6. Graphes

6.1 Graphes et leur représentation

Définition 6.1. *Un **graphe non-orienté** est un couple (V, E) où $E \subseteq \binom{V}{2}$ (paires non-ordonnées). V est l'ensemble des **sommets** et E est l'ensemble des **arêtes**.*

Définition 6.2. *Un **graphe orienté** est un couple (V, E) où $E \subseteq V \times V$ (paires ordonnées). V est l'ensemble des **nœuds** ou sommets, et E est l'ensemble des **arcs**.*

Matrice d'adjacence. C'est une matrice $V \times V$, où la cellule $A[u, v]$ contient information sur l'arête uv.

Listes d'adjacence. La représentation par **listes d'adjacence**[1] est un ensemble de listes $\mathsf{Adj}[u]$ pour chaque sommet u qui stocke l'ensemble $\{v : uv \in E\}$. Usage de mémoire : $\Theta(|E| + |V|)$, et c'est meilleur que la matrice dans le cas d'un graphe *éparse* avec $E = o(|V|^2)$. En pratique, on peut stocker Adj comme un tableau, ou une liste chaînée : la structure doit supporter le parcours des voisins.

[1] W(fr):liste d'adjacence

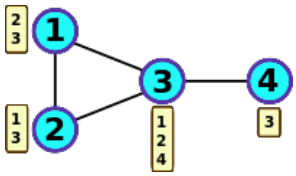


FIG. 1: Listes d'adjacences (en boîtes) dans un graphe à 4 sommets. [Wikimedia Commons]

6.2 Parcours de graphe

On parcourt un graphe à partir d'un **sommet de départ** s, en suivant la logique des algorithmes de parcours des arbres (§4.3). Afin de reconnaître les cycles, on marque les sommets $V = \{0, 1, \ldots, n-1\}$ pendant le parcours (par «coloriage»). On examine les deux approches à explore un graphe : parcours par profondeur et parcours par largeur.

Parcours en profondeur

Dans l'algorithme de **parcours en profondeur**[2] (*depth-first search*), on colorie les sommets par vert au début, puis par jaune (en visite préfixe) et finalement par rouge (en visite postfixe). Dans la version ci-dessous, on stocke la liaison $\mathsf{parent}[u]$ à chaque nœud qui donne le sommet à partir duquel on arrive à u pour la première fois (quand on *découvre* u). Quitte à l'application veut suivre les liasons, il suffit de maintenir le coloriage des sommets pour faire le parcours.

[2] W(fr):parcours en profonderu

```
(init)  parent[s] ← s ; for u ← 0, 1, ..., n − 1 do couleur[u] ← verte
        DFS(s)                          // parcours en profondeur à partir de sommet s
D1      couleur[s] ← jaune                                  // prévisite de s
D2      for st ∈ Adj[s] do                  // pour tout sommet t adjacent à s
D3          if couleur[t] = verte then DFS(t) ; parent[t] ← s   // visite du voisin t
D4      couleur[s] ← rouge                                  // post-visite de s
```

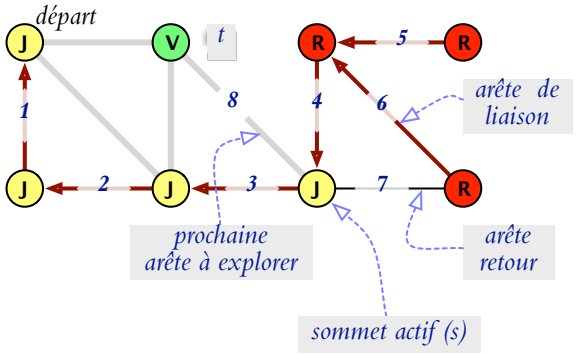


FIG. 2: Parcours en profondeur (DFS). Quand on visite une arête st pour la première fois, la couleur du sommet t peut être **verte** : arête st et sommet t découverts pour la première fois (c'est une arête de **liaison**), **jaune** : arête st découverte pour la première fois, mais t est connu (c'est une arête **retour**), ou **rouge** : jamais, car on visite toutes les arêtes adjacentes, incluant ts, avant de rougir t.

Temps de calcul. Le parcours finit en $O(|V| + |E|)$ temps : on considère chaque arête exactement deux fois (Ligne D2), et on colorie chaque sommet exactement trois fois.

Composantes connexes. On explore tout le graphe en lançant DFS à partir de tout sommet qui reste vert vert.

```
for s ← 0, 1, ... n − 1 do if couleur[s] = verte then DFS(s)
```

Il est facile de voir que le parcours en profondeur peut être employé pour identifier les composantes connexes du graphe. Les arêtes de liaison forment un ensemble d'arborescences, ou un **forêt en profondeur** qui couvre tous les sommets. En suivant les liaisons parent, on trouve un chemin entre un sommet quelconque v et le sommet de départ. (C'est le chemin formé par les sommets jaunes quand v est découvert.)

Graphe biparti. Un **graphe biparti** est un graphe non-orienté (V, E) dans lequel V peut être partionnée en deux ensembles V_1 et V_2 ($V_1 \cup V_2 = V; V_1 \cap V_2 = \emptyset$) tels que toutes les arêtes passent entre V_1 et V_2 : si $uv \in E$, alors $u \in V_1$ et $v \in V_2$ ou $u \in V_2$ et $v \in V_1$. On peut tester si un graphe est biparti pendant le parcours : il suffit de partitionner les sommets quand on les découvre, et tester si les arêtes retour passent bel et bien entre les deux ensembles.

```
(init)  partition[s] ← 1 ; for u ← 0, 1, ..., n − 1 do couleur[u] ← verte
    DFS-BIP(s)                          // tester si la composante connexe des est biparti
1   couleur[s] ← jaune
2   for st ∈ Adj[u] do
3       if couleur[t] = verte then                                  // arête de liaison
4           partition[t] = 3 − partition[s]                         // 1 ↔ 2
5           DFS-BIP(t)
6       else if couleur[t] = jaune then                             // parent ou arête retour
7           if partition[s] ≠ partition[t] then die(«Cycle de longueur impaire»)
```

FIG. 3: Arbre DFS dans un graphe de 250 nœuds. [SW 2011]

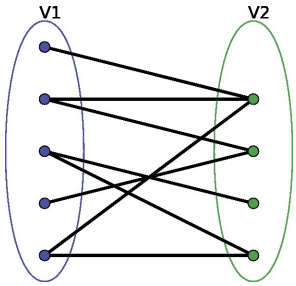
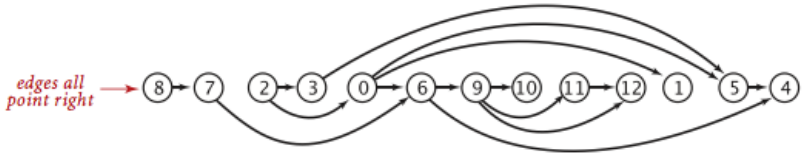


FIG. 4: Graphe biparti : sommets partitionnés en deux (côtés gauche et droit V_1 et V_2), aucune arête entre sommets du même côté.

Tri topologique. Soit $G = (S, A)$ un **graphe acyclique orienté**. Le **tri topologique**[3] cherche une permutation de sommets telle que si $uv \in A$, alors u se trouve avant v dans l'ordre. En fait, l'ordre inverse des visites postfixes est un tri topologique. Il suffit alors d'utiliser une pile P pour stocker les sommets en post-visite.

D4 couleur$[s] \leftarrow$ rouge ; P.push(u)

À la fin, P.pop défile les sommets dans l'ordre du tri topologique.



[3] W(fr):tri topologique

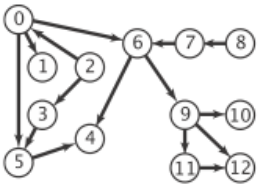


FIG. 5: Un graphe acyclique orienté à 13 nœuds. [SW 2011]

FIG. 6: Ordre de visite et tri topologique. [SW 2011]

Parcours en largeur

Lors d'un **parcours en largeur**[4] (*breadth-first search*), on enfile les voisins dans une file FIFO (queue) — parcours en profondeur correspond à l'usage d'une pile. Dans la version ci-dessous, on maintient la distance d à partir du sommet de source. Les arêtes de liaison forment un **arborescence en largeur** couvrant une composante connexe. Dans cet arbre, enraciné au sommet de départ s, tout $d[u]$ est la profondeur du nœud u.

```
1  BFS(s)
2  for u ← 0, 1, ..., n − 1 do couleur[u] ← jaune
3  d[s] ← 0 ; parent[s] ← s
4  Q.enqueue(s); couleur[s] ← jaune
5  while Q ≠ ∅ do
6  |  u ← Q.dequeue()
7  |  for uv ∈ Adj[u] do
8  |  |  if couleur[v] = verte then
9  |  |  |  d[v] ← d[u] + 1; parent[v] ← u
10 |  |  |  Q.enqueue(v); couleur[v] ← jaune
11 |  |  end
12 |  end
13 |  couleur[u] ← rouge
14 end
```

[4] W(fr):parcours en largeur

FIG. 7: Arbre BFS dans un graphe de 250 nœuds. [SW 2011]

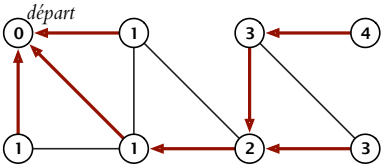


FIG. 8: Le parcours en largeur découvre la distance du sommet de départ.

Temps de calcul. Le parcours prend $O(|V| + |E|)$ avec listes d'adjacence.

Plus courts chemins. À la fin du parcours, $d[u]$ est la longueur minimale d'un chemin entre s et u pour tout sommet s. On peut retracer ce plus court chemin en suivant les liaisons parent.