R2.07 Graphes

Corentin Dufourg, Régis Fleurquin et Thibault Godin

IUT de Vannes Informatique

Comment parcourir un graphe ?

Imaginer un labyrinthe ou une map de jeu vidéo $\rightsquigarrow$ on ne connait pas la topologie du graphe.

Comment parcourir un graphe ?

Imaginer un labyrinthe ou une map de jeu vidéo $\rightsquigarrow$ on ne connait pas la topologie du graphe.

On veut

- Explorer tout le graphe

Comment parcourir un graphe ?

Imaginer un labyrinthe ou une map de jeu vidéo $\rightsquigarrow$ on ne connait pas la topologie du graphe.

On veut

- Explorer tout le graphe
- Le faire efficacement

Comment parcourir un graphe ?

Imaginer un labyrinthe ou une map de jeu vidéo $\rightsquigarrow$ on ne connait pas la topologie du graphe.

On veut

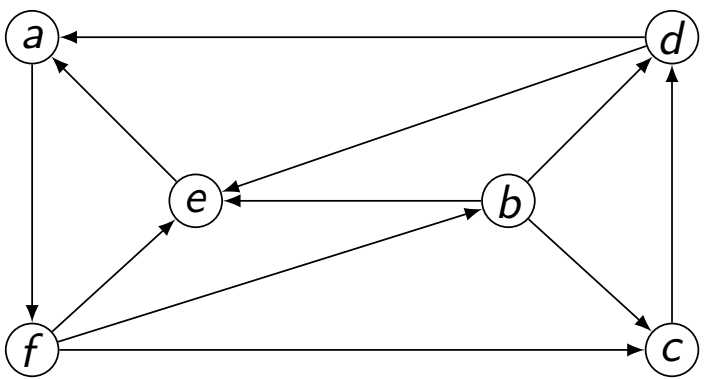
- Explorer tout le graphe
- Le faire efficacement

Comment parcourir un graphe ?

Imaginer un labyrinthe ou une map de jeu vidéo $\rightsquigarrow$ on ne connait pas la topologie du graphe.

On veut

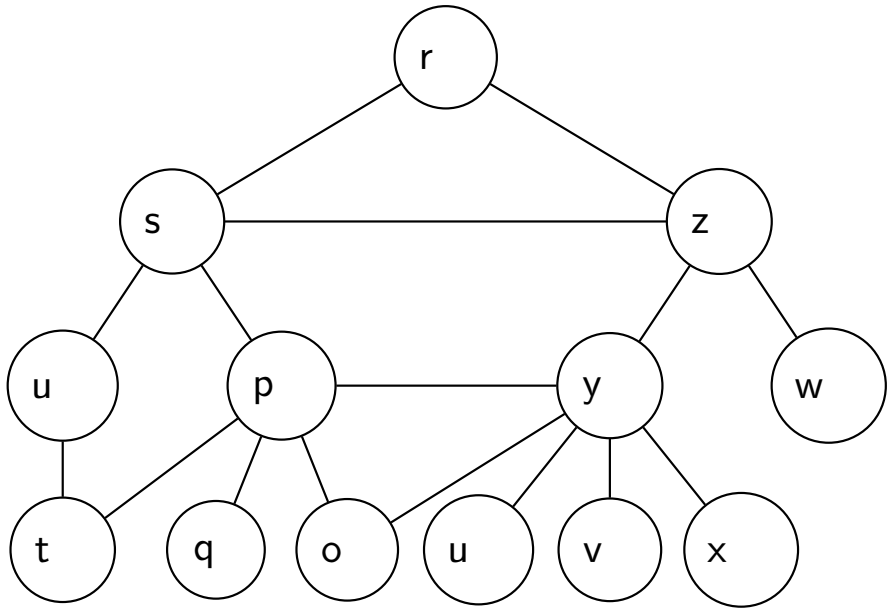
- Explorer tout le graphe
- Le faire efficacement



Plan

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter, {v});
    etat[u]←examine
```



Structure : File (FiFo)

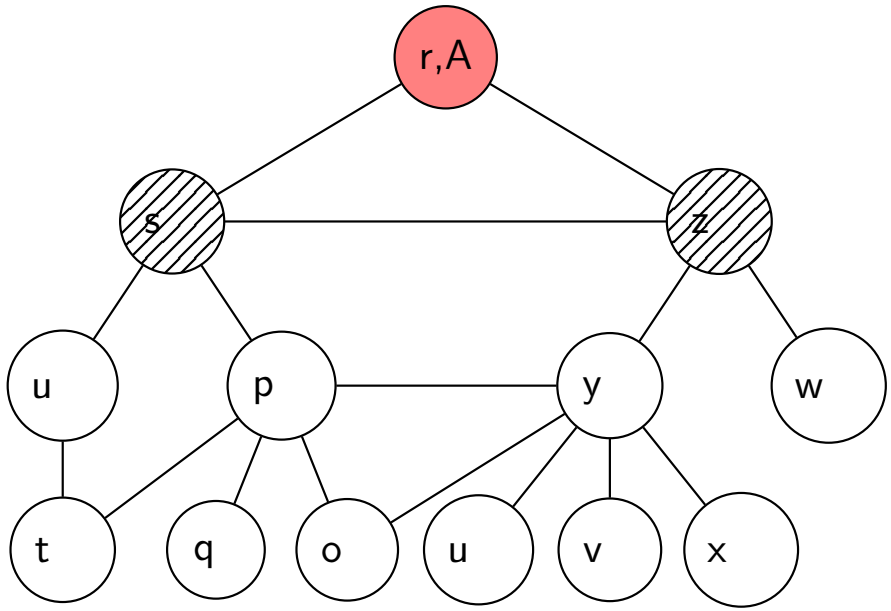
Complexité : $O(S + A)$

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter, {v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | z | s | $\xrightarrow{out}$

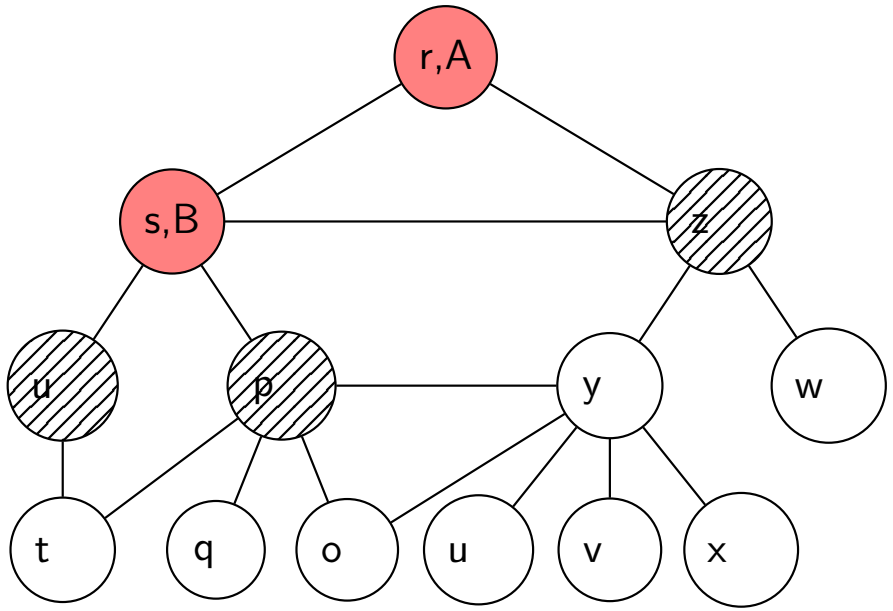
Current : r

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | p | u | z | $\xrightarrow{out}$

Current : s

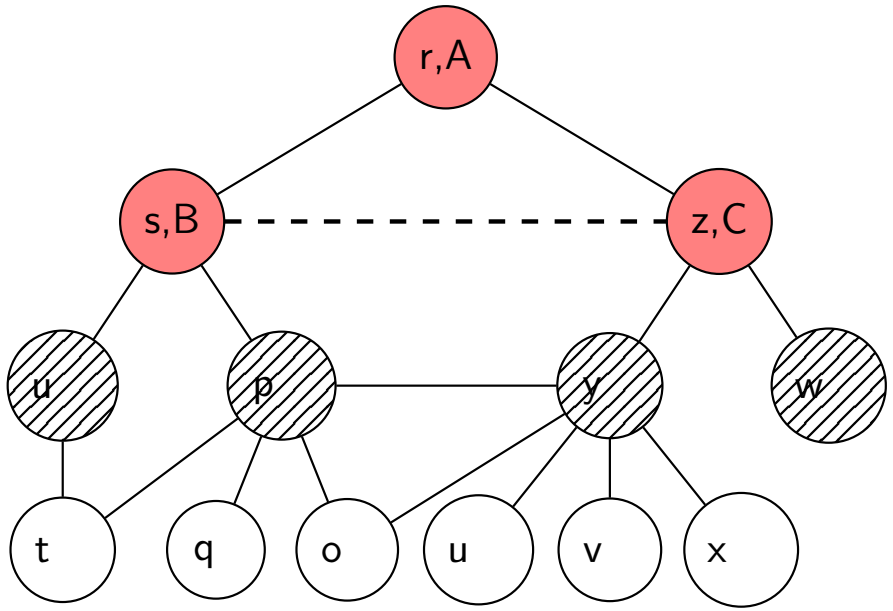
Already reached : z

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | w | y | p | u | $\xrightarrow{out}$

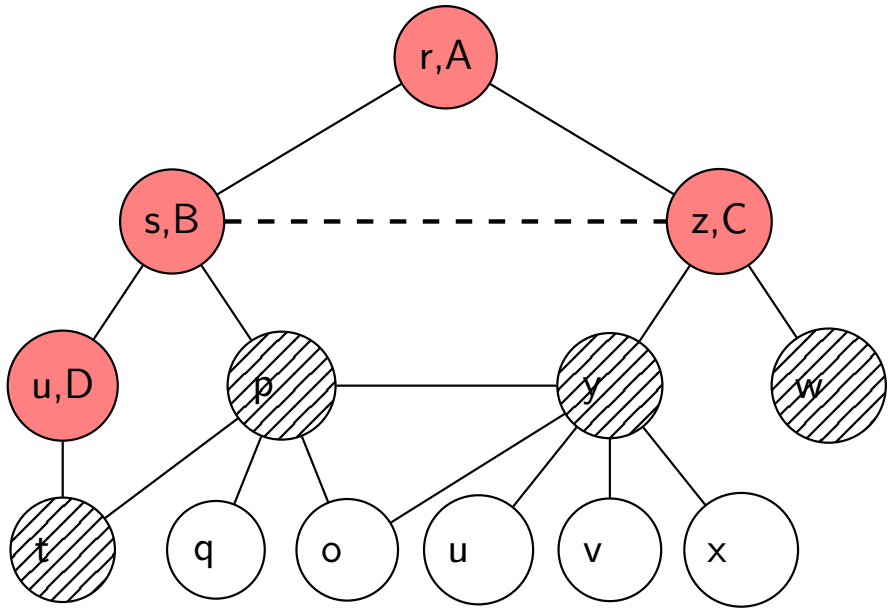
Current : z

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | t | w | y | p | $\xrightarrow{out}$

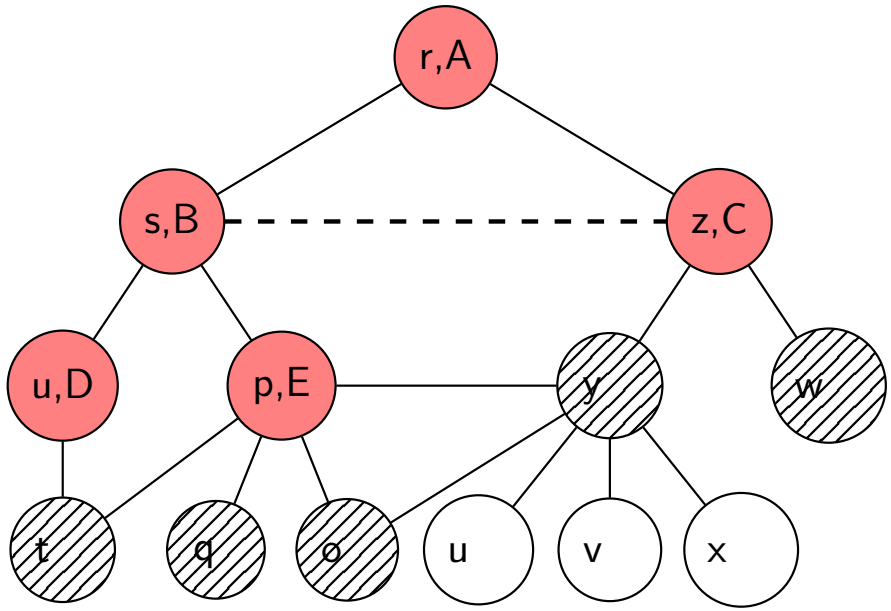
Current : u

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | t | w | y | $\xrightarrow{out}$

Current : p

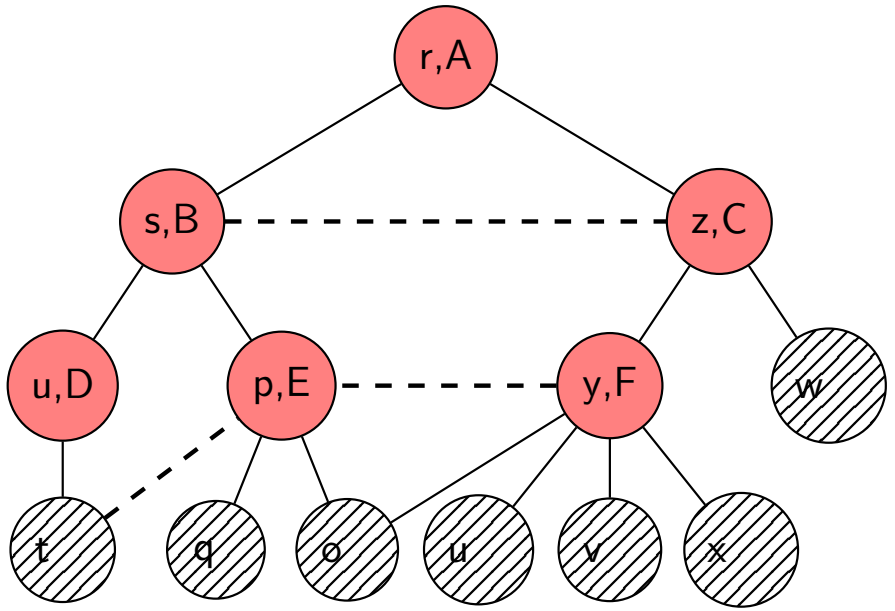
Already reached : t,y

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | o | q | t | w | $\xrightarrow{out}$

Current : y

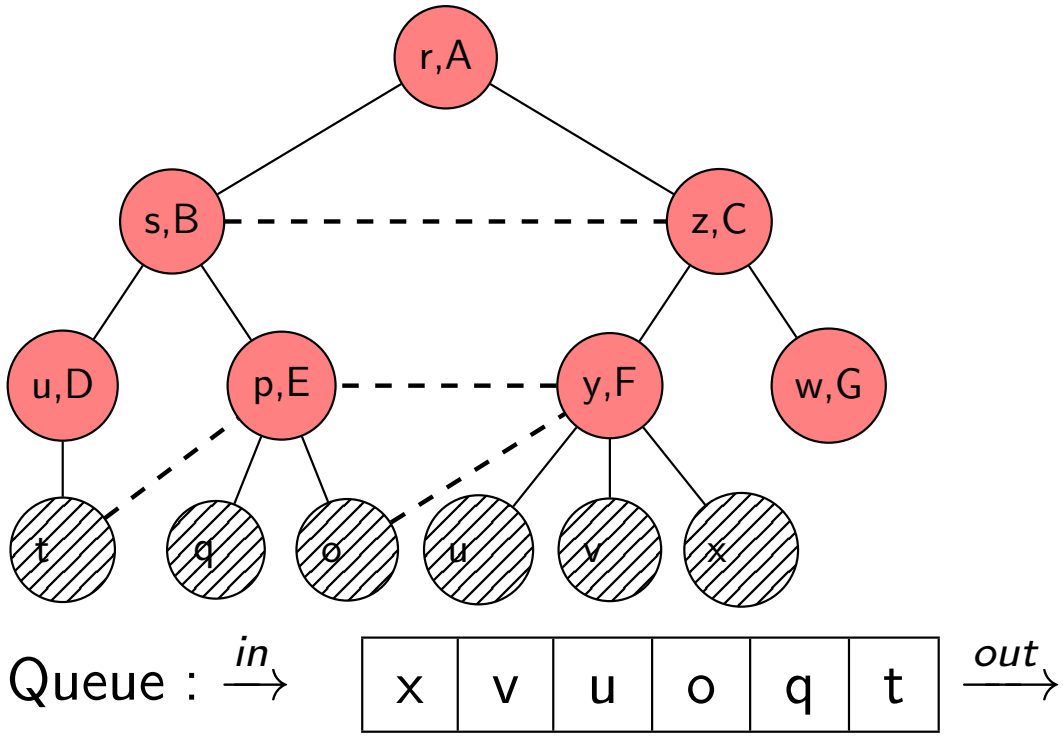
Already reached : o,p

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$

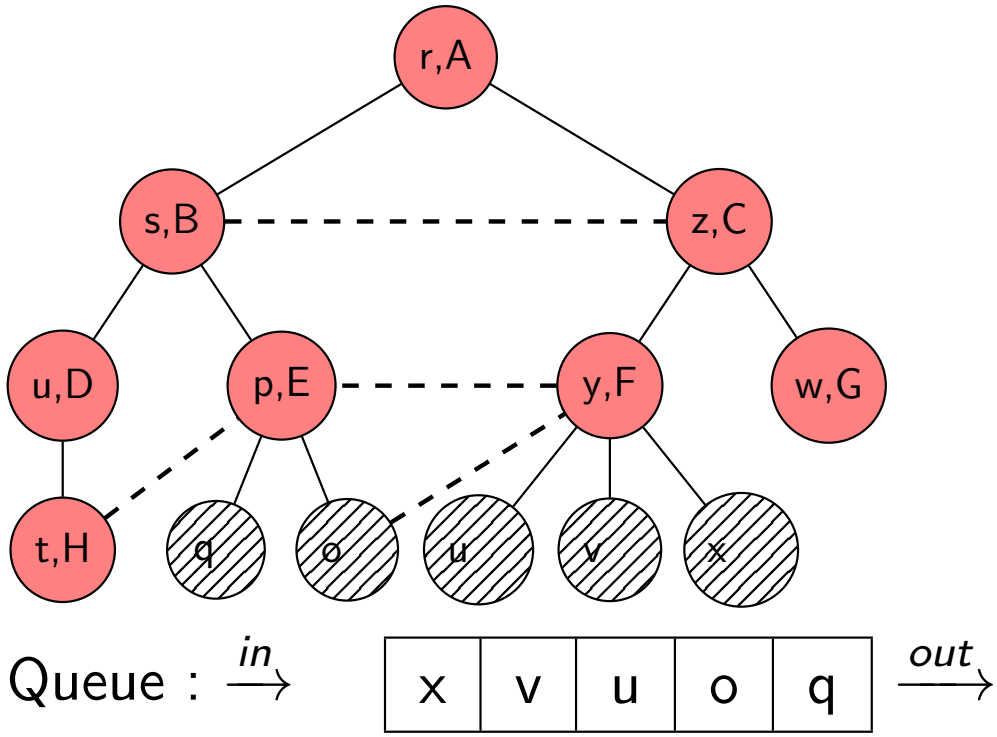


Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$

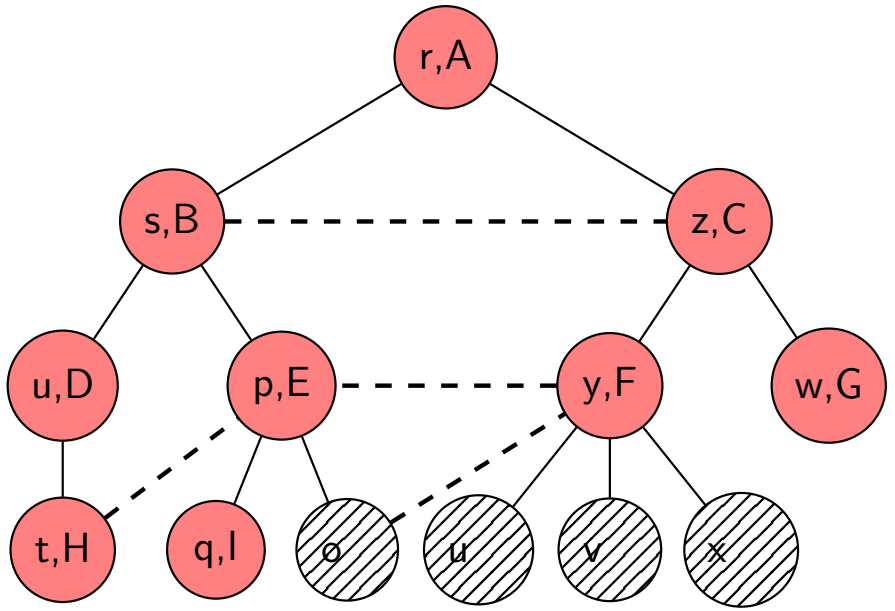


Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Queue : $\xrightarrow{in}$ | x | v | u | o | $\xrightarrow{out}$

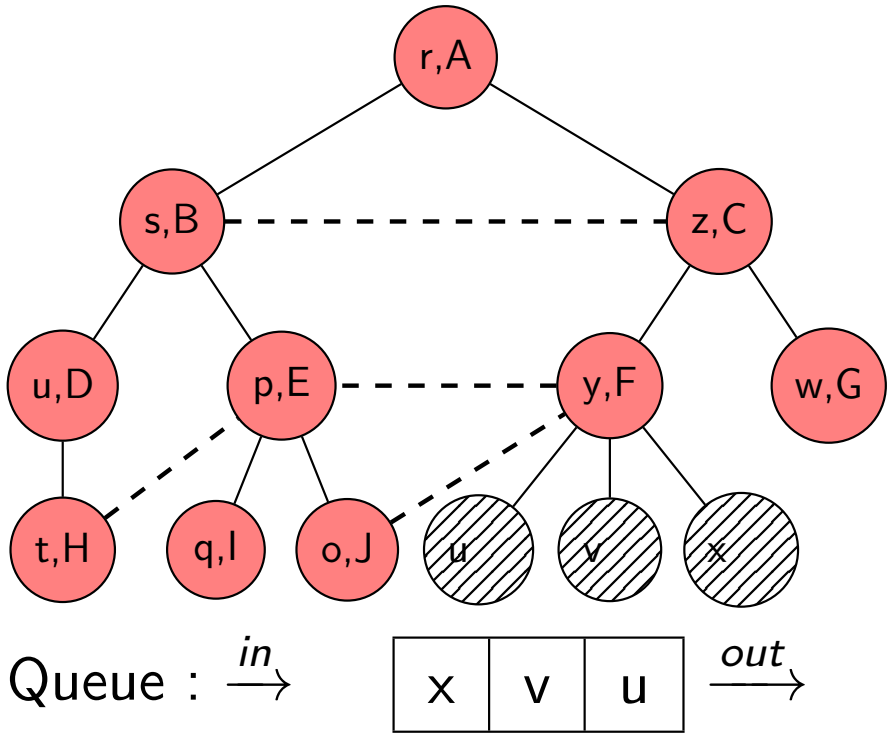
Current : q

Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$

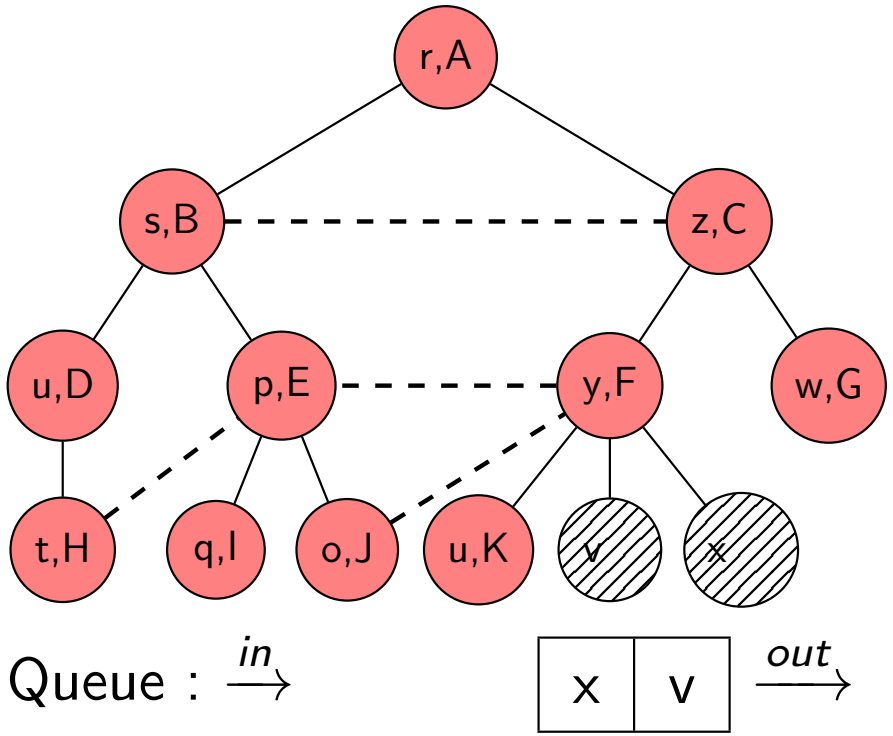


Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$

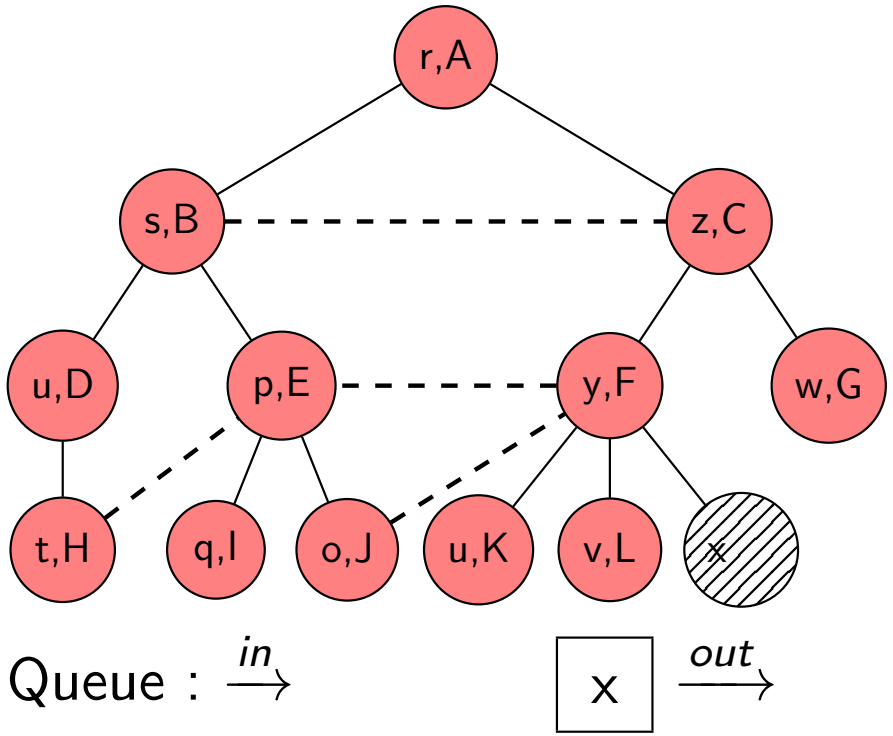


Parcours en largeur (BFS)

```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter,{x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter,{v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$

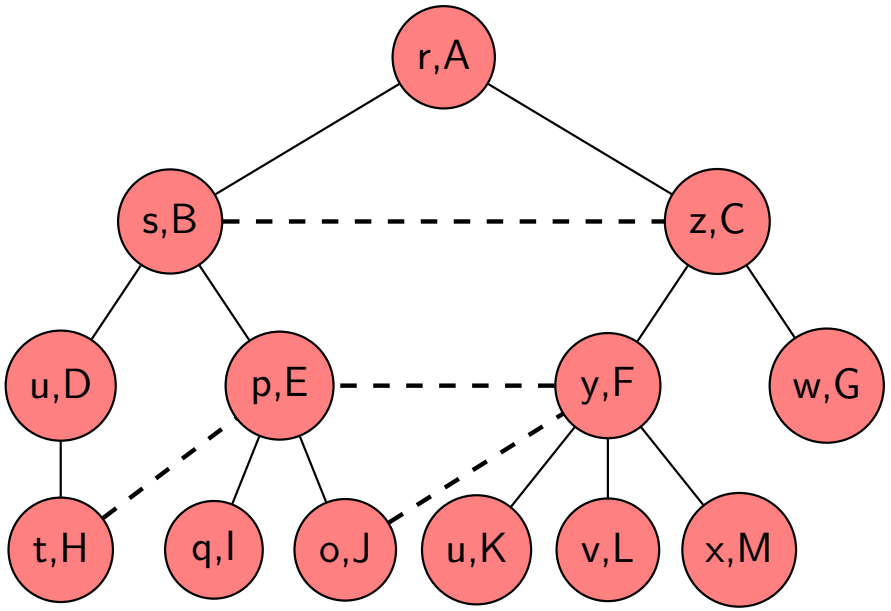


Parcours en largeur (BFS)

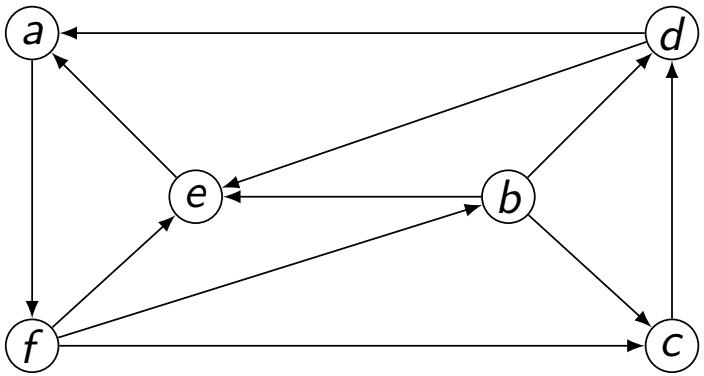
```
for tout x ∈ S do
    etat[x] ← non_atteint;
    parent[x] ← -1;
T ← ∅
à_traiter = empty queue
(*initialisation*);
enqueue(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    u←dequeue(à_traiter)
    à_traiter←à_traiter\{u}
    for tout v voisin de u do
        if etat[v] = non_atteint
        then
            T←T ∪ {{u, v}}
            etat[v]←atteint
            parent[v]← u
            enqueue(à_traiter, {v});
    etat[u]←examine
```

Structure : File (FiFo)

Complexité : $O(S + A)$



Exemple–BFS



Plan

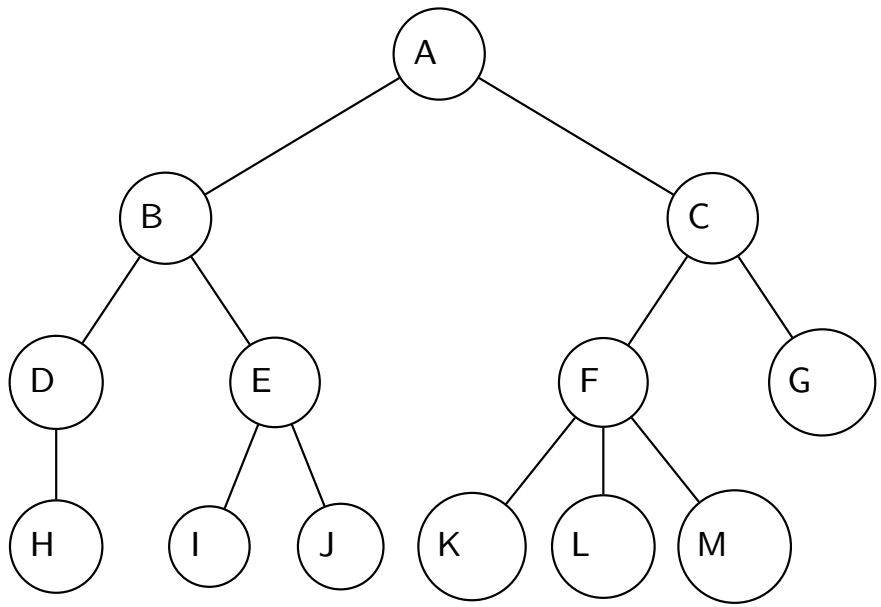
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



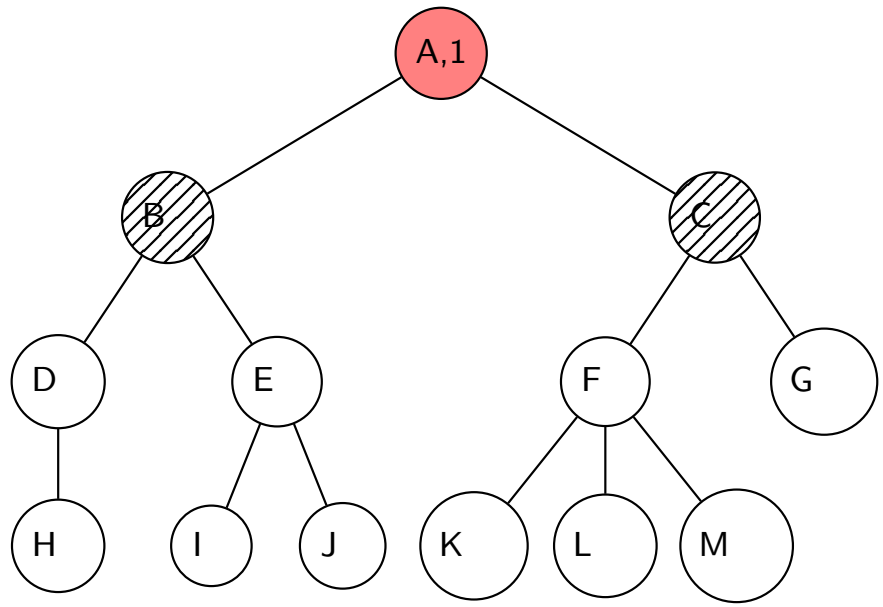
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | B | C |

Current : A

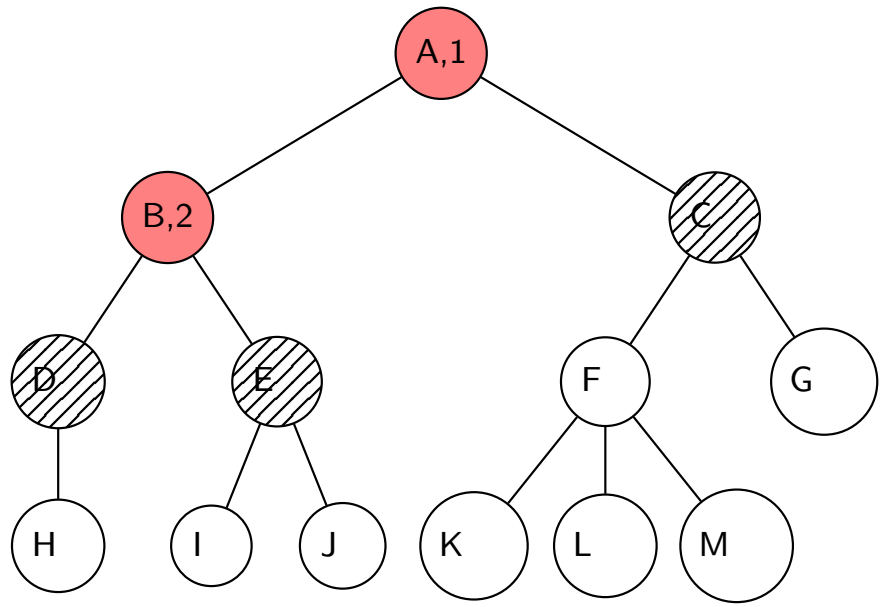
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | D | E | C |

Current : B

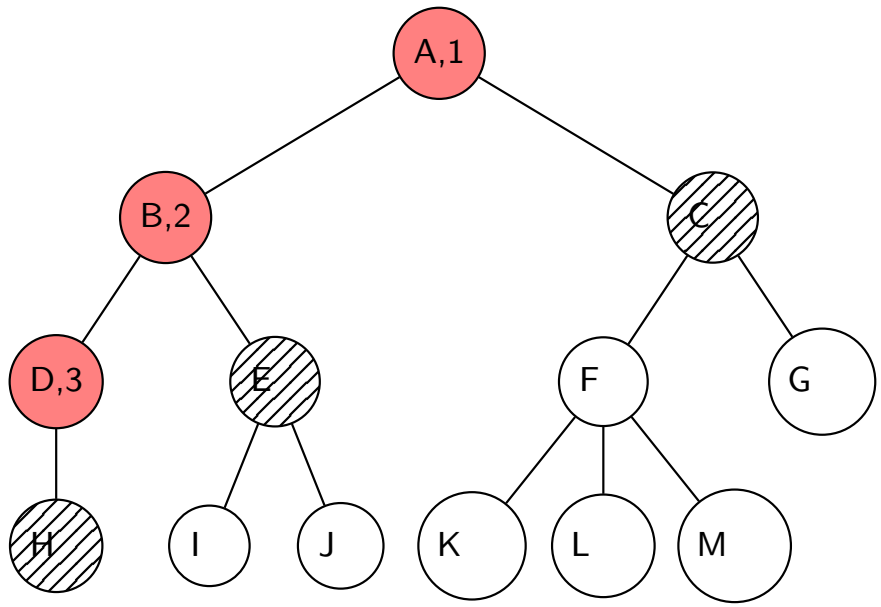
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | H | E | C |

Current : D

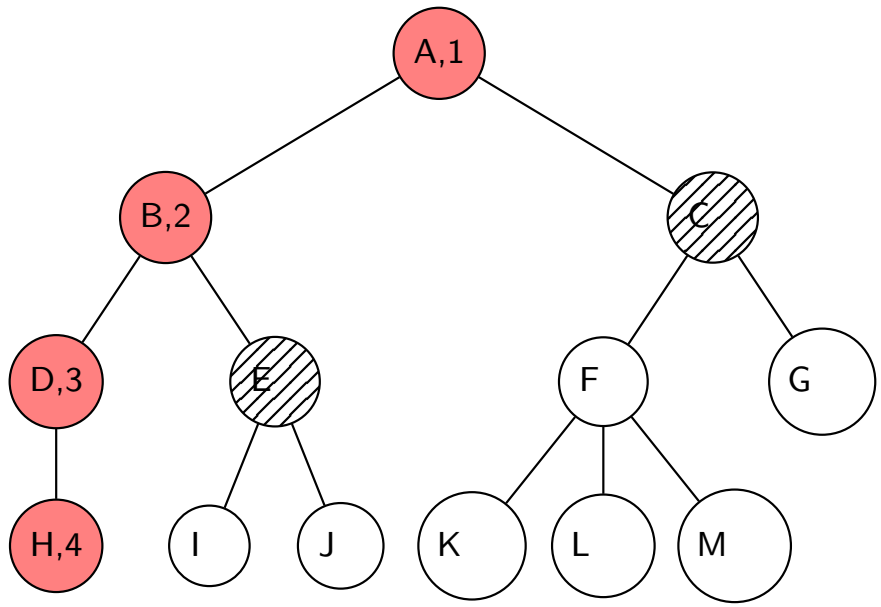
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | E | C |

Current : H

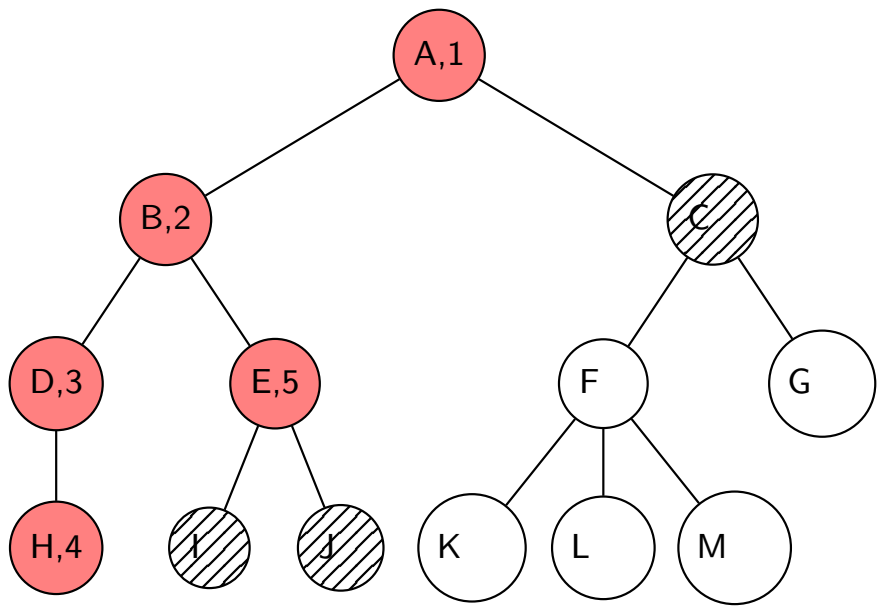
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | I | J | C |

Current : E

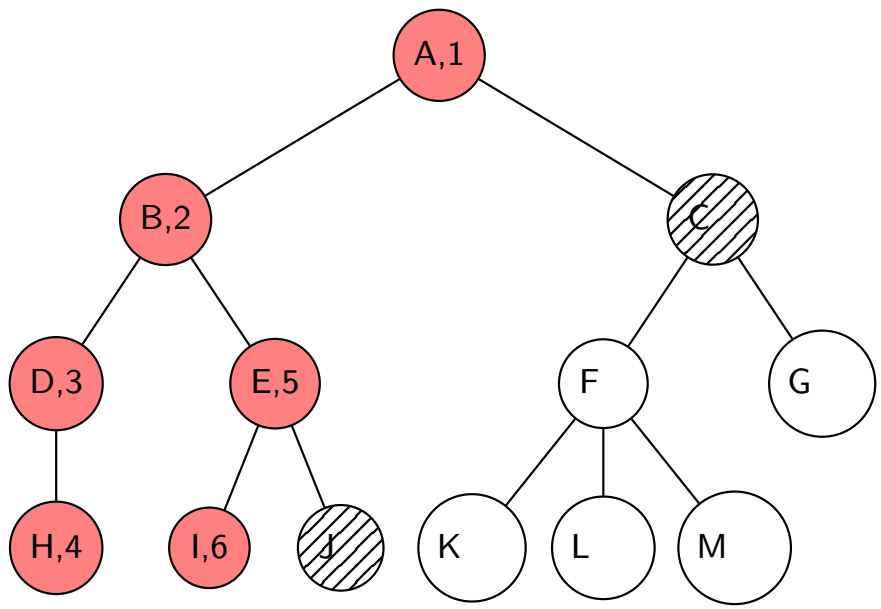
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | J | C |

Current : I

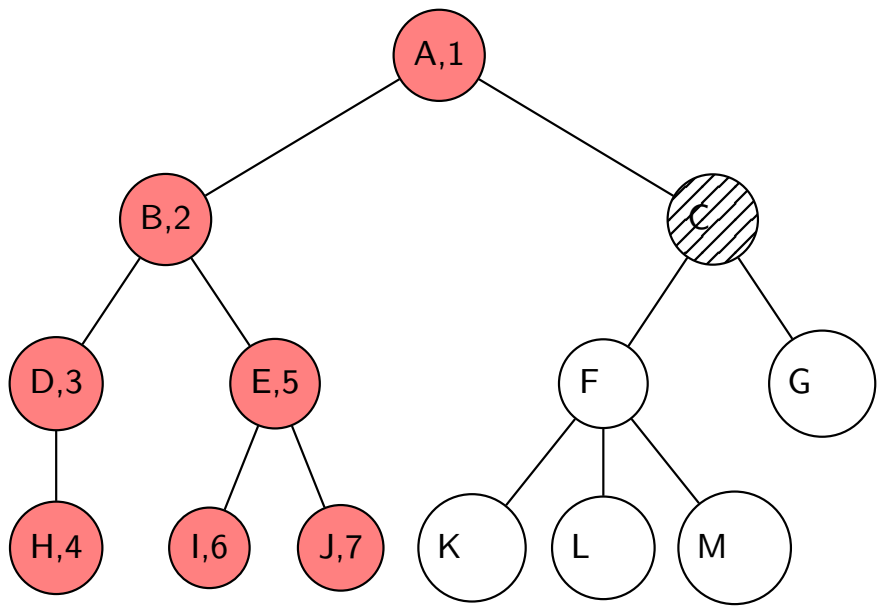
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ C

Current : J

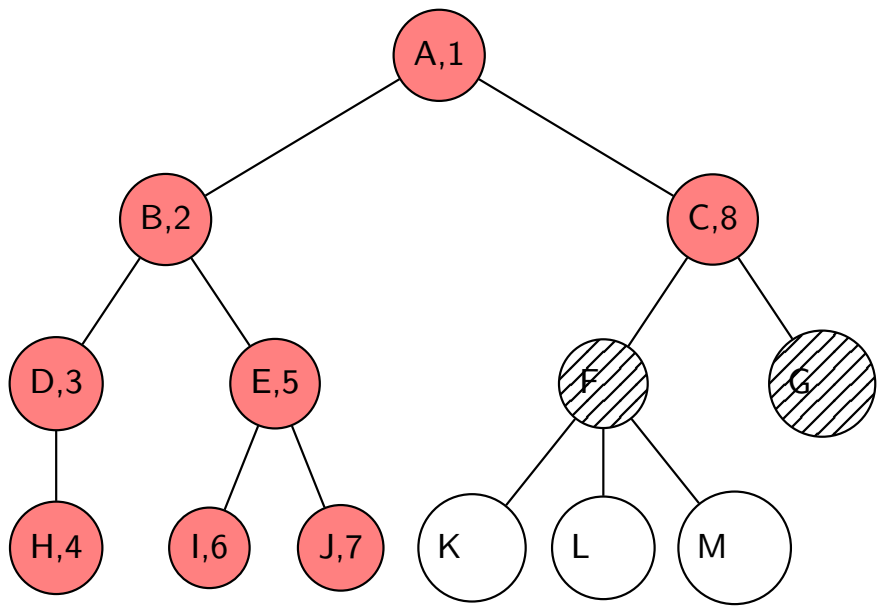
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | F | G |

Current : C

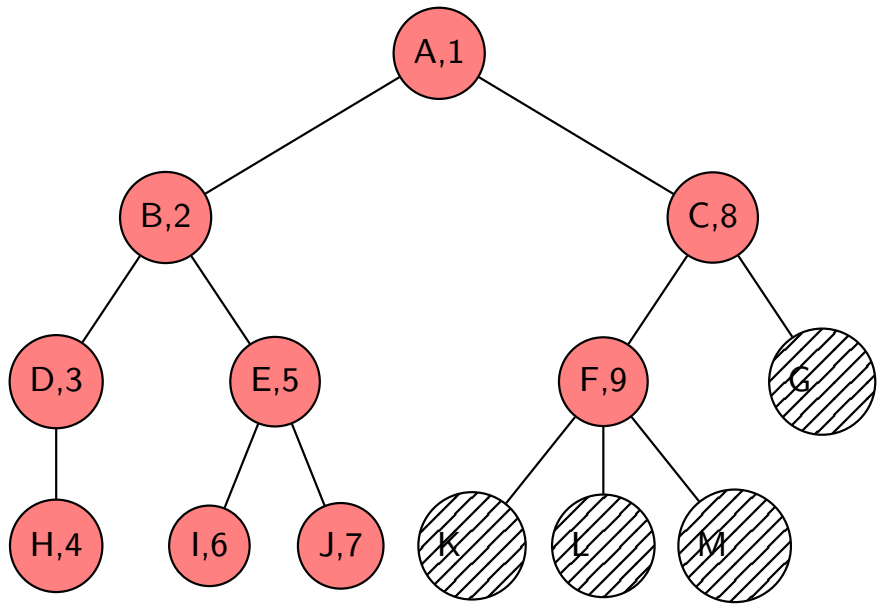
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

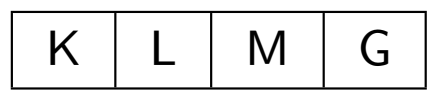
```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋



Current : F

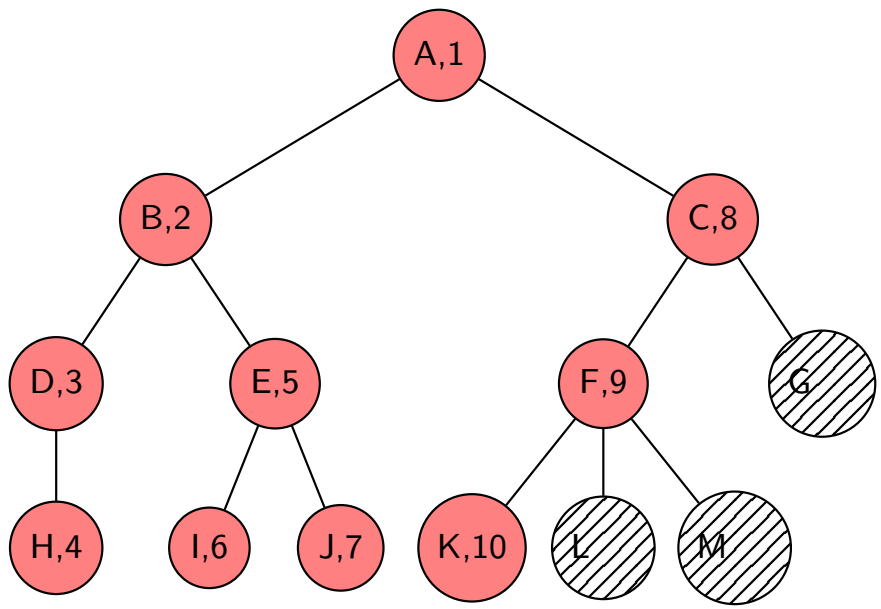
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

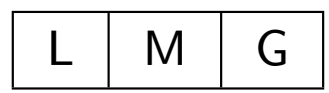
```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋



Current : K

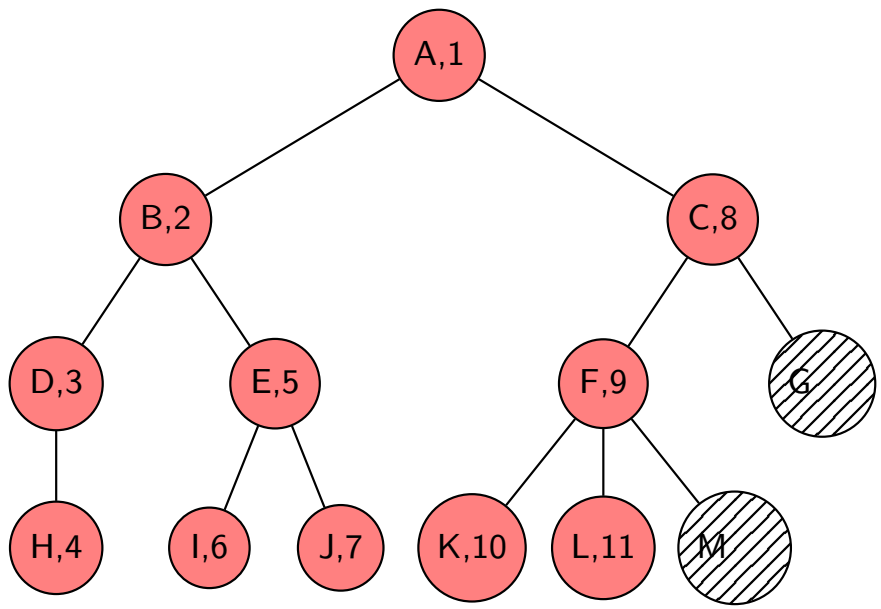
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ | M | G |

Current : L

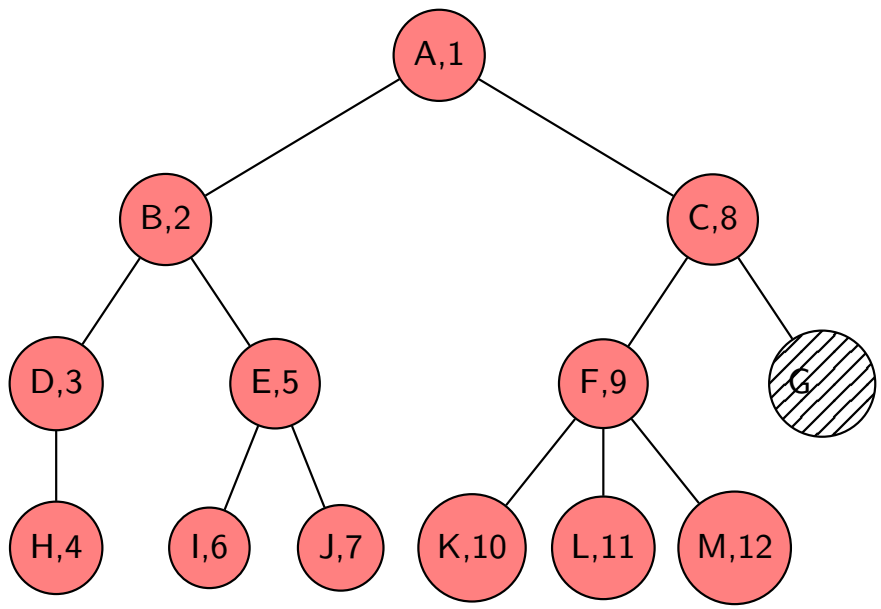
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

Complexité : $O(S + A)$



Stack : ⇋ G

Current : M

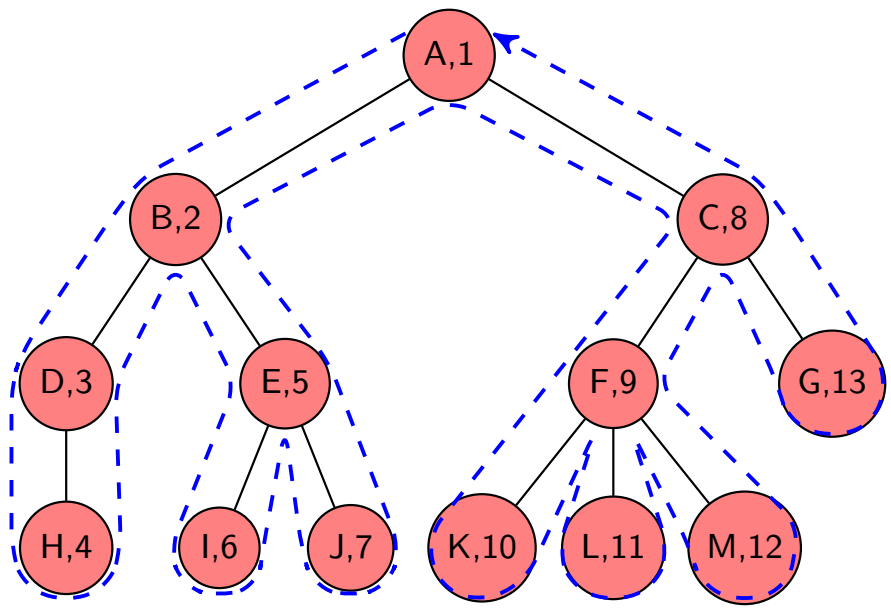
Parcours en profondeur (DFS)

Ce parcours essaie de visiter le nœud (à distance 1 de la partie déjà explorée) le plus éloigné de la racine. On va d'abord explorer la branche la plus à gauche, puis la suivante etc.

```
for tout x ∈ S do
    etat[x] ← non_atteint;
T ← ∅
à_traiter ← ∅
(*initialisation*);
push(à_traiter, {x});
etat[x]←atteint;
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] = non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter, {z});
```

Structure : Pile (LiFo)

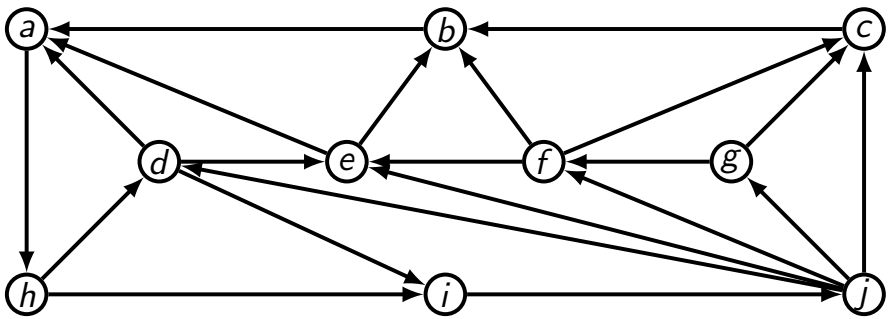
Complexité : $O(S + A)$



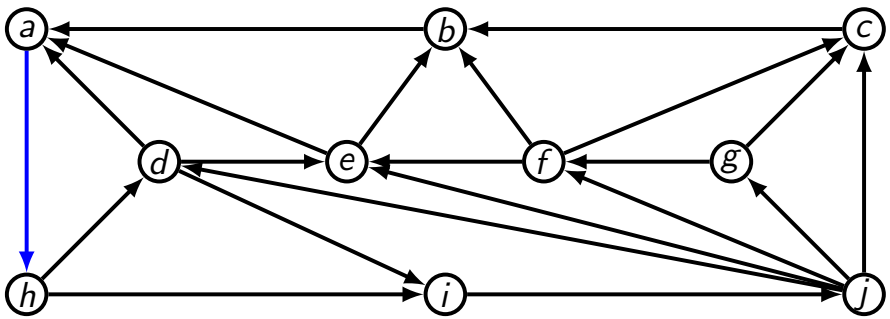
Stack : $\leftrightharpoons$ $\boxed{\emptyset}$

Current : G

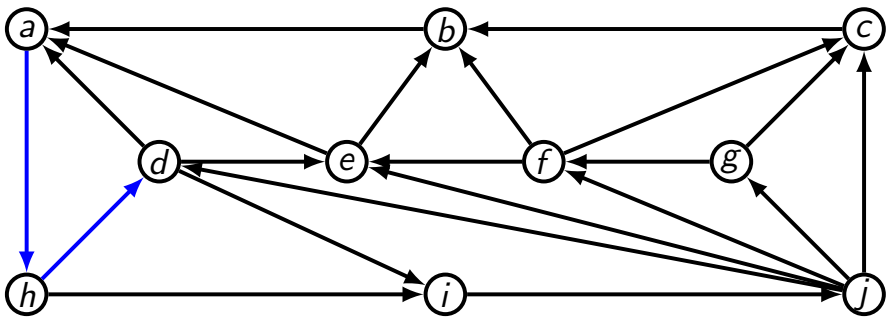
Exemple DFS



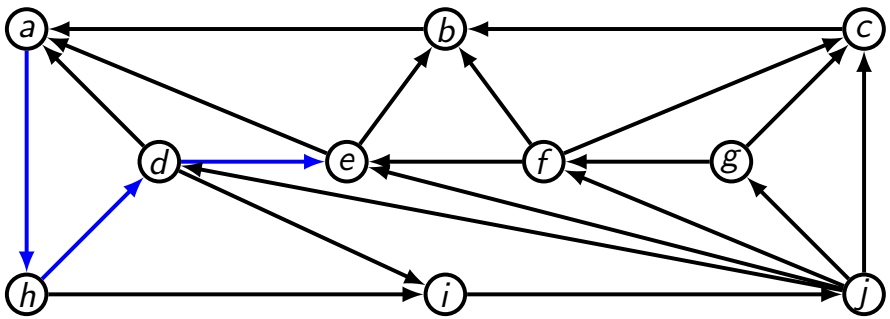
Exemple DFS



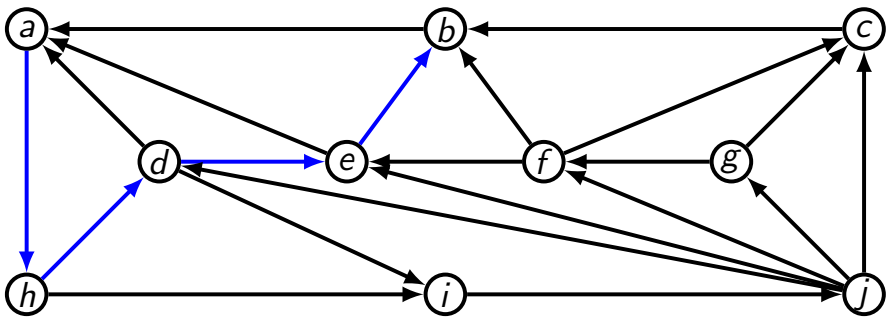
Exemple DFS



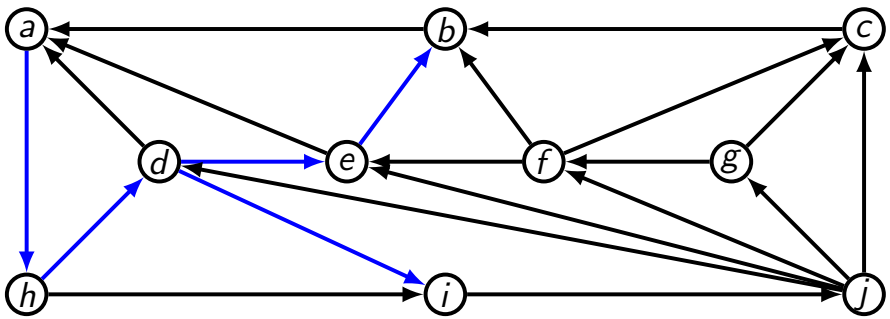
Exemple DFS



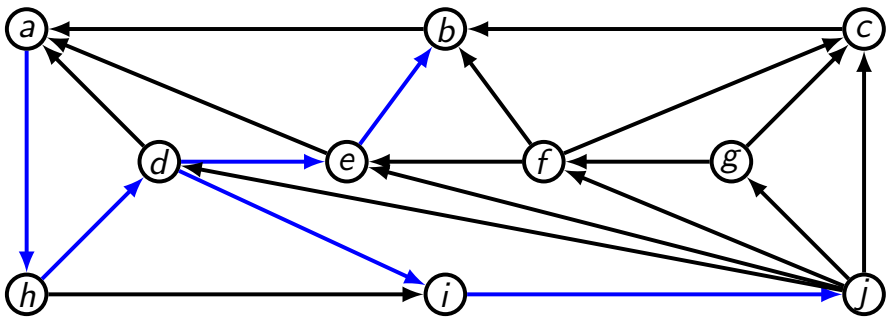
Exemple DFS



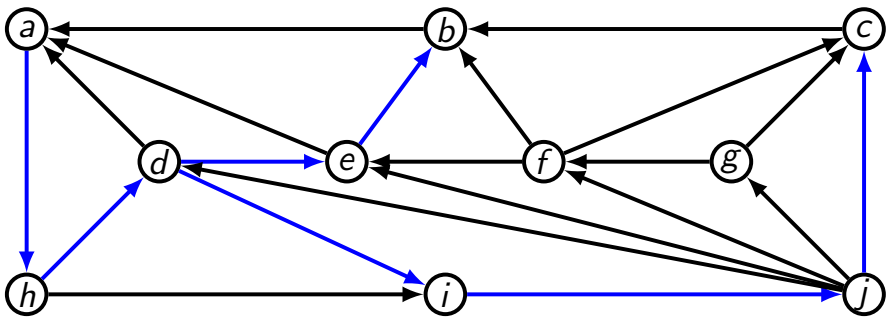
Exemple DFS



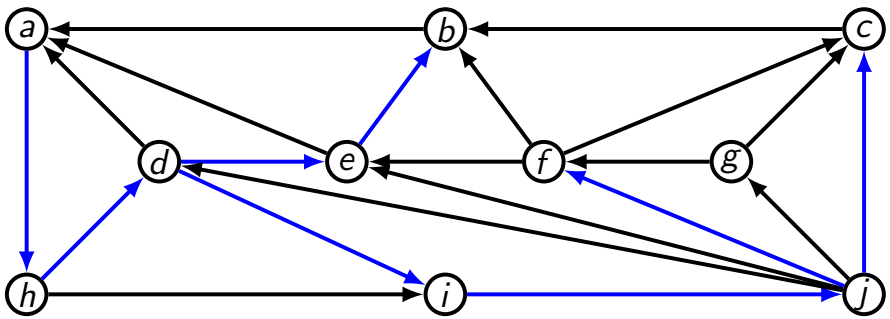
Exemple DFS



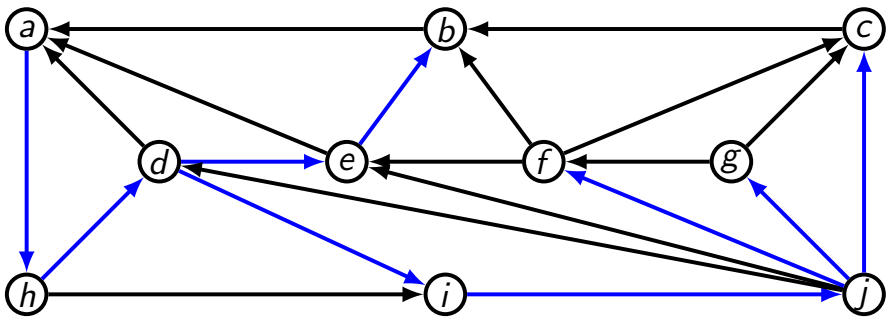
Exemple DFS



Exemple DFS



Exemple DFS



Parcours préfixe

```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
```

Parcours préfixe

```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
```

Parcours postfixe ou suffixe

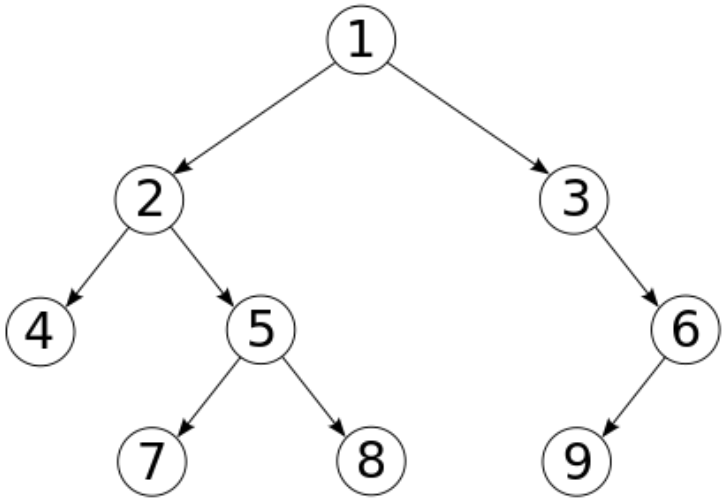
```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
    etat[y]←examine;
```

Parcours préfixe

```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
```

Parcours postfixe ou suffixe

```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
    etat[y]←examine;
```

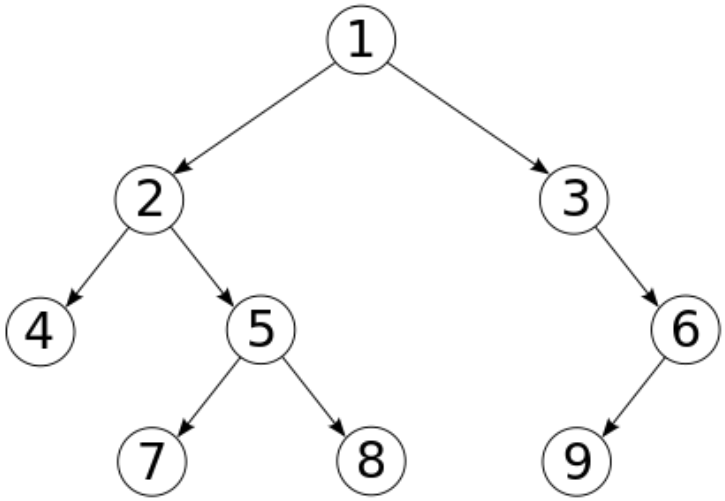


Parcours préfixe

```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    etat[y]←examine;
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
```

Parcours postfixe ou suffixe

```
while à_traiter ≠ ∅ do
    y =pop(à_traiter)
    for tout z voisin de y do
        if etat[z] =
          non_atteint then
            T←T ∪ {{y, z}}
            etat[z]←atteint
            push(à_traiter,
              {z});
    etat[y]←examine;
```

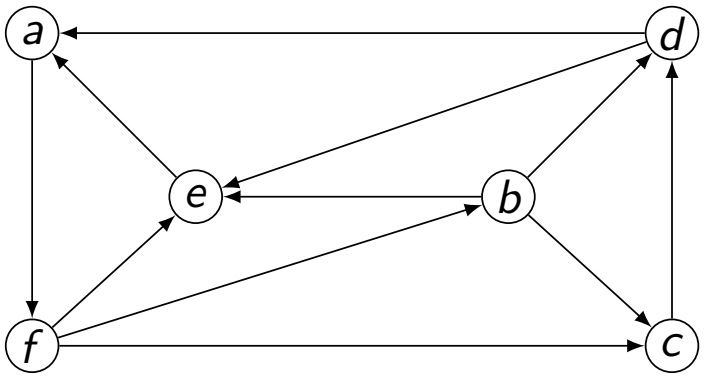


préfixe : 1, 2, 4, 5, 7, 8, 3, 6, 9

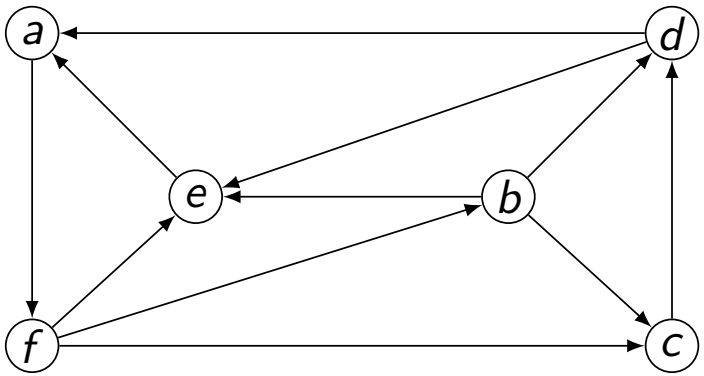
postfixe : 4, 7, 8, 5, 2, 9, 6, 3, 1

BFS : 1, 2, 3, 4, 5, 6, 7, 8, 9

Exemple–DFS–préfixe



Exemple–DFS–postfixe



Applications

- calcul des composantes connexes

Applications

- calcul des composantes connexes
- détection de cycle

Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

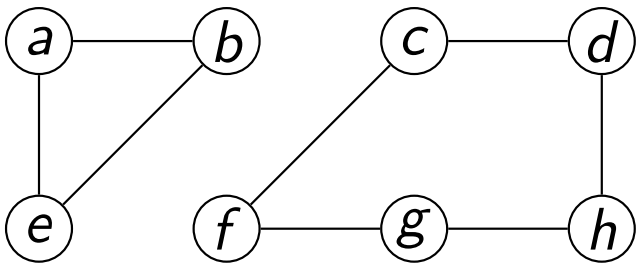
On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

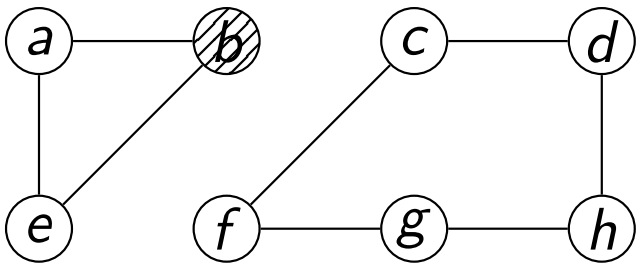


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

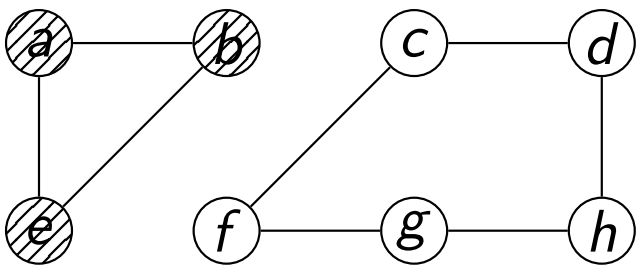


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

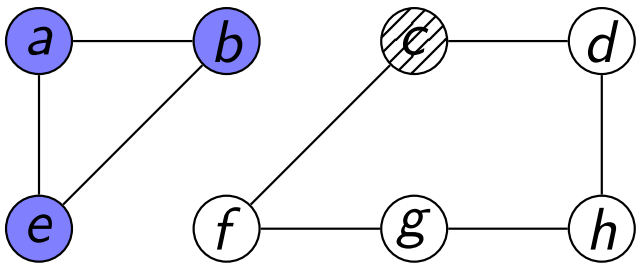


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

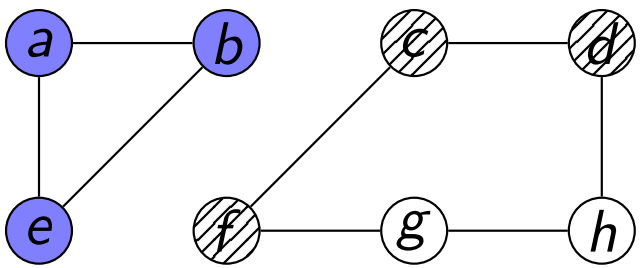


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

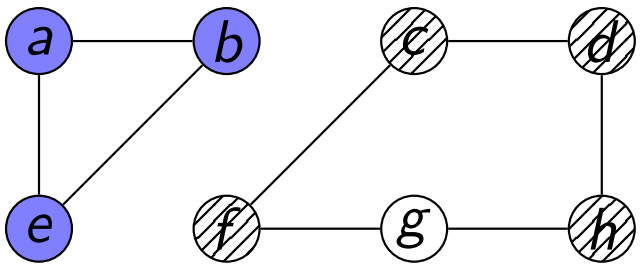


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

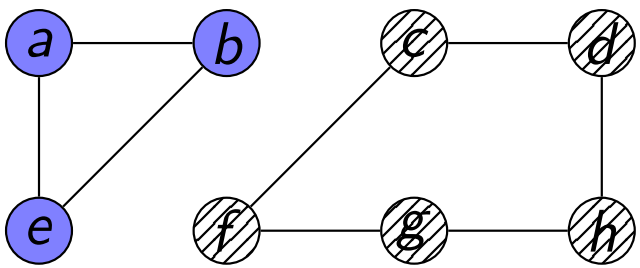


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

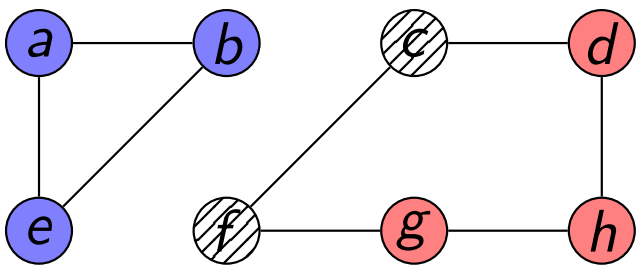


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.

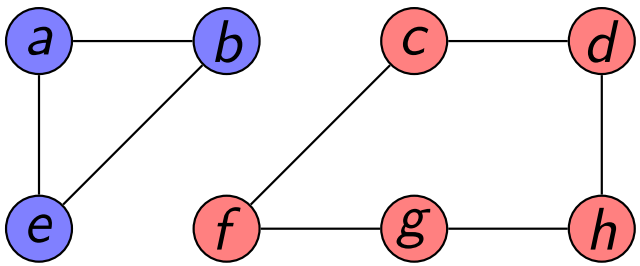


Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.



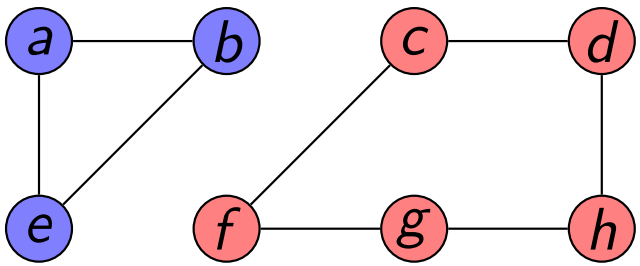
Ordre post-fixe : eab ghdfc

Applications

- calcul des composantes connexes
- détection de cycle
- arbres syntaxiques

Composantes connexes

On effectue un parcours et on change de couleur/composante connexe à chaque fois que la pile/file est vide.



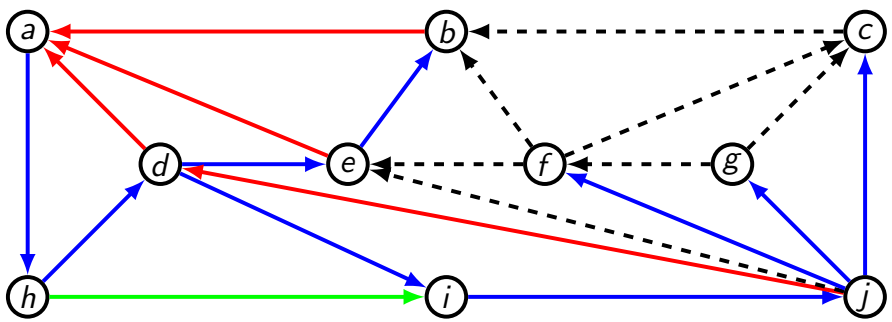
Ordre post-fixe : eab ghdfc

Deux composantes connexes : $\{e, a, b\}$ et $\{g, h, d, f, c\}$

Détection de circuit

Le DFS de G (orienté) $\rightsquigarrow$ quatre types d'arc :

- **arcs couvrants** : arcs de l'arborescence du DFS
- **arcs directs** : arcs hors arbo. reliant un sommet à un descendant
- **arcs retour** : arcs hors arbo. reliant un sommet à un ascendant (ou à lui même)
- **arcs traversiers** : arcs hors arbo. reliant deux branches distinctes de l'arbre



Détection des circuits

Un graphe orienté G est sans circuit ssi un DFS ne génère aucun arc retour.

Graphe Valué

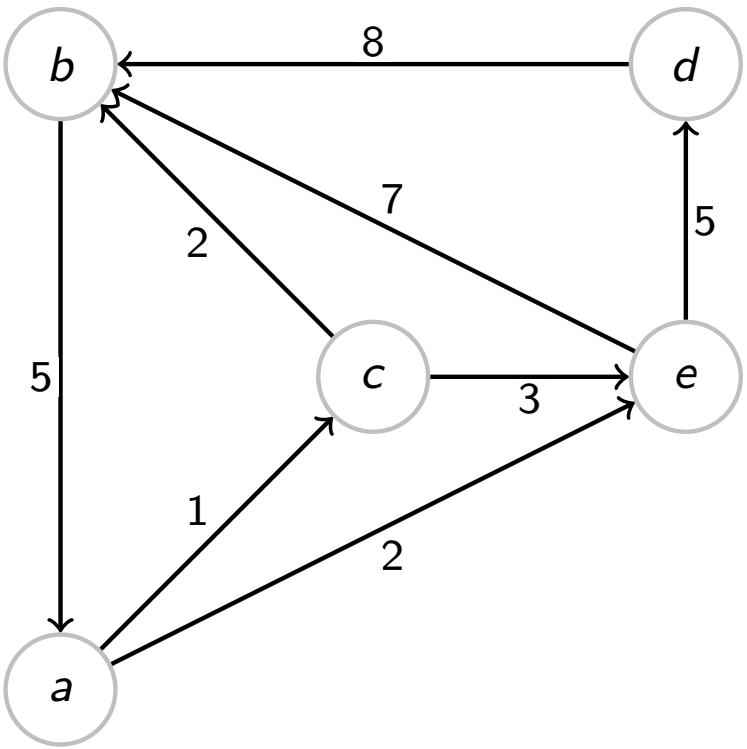
Soit $\mathcal{G} = (S, A)$ un graphe est dit **valué** (ou pondéré, en anglais weighted) s'il existe une application $w : A \to \mathbb{R}$. On utilise souvent une matrice de valuation W de taille n.

$$W_{i,j} = \begin{cases} w((i,j)) \text{ si } iAj \\ \infty \text{ sinon.} \end{cases}$$

Graphe Valué

Soit $\mathcal{G} = (S, A)$ un graphe est dit **valué** (ou pondéré, en anglais weighted) s'il existe une application $w : A \to \mathbb{R}$. On utilise souvent une matrice de valuation W de taille n.

$$W_{i,j} = \begin{cases} w((i,j)) \text{ si } iAj \\ \infty \text{ sinon.} \end{cases}$$



A pour matrice de valuation :

$$\begin{pmatrix} \infty & \infty & 1 & \infty & 2 \\ 5 & \infty & \infty & \infty & \infty \\ \infty & 2 & \infty & \infty & 3 \\ \infty & 8 & \infty & \infty & \infty \\ \infty & 7 & \infty & \infty & 5 \end{pmatrix}$$

Plan

Gestion d'un projet

Un groupe d'élèves de l'IUT travaille sur sa SAÉ. Comment organiser les tâches ?

Gestion d'un projet

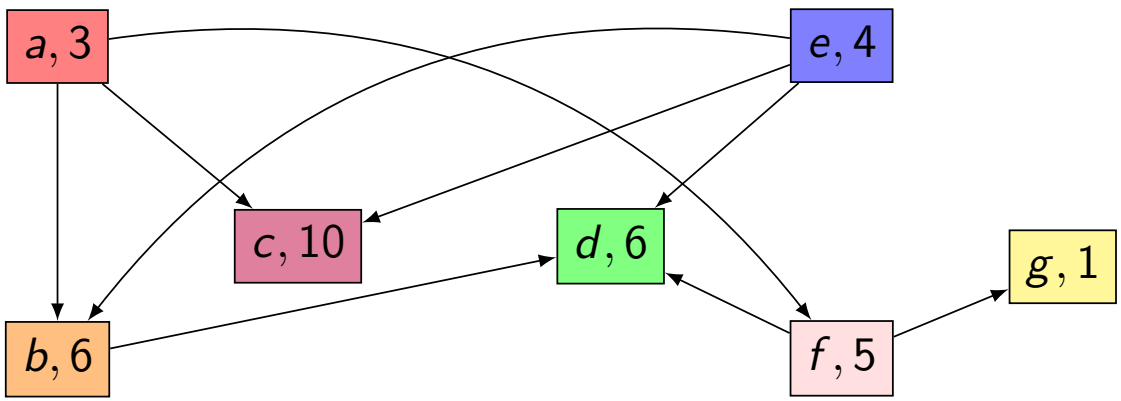
Un groupe d'élèves de l'IUT travaille sur sa SAÉ. Comment organiser les tâches ?

Par ex : la rédaction du rapport prendra 6 jours, mais n'est possible qu'après le test du prototype, qui lui prend 4 jours etc...

Gestion d'un projet

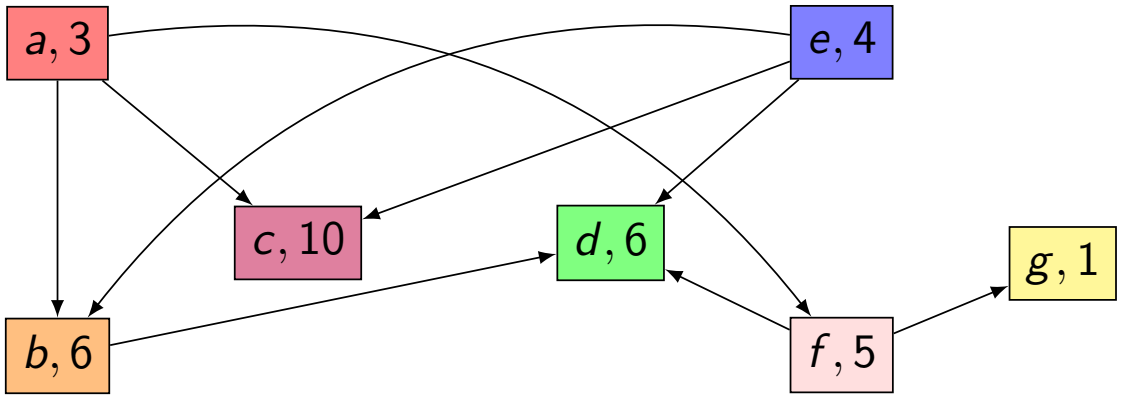
Un groupe d'élèves de l'IUT travaille sur sa SAÉ. Comment organiser les tâches ?

Par ex : la rédaction du rapport prendra 6 jours, mais n'est possible qu'après le test du prototype, qui lui prend 4 jours etc...



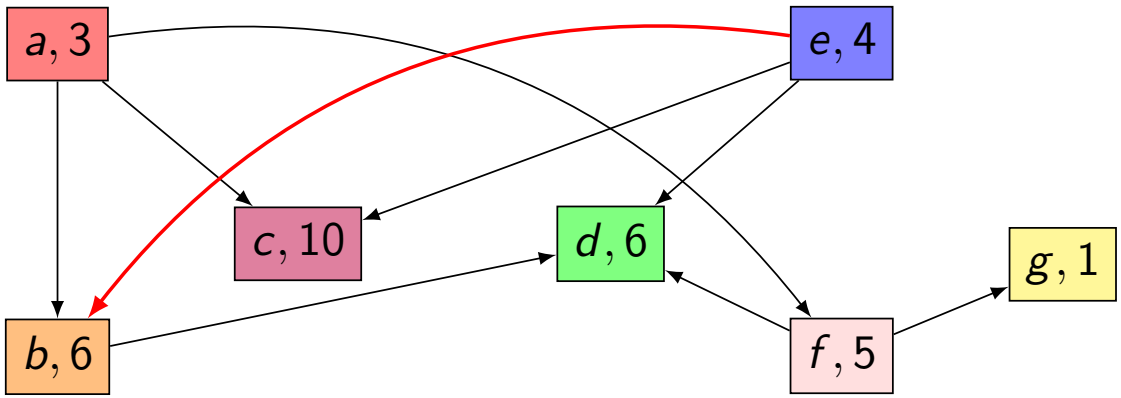
Tri topologique

1 processeur :



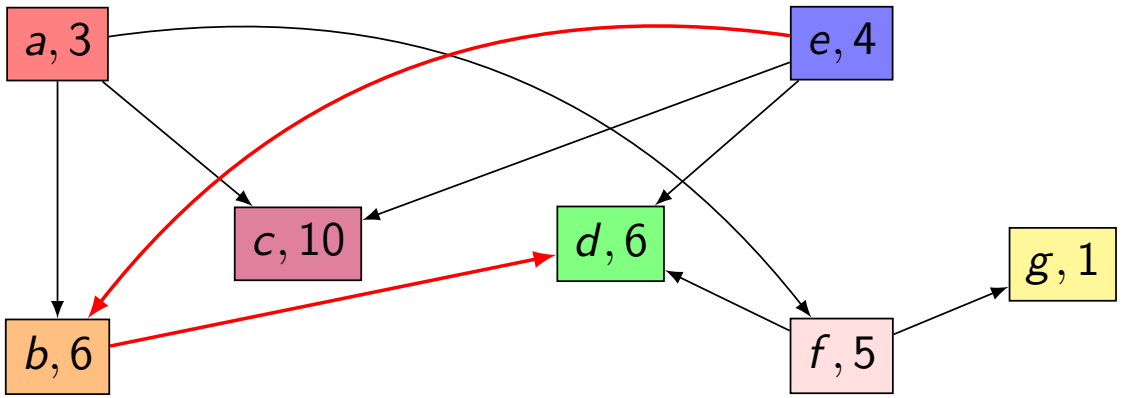
Tri topologique

1 processeur :



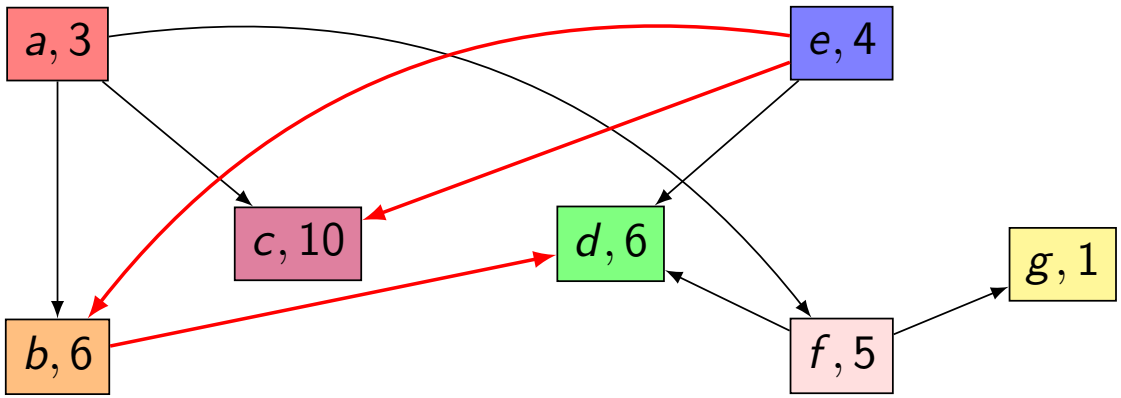
Tri topologique

1 processeur :



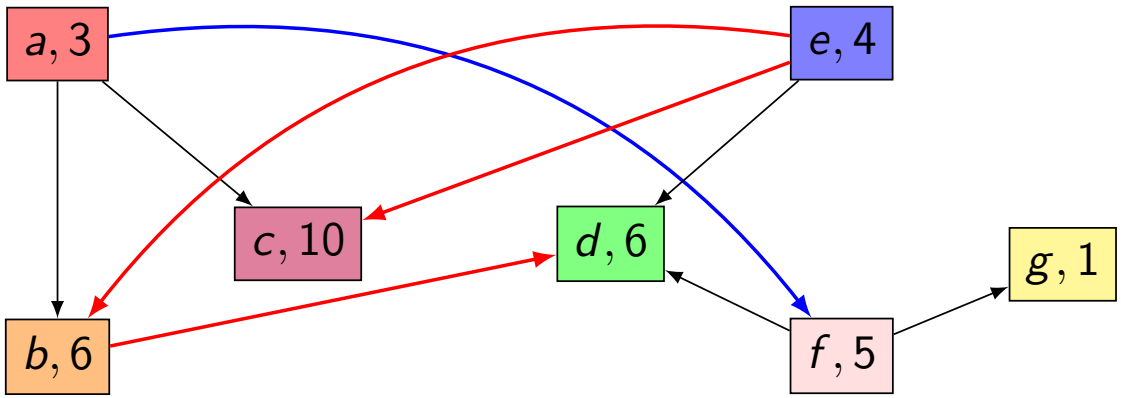
Tri topologique

1 processeur :



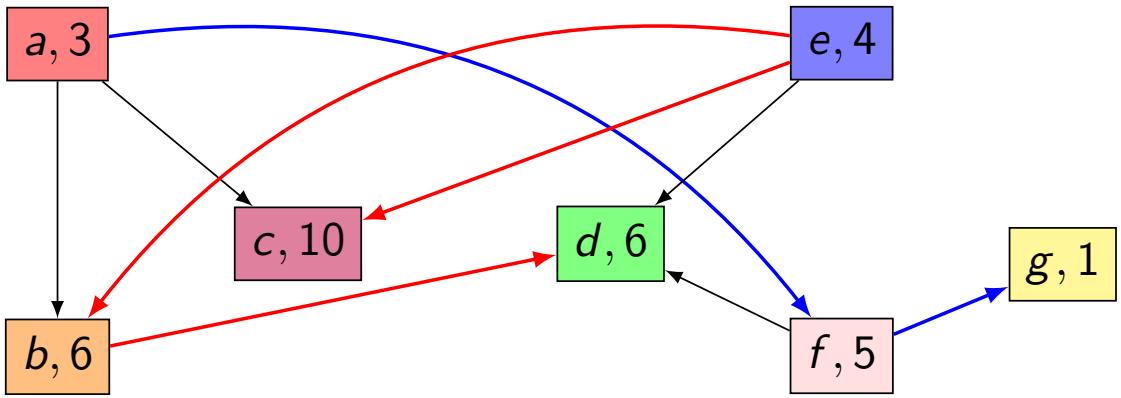
Tri topologique

1 processeur :



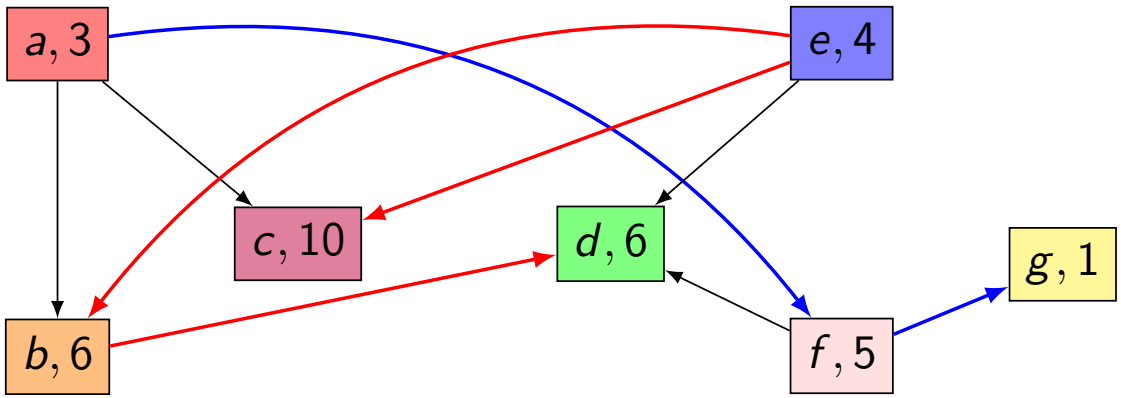
Tri topologique

1 processeur :



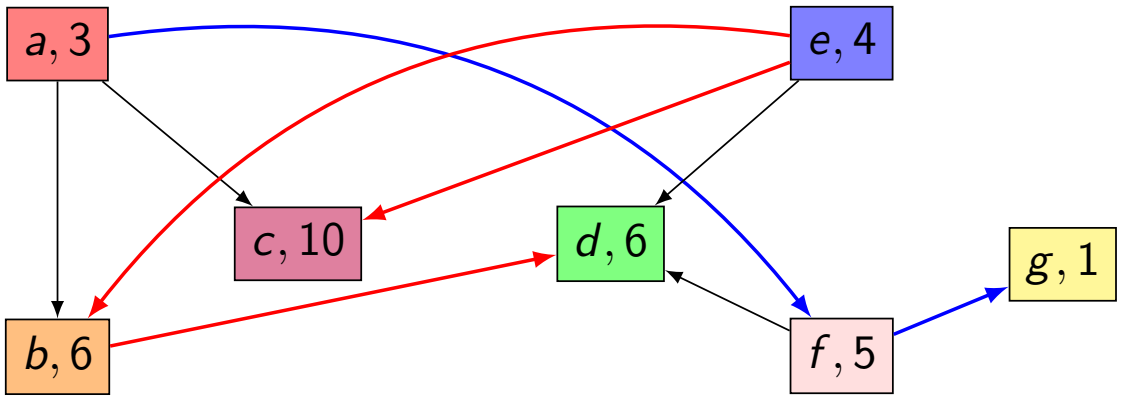
Tri topologique

1 processeur :

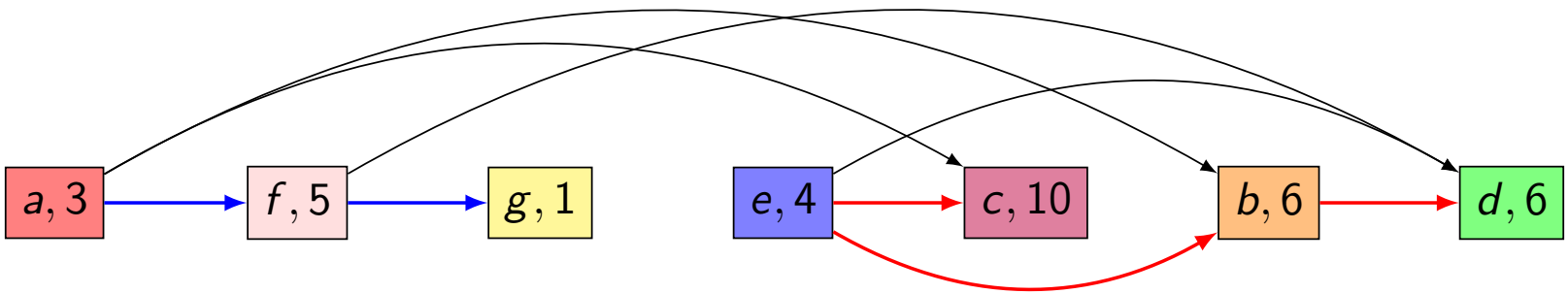


Tri topologique

1 processeur :

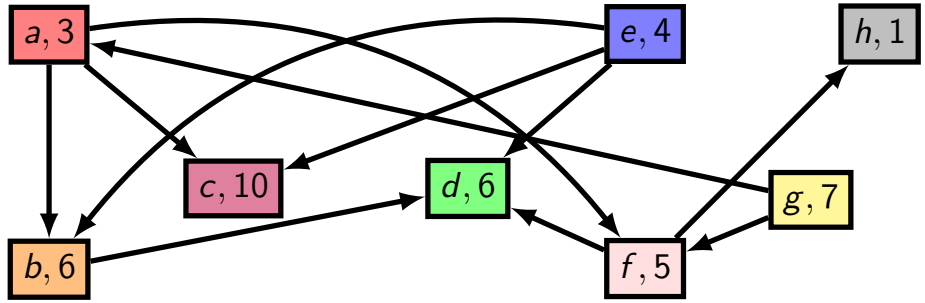


On calcule l'ordre post-fixe que l'on inverse, on obtient alors un ordre (total) sur les sommets :

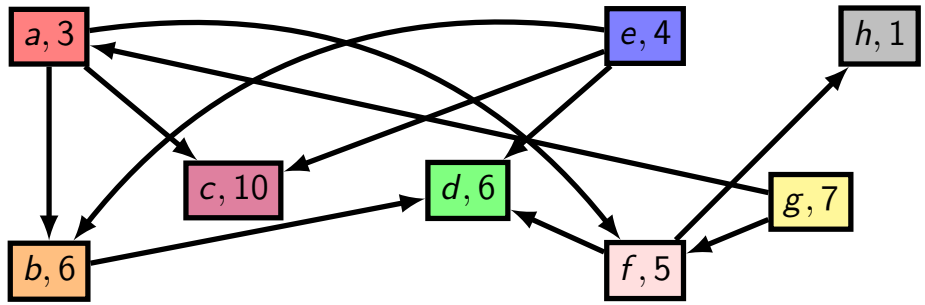


Plan

Tri par niveau



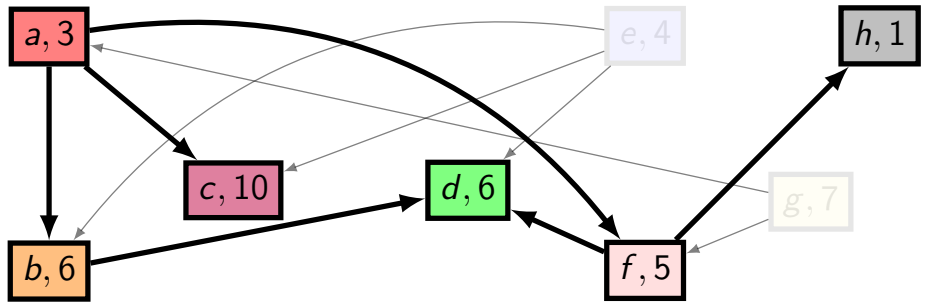
Tri par niveau



g

e

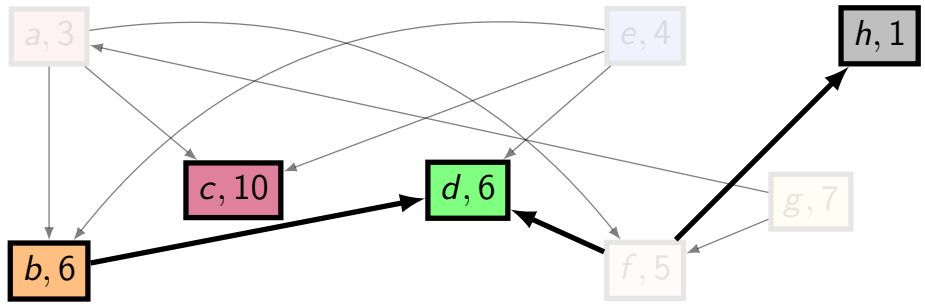
Tri par niveau



g

e

Tri par niveau

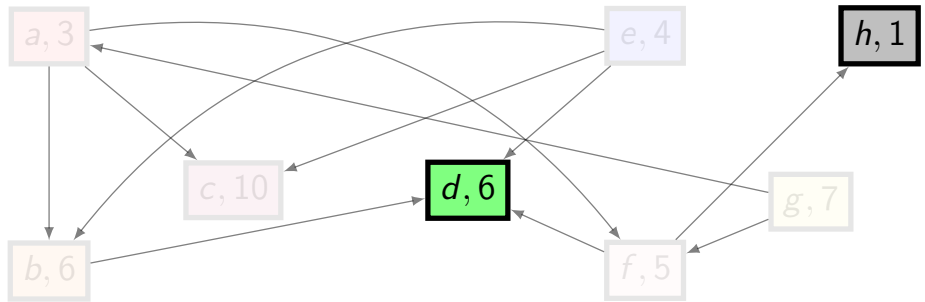


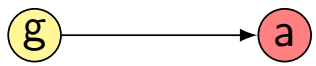
g

a

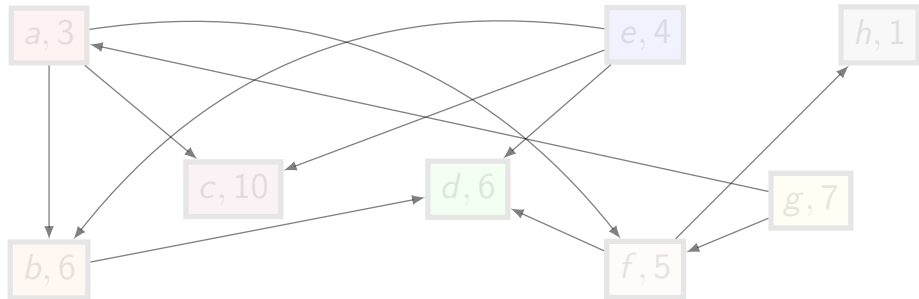
e

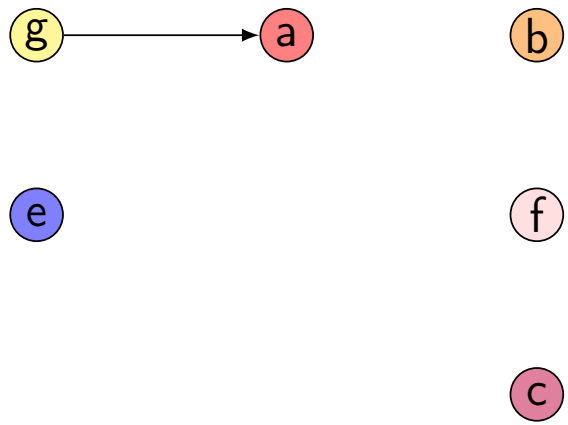
Tri par niveau



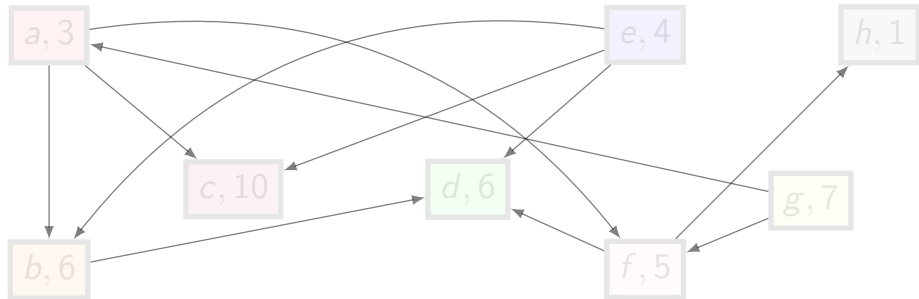


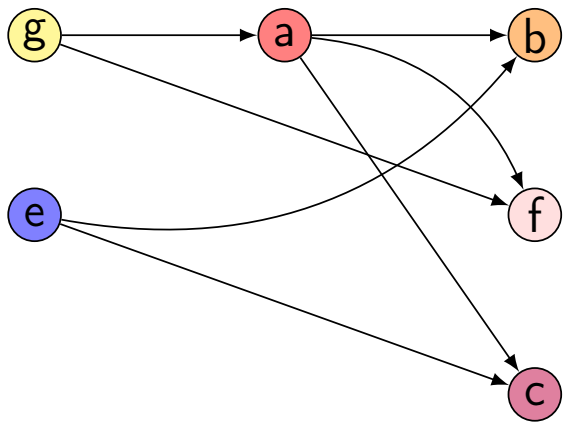
Tri par niveau



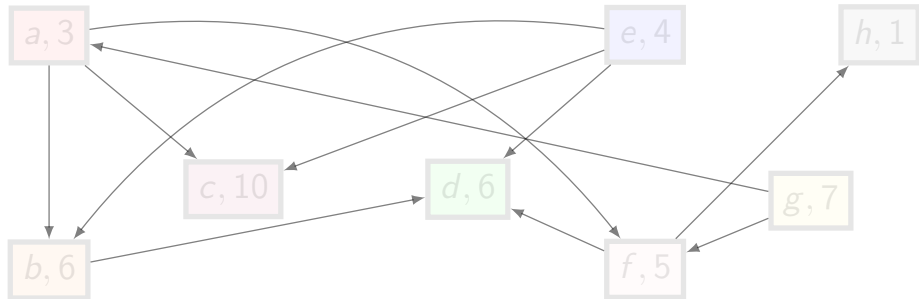


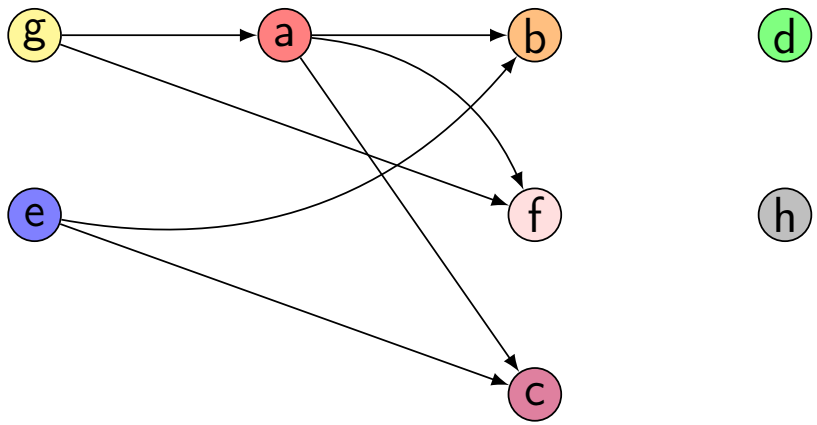
Tri par niveau



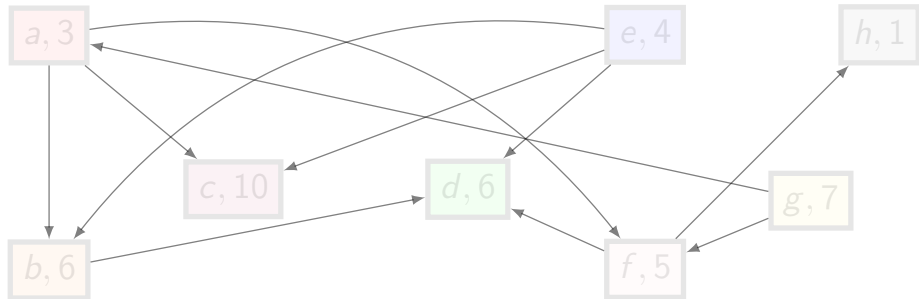


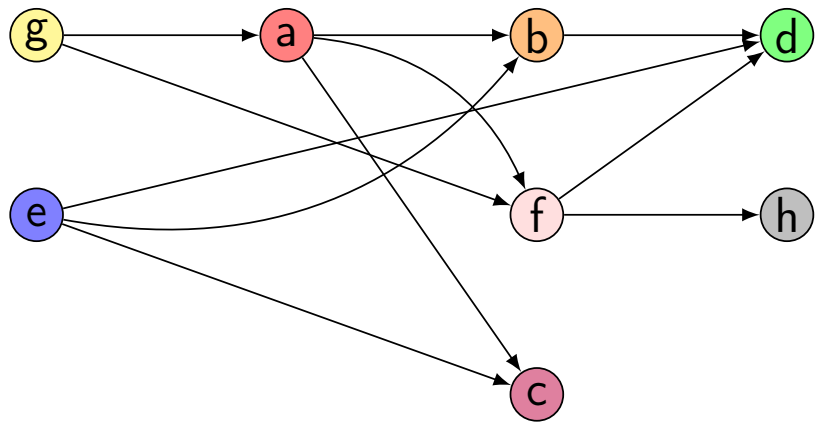
Tri par niveau



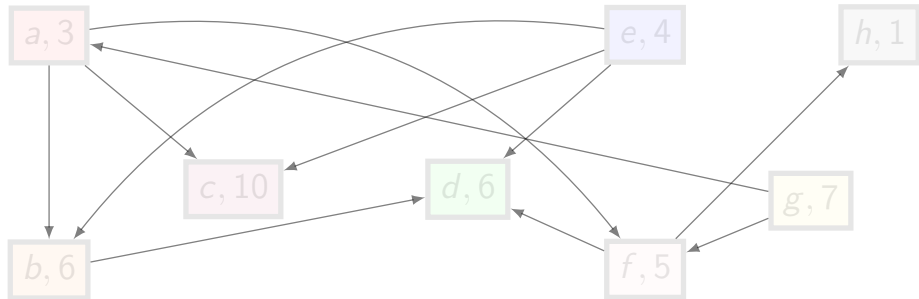


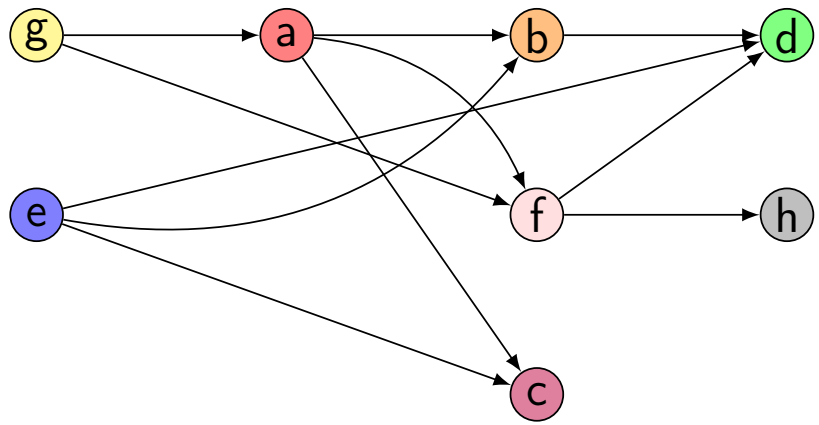
Tri par niveau



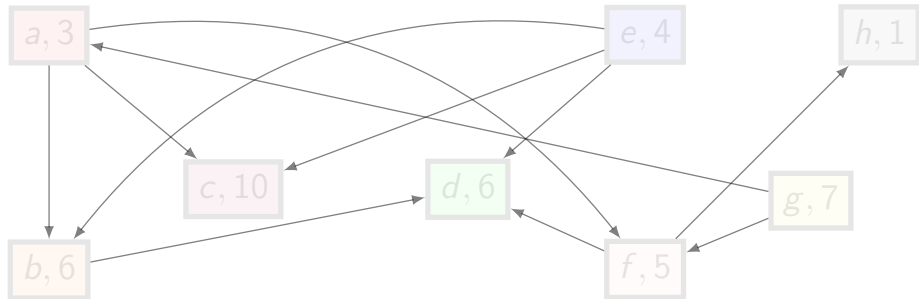


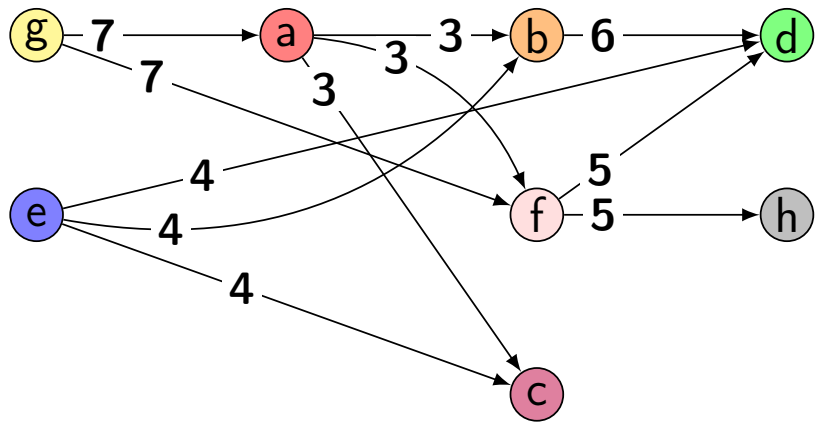
Tri par niveau



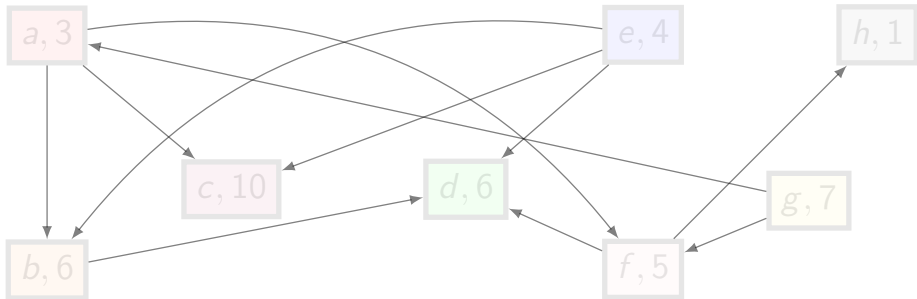


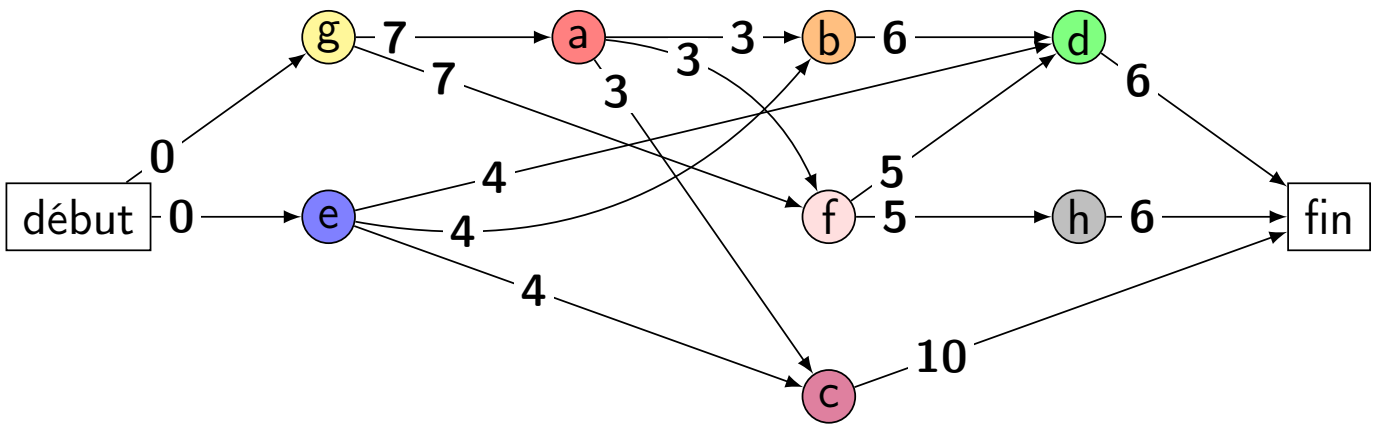
Tri par niveau



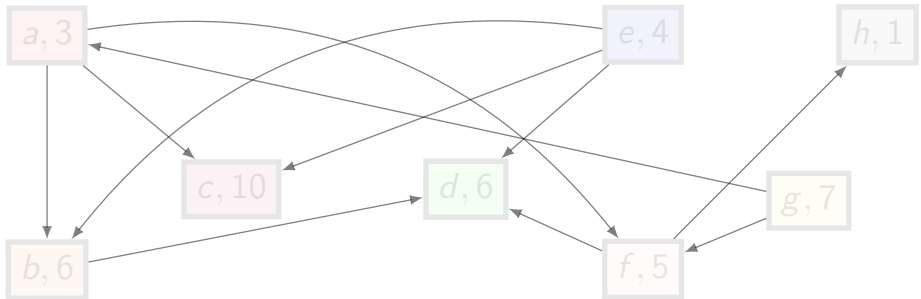


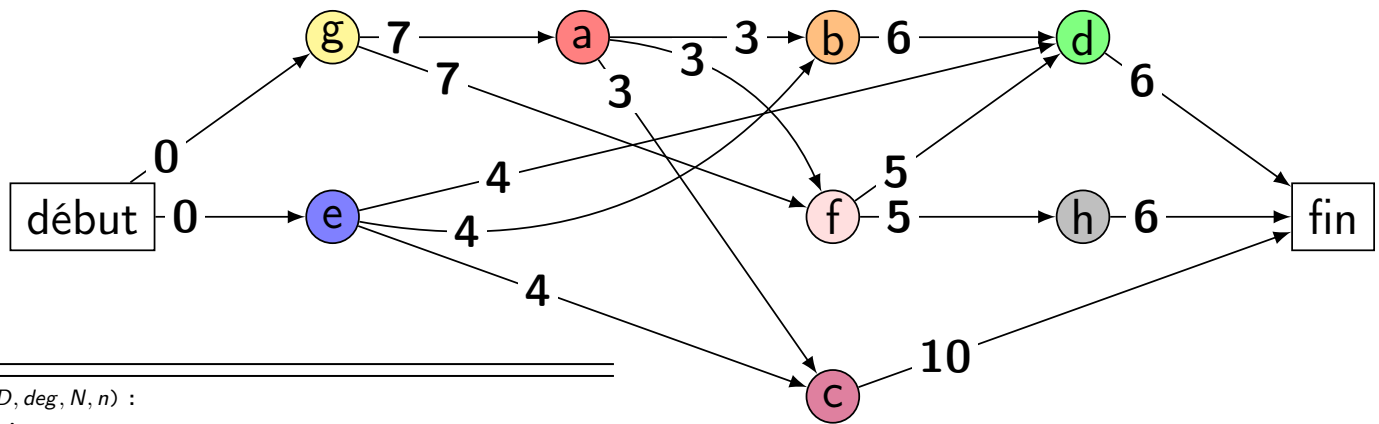
Tri par niveau





Tri par niveau





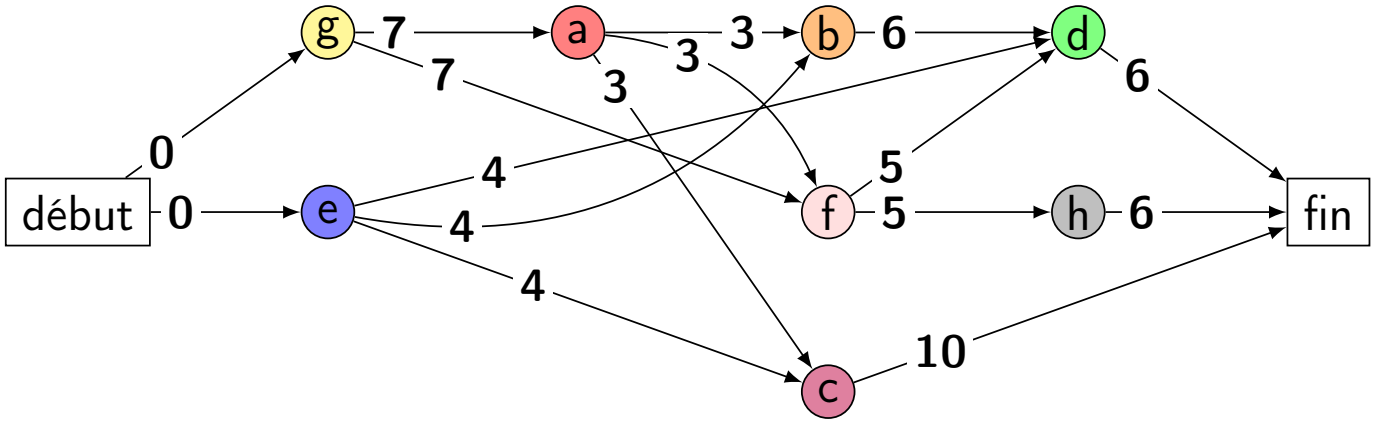
Function `TriNiveauRec`(D, deg, N, n) :

- $deg' = deg$ **for** $s \in S$ **do**
  - **if** $deg[s] = 0$ **then**
    - **if** $N[s] = -1$ **then**
      - $N[s] = n$
      - **for** v *voisin de* s **do**
        - $deg'[v] = deg'[v] - 1$
- N= `TriNiveauRec`$(D, deg', N, n+1)$
- **return** N

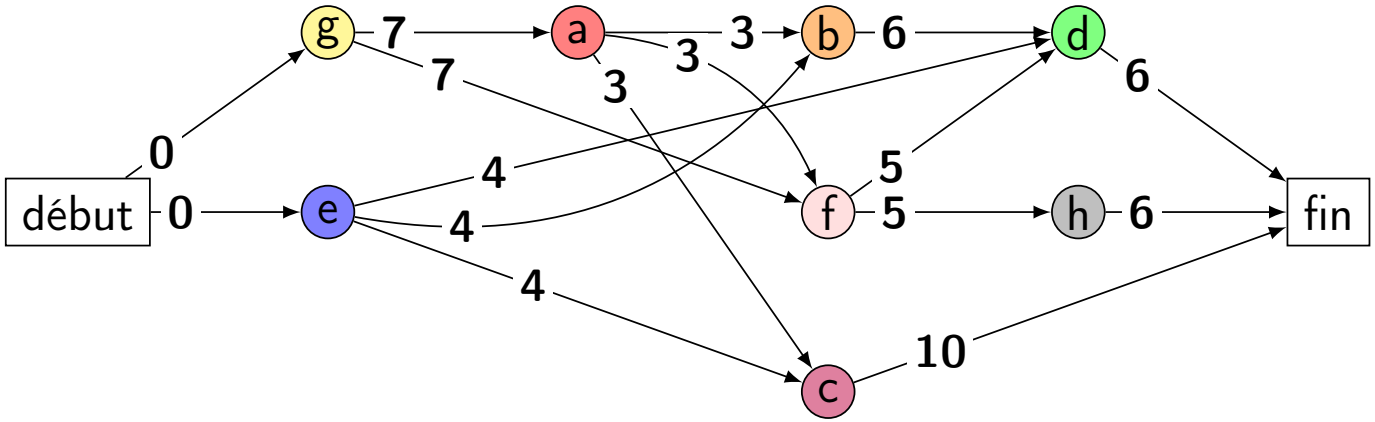
Plan

Ordonnancement au plus tôt

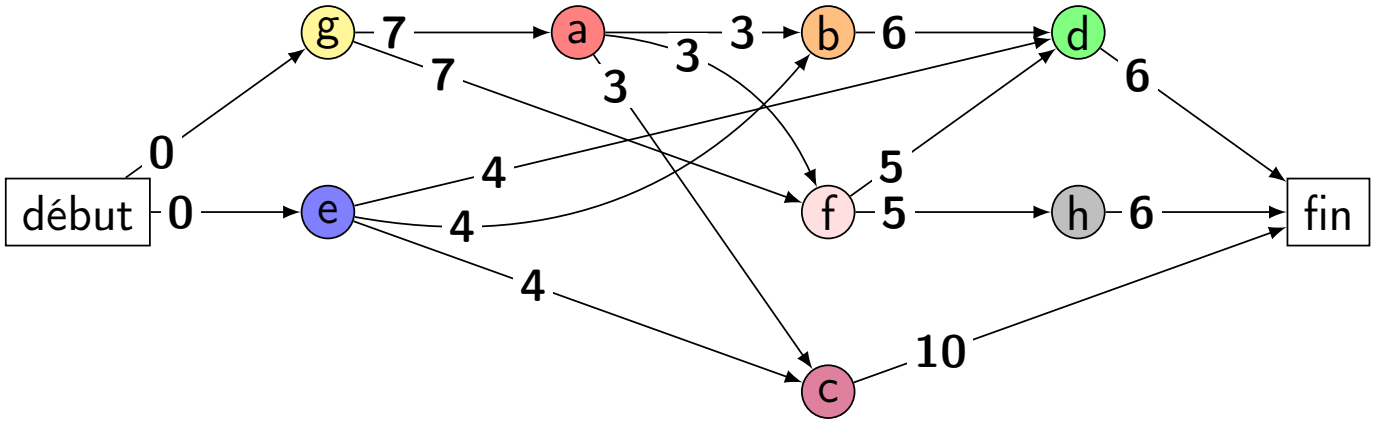
Ordonnancement au plus tôt



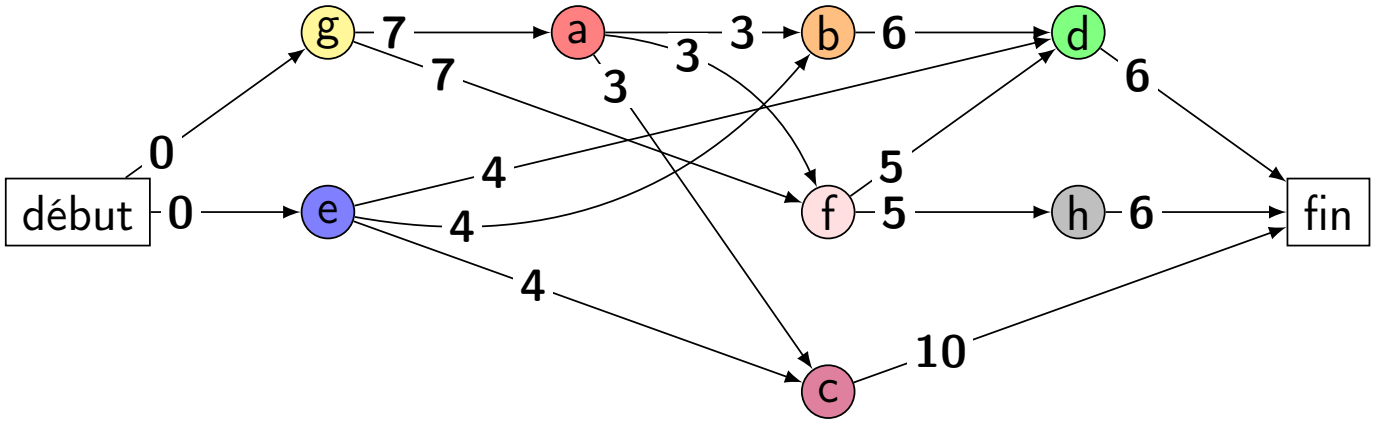
Ordonnancement au plus tôt

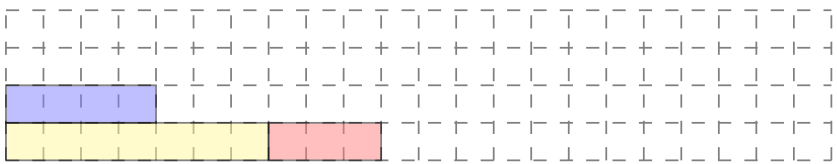


Ordonnancement au plus tôt

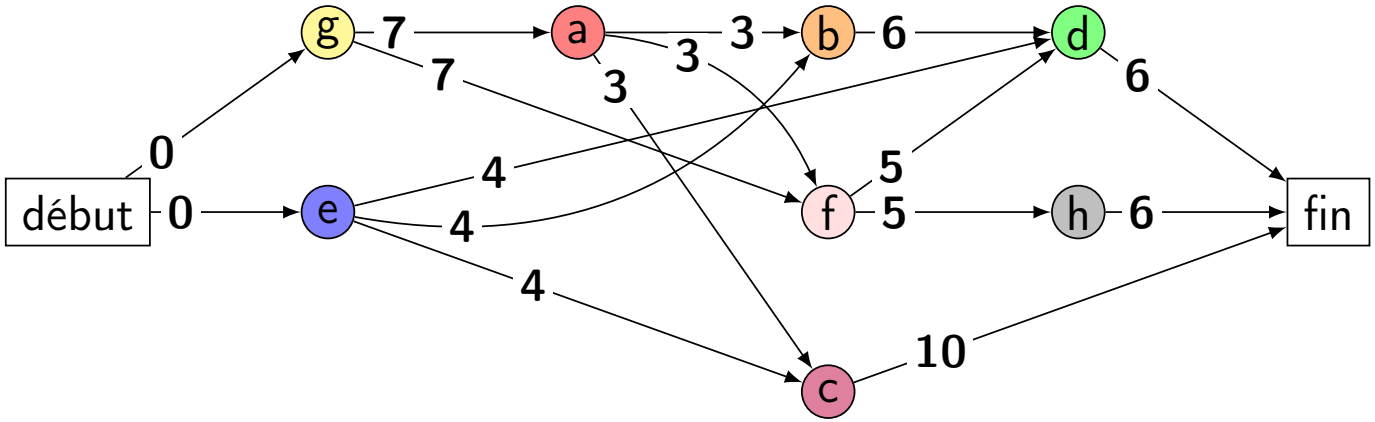


Ordonnancement au plus tôt

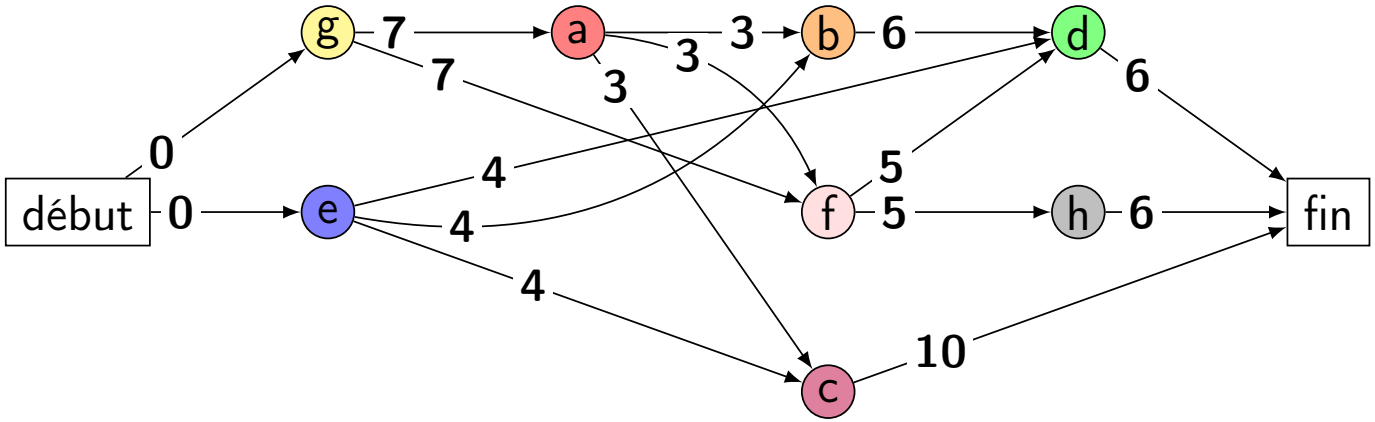




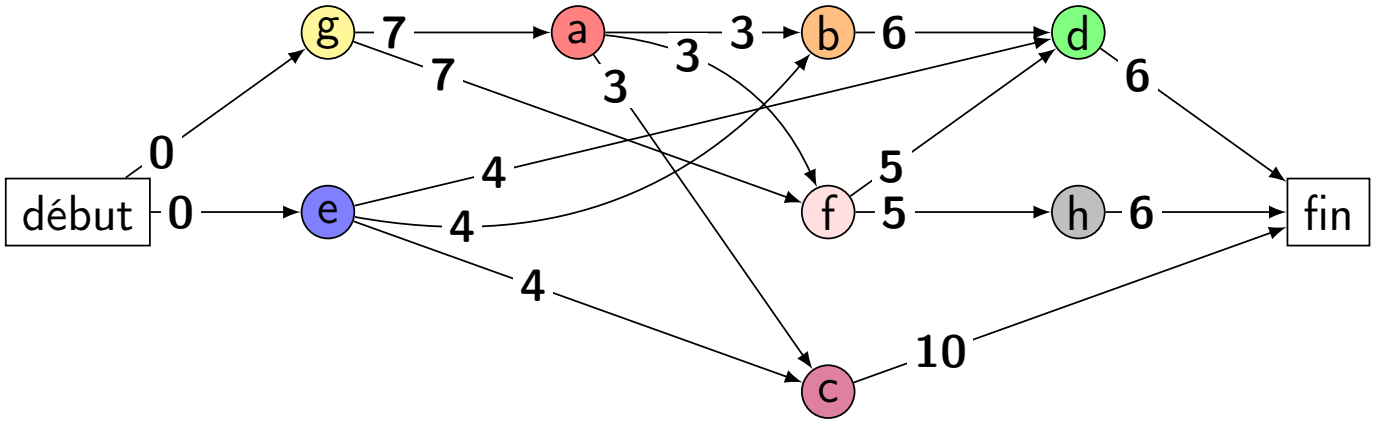
Ordonnancement au plus tôt



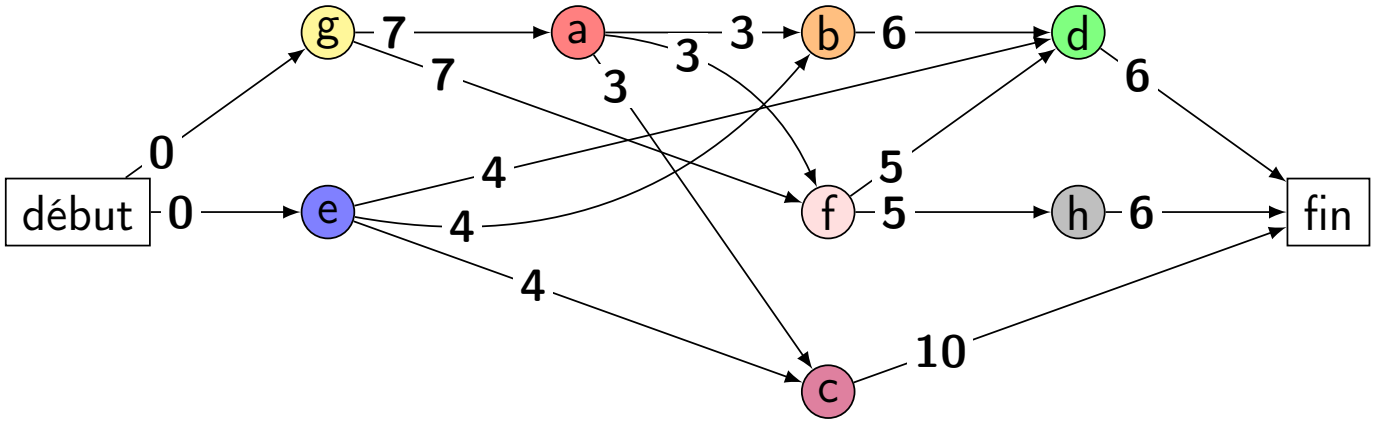
Ordonnancement au plus tôt



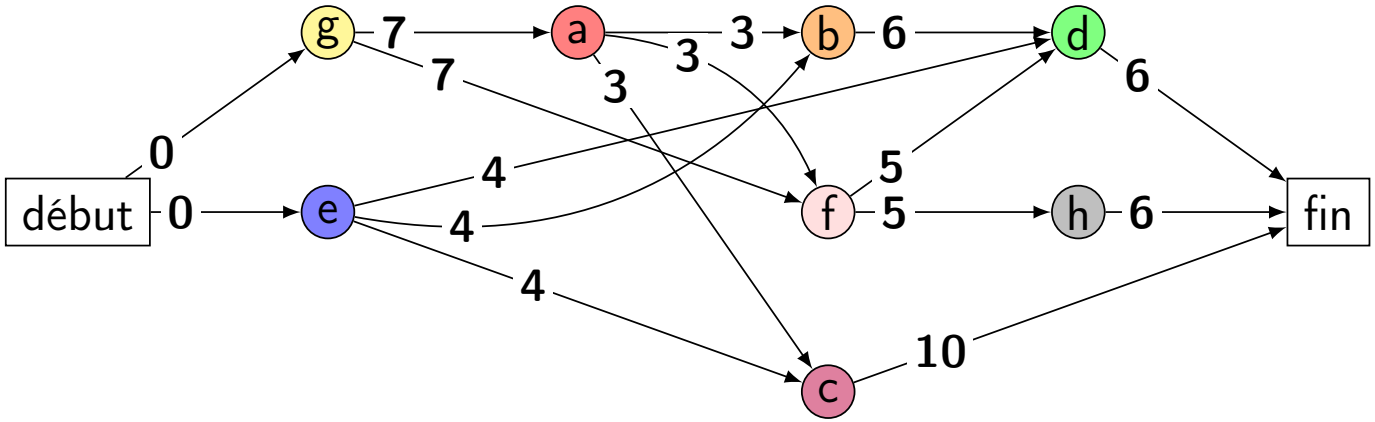
Ordonnancement au plus tôt



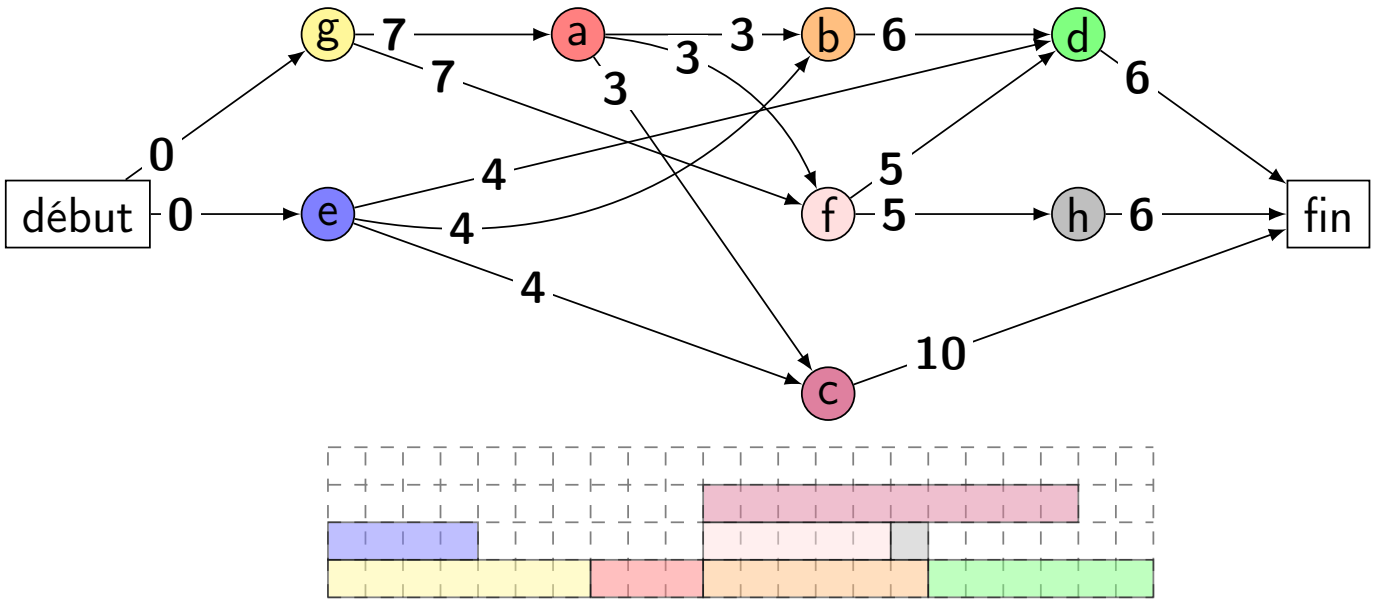
Ordonnancement au plus tôt



Ordonnancement au plus tôt



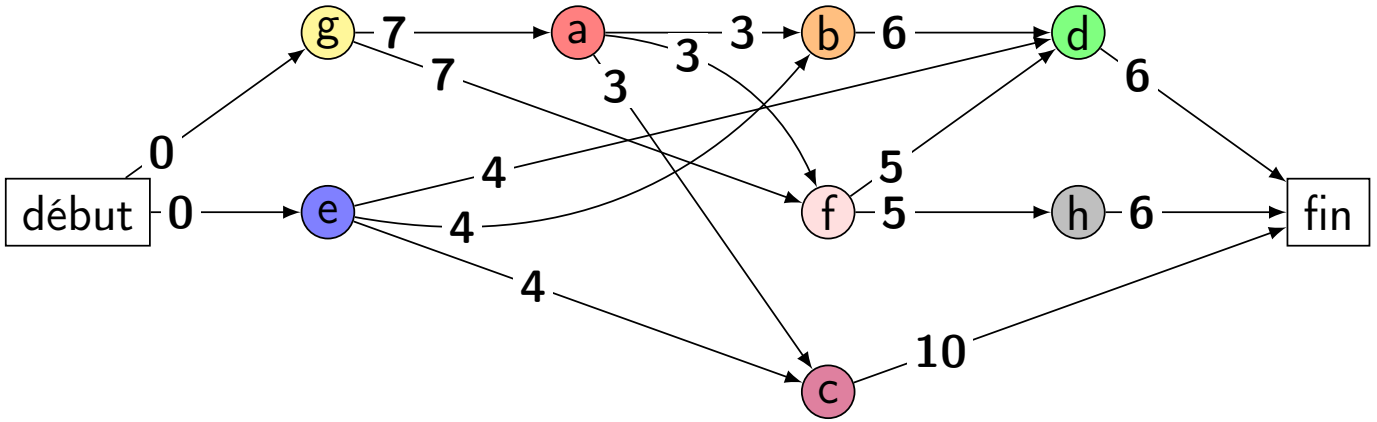
Ordonnancement au plus tôt



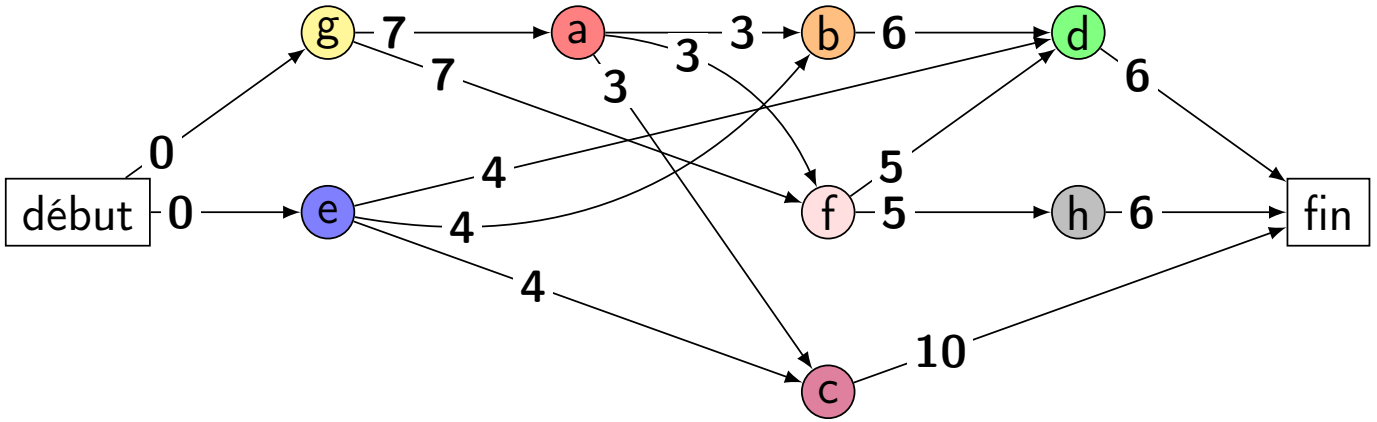
Algo : Dans l'ordre des niveaux $\rightsquigarrow \max_{t \in pred(s)} time(t) + length(t)$

Plan

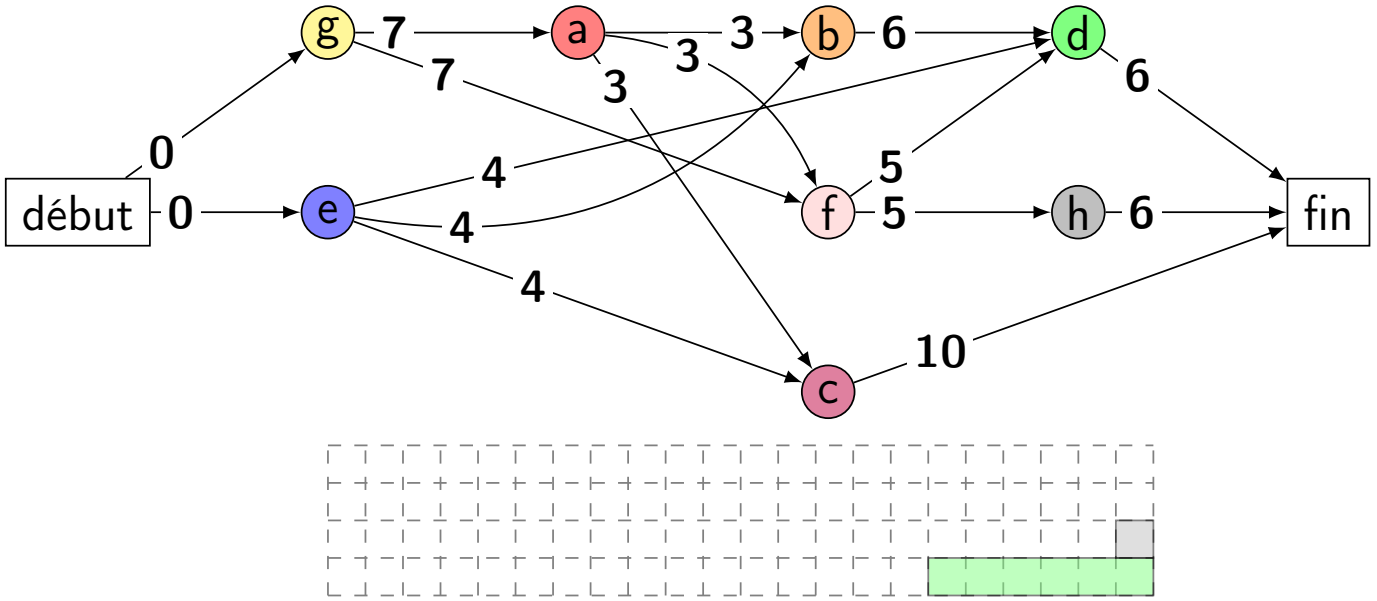
Ordonnancement au plus tard



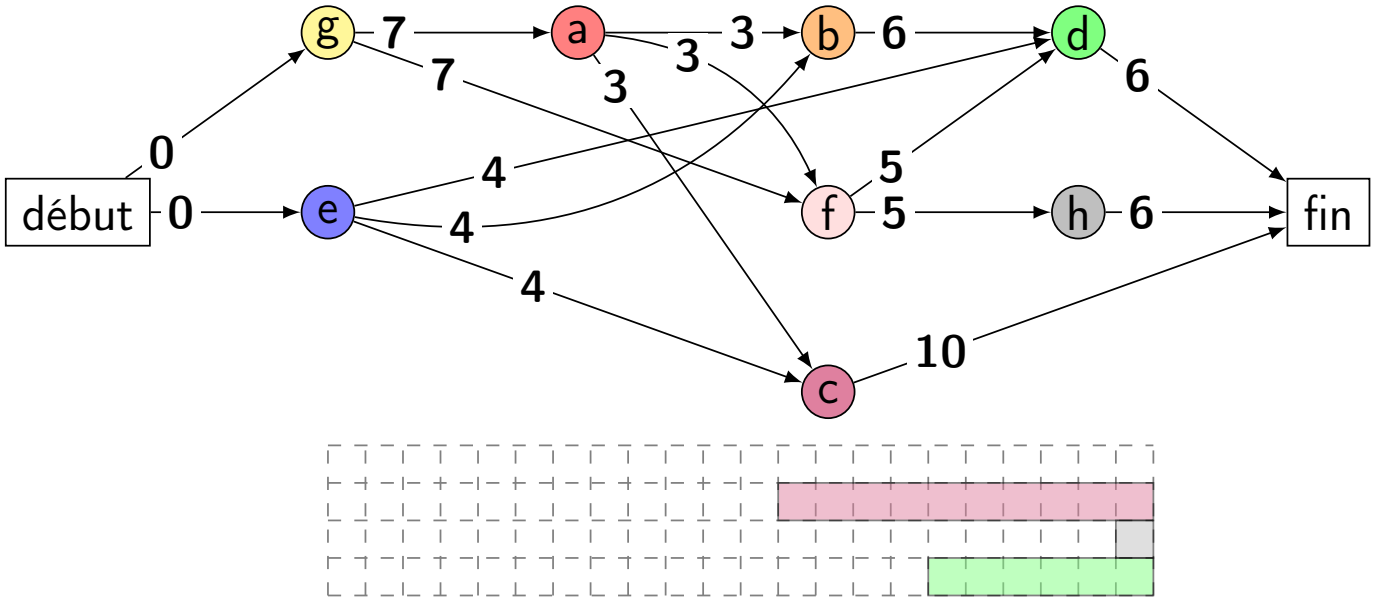
Ordonnancement au plus tard



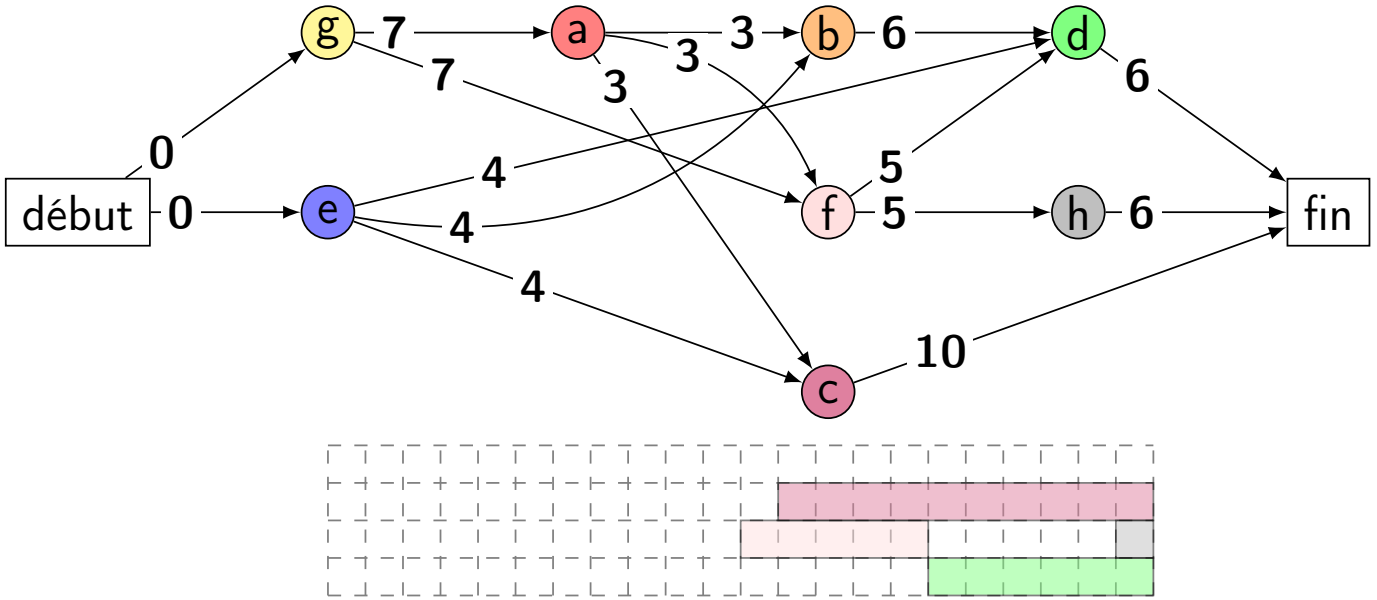
Ordonnancement au plus tard



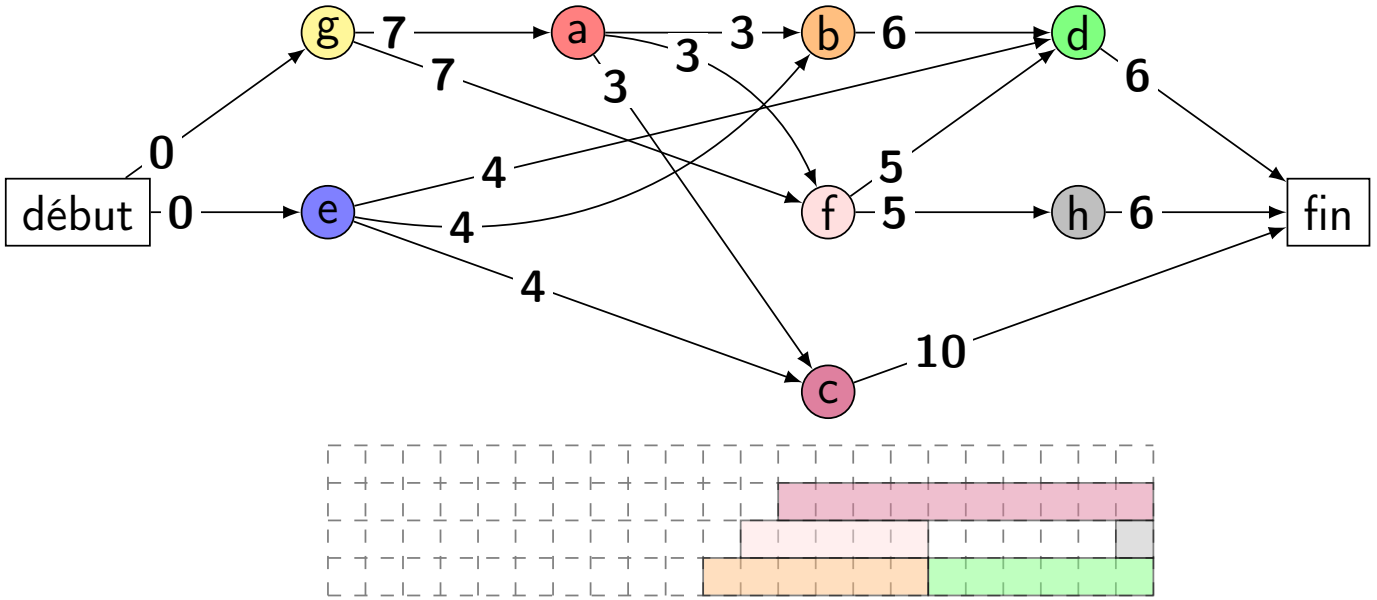
Ordonnancement au plus tard



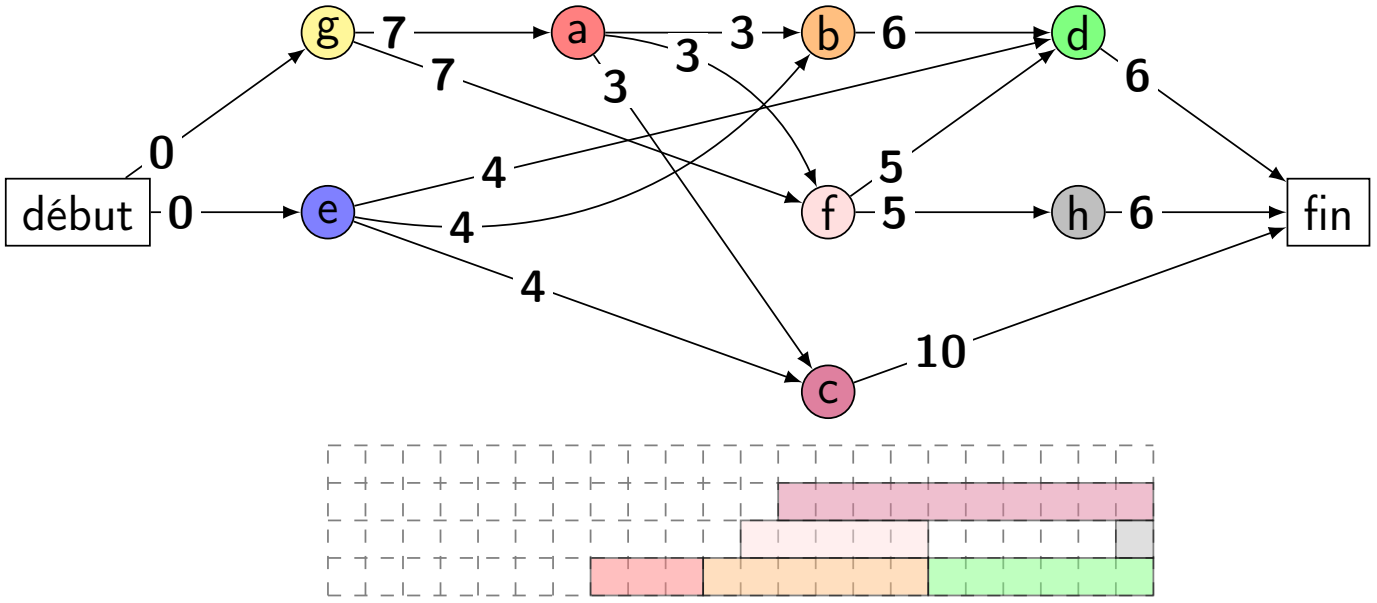
Ordonnancement au plus tard



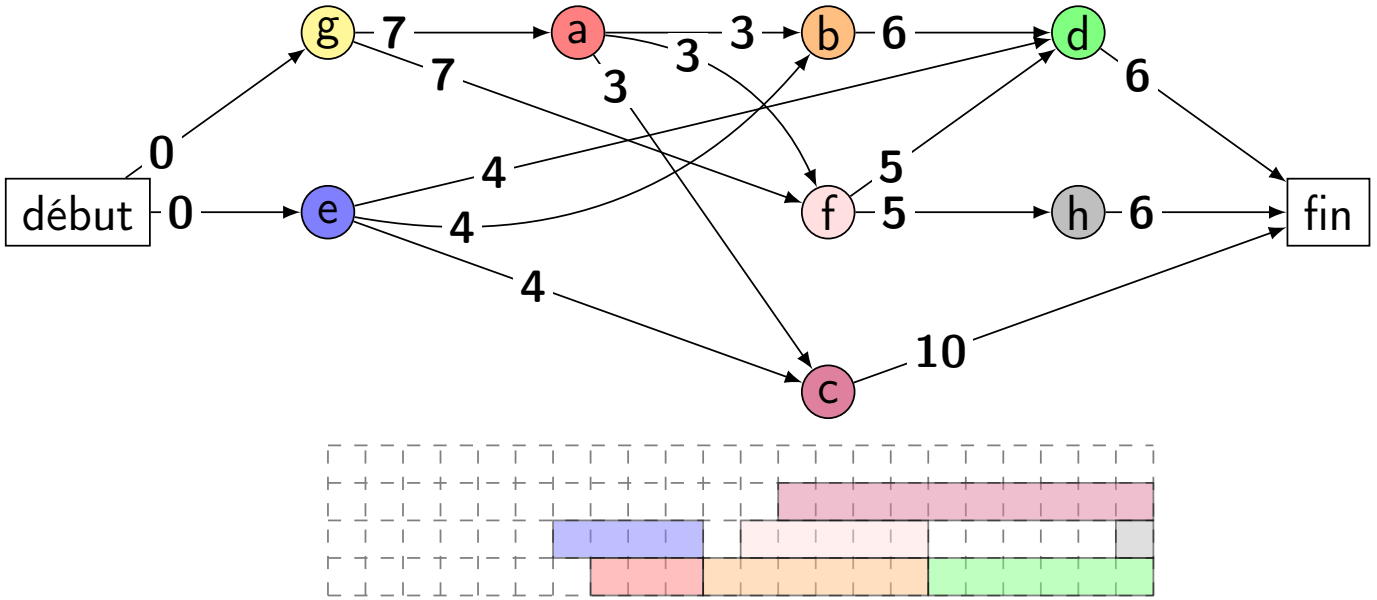
Ordonnancement au plus tard



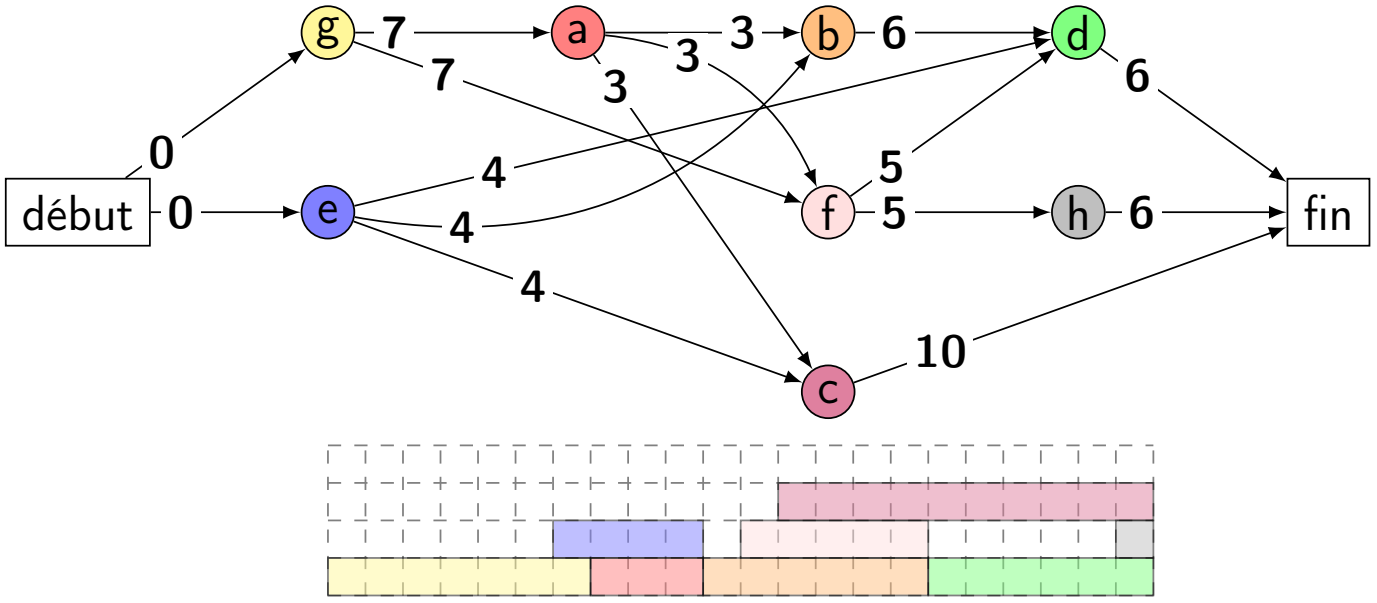
Ordonnancement au plus tard



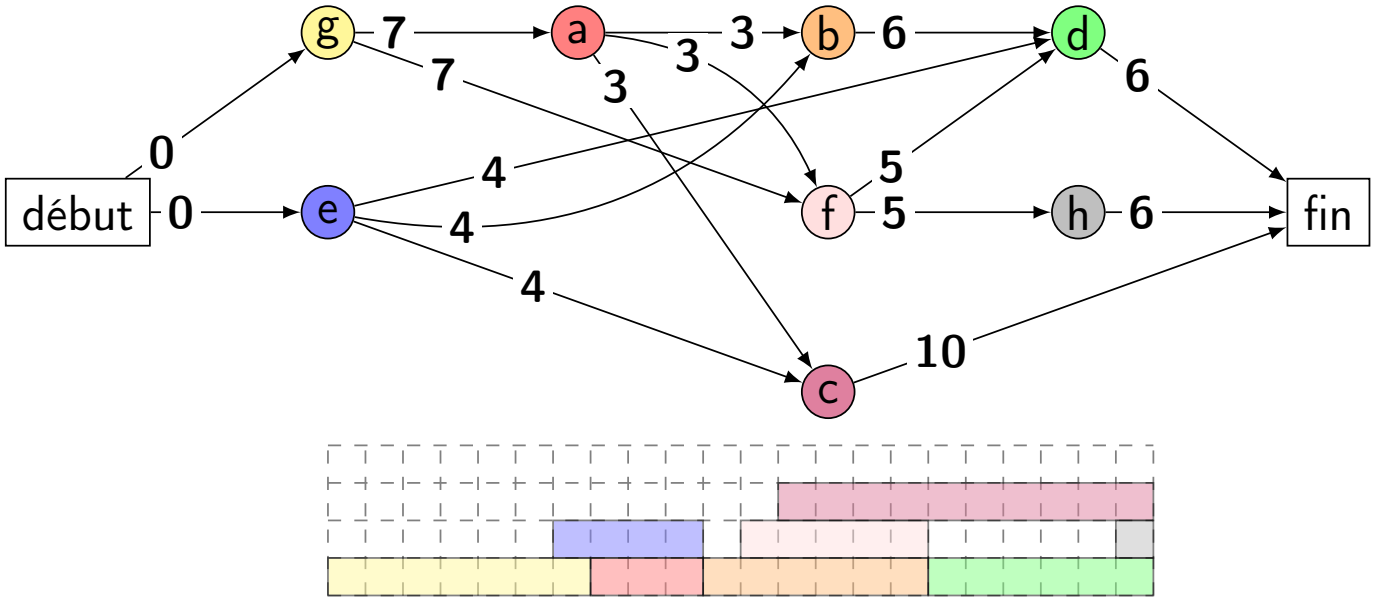
Ordonnancement au plus tard



Ordonnancement au plus tard

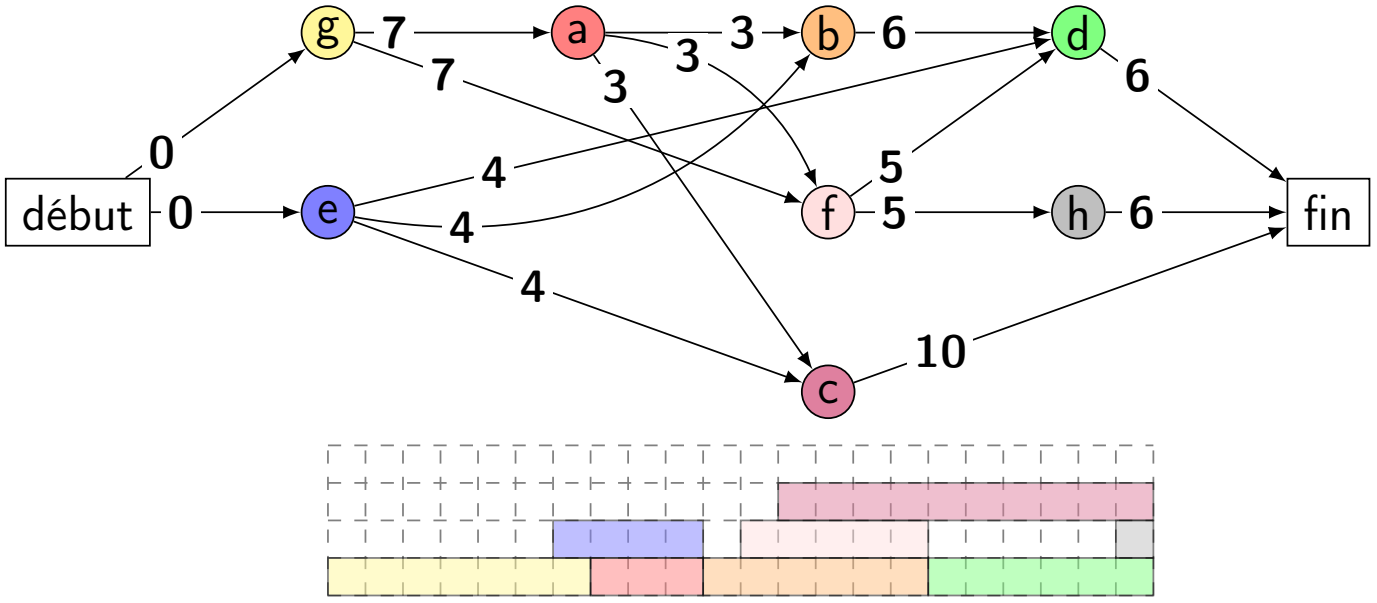


Ordonnancement au plus tard



Algo : Dans l'ordre inverse des niveaux $\rightsquigarrow \min_{t \in succ(s)} time(t) - length(s)$

Ordonnancement au plus tard



Algo : Dans l'ordre inverse des niveaux $\rightsquigarrow \min_{t \in succ(s)} time(t) - length(s)$

Différence temps min/temps max : marge

Ordonnancement/Tri topologique

Hypothèse implicites :

- Pas de circuit (DAG)

Ordonnancement/Tri topologique

Hypothèse implicites :

- Pas de circuit (DAG)
- Ressources illimitées/ égale à 1

Ordonnancement/Tri topologique

Hypothèse implicites :

- Pas de circuit (DAG)
- Ressources illimitées/ égale à 1
- Durée connues

Ordonnancement/Tri topologique

Hypothèse implicites :

- Pas de circuit (DAG)
- Ressources illimitées/ égale à 1
- Durée connues

- Il existe un algo polynomial optimal pour $p = 2$ processeurs (Coffman–Graham)

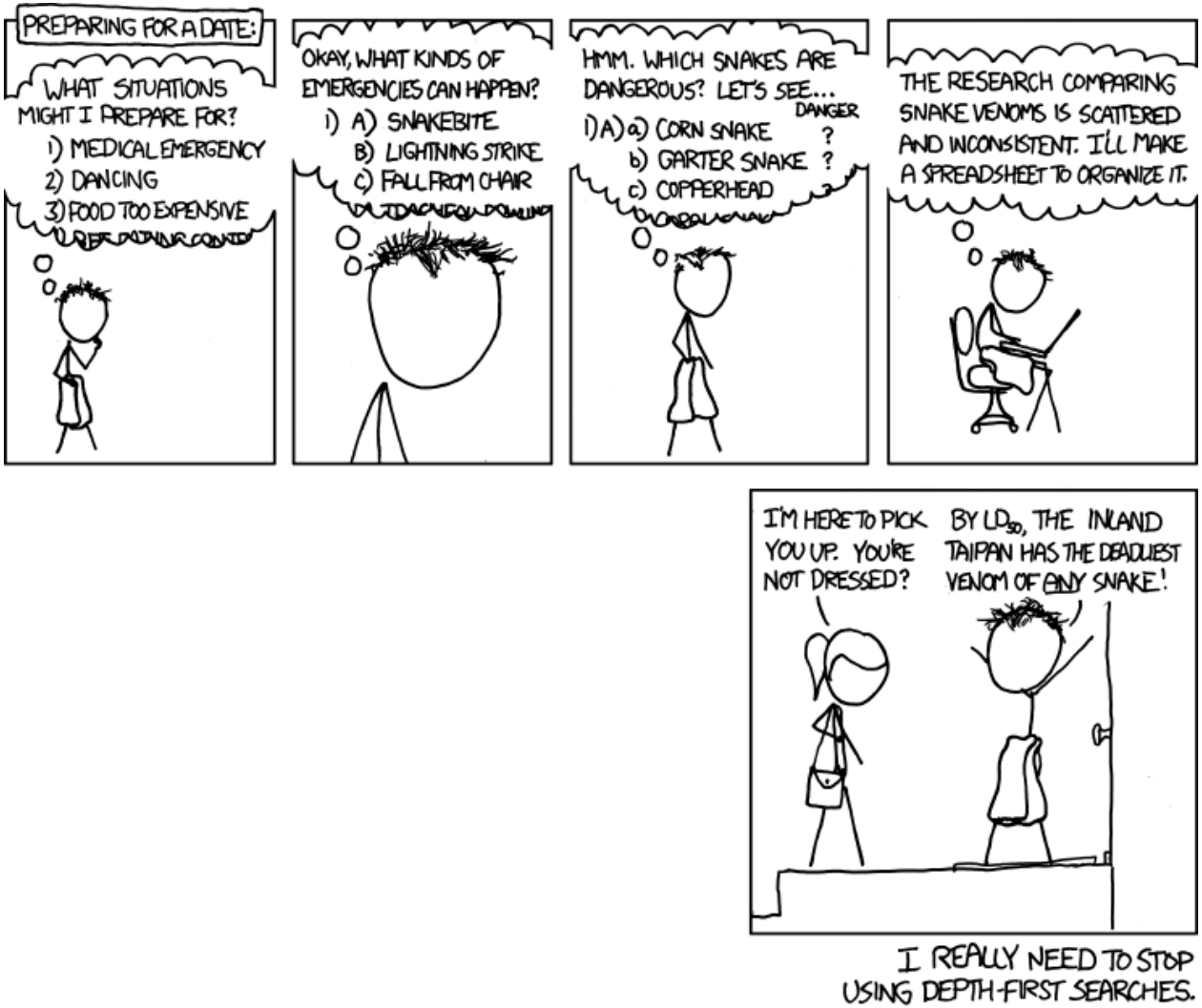
Ordonnancement/Tri topologique

Hypothèse implicites :

- Pas de circuit (DAG)
- Ressources illimitées/ égale à 1
- Durée connues

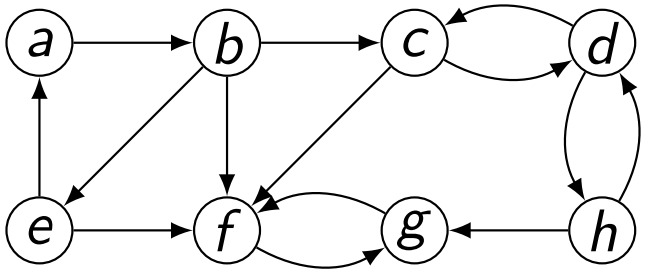
- Il existe un algo polynomial optimal pour $p = 2$ processeurs (Coffman–Graham)
- pour p le problème est NP-complet

Questions ? Commentaires ?



source : https://xkcd.com/761/, CC ANC 2.5

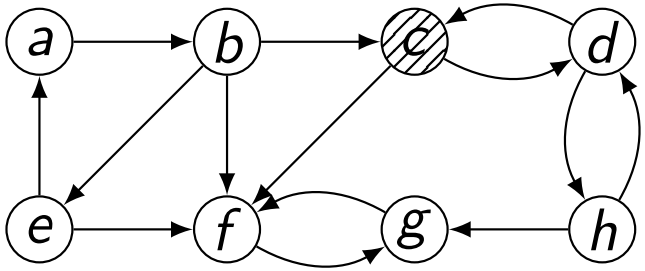
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe :

Algorithme :

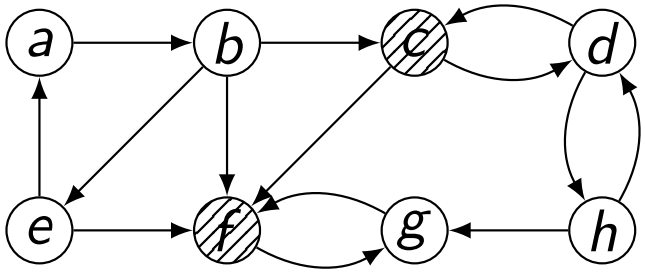
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe :

Algorithme :

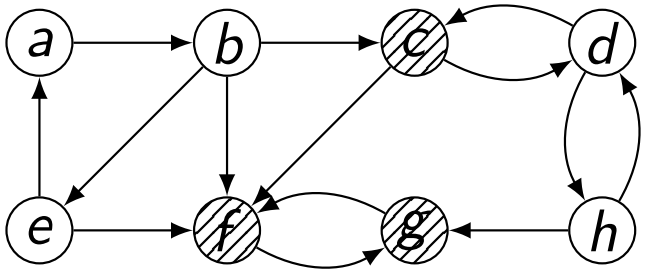
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe :

Algorithme :

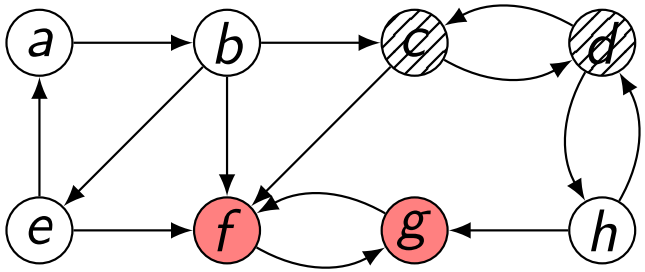
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe :

Algorithme :

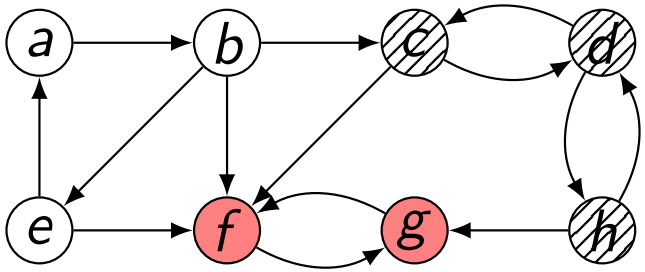
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f

Algorithme :

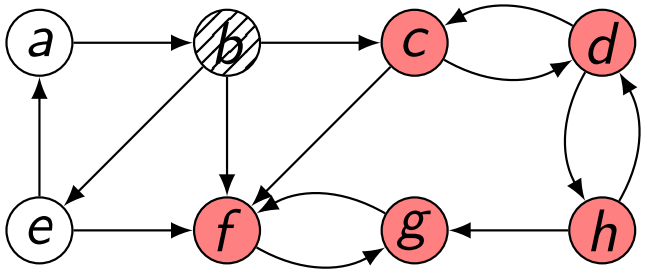
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c

Algorithme :

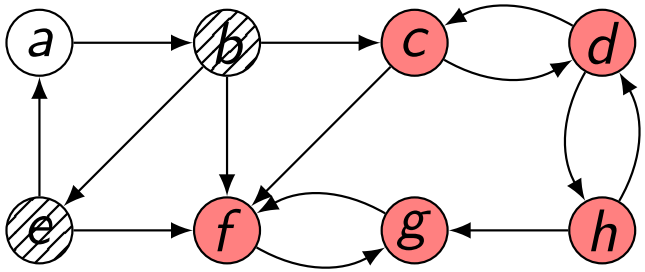
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c

Algorithme :

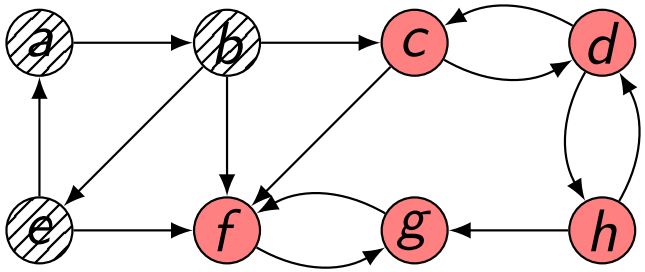
Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c

Algorithme :

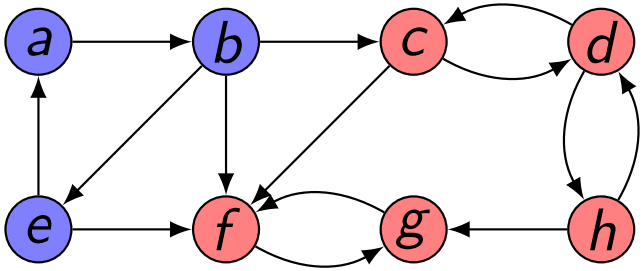
Contenu supplémentaire : Algorithme de Kosaraju



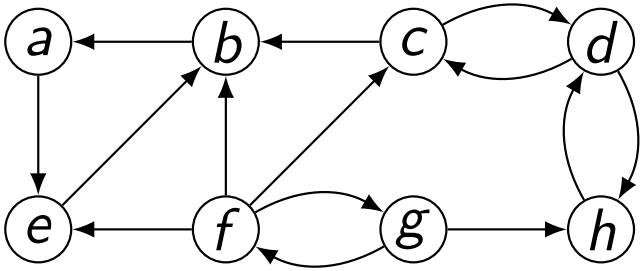
Ordre post-fixe : g,f,h,d,c,a,e,b

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

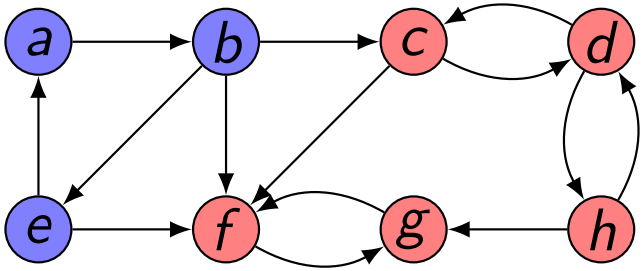


On renverse l'ordre :

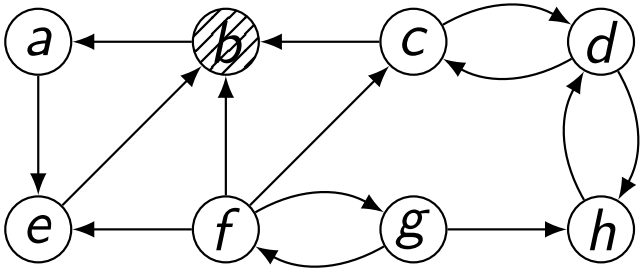
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

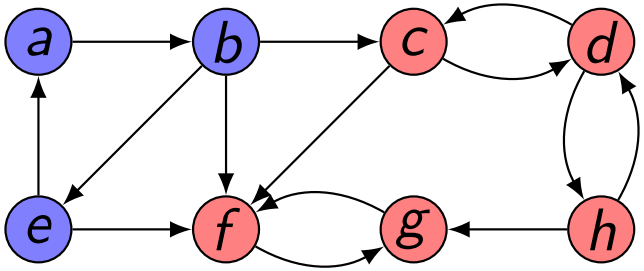


On renverse l'ordre :

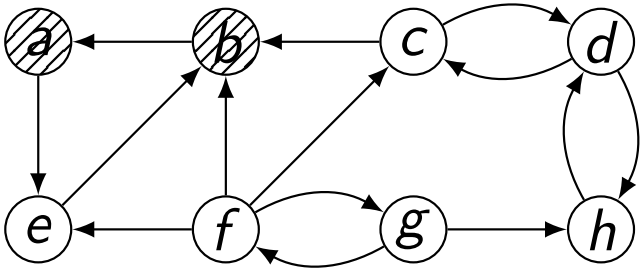
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

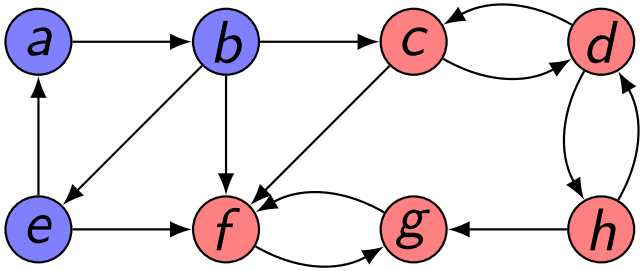


On renverse l'ordre :

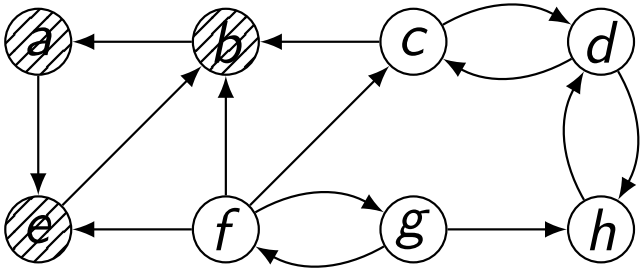
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

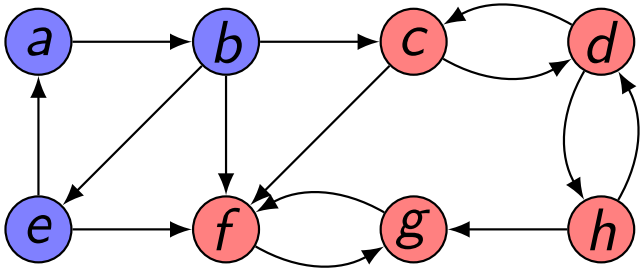


On renverse l'ordre :

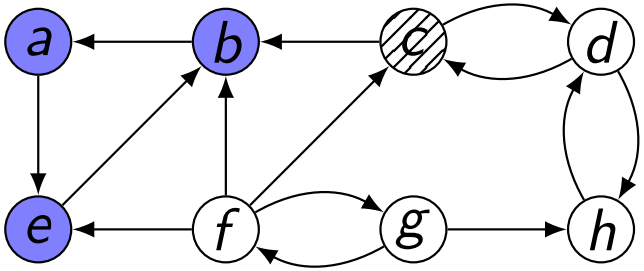
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

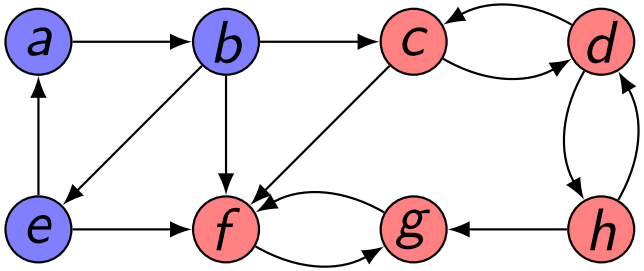


On renverse l'ordre :

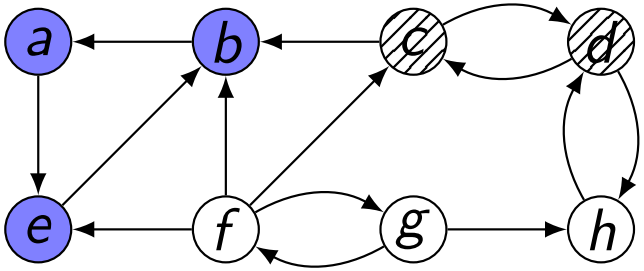
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

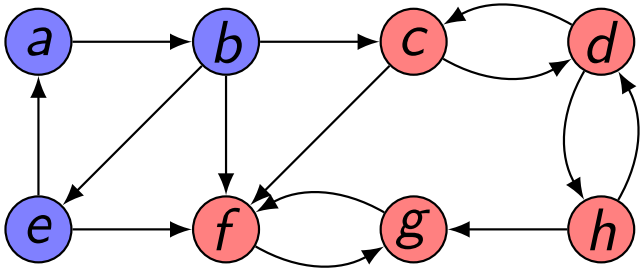


On renverse l'ordre :

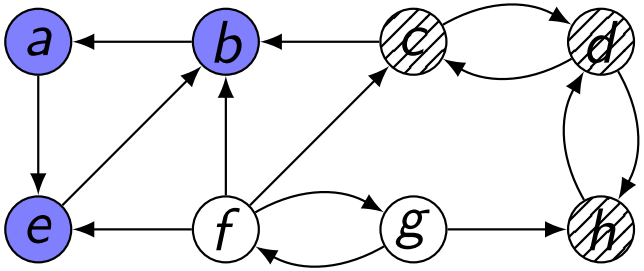
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

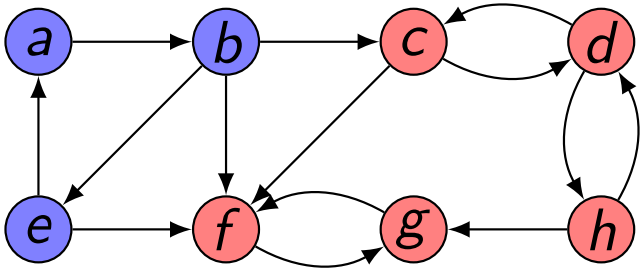


On renverse l'ordre :

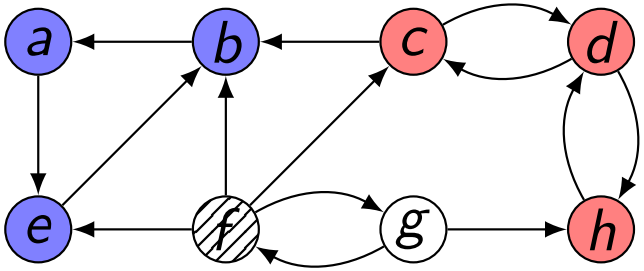
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

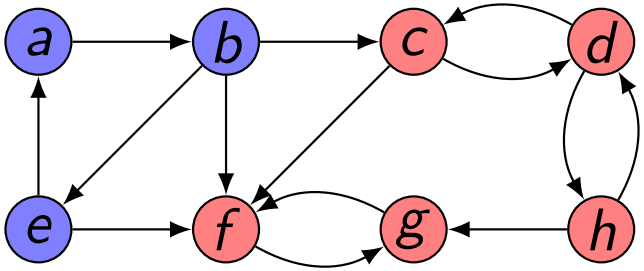


On renverse l'ordre :

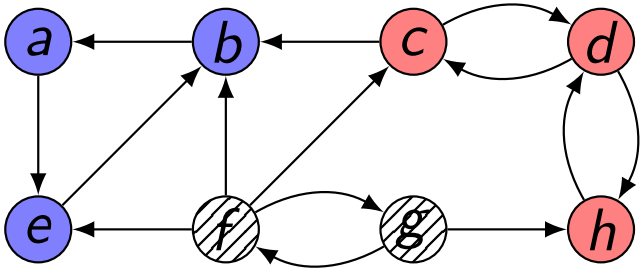
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b

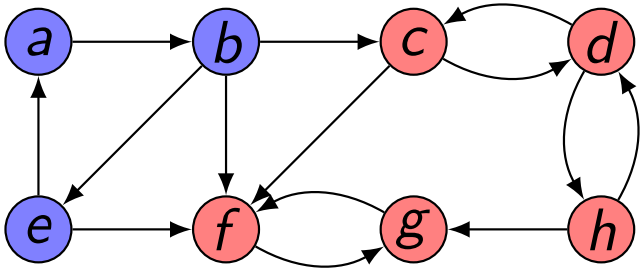


On renverse l'ordre :

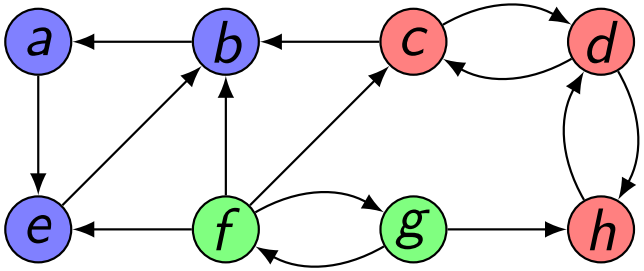
b,e,a,c,d,h,f,g

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b



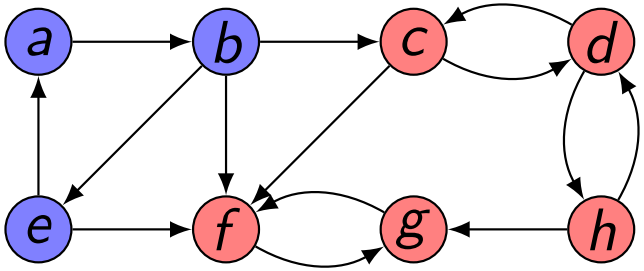
On renverse l'ordre :

b,e,a,c,d,h,f,g 3 composantes fortement connexes :

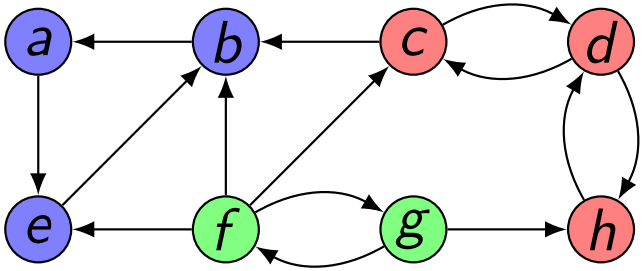
$\{b, e, a\}; \{c, d, g\}; \{f, g\}$

Algorithme :

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b



On renverse l'ordre :

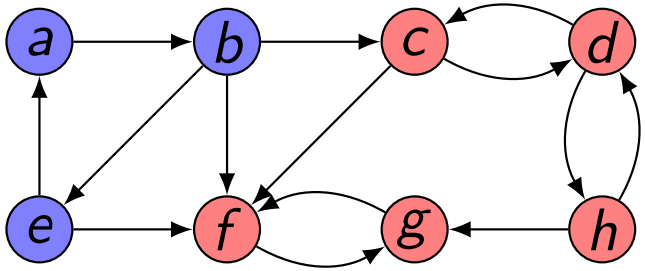
b,e,a,c,d,h,f,g 3 composantes fortement connexes :

$\{b, e, a\}; \{c, d, g\}; \{f, g\}$

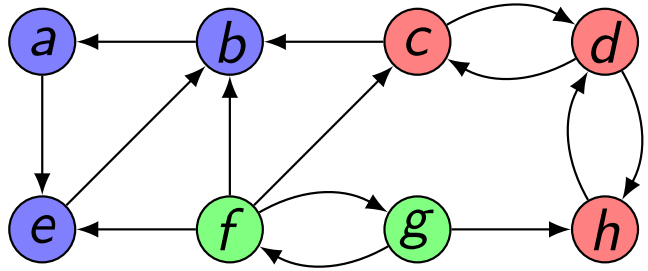
Algorithme :

1- Calculer un ordre postfixe du graphe

Contenu supplémentaire : Algorithme de Kosaraju



Ordre post-fixe : g,f,h,d,c,a,e,b



On renverse l'ordre :

b,e,a,c,d,h,f,g 3 composantes fortement connexes :

$\{b, e, a\}; \{c, d, g\}; \{f, g\}$

Algorithme :

1- Calculer un ordre postfixe du graphe

2- Inverser les arcs et faire un parcours dans l'ordre post-fixe renversé