Algorithmique 2 (INFO-F203)
Parcours et Cycles

Jean Cardinal

Mars 2021

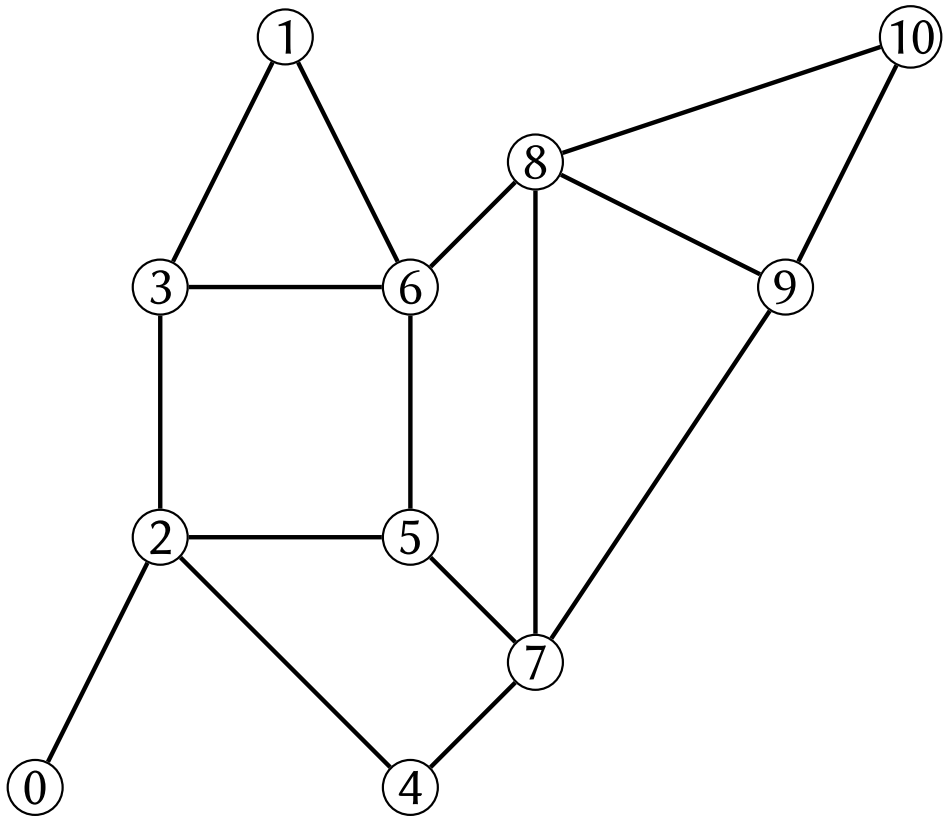
Erratum : Arbres aléatoires

Moyenne de la somme des profondeurs des nœuds d'un arbre de recherche binaire aléatoire :

$$T(n) = 1 + \sum_{i=0}^{n-1} \frac{1}{n}(T(i) + T(n-i-1)) + (n-1).$$

Graphes

Sommets
Arêtes
Degré
Chemin
Connexité
Cycle



Représentation

Matrice d'adjacence $M_{ij} = 1 \Leftrightarrow ij \in \mathcal{E}$

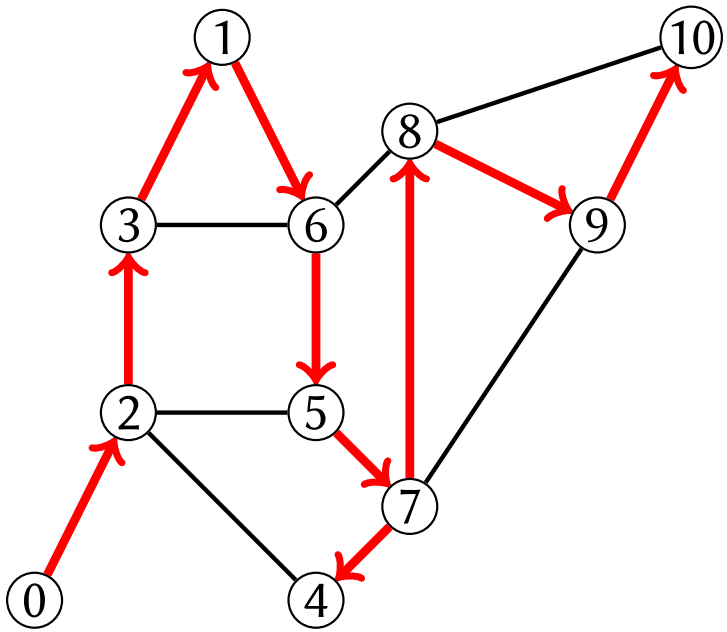
Listes d'adjacence

```
for (int v = 0; v < G.V(); v++)
  for (int w : G.adj(v))
    StdOut.println(v + "-" + w);
```

Parcours en profondeur

```
private void dfs(Graph G, int v) {
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w]) {
         dfs(G, w);
         edgeTo[w] = v;
      }
}
```

Exemple



- 0,2,3,1,6,5,7,4,8,9,10

Composantes connexes

L'algorithme de parcours en profondeur permet de compter le nombre de composantes connexes du graphes. Il suffit d'appeler de manière répétée la procédure de parcours sur les sommets non encore visités.

```
for (int v = 0; v < G.V(); v++) {
  if (!marked[v]) {
    dfs(G, v);
    count++;
  }
}
```

Parcours en largeur

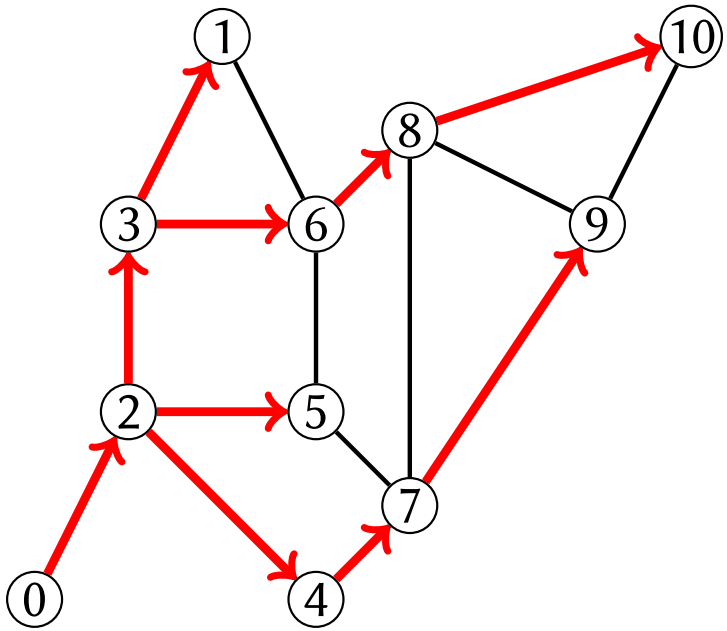
L'algorithme de parcours en largeur visite les sommets par ordre croissant de distance à partir du sommet initial. La distance entre deux sommets est ici définie comme le nombre d'arêtes minimum dans un chemin entre deux sommets.

File

1. Choisir le sommet suivant sur la file
2. Marquer et insérer dans la file tous ses voisins
3. Retenir les arêtes parcourues et la distance

```
private void bfs(Graph G, int s) {
    Queue<Integer> q = new Queue<Integer>();
    q.enqueue(s);
    marked[s] = true;
    distTo[s] = 0;
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                q.enqueue(w);
                marked[w] = true;
                edgeTo[w] = v;
                distTo[w] = distTo[v] + 1;
            }
        }
    }
}
```

Exemple



distance 0 : 0

distance 1 : 2

distance 2 : 3,4,5

distance 3 : 1,6,7

distance 4 : 8,9

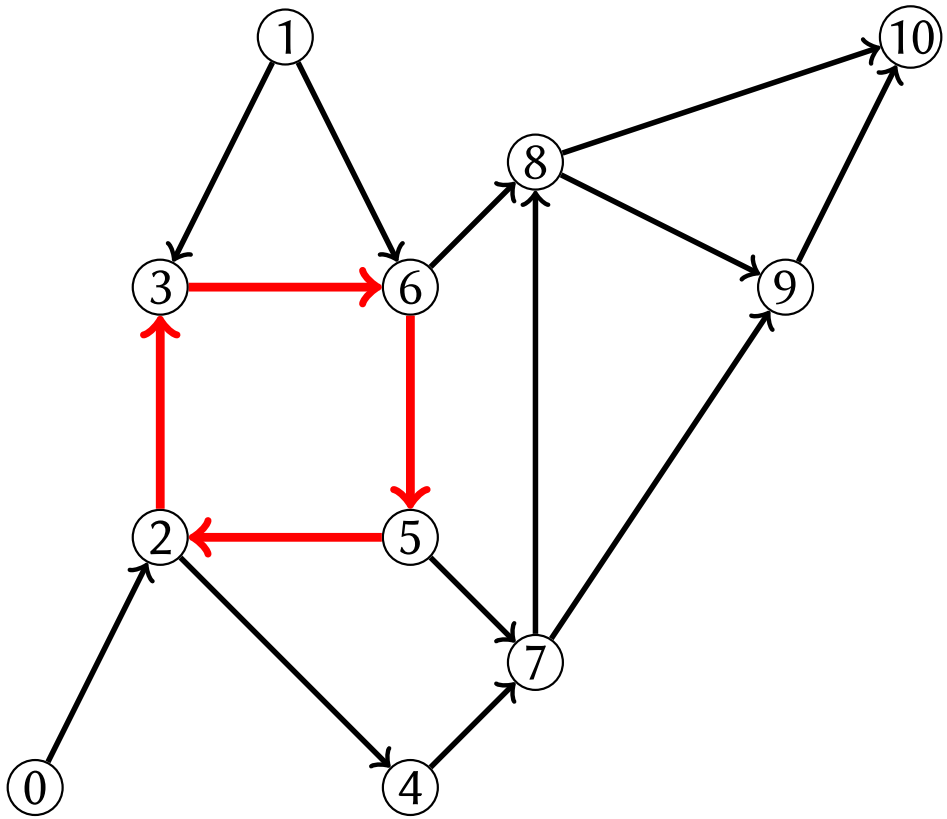
distance 5 : 10

Plus courts chemins

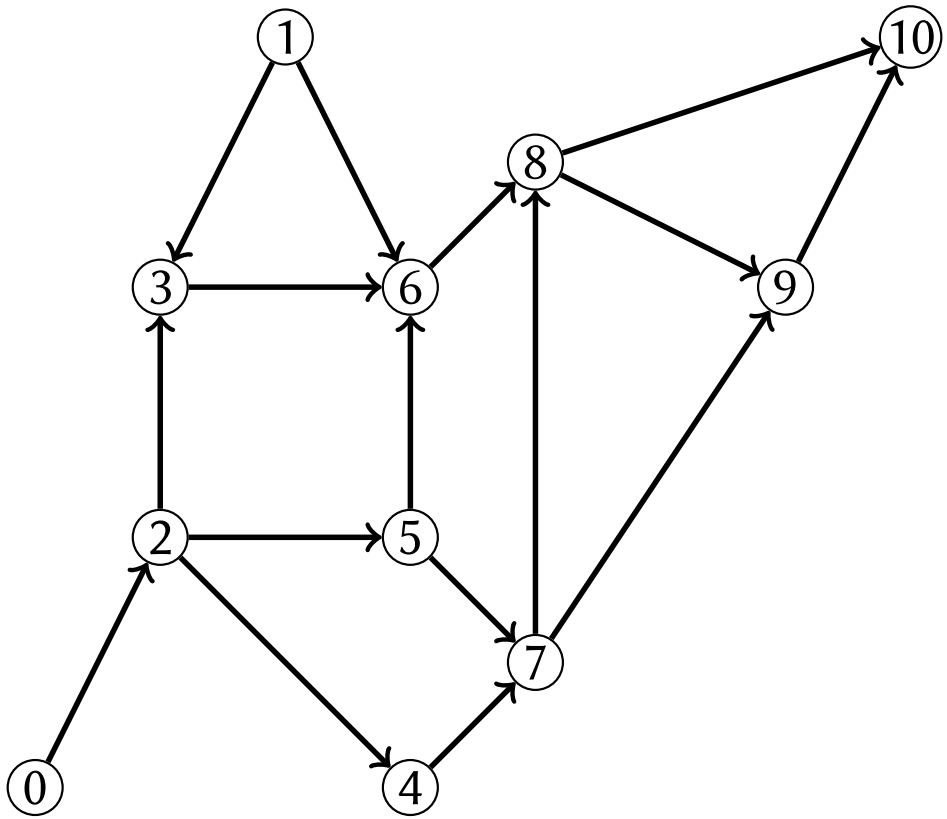
Proposition 1

L'algorithme de parcours en largeur calcule un plus court chemin entre le sommet de départ et tous les sommets accessibles, en un temps proportionnel à $E + V$ au pire cas.

Graphes dirigés



Graphes dirigés acycliques (DAG)



Détection de cycles

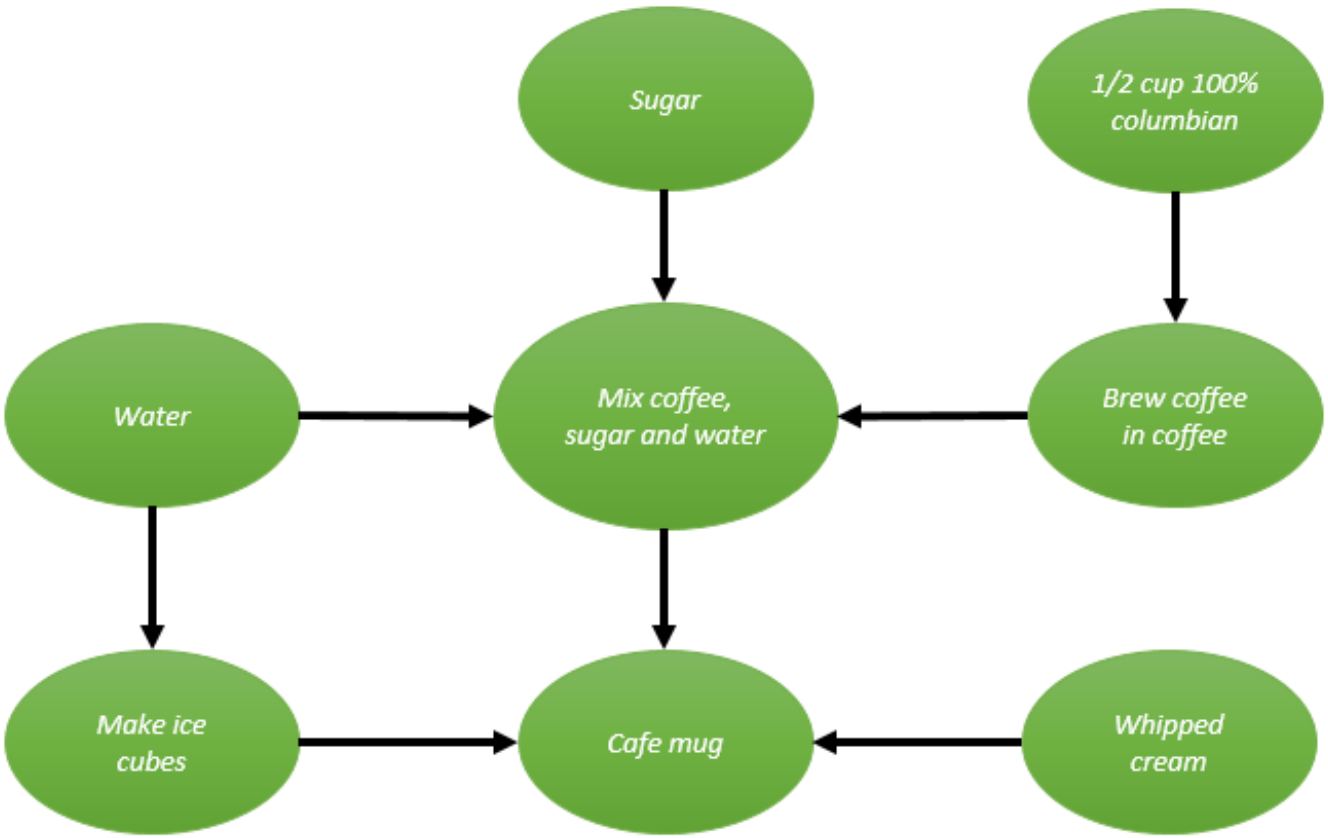
Lors d'un parcours en profondeur, on détecte un cycle dès qu'un sommet u pour lequel un appel récursif est en cours est l'extrémité d'un arc émanant du sommet v courant.

- Supposons que cela se produise. Alors, par définition, il existe un chemin de u vers v. Si de plus il existe un arc de v vers u, celui-ci "ferme" le cycle.
- Supposons d'autre part l'existence d'un cycle. La première fois que le parcours rencontre un des sommets du cycle, l'appel récursif à ce sommet ne se termine pas avant d'avoir exploré tous les sommets accessibles, donc tous les sommets du cycle.

Mise en œuvre

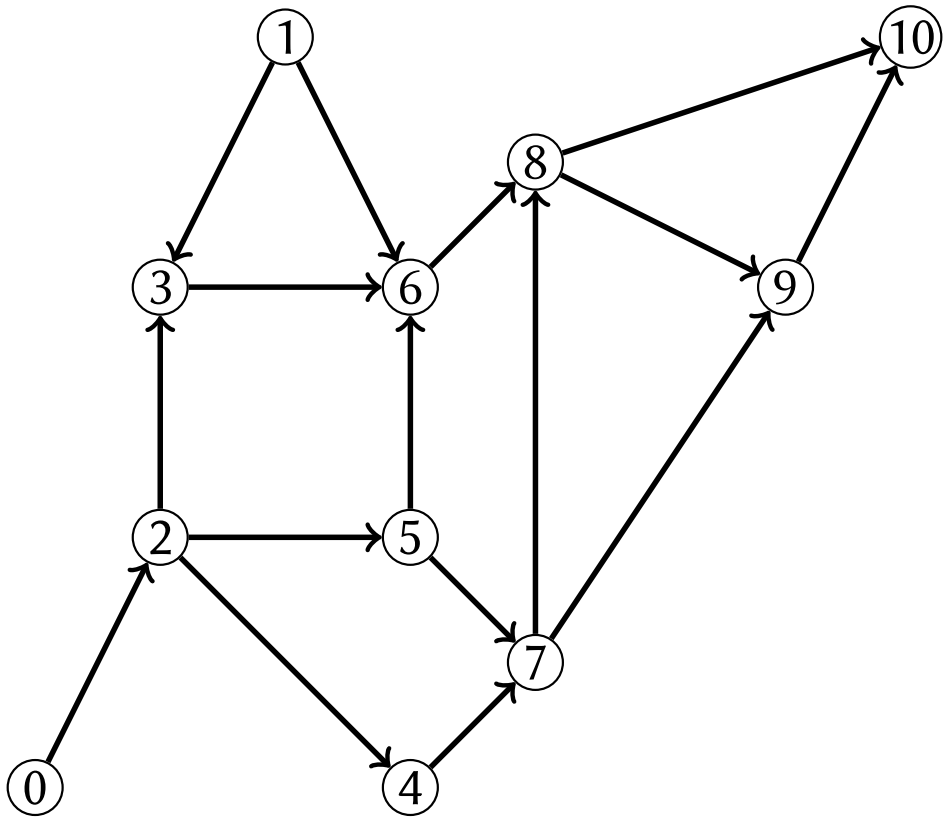
```
private void dfs(Digraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v)) {

        // termine si un cycle est construit
        if (cycle != null) return;

        // on trouve un nouveau sommet accessible
        else if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }

        // si le sommet est sur la pile, il y a un cycle
        else if (onStack[w]) {
          // construire le cycle
        }
    }
    onStack[v] = false;
}
```

Tri topologique et organisation

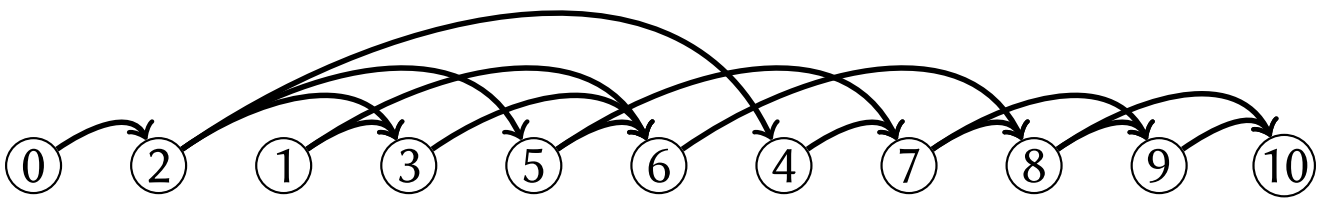


Steps To Make Iced Coffee

Graphes dirigés acycliques (DAG)



Tri topologique



Existence

Proposition 2

Un graphe dirigé possède un ordre topologique si et seulement s'il est acyclique.

Démonstration et construction

Pour démontrer cette proposition, nous considérons les deux implications séparément. Tout d'abord, il est clair que si le graphe dirigé comporte un cycle, alors il est impossible d'ordonner totalement les sommets de ce cycle sans avoir d'arc dans la mauvaise direction.
Nous devons maintenant, en faisant l'hypothèse de l'absence de cycle, démontrer l'existence d'un ordre topologique. Considérons un parcours en profondeur du graphe dirigé G. Celui-ci induit deux ordres sur les sommets :

- le *préordre* : l'ordre dans lequel la procédure `dfs` est appelée,
- le *postordre* : l'ordre dans lequel la procédure `dfs` se termine.

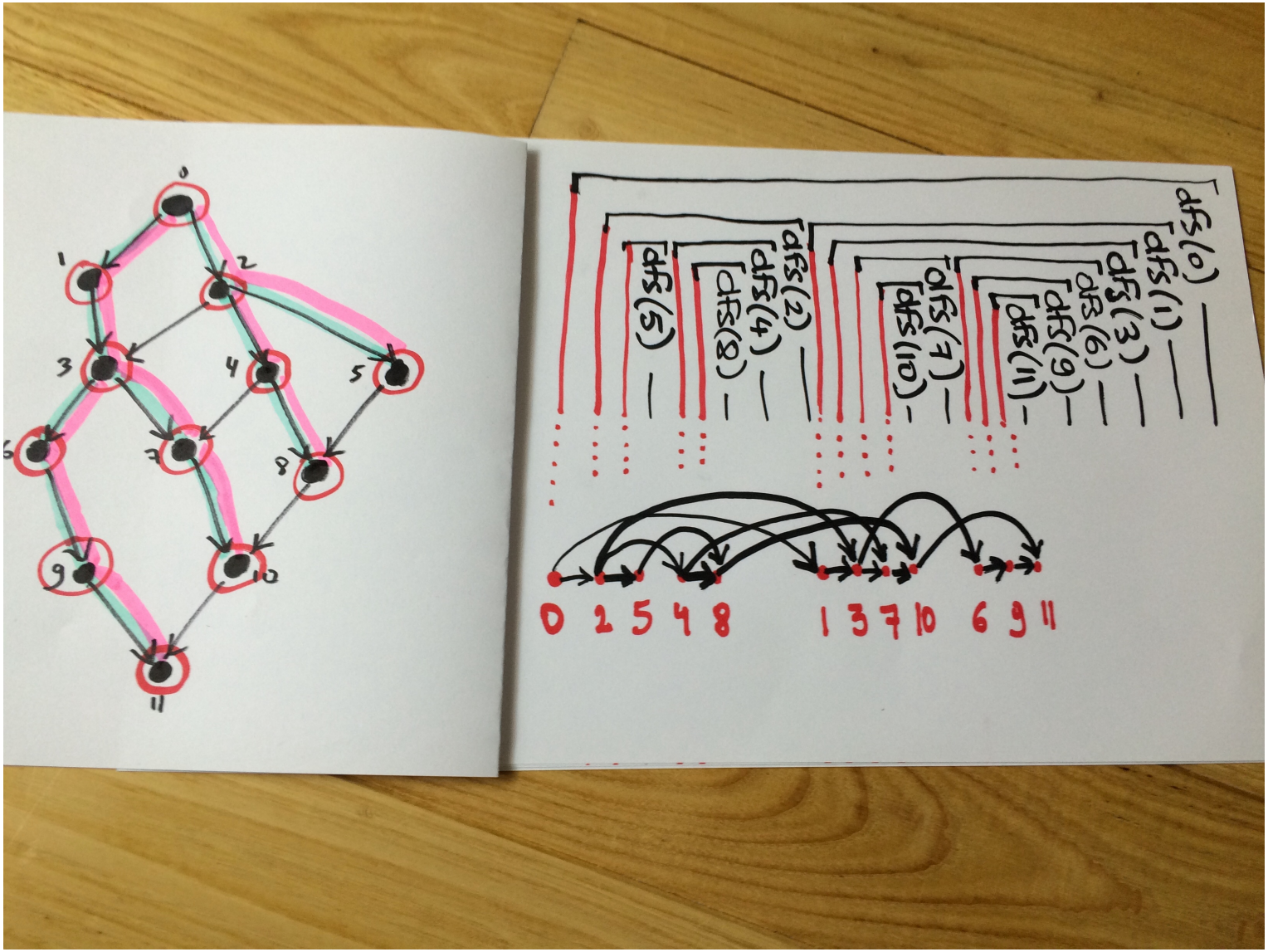
Enfin, un troisième ordre est le *postordre inverse*, qui est simplement l'image miroir du postordre. Nous pouvons nous convaincre qu'en l'absence de cycle, le postordre inverse est un ordre topologique.

Démonstration et construction

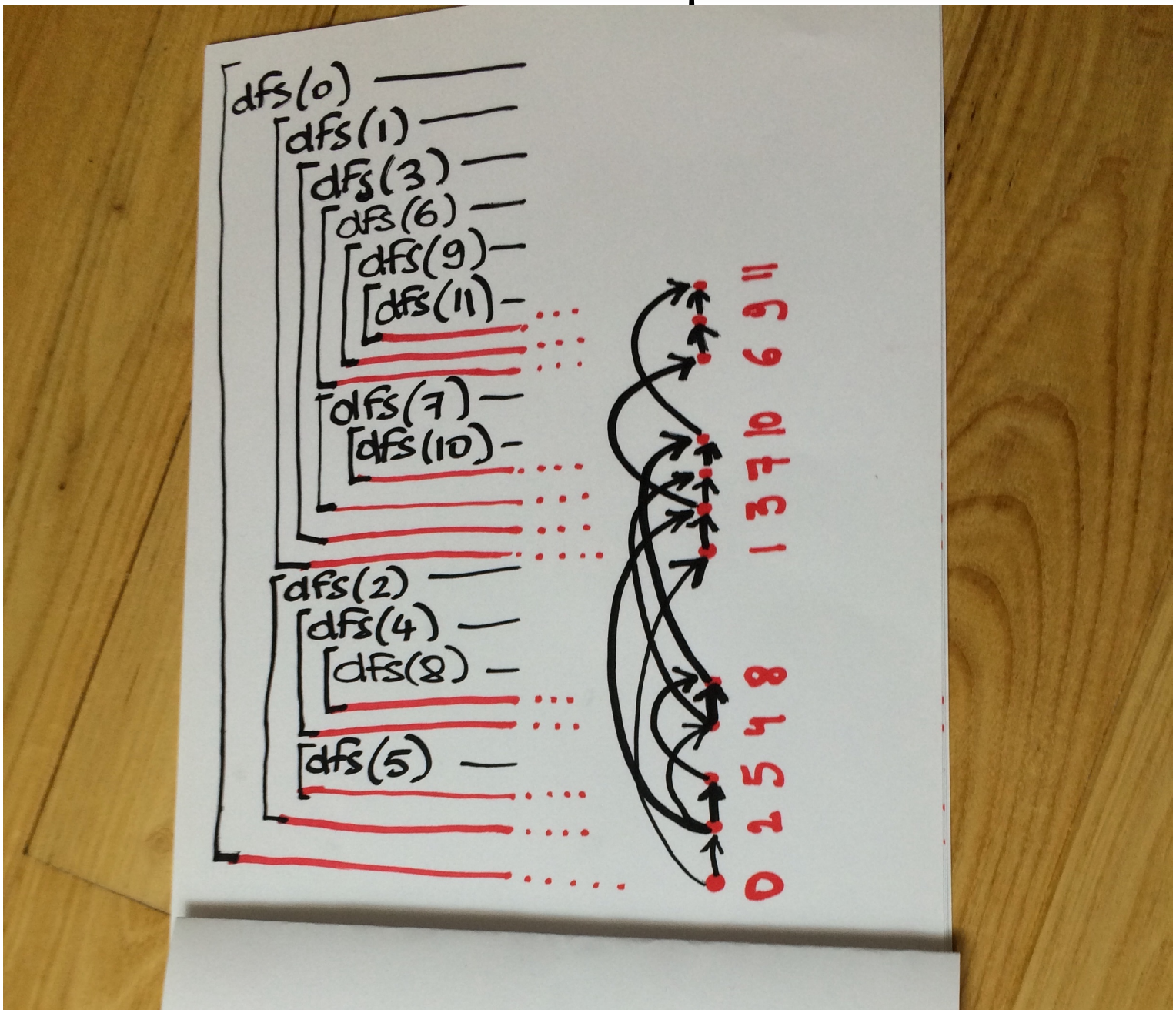
En effet, considérons un arc du sommet `v` au sommet `w`. Lorsque l'appel `dfs(G,v)` est effectué, trois cas de figure peuvent survenir :

1. L'appel `dfs(G,w)` a déjà été effectué, et s'est terminé. Le sommet `w` apparaît donc après `v` dans le postordre inverse, et celui-ci n'est donc pas incompatible avec l'arc.
2. L'appel `dfs(G,w)` n'a pas encore été effectué. Alors celui-ci sera effectué directement ou indirectement par l'appel courant `dfs(G,v)`. Par conséquent, il terminera avant `dfs(G,v)`, et `w` apparaîtra à nouveau après `v` dans le postordre inverse.
3. L'appel `dfs(G,w)` a déjà été effectué, mais ne s'est pas terminé. Dans ce cas, comme vu précédemment, il existe un cycle dans G, ce qui contredit notre hypothèse de travail.

Exemple


dfs(0)
dfs(1)
dfs(3)
dfs(6)
dfs(9)
dfs(11)
dfs(10)
dfs(7)
dfs(2)
dfs(4)
dfs(8)
dfs(5)
0 2 5 4 8 1 3 7 10 6 9 11

Exemple


dfs(0)
dfs(1)
dfs(3)
dfs(6)
dfs(9)
dfs(11)
dfs(7)
dfs(10)
dfs(2)
dfs(4)
dfs(8)
dfs(5)
0 2 5 4 8 7 3 1 6 9 11