector<ListeAdj> listesAdj;
public:
    Graphe(unsigned n) {
        listesAdj.resize(n);
    }

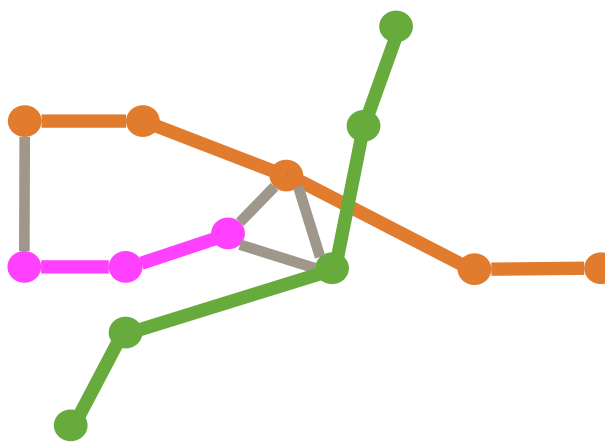
    void ajoutArete( size_t i, size_t j ) {
        listesAdj[i].push_back(j);
        listesAdj[j].push_back(i);
    }

    bool sontAdjacents( size_t i, size_t j ) const {
        return std::find(listesAdj[i].begin(),
                        listesAdj[i].end(), j)
            != listesAdj[i].end();
    }

    int nSommets() const {
        return listesAdj.size();
    }

    ListeAdj const& sommetsAdjacents(int i) const {
        return listesAdj[i];
    }
};
```

Multigraphe non orienté en C++



Exemple de code client :

```
int degreSommet(Graphe const& G, int v) {
    int deg = 0;
    for (int w : G.sommetsAdjacents(v))
        ++deg;
    return deg;
}

int degreMaxGraphe(Graphe const& G) {
    int degMax = 0;
    for (int v = 0; v < G.nSommets(); ++v)
        degMax = max(degMax,
                    degreSommet(G, v));
    return dm;
}
```

```
class Graphe {
    using ListeSucc = set<unsigned>;
    vector<ListeSucc> succ;
public:
    Graphe(unsigned n) {
        succ.resize(n);
    }

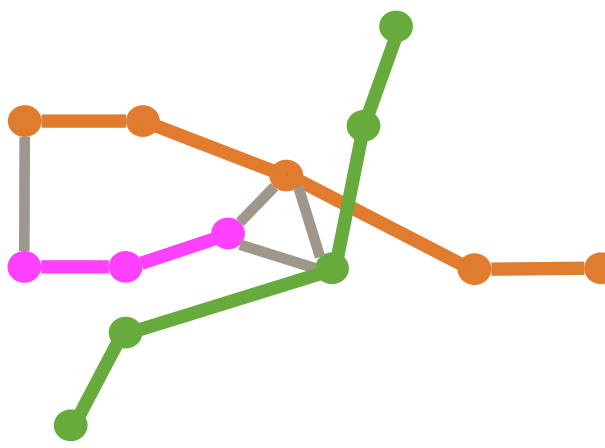
    void ajoutArc( size_t i, size_t j ) {
        succ[i].insert(j);
    }

    bool estArc( size_t i, size_t j ) const {
        return std::find(succ[i].begin(),
                        succ[i].end(), j)
               != succ[i].end();
    }

    int nSommets() const {
        return succ.size();
    }

    ListeSucc const& successeurs(int i) const {
        return succ[i];
    }
};
```

Graphe simple orienté en C++

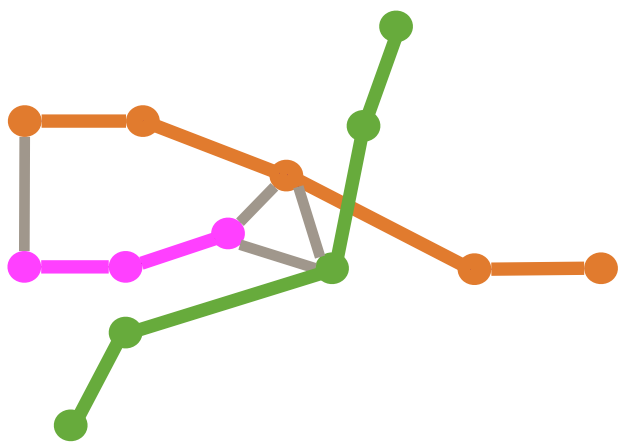


Exemple de code client :

```
int degreSortant(Graphe const& G, int v) {
    int deg = 0;
    for (int w : G.successeurs(v))
        ++deg;
    return deg;
}

int degreEntrant(Graphe const& G, int v) {
    int deg = 0;
    for (int w = 0; w < G.nSommets(); ++w)
        deg += G.estArc(w,v);
    return deg;
}
```

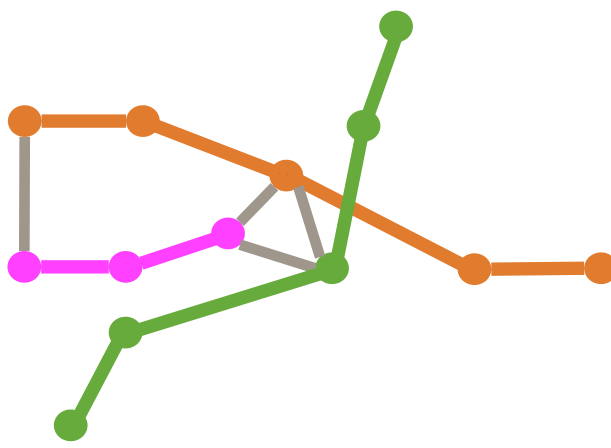
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*) représenté par listes d'adjacence (*de successeurs*),

- **Stocker** les listes a un coût mémoire de

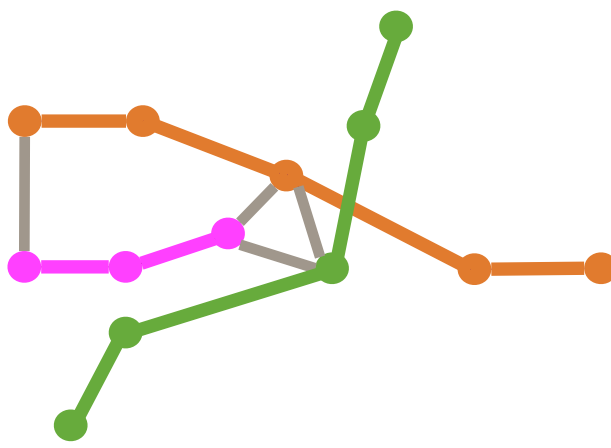
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*) représenté par listes d'adjacence (*de successeurs*),

- **Stocker** les listes a un coût mémoire de $O(n + m)$
- **Savoir si** $\{v_i, v_j\}$ est une arête s'effectue en

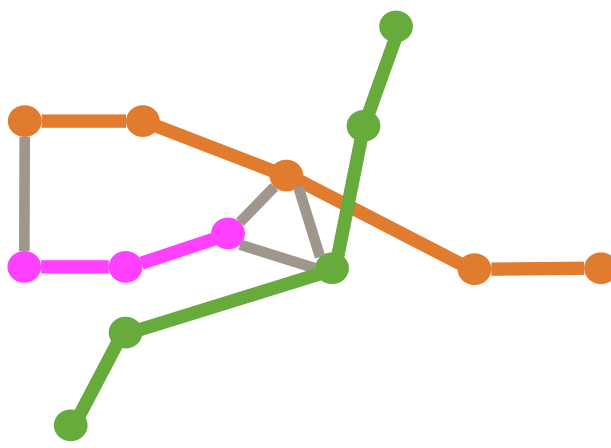
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*) représenté par listes d'adjacence (*de successeurs*),

- **Stocker** les listes a un coût mémoire de $O(n + m)$
- **Savoir si** $\{v_i, v_j\}$ est une arête s'effectue en $O(\deg(v_i))$
si (v_i, v_j) est un arc en $O(\deg_+(v_i))$
- **Parcourir** les arêtes incidentes à v_i s'effectue en

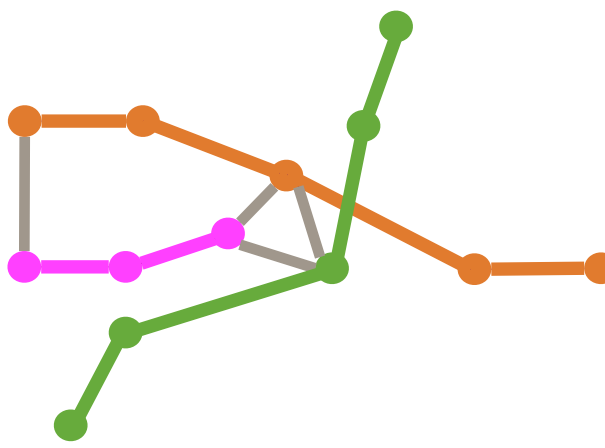
Complexités



Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*) représenté par listes d'adjacence (*de successeurs*),

- **Stocker** les listes a un coût mémoire de $O(n + m)$
- **Savoir si** $\{v_i, v_j\}$ est une arête s'effectue en $O(\deg(v_i))$
si (v_i, v_j) est un arc en $O(\deg_+(v_i))$
- **Parcourir** les arêtes incidentes à v_i s'effectue en $O(\deg(v_i))$
les arcs partants de v_i en $O(\deg_+(v_i))$
- **Parcourir toutes** les arêtes (*arcs*) du graphe s'effectue en

Complexités

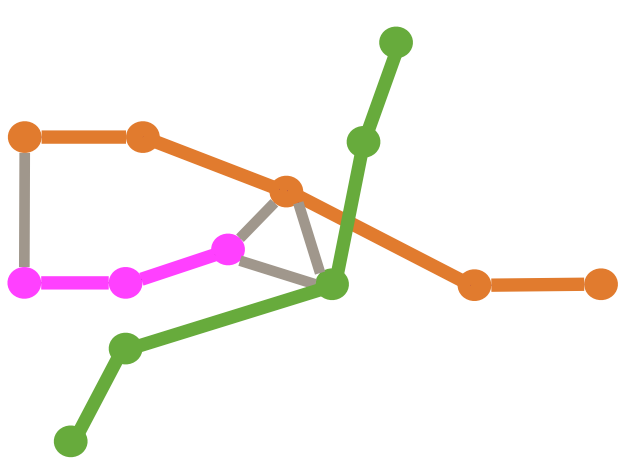


Pour un graphe $G = (V, E)$ de $n = |V|$ sommets et $m = |E|$ arêtes (*arcs*) représenté par listes d'adjacence (*de successeurs*),

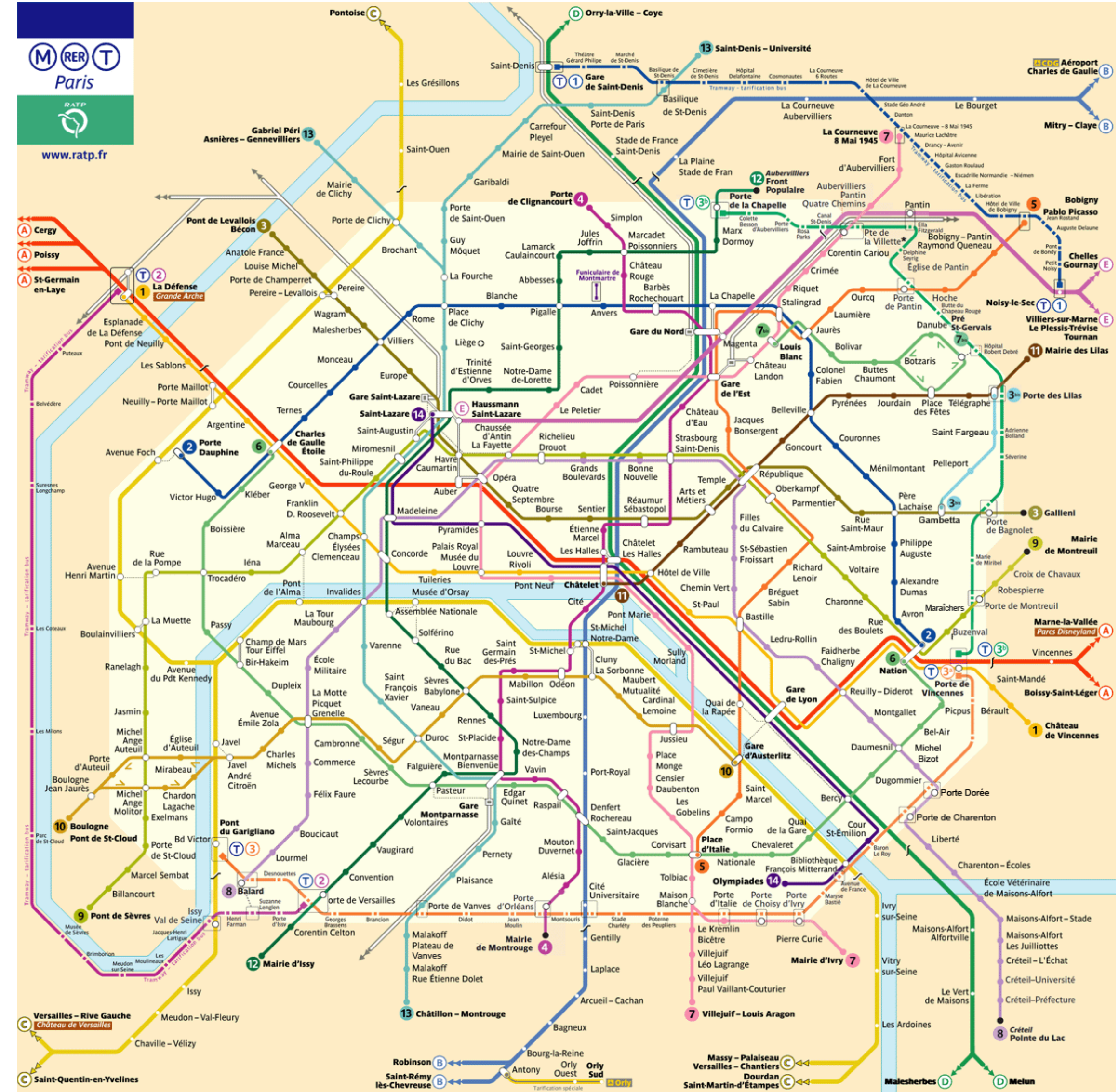
- **Stocker** les listes a un coût mémoire de $O(n + m)$
- **Savoir si** $\{v_i, v_j\}$ est une arête s'effectue en $O(\deg(v_i))$
si (v_i, v_j) est un arc en $O(\deg_+(v_i))$
- **Parcourir** les arêtes incidentes à v_i s'effectue en $O(\deg(v_i))$
les arcs partants de v_i en $O(\deg_+(v_i))$
- **Parcourir toutes** les arêtes (*arcs*) du graphe s'effectue en $O(n + m)$
- **Parcourir** les arcs dont v_i est l'extrémité finale requiert une boucle sur tous les arcs en l'absence de listes de prédécesseur, i.e s'effectue en $O(n + m)$. Avec ces listes, on le fait en $O(\deg_-(v_i))$

Notons que $O(\deg(v_i))$ équivaut à $O(m/n)$ en moyenne

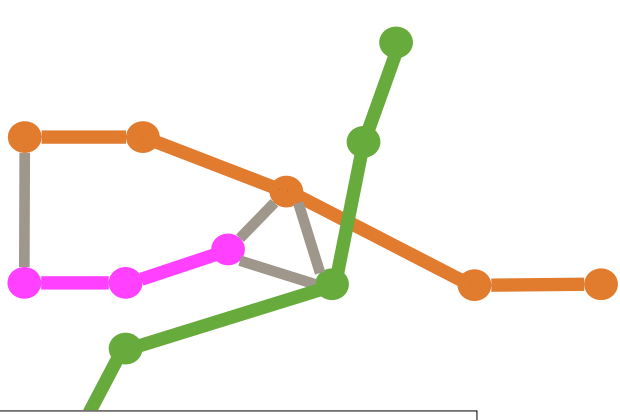
En pratique



- La représentation par listes d'adjacence - de successeurs - est efficace pour les **graphes creux** pour lesquels $|E| \ll |V|^2$
- C'est le cas pour notre exemple de réseau du métro parisien



Représentation implicite

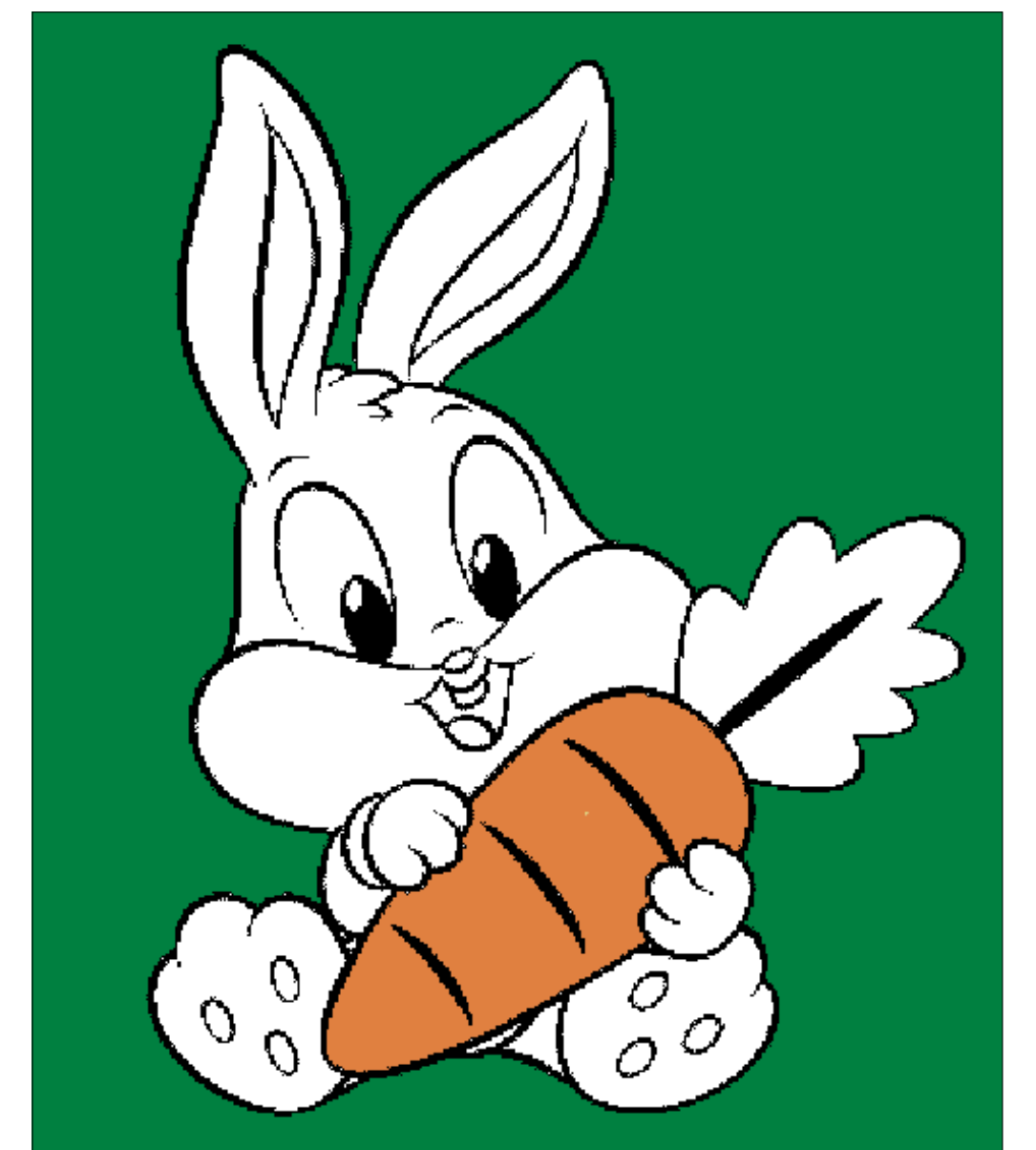
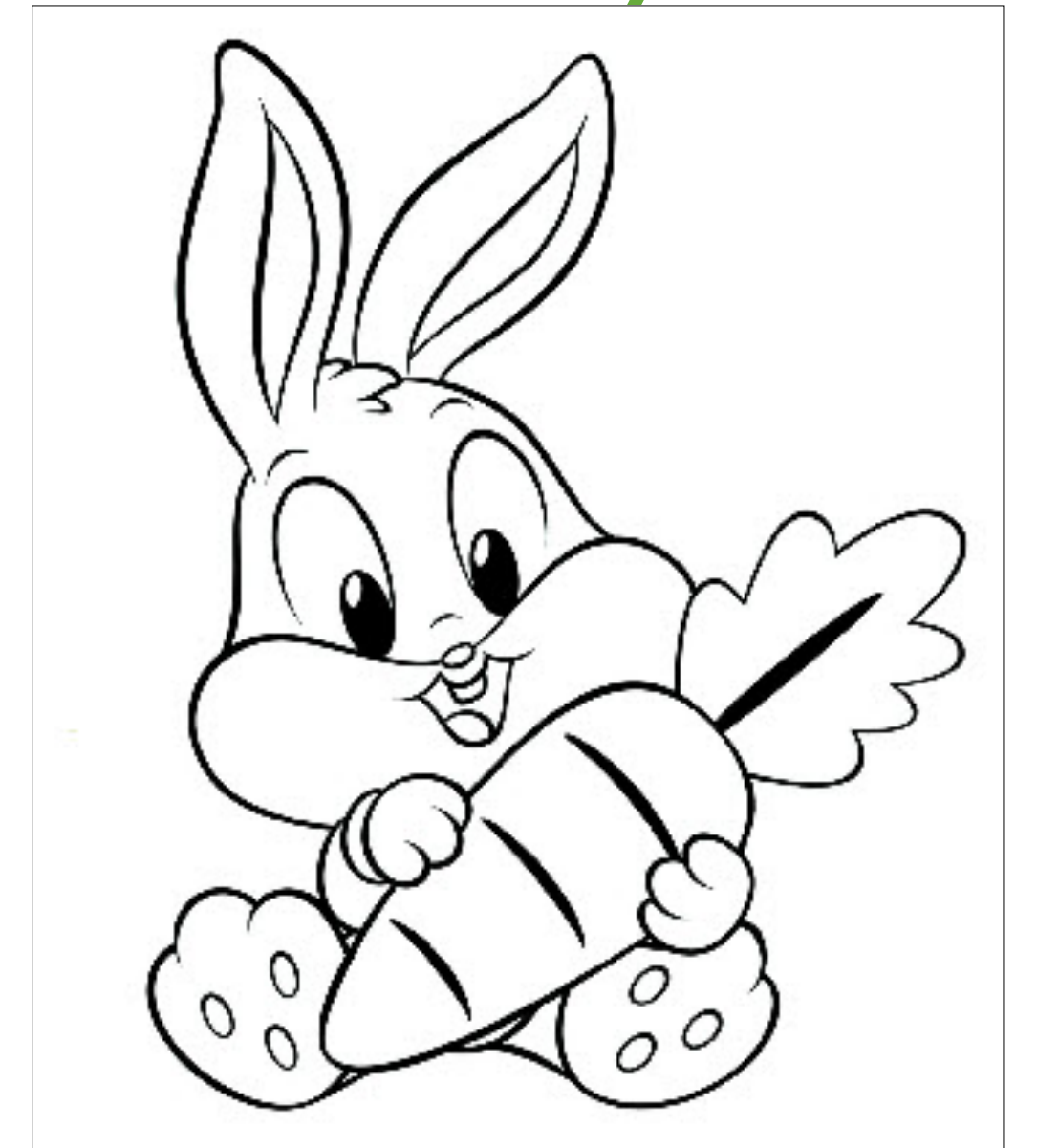


- Parfois, il n'est pas nécessaire de représenter explicitement ni sommets ni arêtes (*arcs*) du graphe



Par exemple, pour colorier une image avec un outil de remplissage de couleur, on peut la représenter sous la forme d'un graphe dont

- les sommets sont les pixels (x, y) de l'image
- les sommets adjacents de (x, y) sont les 4 pixels voisins $\{(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)\}$ à condition qu'ils soient dans l'image et de la même couleur que (x, y)
- Parcourir ce graphe depuis le pixel cliqué par la souris permet de colorier entièrement la zone connexe de l'image



```

struct Pixel { int x, int y };

class Graphe {
    const ImageType& image;

public:
    Graphe(ImageType i) : image(i) { }

    int indice(Pixel p) const {
        return p.x + image.width * p.y;
    }

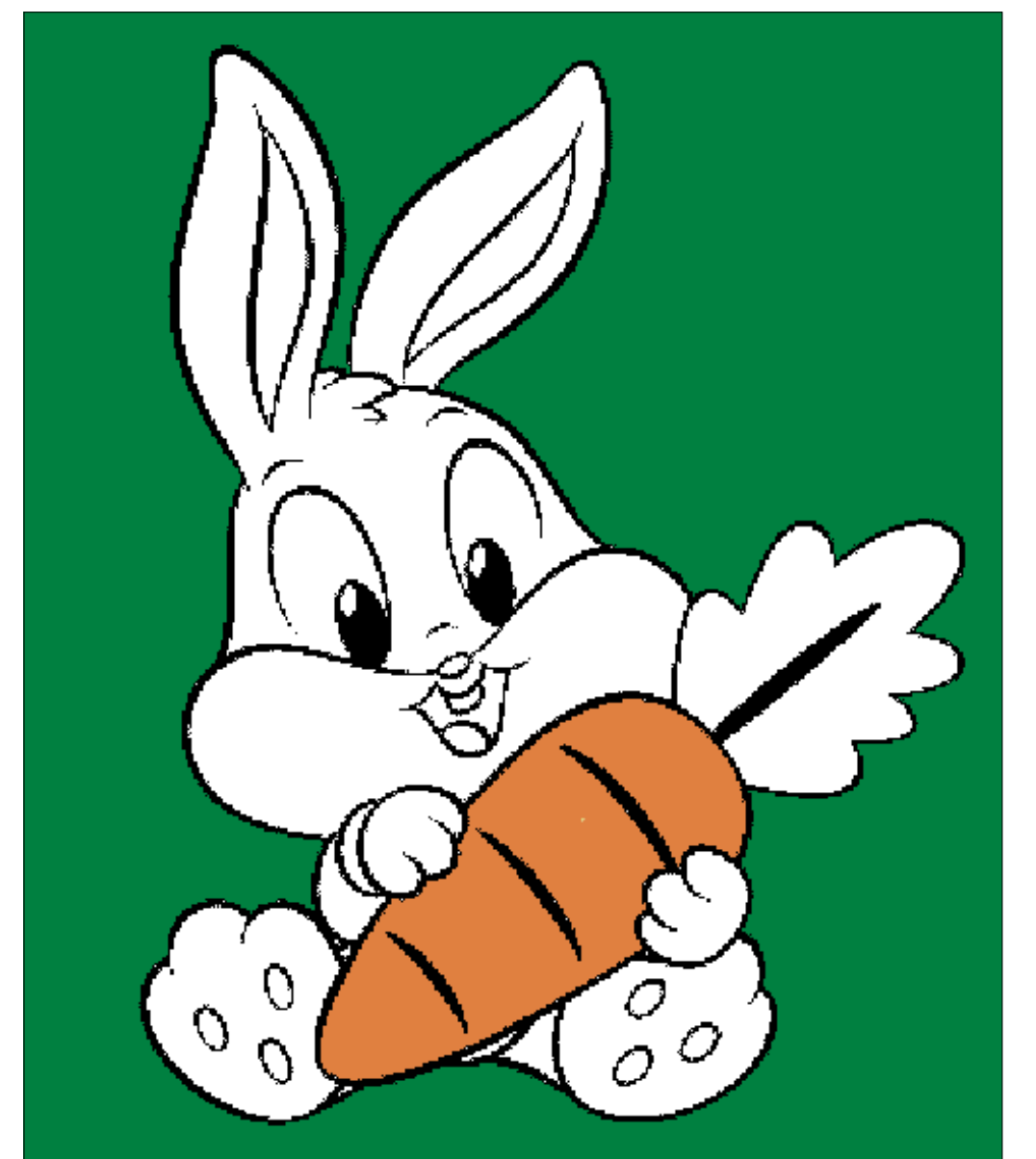
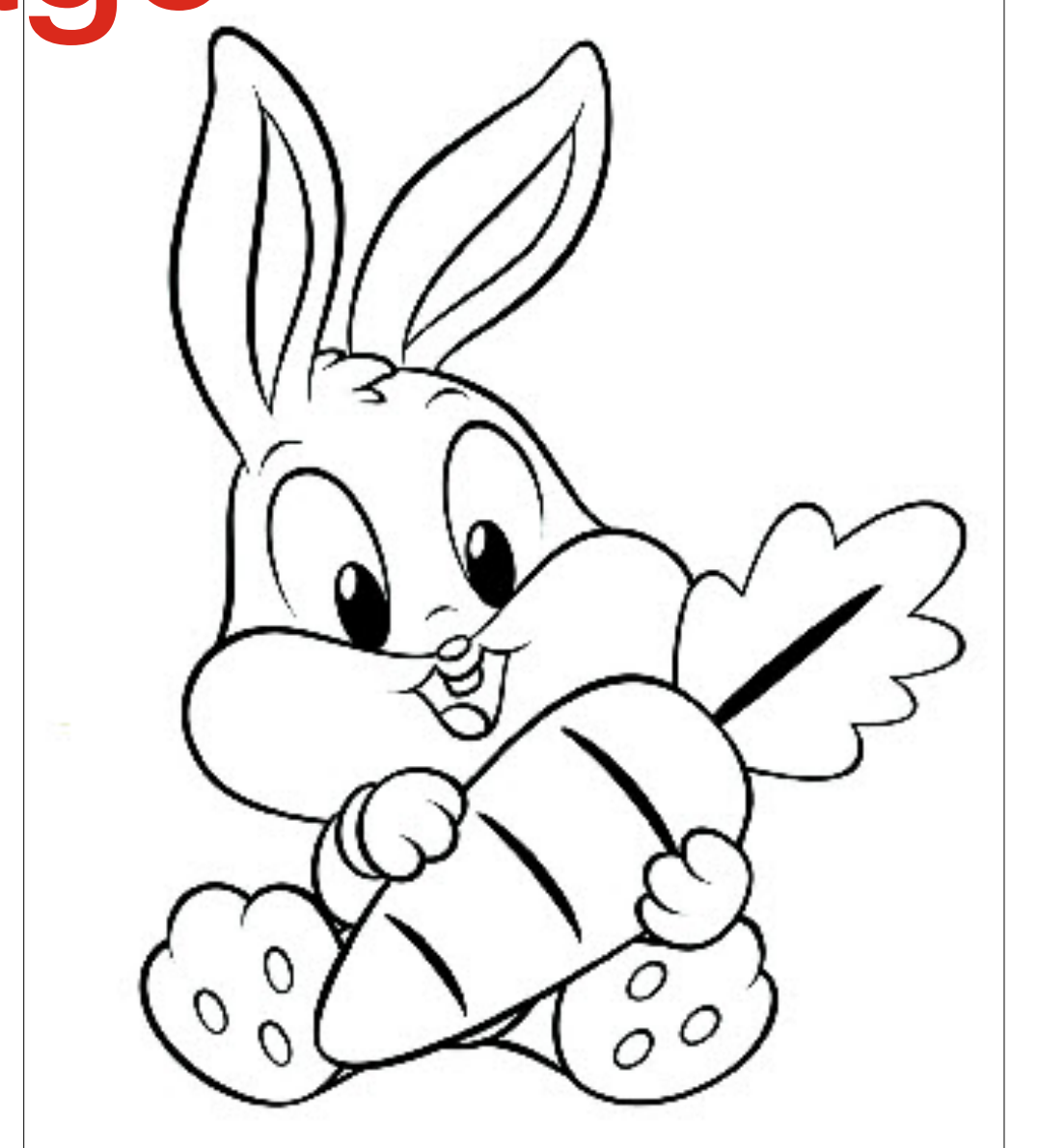
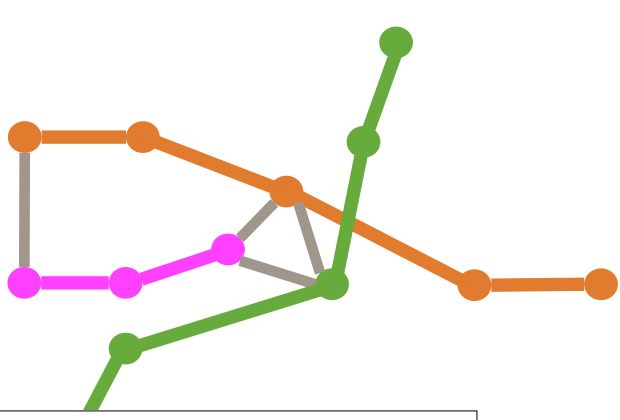
    Pixel pixel(int i) const {
        return Pixel {i % image.width, i / image.width};
    }

    int nSommets() const {
        return image.height * image.width;
    }

    vector<int> adjacent(int v) const {
        vector<int> adj;
        Pixel p = pixel(v);
        if(p.x > 0 and image.get(p.x-1,p.y) == image.get(p.x,p.y))
            adj.push_back(indice(Pixel{p.x-1,y}));
        //...
        return adj;
    }
};

```

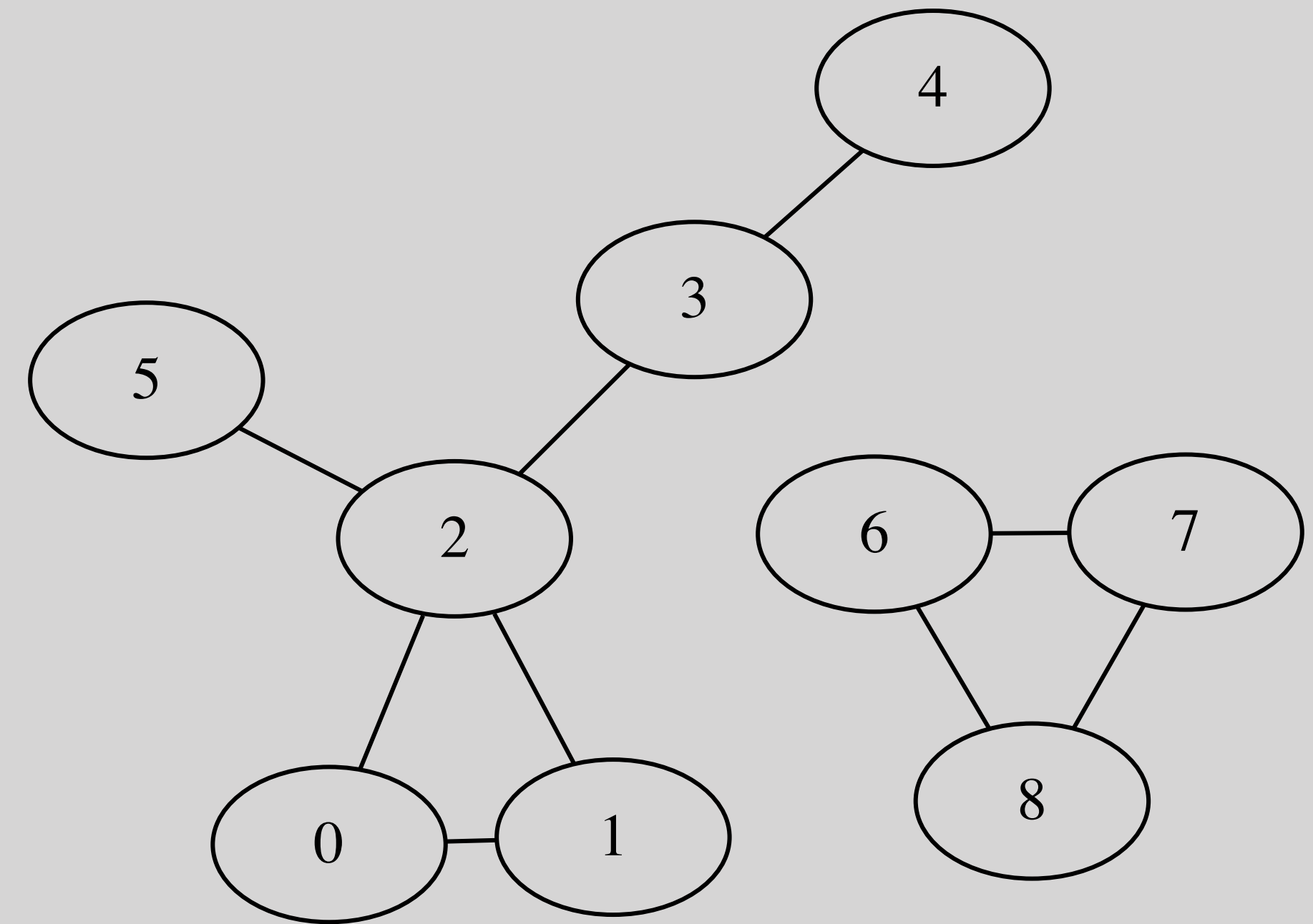
Graphe implicite à partir d'une image



Exercice 1

Donnez la représentation du graphe ci-contre

- sous forme de matrice d'adjacence
- sous forme de listes d'adjacence



Exercice 2

- Dessinez le graphe dont la matrice d'adjacence est A
- Donnez sa représentation en listes d'adjacence

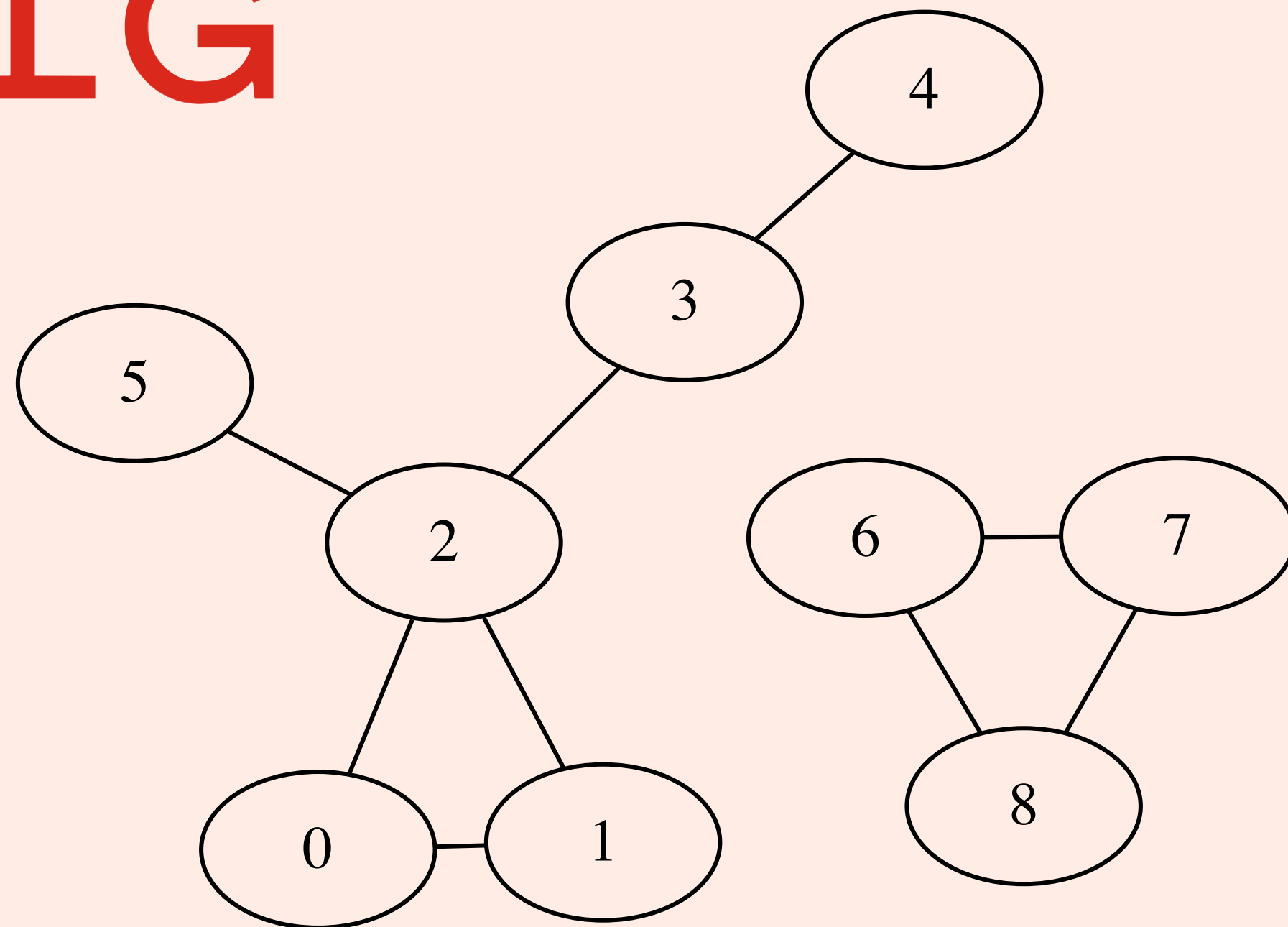
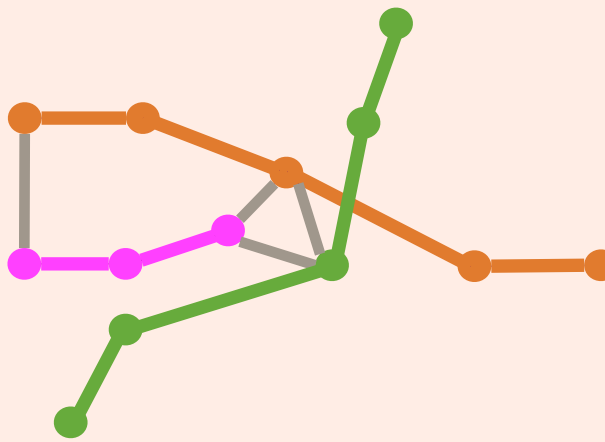
$$A = \begin{bmatrix} [0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0] \\ [0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0] \\ [1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0] \\ [0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0] \\ [0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0] \\ [0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0] \\ [0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1] \\ [0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0] \\ [0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0] \end{bmatrix}$$

$$B = \begin{bmatrix} [2, 8], \\ [3, 2, 4], \\ [1, 5, 7, 0], \\ [1], \\ [5, 1], \\ [2, 4], \\ [7, 8], \\ [2, 6], \\ [0, 6] \end{bmatrix}$$

Exercice 3

- Dessinez le graphe dont la représentation par listes d'adjacence est B
- Donnez sa représentation par matrice d'adjacence

Solution 1



Matrice d'adjacence

```

[[0 1 1 0 0 0 0 0 0]
 [1 0 1 0 0 0 0 0 0]
 [1 1 0 1 0 1 0 0 0]
 [0 0 1 0 1 0 0 0 0]
 [0 0 0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 1]
 [0 0 0 0 0 0 1 0 1]
 [0 0 0 0 0 0 1 1 0]]

```

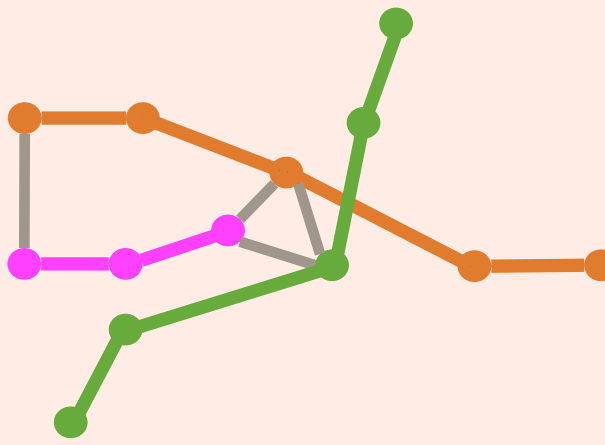
Listes d'adjacence

```

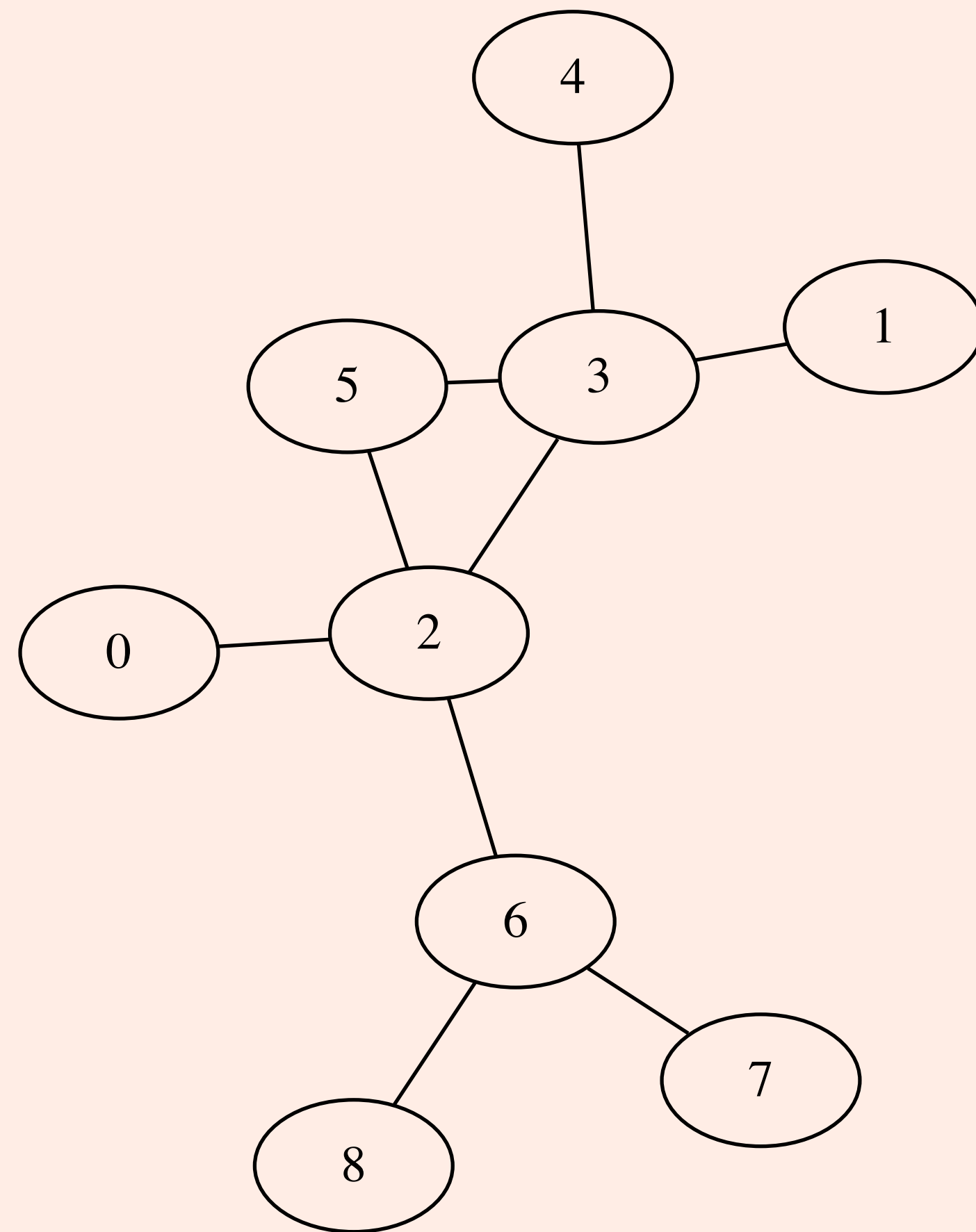
[[1, 2],
 [0, 2],
 [0, 1, 3, 5],
 [2, 4],
 [3],
 [2],
 [7, 8],
 [8, 6],
 [7, 6]]

```

Solution 2



Le graphe dont la matrice d'adjacence est

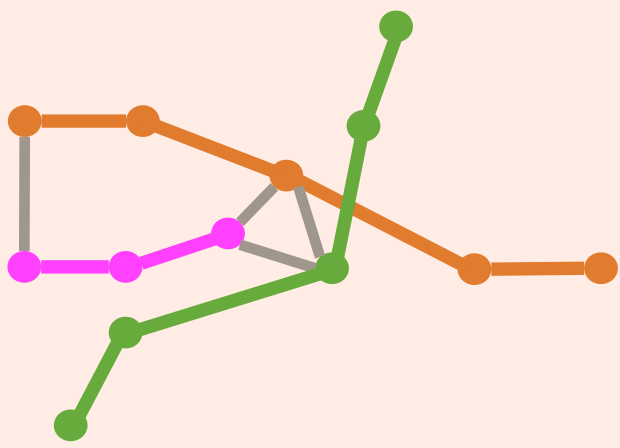


$$\begin{bmatrix}
 [0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0] \\
 [0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0] \\
 [1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0] \\
 [0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0] \\
 [0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0] \\
 [0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0] \\
 [0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1] \\
 [0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0] \\
 [0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0]
 \end{bmatrix}$$

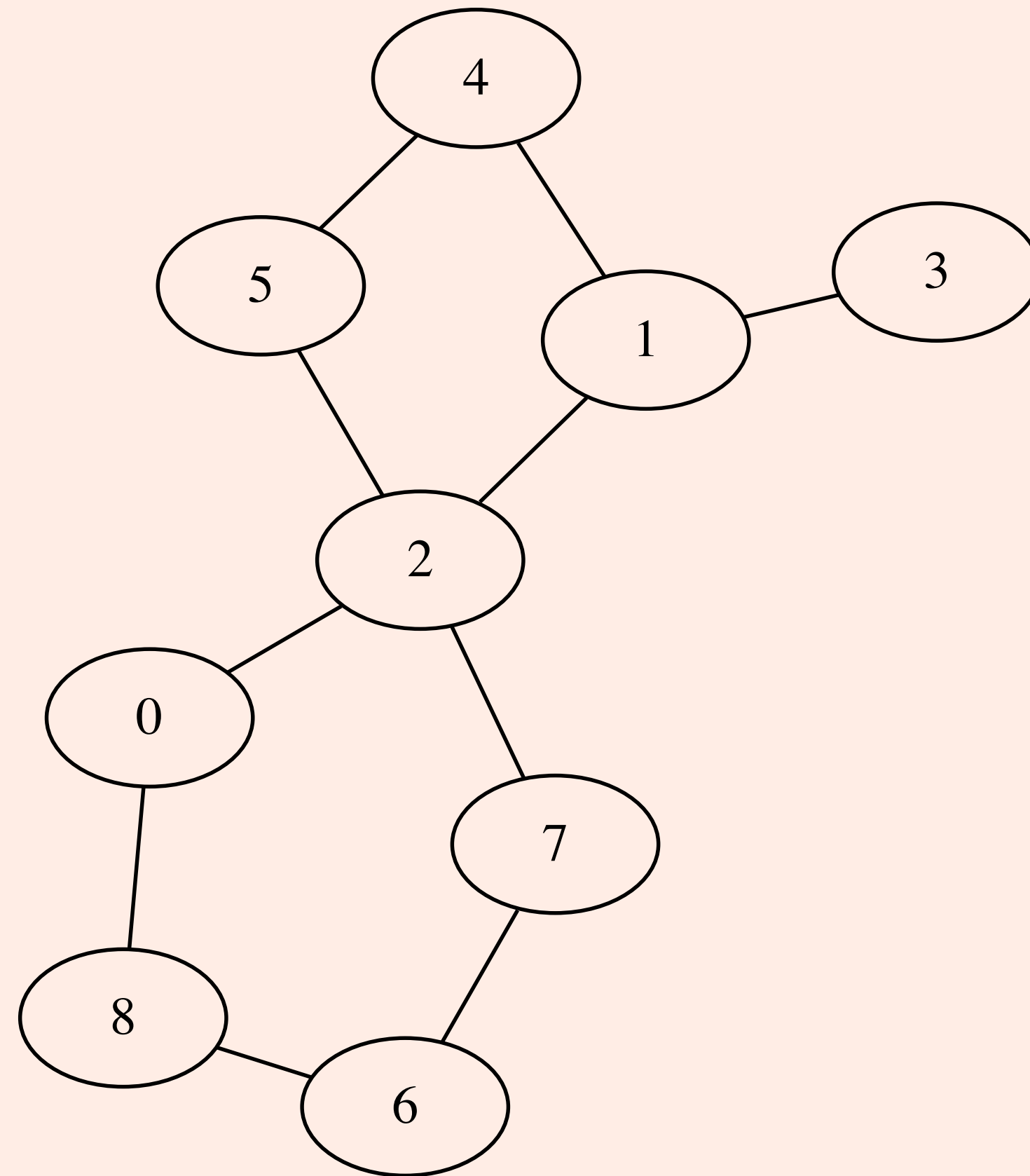
$$\begin{aligned}
 &[[2], \\
 &[3], \\
 &[0, 3, 5, 6], \\
 &[1, 2, 4, 5], \\
 &[3], \\
 &[2, 3], \\
 &[2, 7, 8], \\
 &[6], \\
 &[6]]
 \end{aligned}$$

Sous forme de listes :

Solution 3



Le graphe dont la représentation par listes d'adjacence est



Sous forme de matrice :

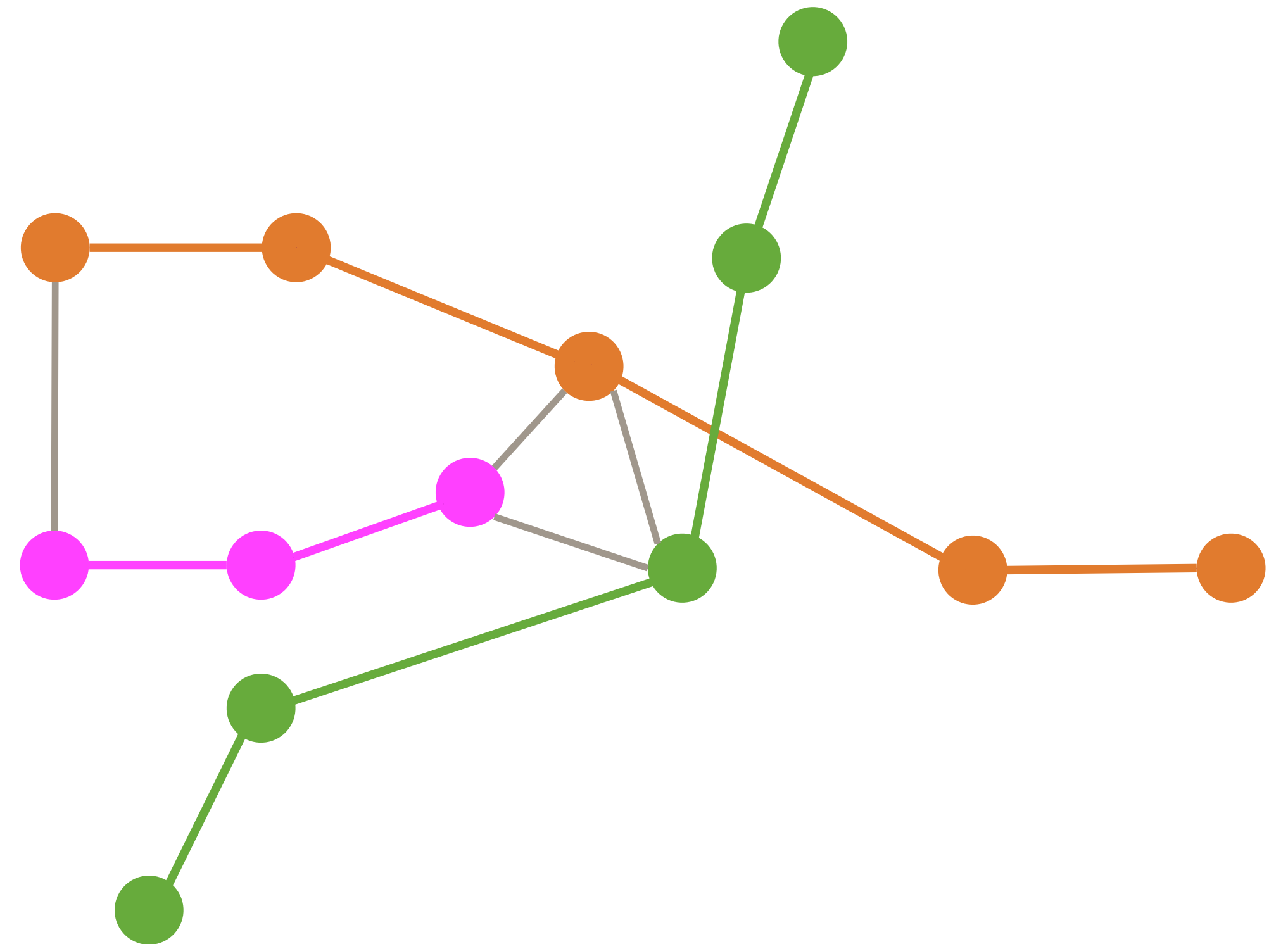
```

[ [0 0 1 0 0 0 0 0 1]
  [0 0 1 1 1 0 0 0 0]
  [1 1 0 0 0 1 0 1 0]
  [0 1 0 0 0 0 0 0 0]
  [0 1 0 0 0 1 0 0 0]
  [0 0 1 0 1 0 0 0 0]
  [0 0 0 0 0 0 0 1 1]
  [0 0 1 0 0 0 1 0 0]
  [1 0 0 0 0 0 1 0 0] ]
  
```

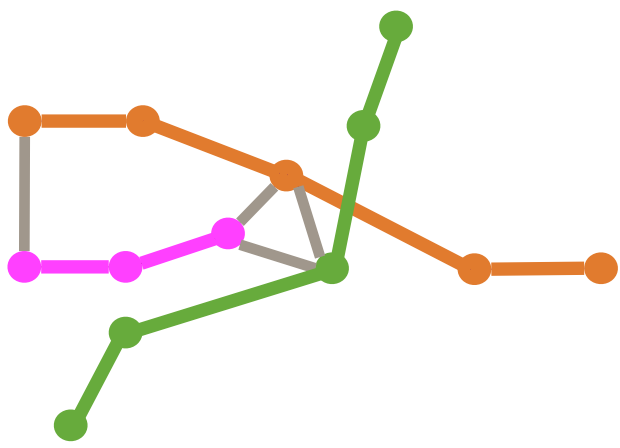
```

[ [2, 8],
  [3, 2, 4],
  [1, 5, 7, 0],
  [1],
  [5, 1],
  [2, 4],
  [7, 8],
  [2, 6],
  [0, 6] ]
  
```

6.4. Parcours en profondeur et en largeur



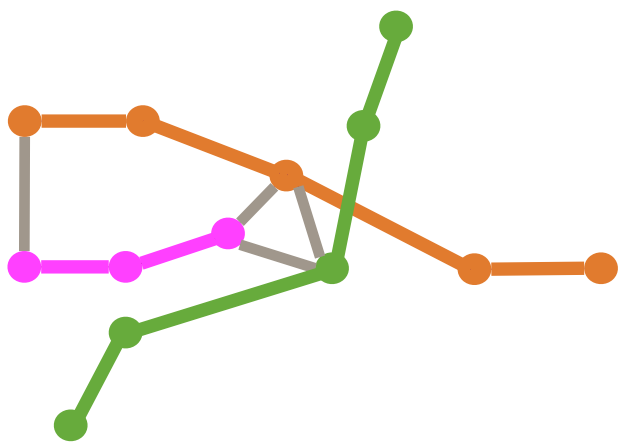
Similarités avec les parcours d'arbre



Comme pour les arbres, on a deux stratégies possible pour parcourir tous les sommets d'un graphe

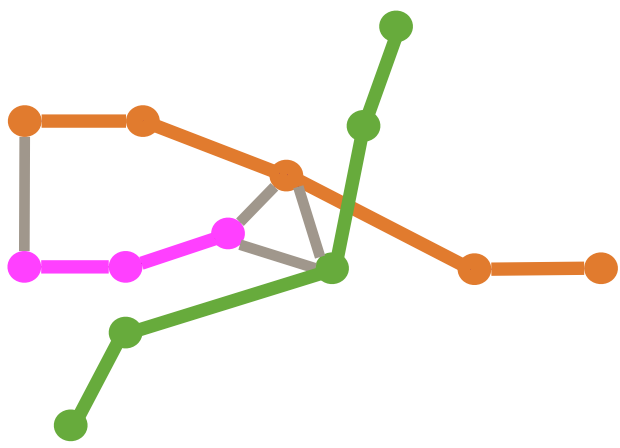
- **en profondeur** (depth first search) - méthode **récursive** qui s'appelle pour tous les sommets adjacents
- **en largeur** (breadth first search) - utilisation d'une **file FIFO** qui parcourt d'abord tous les sommets adjacents avant ceux à distance 2, puis 3, ...

Différences avec les parcours d'arbre

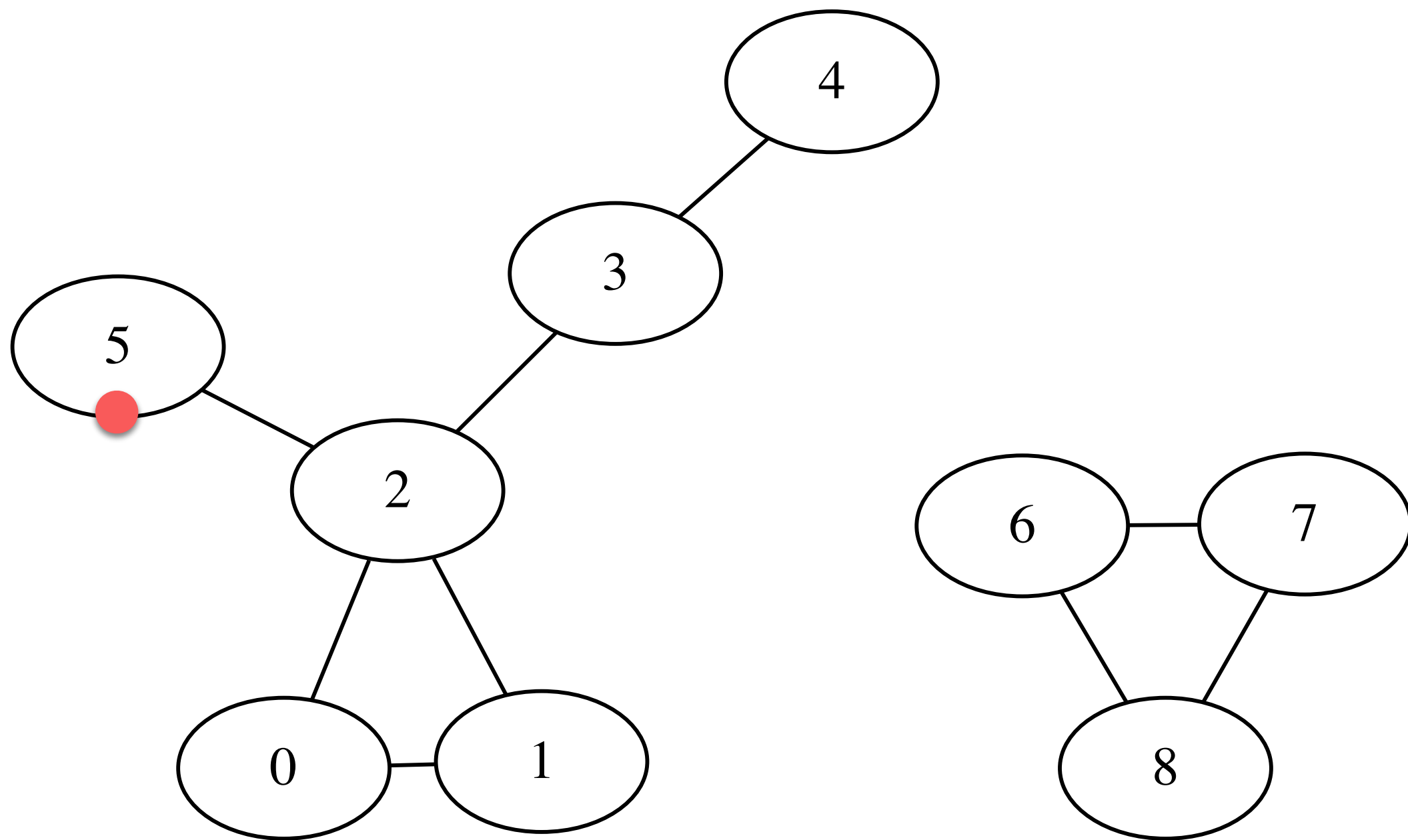


- Mais c'est plus compliqué ... parce que
 - Pas de hiérarchie de parent vers enfant : il faut donc éviter d'appeler le sommet appelant, et créer une **boucle infinie**.
 - Il peut y avoir des **cycles** dans le graphe.
- La solution consiste à **marquer les sommets** déjà atteints par le parcours.

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

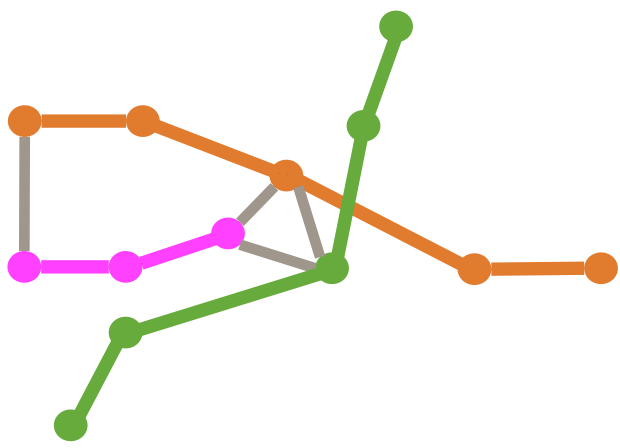
```
  ● marquer v visité
```

```
  pour tout w adjacent à v
```

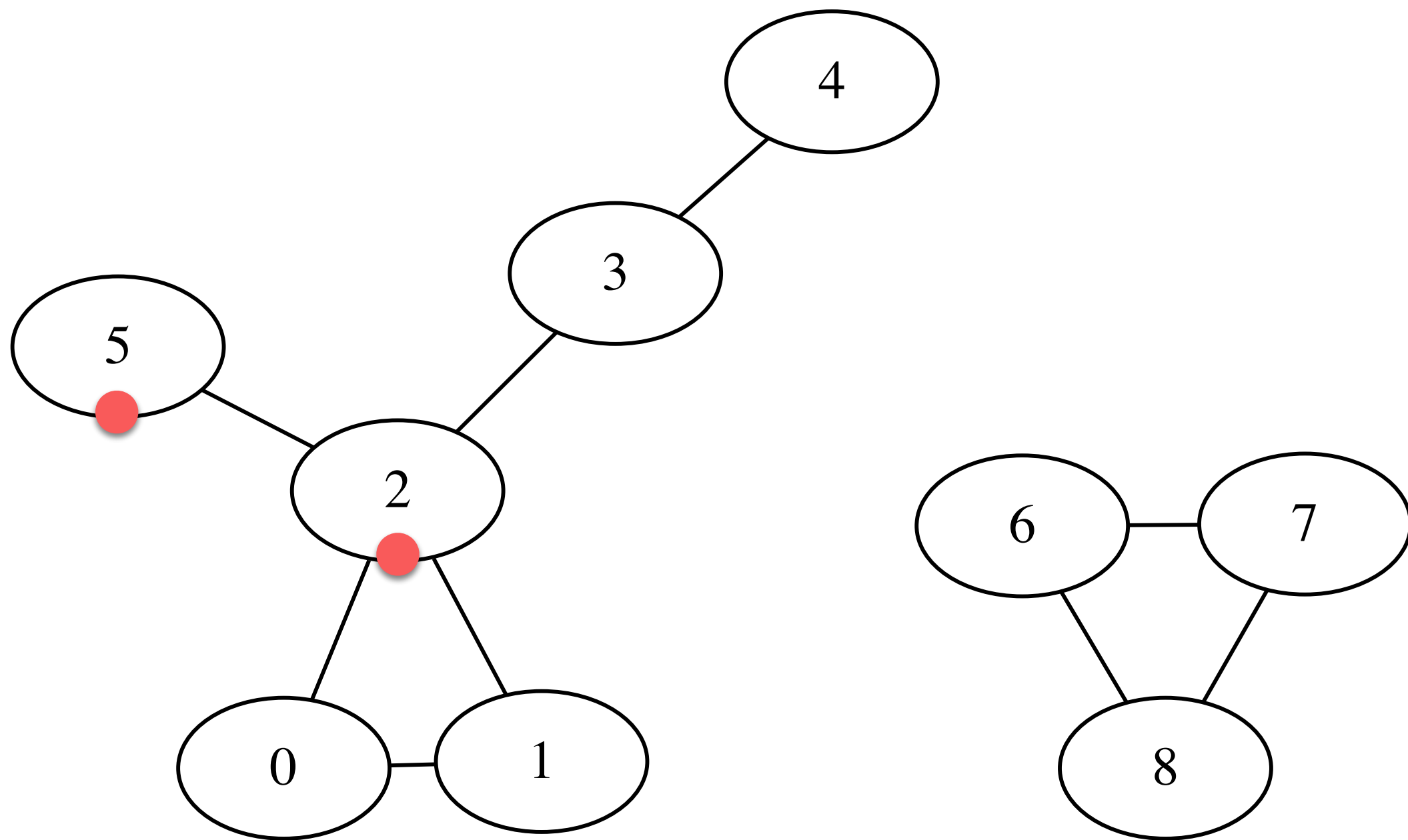
```
    si w n'est pas visité
```

```
      profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommets v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

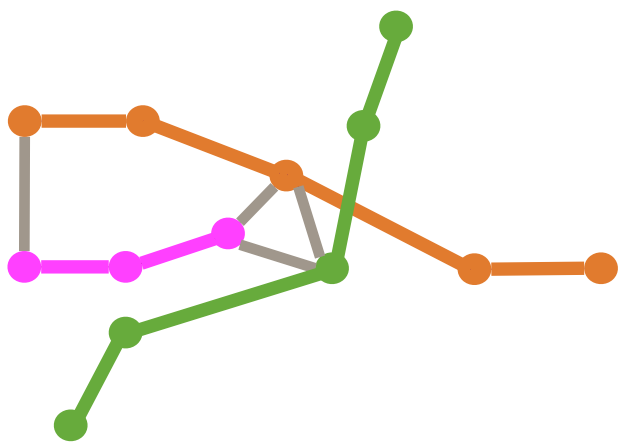
● marquer v visité

```
pour tout w adjacent à v
```

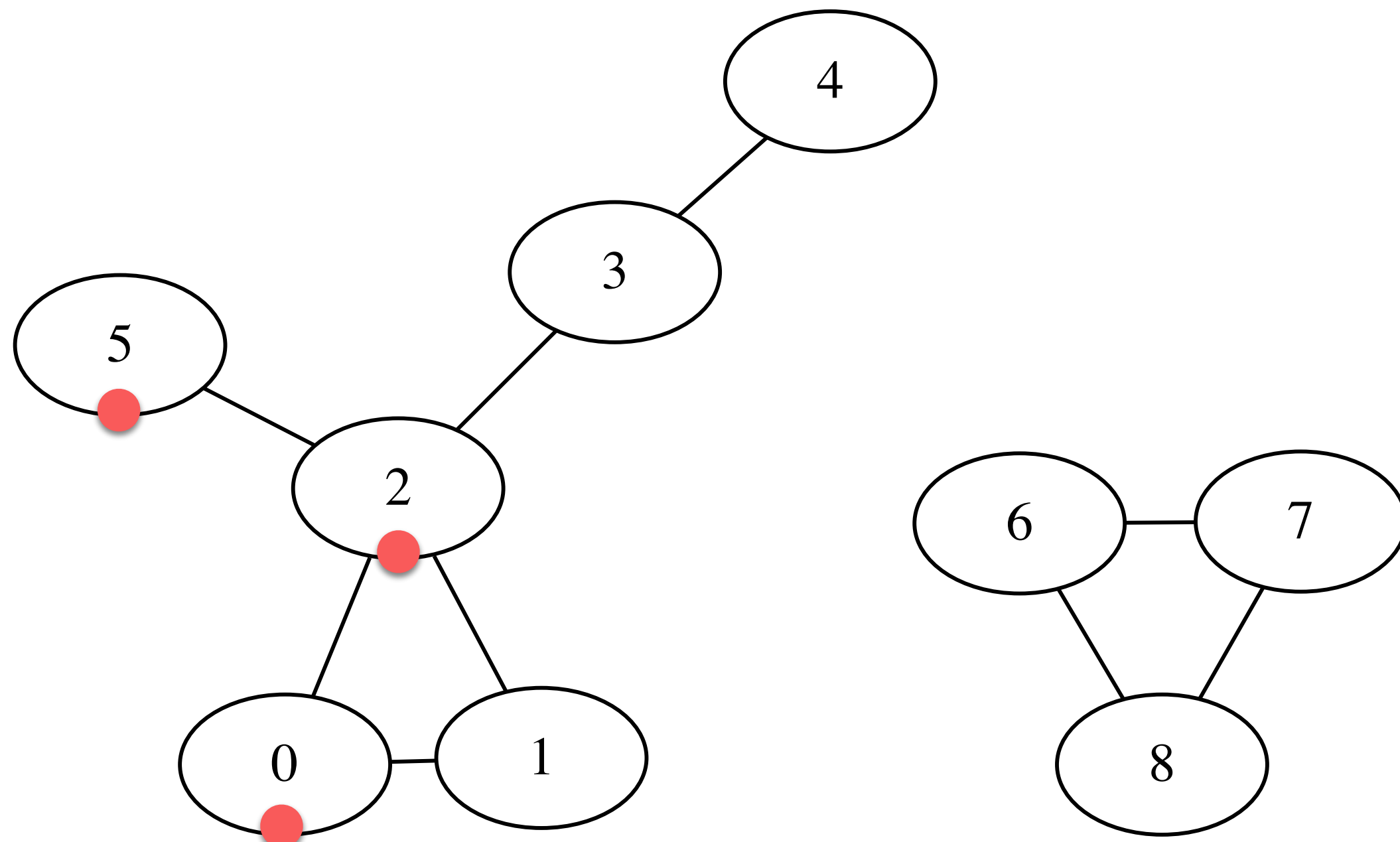
```
  si w n'est pas visité
```

```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

● marquer v visité

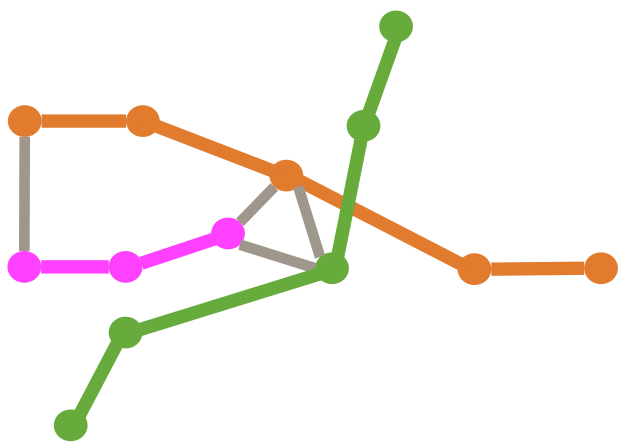
```
pour tout w adjacent à v
```

```
  si w n'est pas visité
```

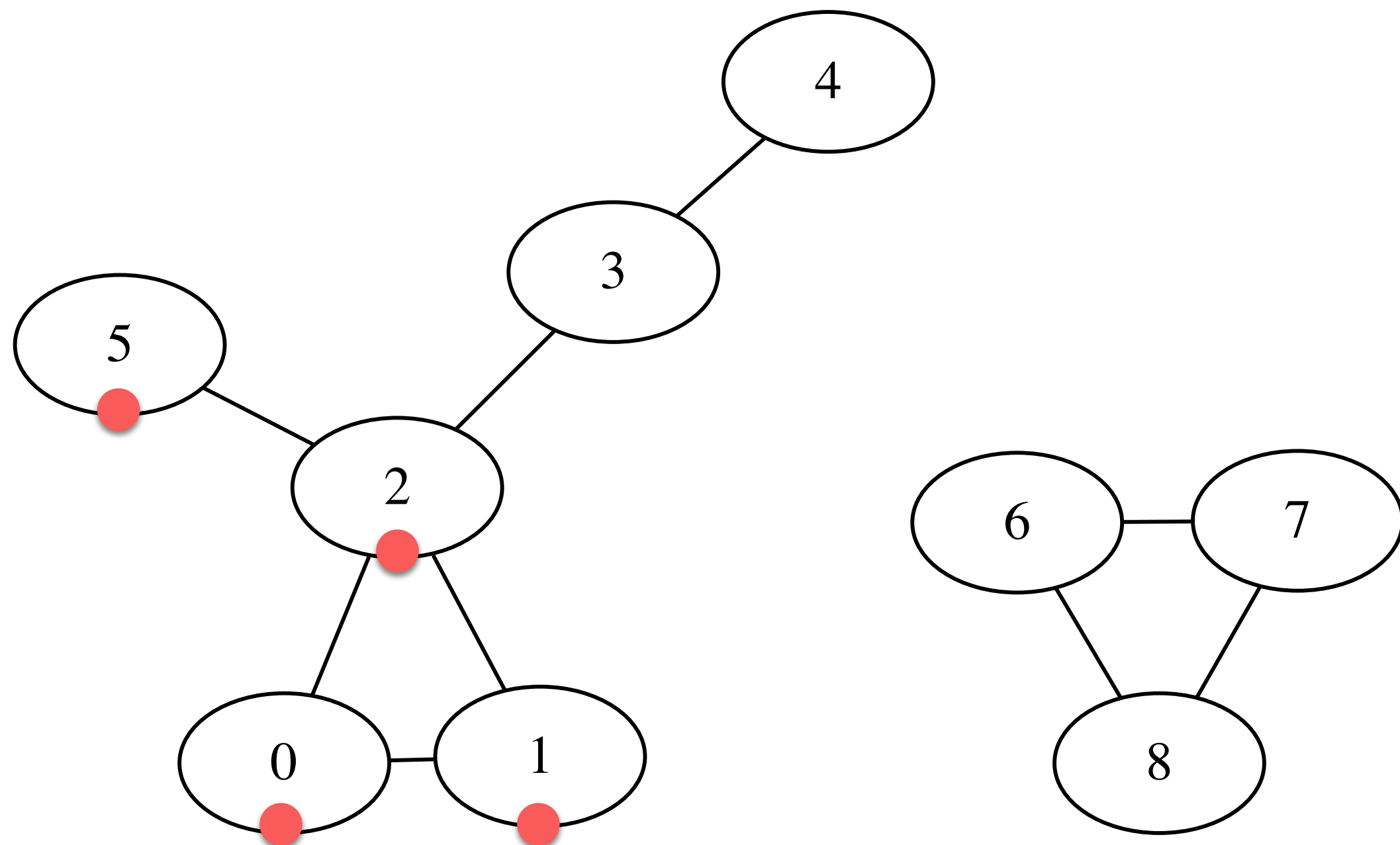
```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

● marquer v visité

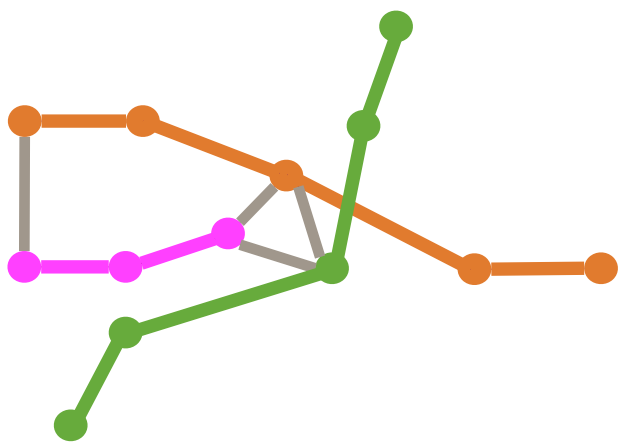
```
pour tout w adjacent à v
```

```
  si w n'est pas visité
```

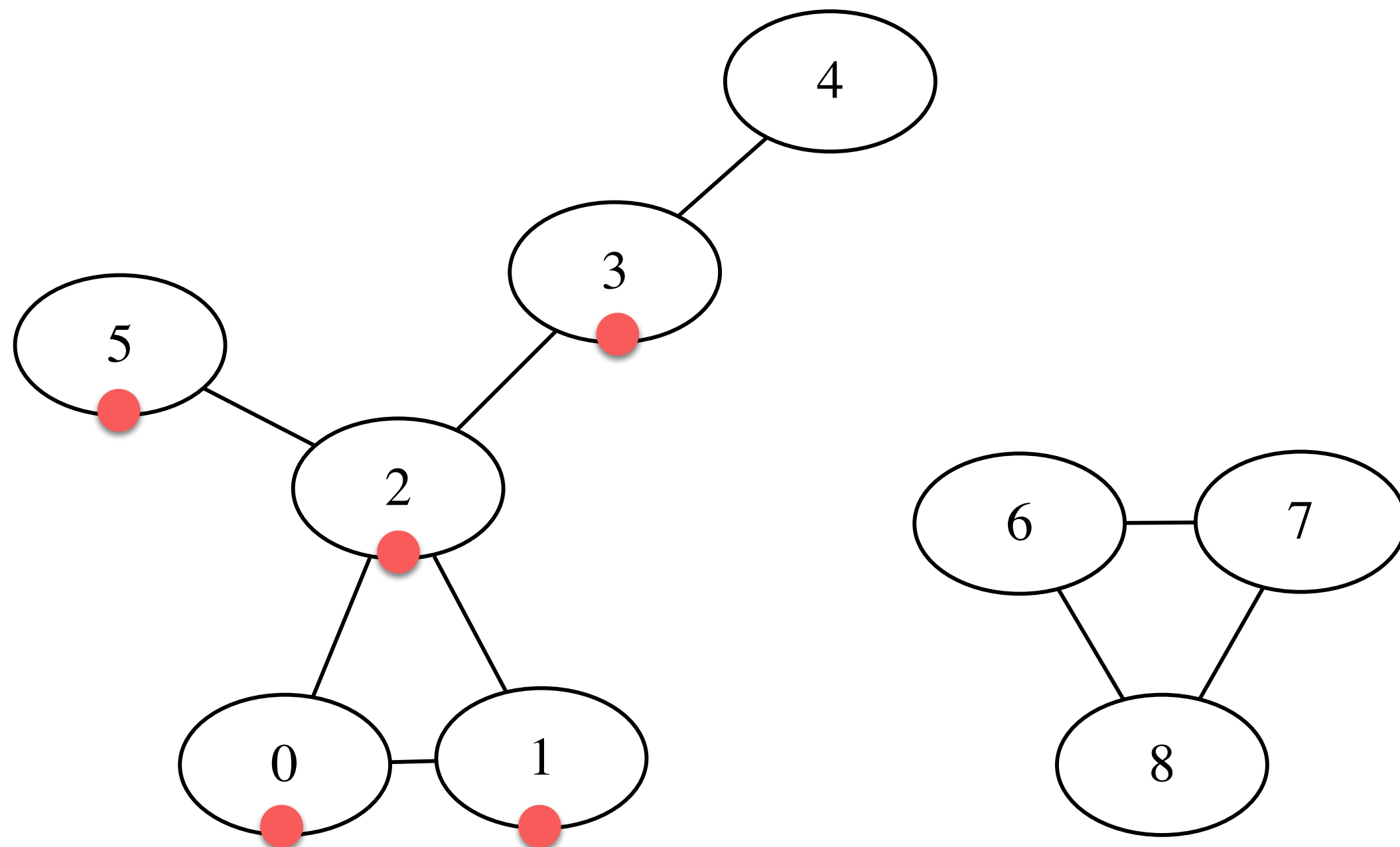
```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

● marquer v visité

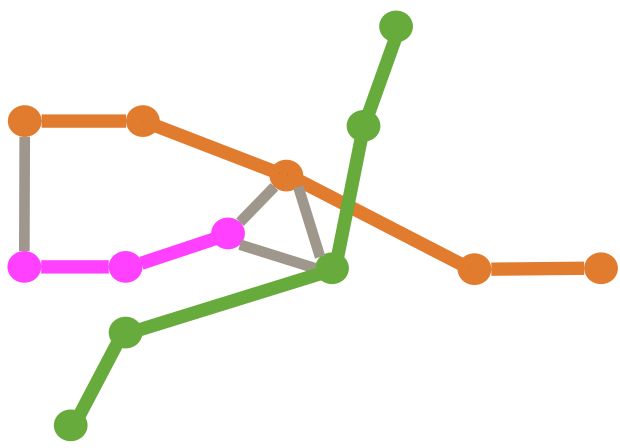
```
  pour tout w adjacent à v
```

```
    si w n'est pas visité
```

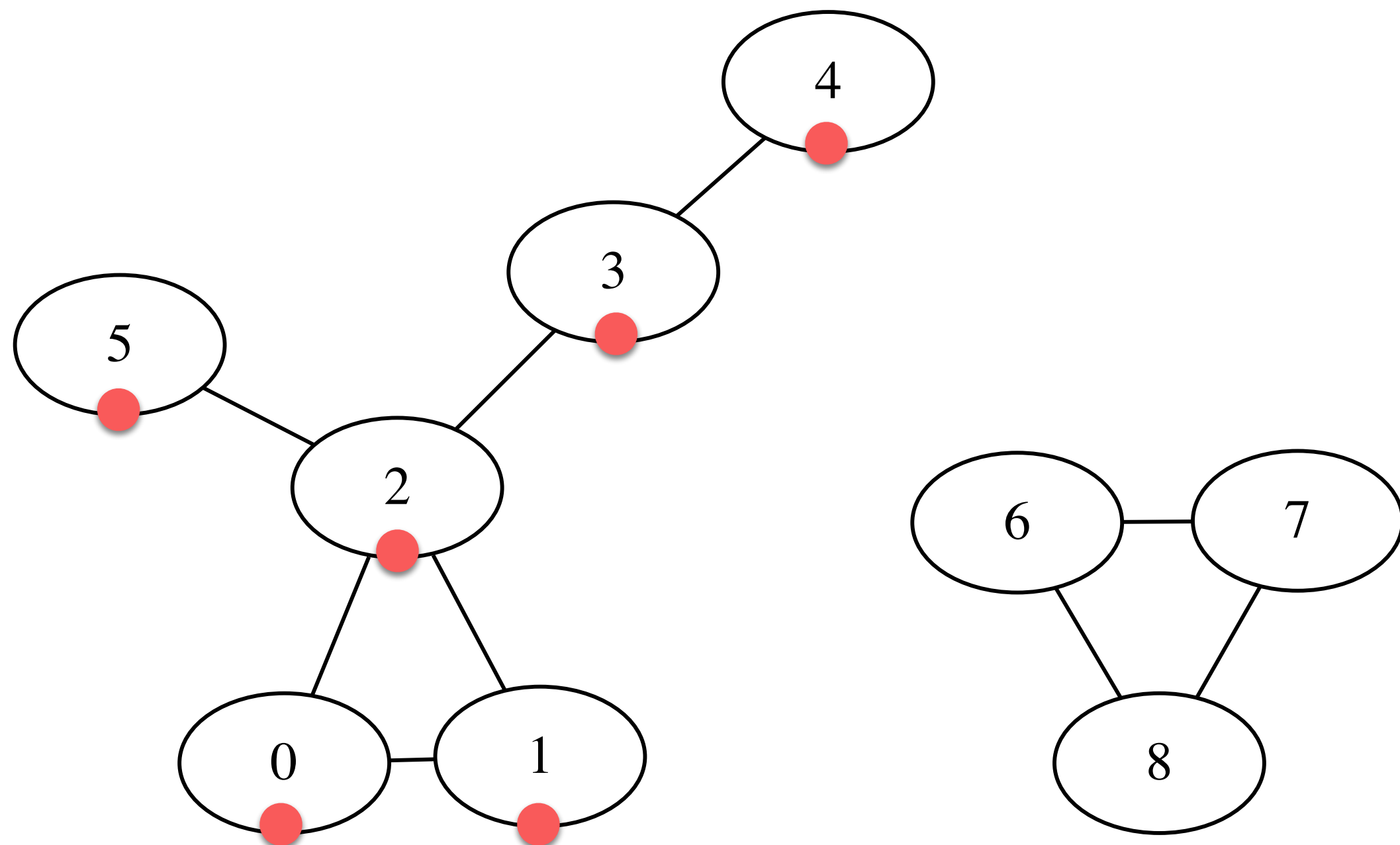
```
      profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

```
  pre(v) // en pré-ordre
```

● marquer v visité

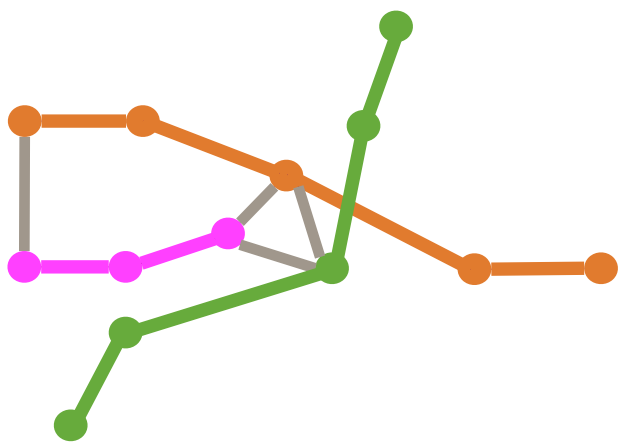
```
pour tout w adjacent à v
```

```
  si w n'est pas visité
```

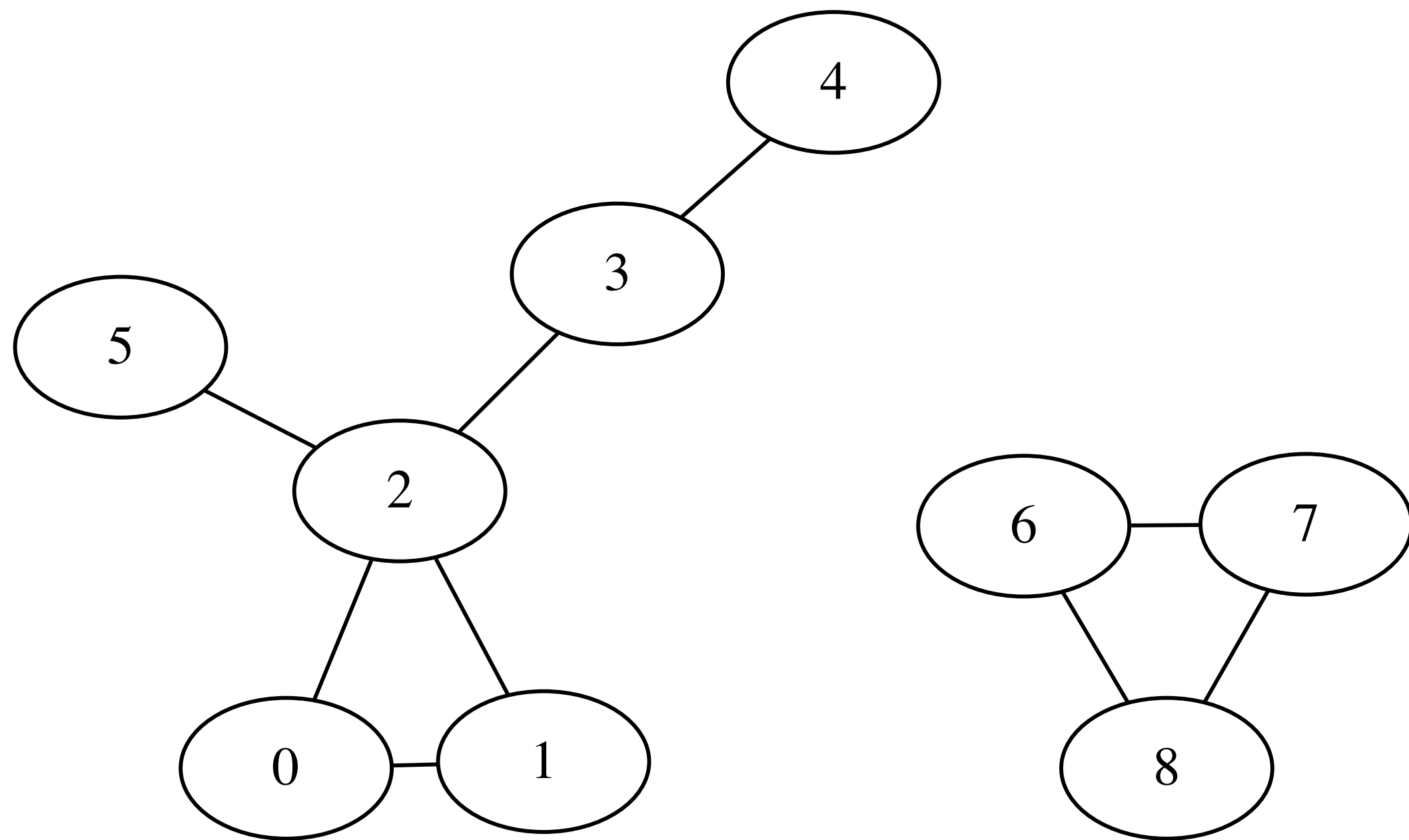
```
    profondeur(w, pre, post)
```

```
  post(v) // en post-ordre
```

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● pre(v) // en pré-ordre

marquer v visité

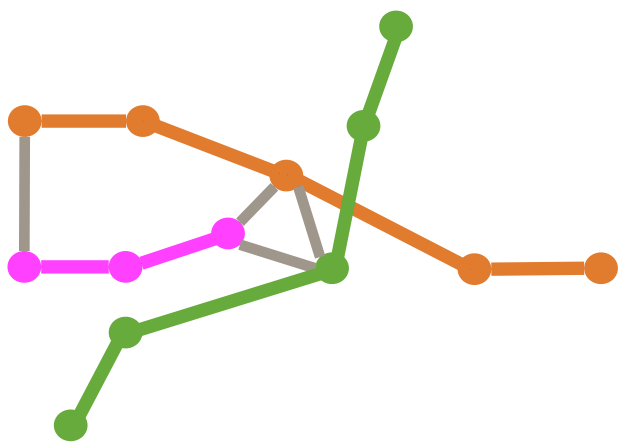
pour tout w adjacent à v

si w n'est pas visité

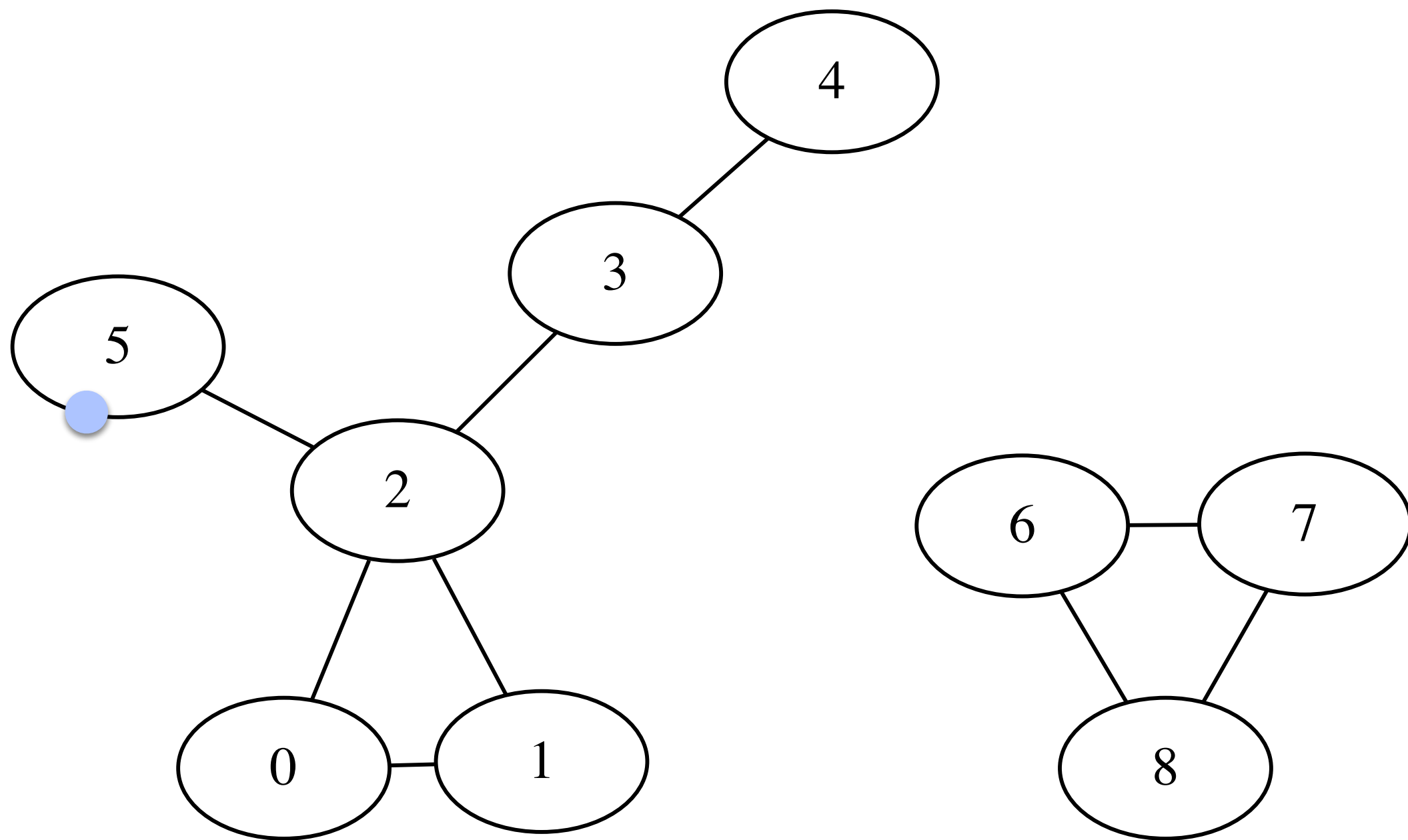
profondeur(w, pre, post)

● post(v) // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● pre(v) // en pré-ordre

marquer v visité

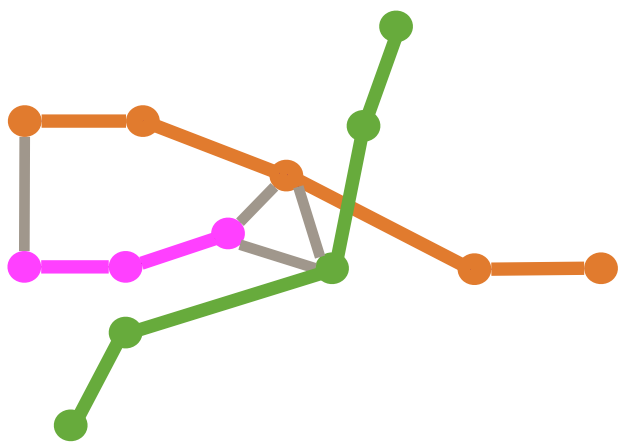
pour tout w adjacent à v

si w n'est pas visité

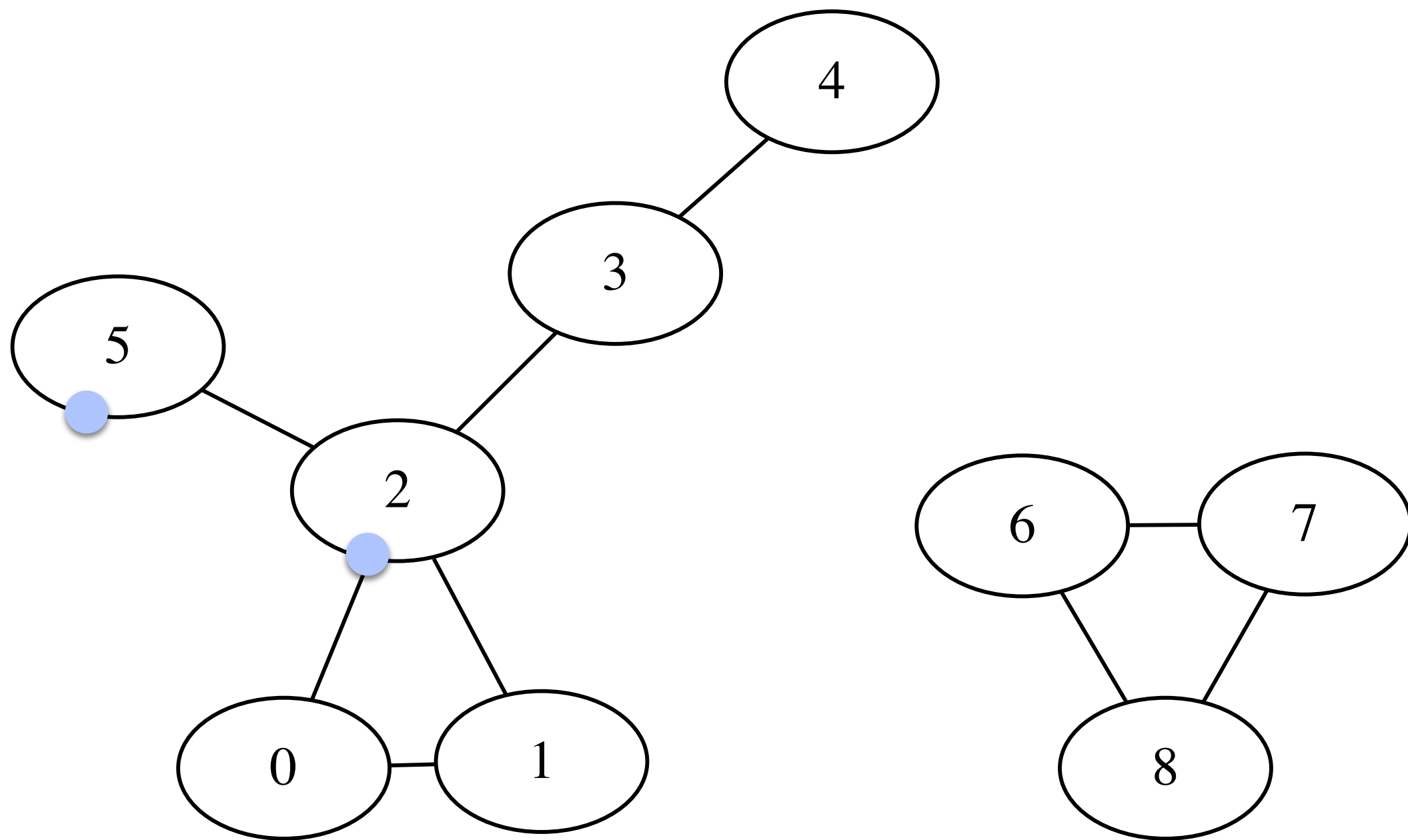
profondeur(w, pre, post)

● post(v) // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

marquer *v* visité

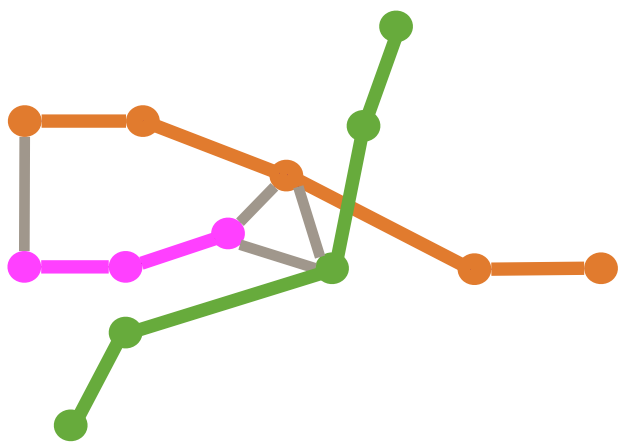
pour tout *w* adjacent à *v*

si *w* n'est pas visité

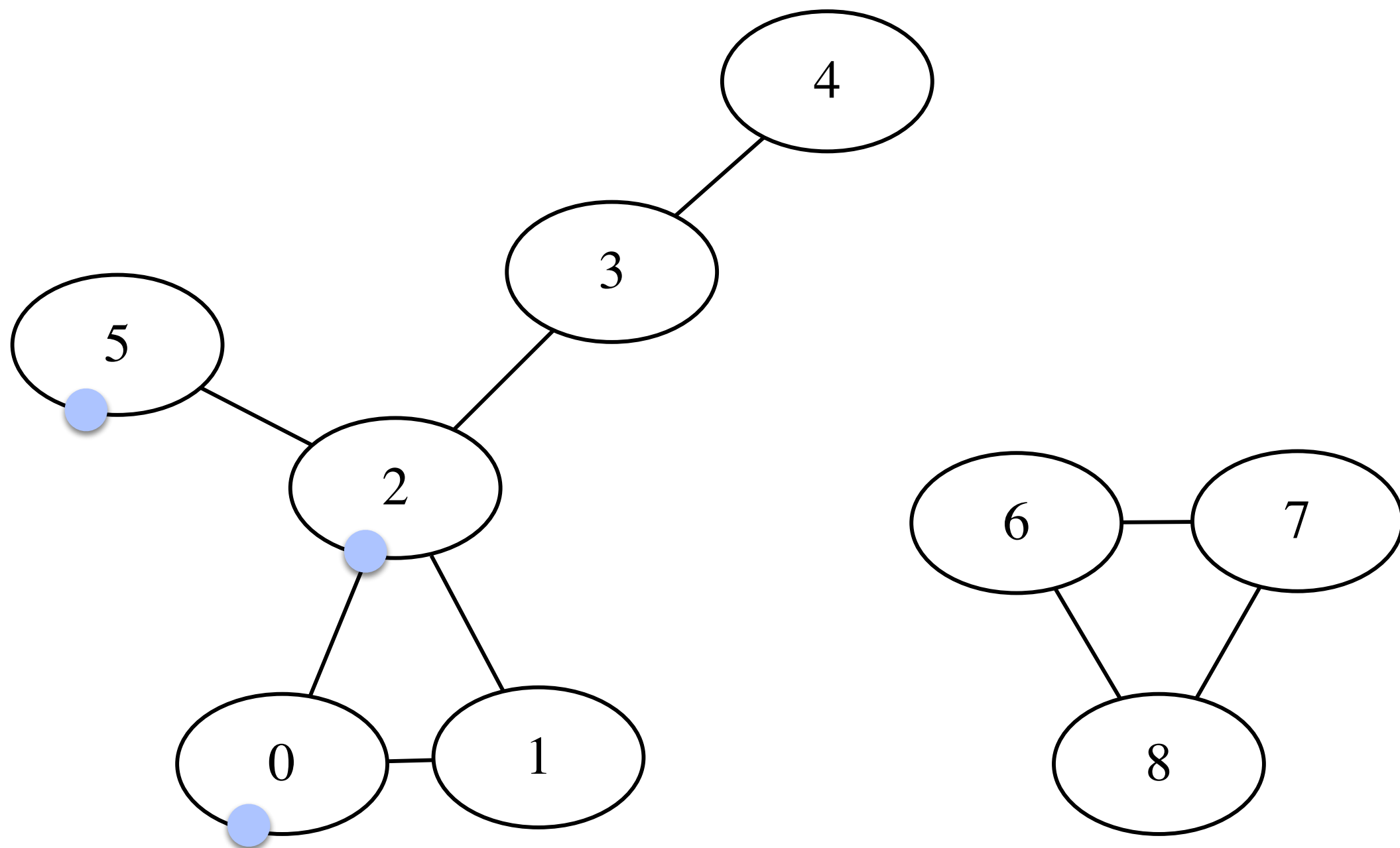
profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

marquer v visité

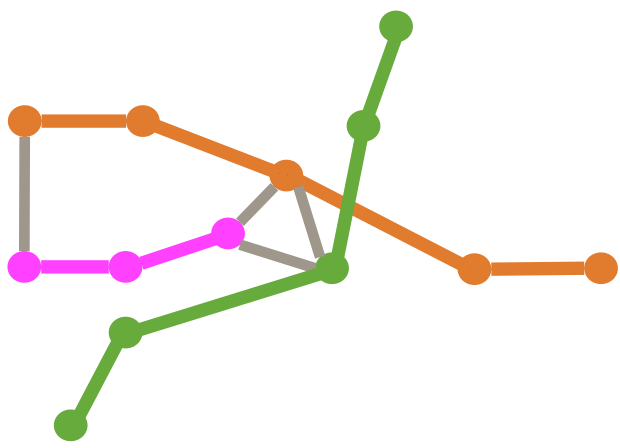
pour tout w adjacent à v

si w n'est pas visité

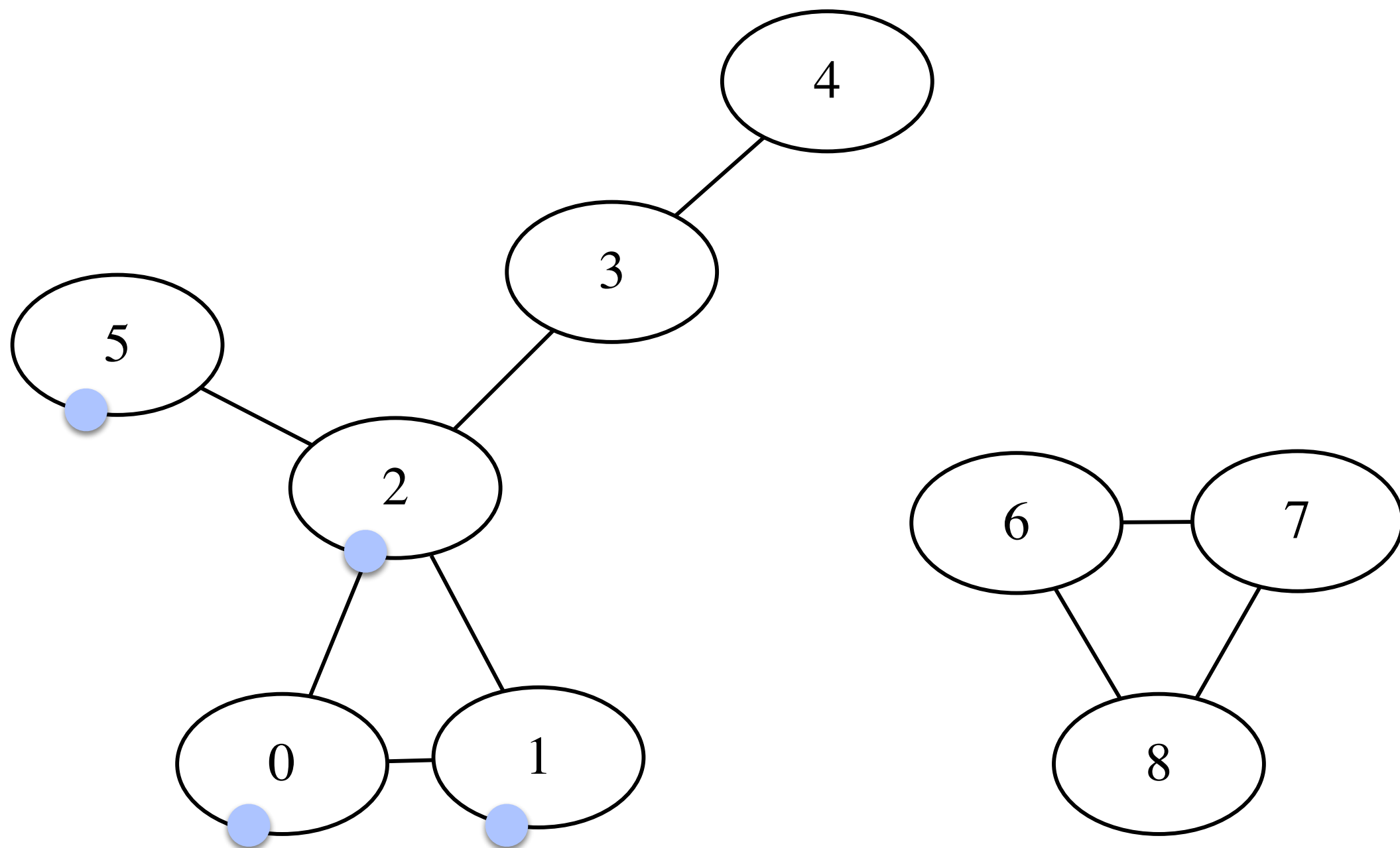
`profondeur(w, pre, post)`

● `post(v)` // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

marquer v visité

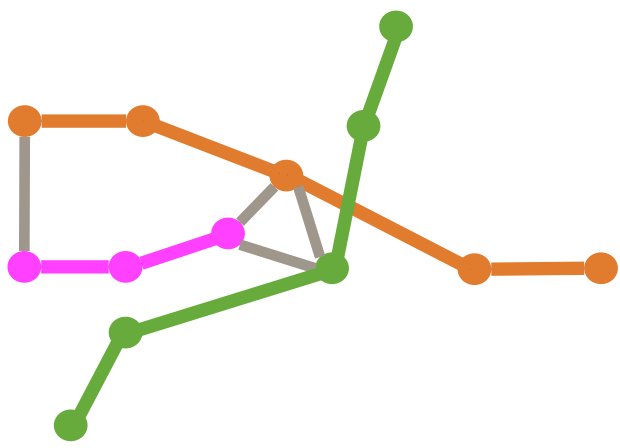
pour tout w adjacent à v

si w n'est pas visité

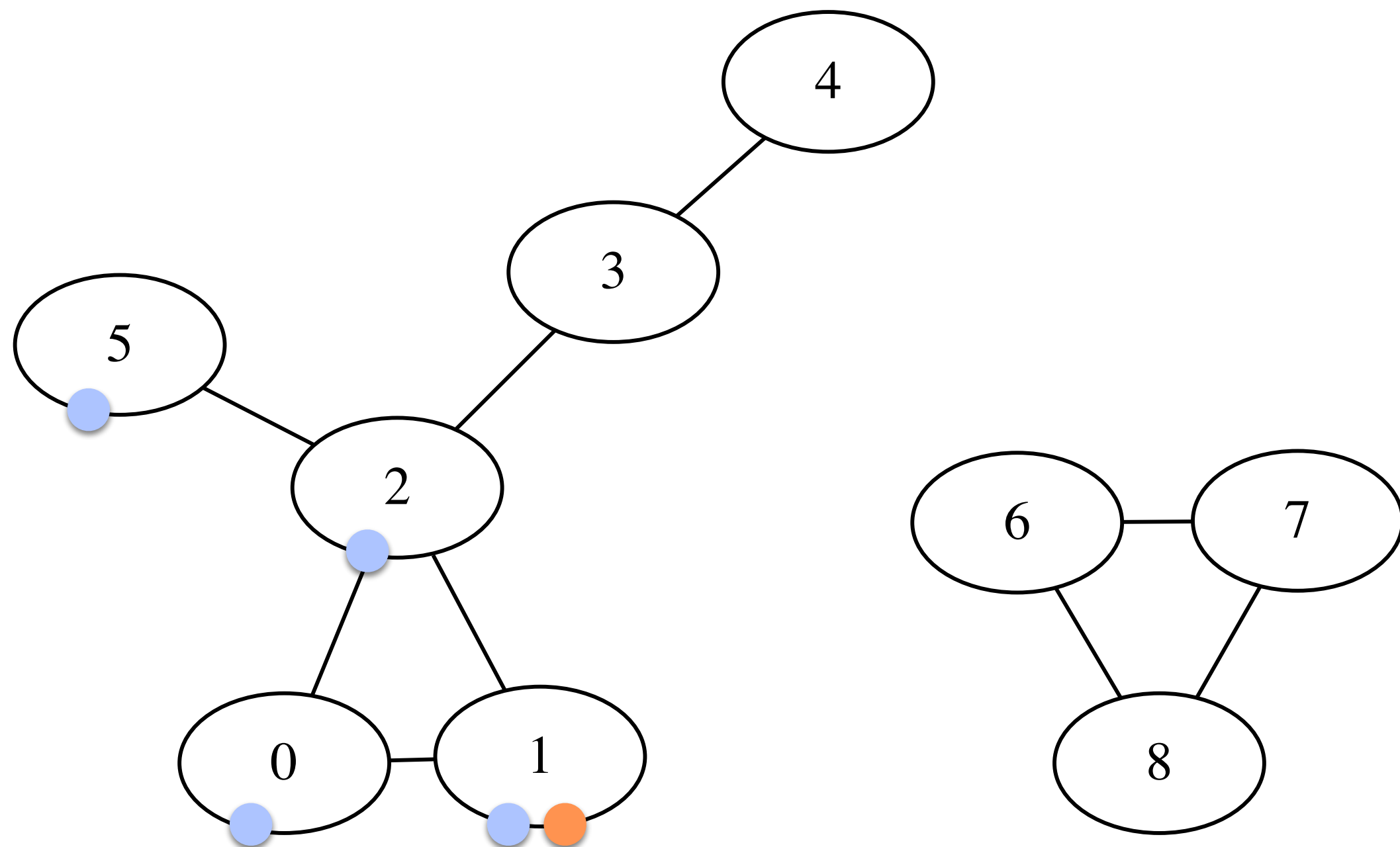
`profondeur(w, pre, post)`

● `post(v)` // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

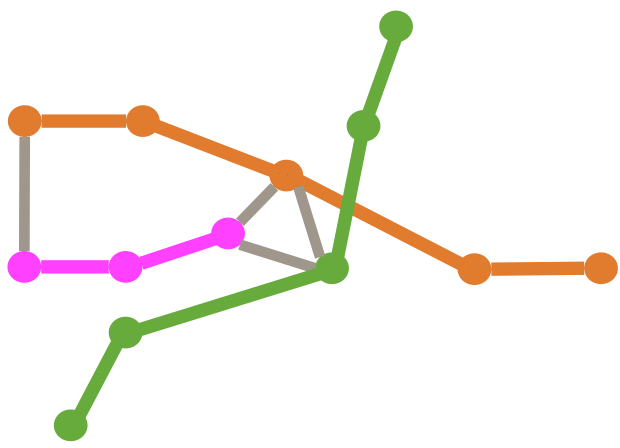
marquer v visité

pour tout w adjacent à v

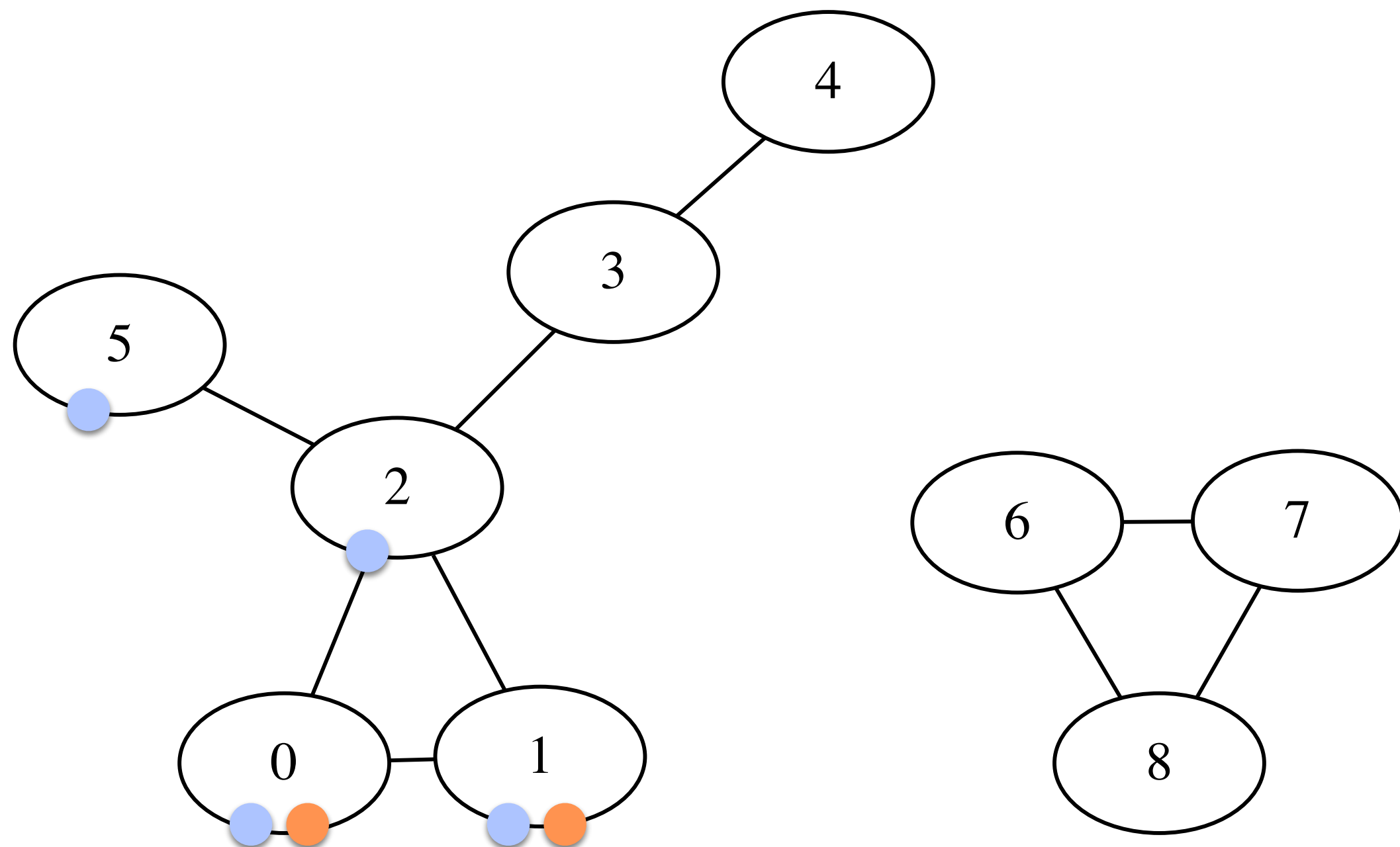
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

marquer *v* visité

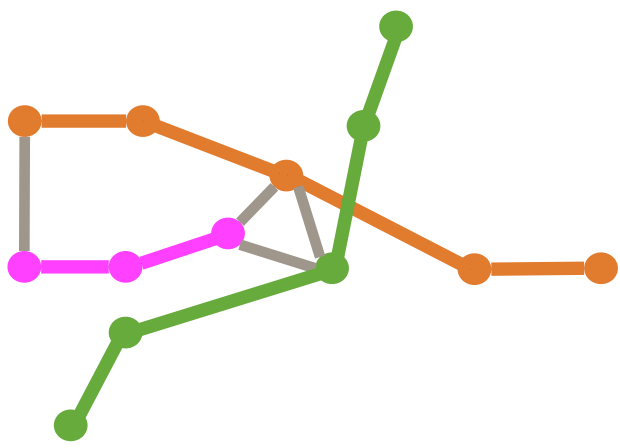
pour tout *w* adjacent à *v*

si *w* n'est pas visité

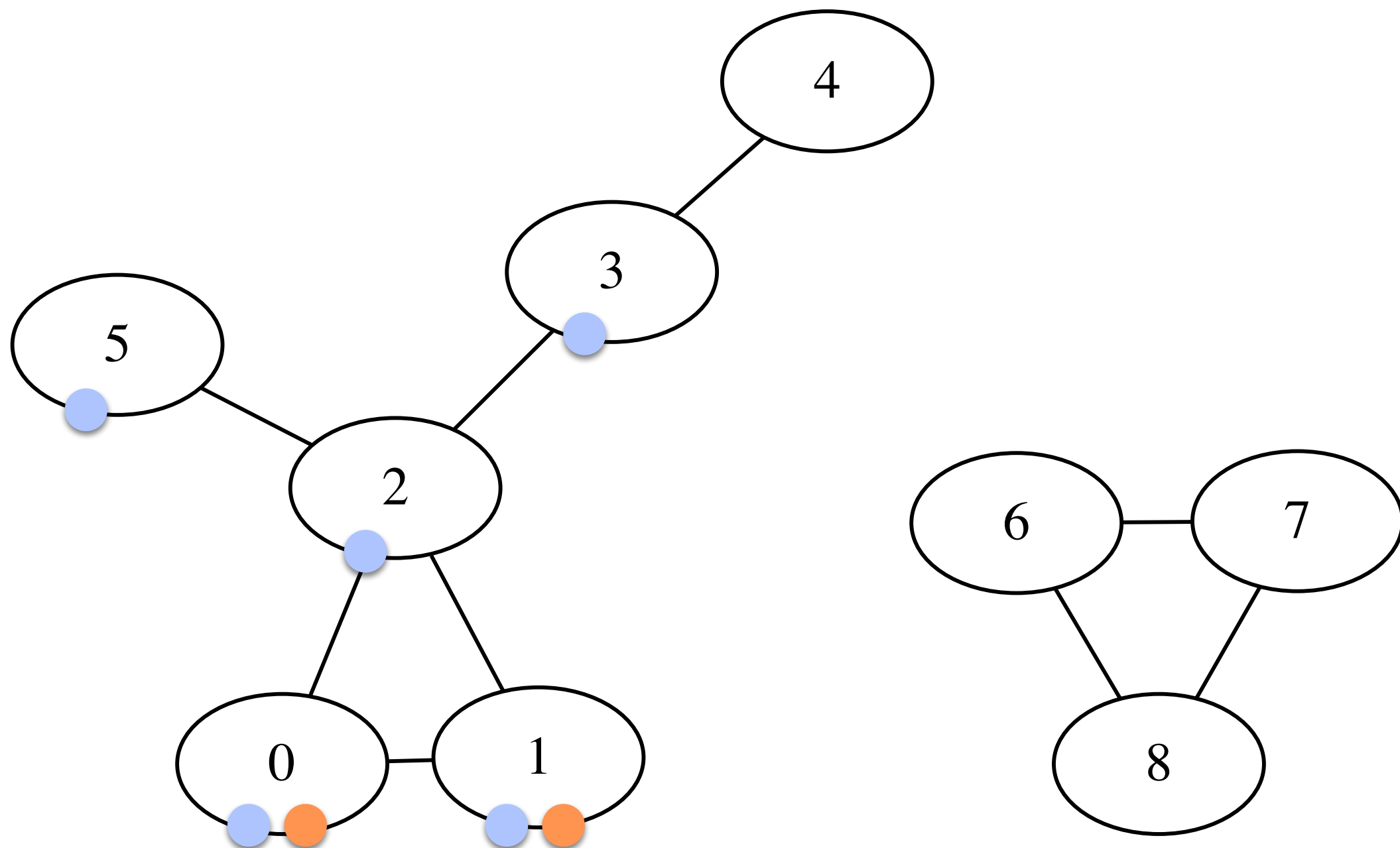
profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

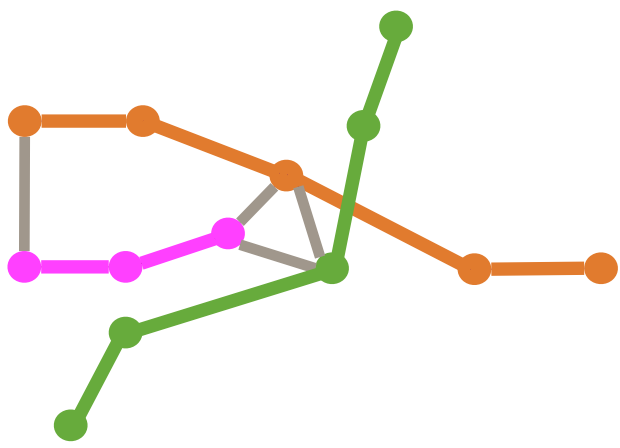
marquer v visité

pour tout w adjacent à v

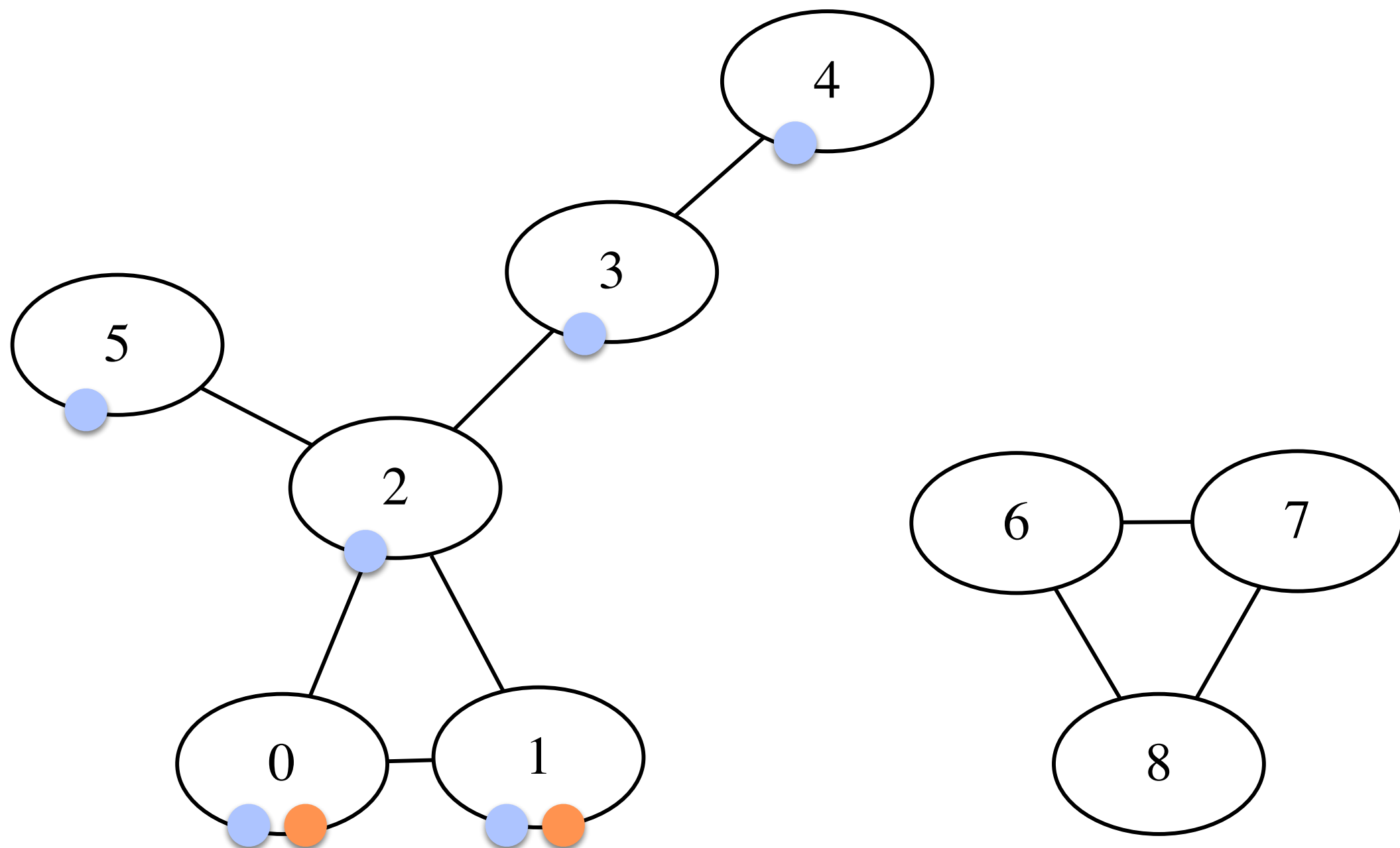
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

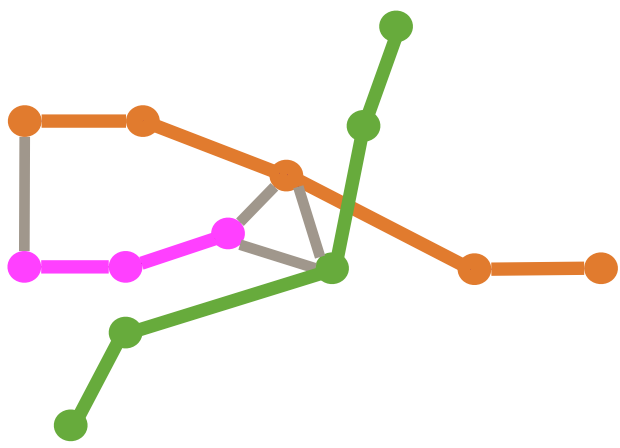
marquer *v* visité

pour tout *w* adjacent à *v*

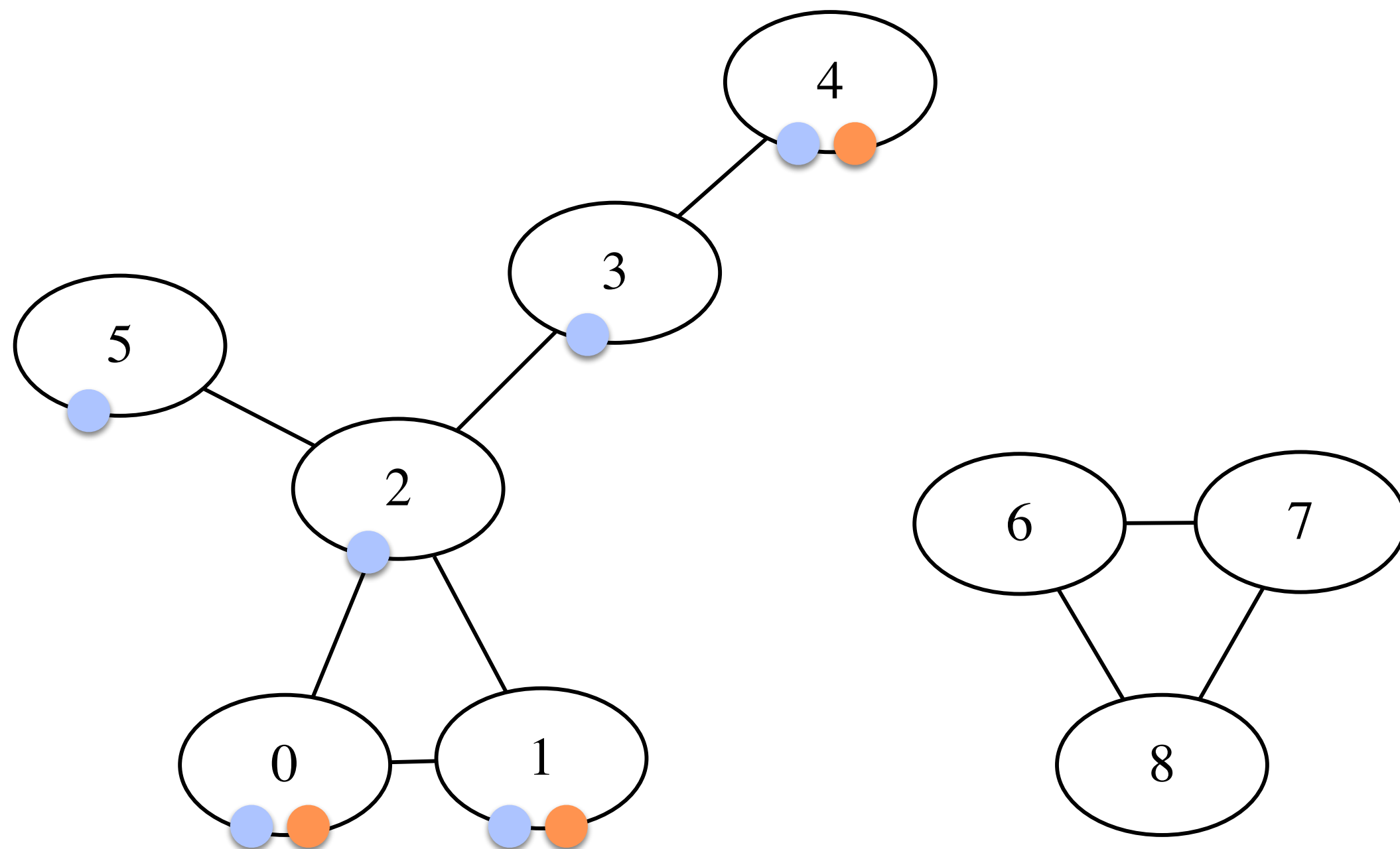
si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

marquer v visité

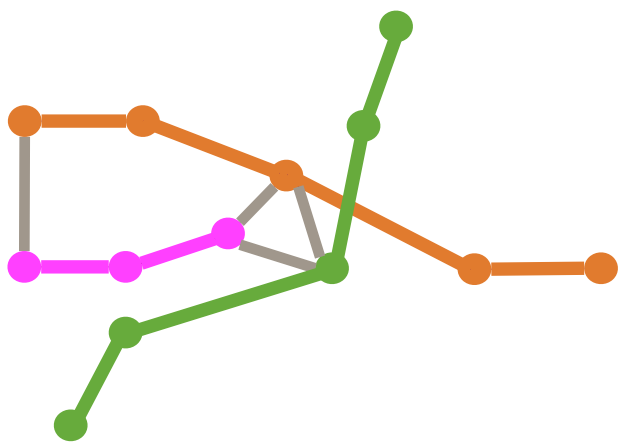
pour tout w adjacent à v

si w n'est pas visité

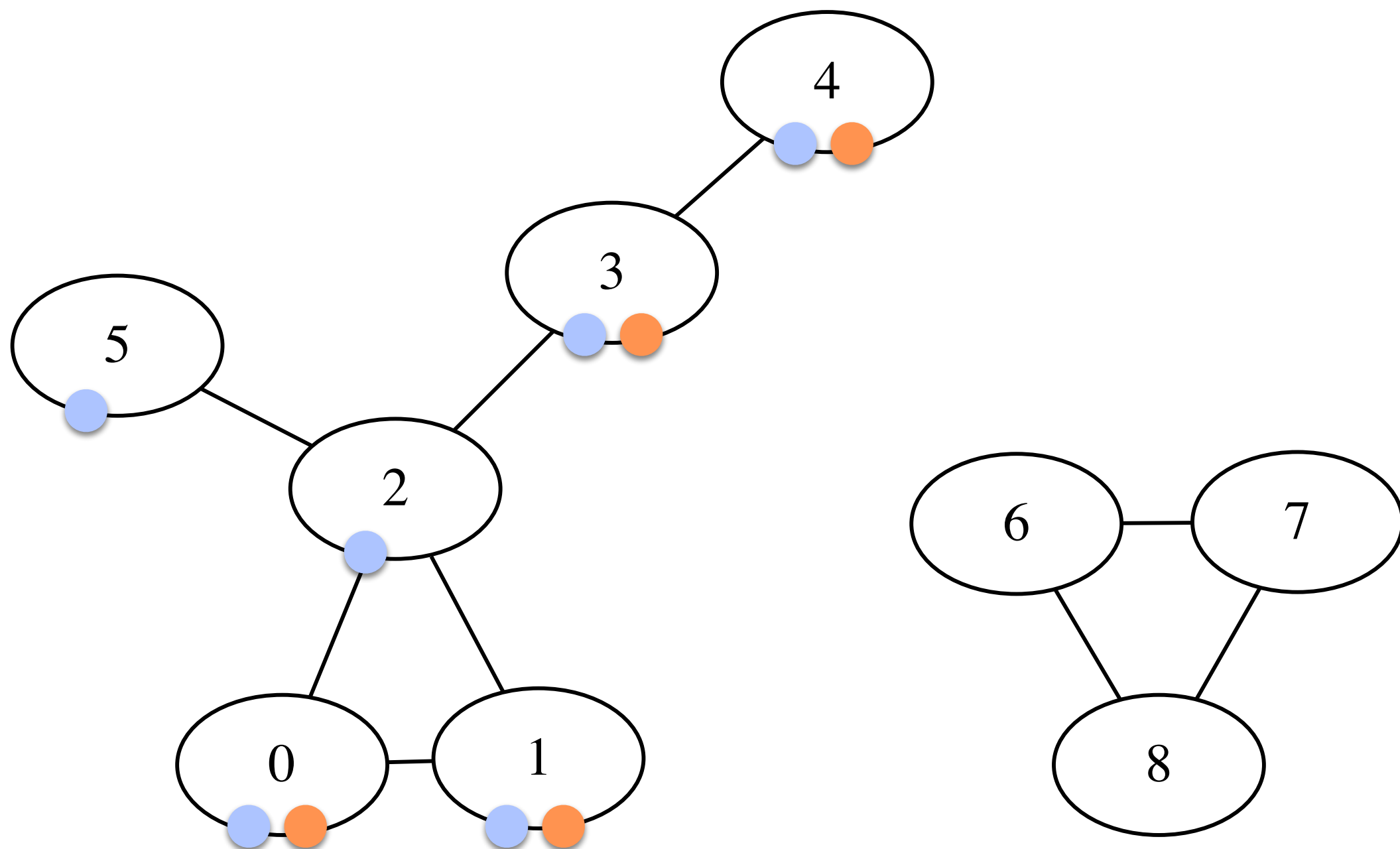
`profondeur(w, pre, post)`

● `post(v)` // en post-ordre

Parcours en profondeur



DFS(5)



Précondition: les sommets sont marqués comme non visités

```
fonction profondeur (sommet v,  
                      Fn pre, (opt.) Fn post)
```

● `pre(v)` // en pré-ordre

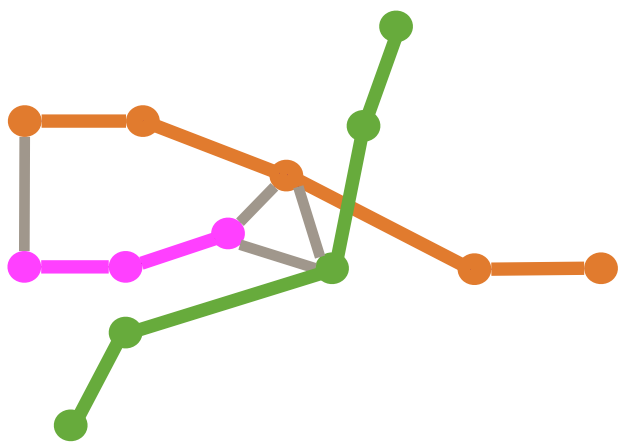
marquer v visité

pour tout w adjacent à v

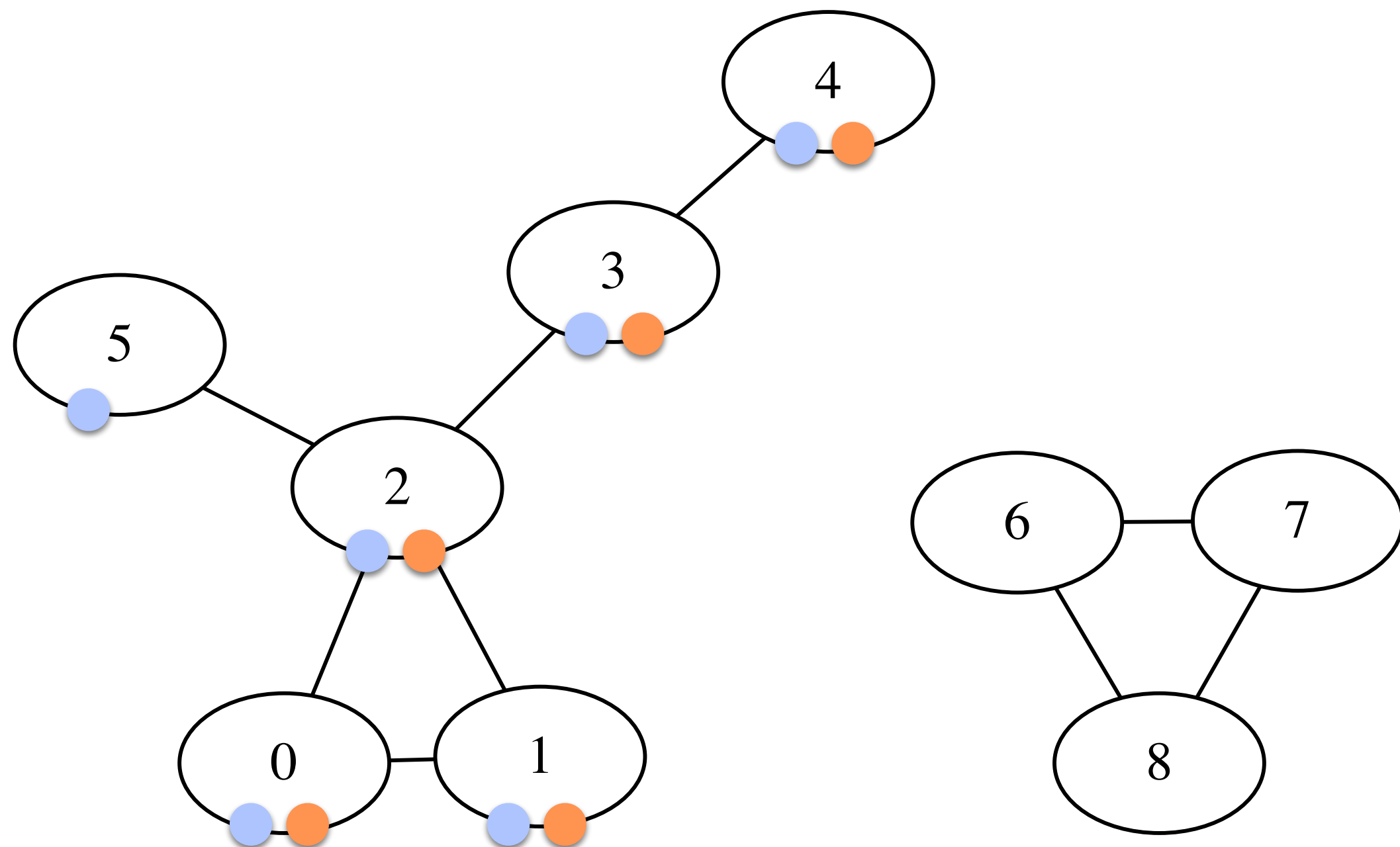
si w n'est pas visité

`profondeur(w, pre, post)`

● `post(v)` // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

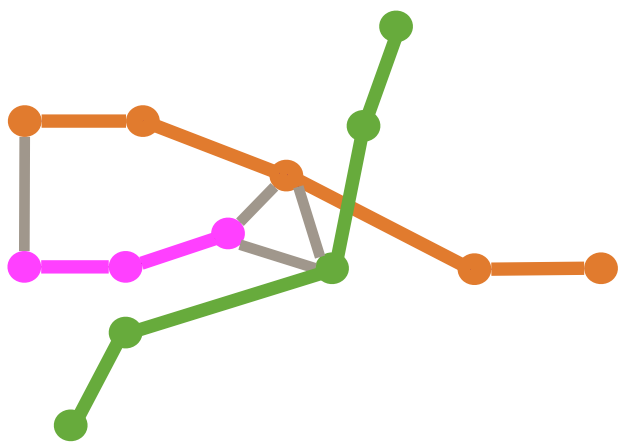
marquer *v* visité

pour tout *w* adjacent à *v*

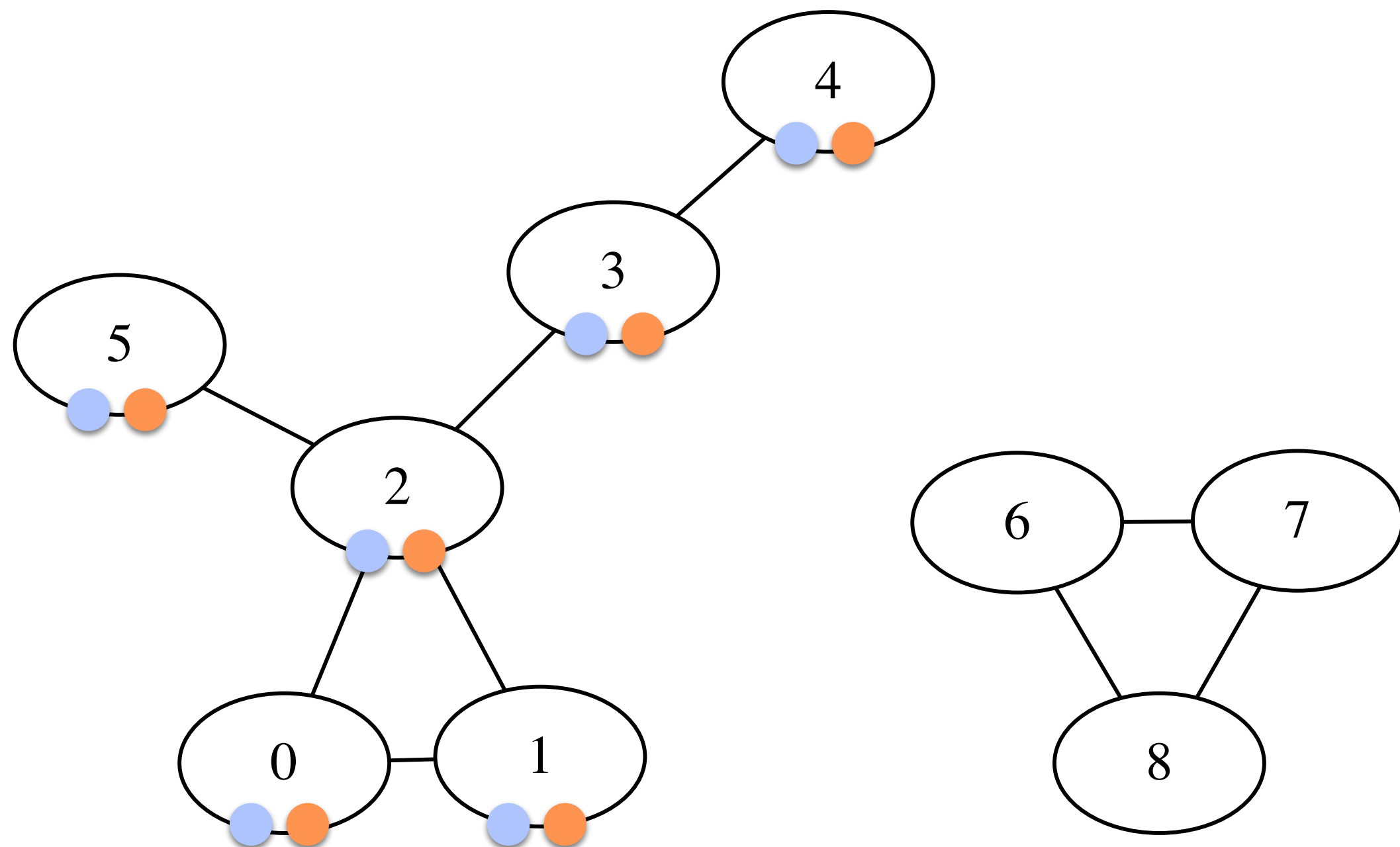
si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre



DFS(5)



Précondition: les sommets sont marqués comme non visités

fonction *profondeur* (sommets *v*,
Fn *pre*, (opt.) Fn *post*)

● *pre(v)* // en pré-ordre

marquer *v* visité

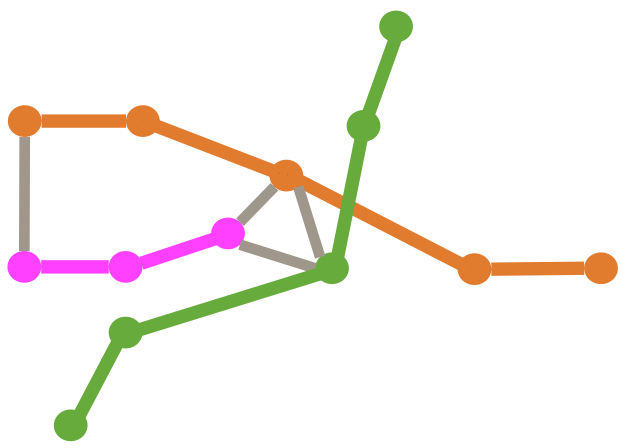
pour tout *w* adjacent à *v*

si *w* n'est pas visité

profondeur(*w*, *pre*, *post*)

● *post(v)* // en post-ordre

Parcours de tout le graphe



- Le parcours depuis un sommet n'atteint pas nécessairement tous les sommets du graphe.

Pourquoi ? Parce que tous deux sommets ne sont pas toujours connectés

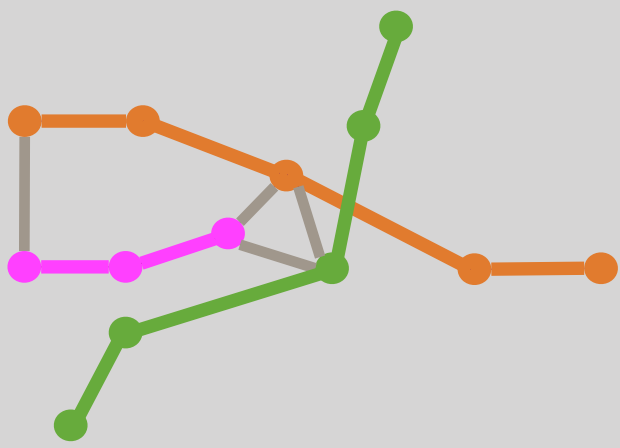
- On peut parcourir tout le graphe en lançant un parcours DFS depuis chaque sommet non atteint par les parcours précédents

```
fonction profondeurG (Graphe G,  
                      Fn action)
```

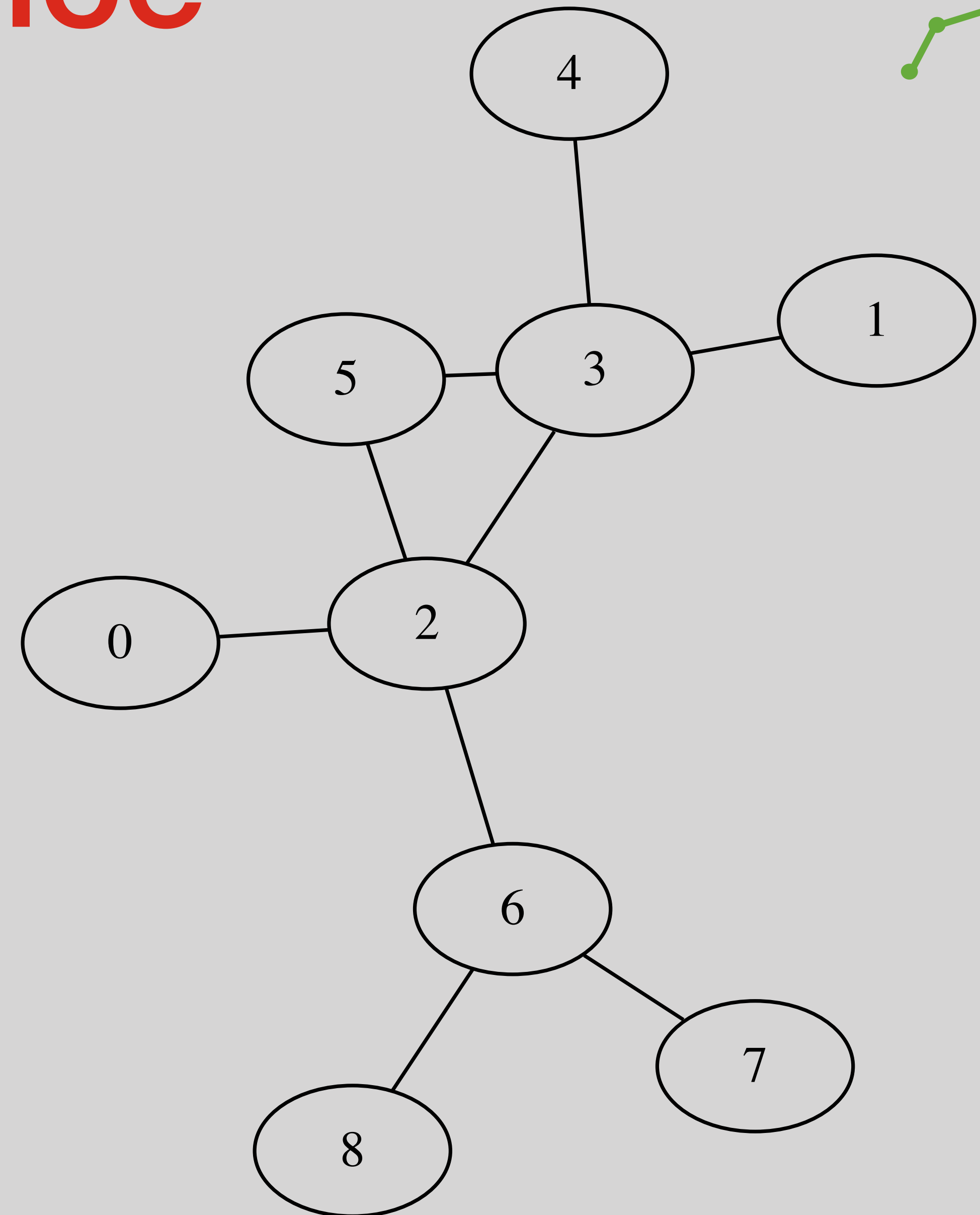
```
    marquer tous les sommets  
    comme non visités
```

```
    pour tout sommet v de G  
        si v n'est pas visité  
            profondeur(v, action)
```

Exercice



- Effectuer les parcours pré et post-ordonnés du graphe suivant à partir du sommet 5.
- On suppose utiliser des listes d'adjacences triées par indice de sommet croissant



Solution

- Effectuer les parcours pré et post-ordonnés du graphe suivant à partir du sommet 5.
- On suppose utiliser des listes d'adjacences triées par indice de sommet croissant

Pré-ordre : 5 2 0 3 1 4 6 7 8
 Post-ordre : 0 1 4 3 7 8 6 2 5

