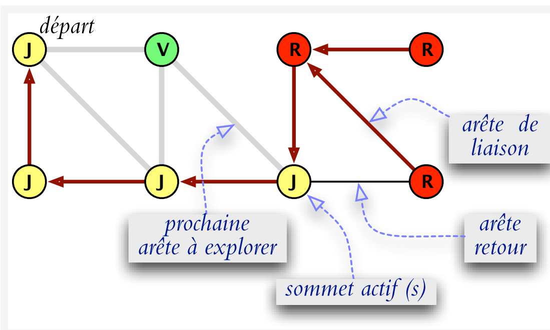
10 Parcours de graphes

10.1 Parcours en profondeur

W(fr)

On parcourt un graphe à partir d'un **sommet de départ** s, en suivant la logique des algorithmes de parcours des arbres (§6.4). Afin de reconnaître les cycles, on doit marquer les sommets $V = \{0, 1, \ldots, n-1\}$ pendant le parcours. Dans l'algorithme de **parcours en profondeur** (*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 parent$[u]$ à chaque nœud qui donne le sommet à partir duquel on arrive à u pour la première fois (quand on *découvre* u).

(init)	parent$[s] \leftarrow s$; **for** $u \leftarrow 0, 1, \ldots, n-1$ **do** couleur$[u] \leftarrow$ verte	
	DFS(s)	*// parcours en profondeur à partir de sommet s*
D1	couleur$[s] \leftarrow$ jaune	*// prévisite de s*
D2	**for** $st \in$ Adj$[u]$ **do**	*// pour tout sommet t adjacent à u*
D3	**if** couleur$[t]$ = verte **then** DFS(t) ; parent$[t] \leftarrow s$	*// visite du voisin t ; stocker la liaison*
D4	couleur$[s] \leftarrow$ rouge	*// post-visite de s*



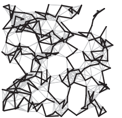
Quand on visite une arête st pour la première fois dans la ligne D3, la couleur du sommet t peut être

- **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**)
- **rouge** : jamais, parce qu'on visite toutes les arêtes adjacentes avant de colorier t par rouge ; st a déjà été visitée à partir de t

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

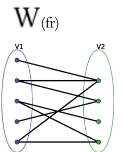
Forêt en profondeur. On peut explorer tout le graphe en lançant DFS à partir de tout sommet :

for $s \leftarrow 0, 1, \ldots n-1$ **do if** couleur$[s]$ = verte **then** DFS(s)



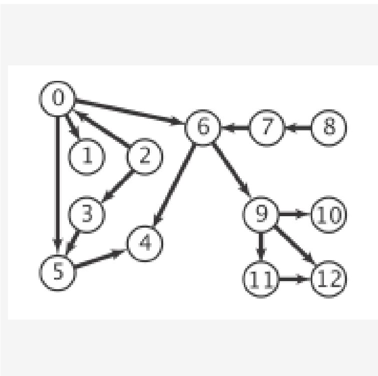
Il est facile de voir que le parcours en profondeur peut être employé pour identifier les composantes connexes du graphe. Les arêtes de liaison forment un ensemble d'arborescences, ou un **forêt en profondeur** qui couvre tous les sommets. En suivant les liaisons parent, on trouve un chemin entre un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

W(fr)



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

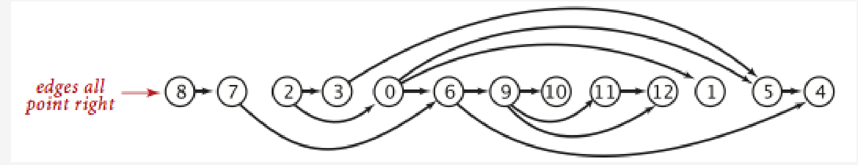
```
(init)  placement[s] ← gauche ; for u ← 0, 1, ..., n − 1 do couleur[u] ← verte
        DFS-BIP(s)                         // tester si la composante connexe des est biparti
  1     couleur[s] ← jaune
  2     for st ∈ Adj[u] do
  3         if couleur[t] = verte then                               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



Soit $G = (S, A)$ un **graphe acyclique orienté**. Le **tri topologique** 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] ← rouge ; P.push(u)
```

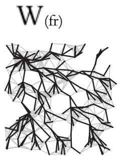
À la fin, P.pop défile les sommets dans l'ordre du tri topologique.



W(fr)

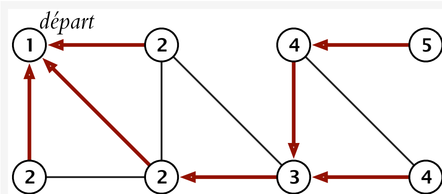
10.2 Parcours en largeur

W(fr)



Lors d'un parcours en largeur (*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.

```
(init)  parent[s] ← s ; initialiser queue Q ← ∅ ; for u ← 0, 1, ..., n − 1 do couleur[u] ← verte
        BFS(s)                                          // parcours en largeur à partir de s
  B1    couleur[s] ← jaune ; d[u] ← 0
  B2    Q.enqueue(s)
  B3    while Q ≠ ∅                                          // tandis que la file n'est vide
  B4        u ← Q.dequeue()
  B5        for uv ∈ Adj[u] do
  B6                if couleur[v] = verte
  B7                then couleur[v] ← jaune; Q.enqueue(v) ; d[v] ← d[u] + 1; parent[v] ← u
  B8        couleur[u] ← rouge
```



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. On peut retrouver ce plus court chemin en suivant les liaisons parent.