Algorithmique (AL5)
TD n° 3 : Parcours en profondeur

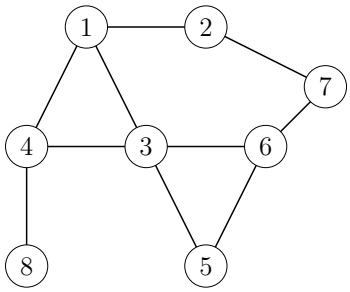
Rappels

Le parcours en profondeur (DFS en anglais abrégé) peut s'effectuer à la fois pour un graphe non-orienté ou un graphe orienté. Voici un algorithme en pseudo-code pour la version récursive

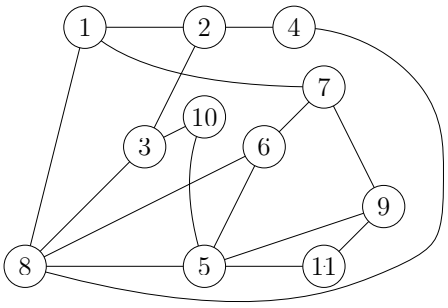
```
1  explorer(G,u) : //la procédure de base
2      marqué[u]=Vrai
3      //pre-visite si besoin (voir cours)
4      Pour tout uv ∈ E faire
5        Si marqué[v] = Faux alors
6          pere[v]=u
7          explorer(G,v)
8      //post-visite si besoin
9
10 DFS(G): // la procédure complète
11   pour tout u de V faire
12     marqué[u]=Faux
13     pere[u]=Vide
14   pour tout u de V faire
15     si marque[u]=Faux faire
16       explorer(G,u)
17   retourne pere
```

Exercice 1 : Parcours en profondeur de graphes non orientés

1. Appliquer aux deux graphes ci-dessous l'algorithme de parcours en profondeur à partir du sommet 1. On suppose que le graphe est présenté par liste d'adjacences et que celles-ci sont triées par étiquette croissante de sommets.



Graphe G_1



Graphe G_2

2. Dans les exemples précédents combien de fois la ligne 16 du pseudocode plus haut a été exécutée ? Dans quels types de graphes cette ligne peut s'exécuter plusieurs fois ?

Rappels 2

Lorsque l'on effectue le parcours en profondeur d'un graphe orienté, en plus de noter la relation de parent qui constitue l'arbre ou la forêt de parcours on classifie les arcs du digraphe en 4 types :

- les arcs avant, c'est à dire des arcs d'un sommet vers un de ses descendants dans l'arbre (les arcs de l'arbre sont comptés comme des arcs avant)
- les arcs retour, c'est à dire des arcs d'un sommet vers un de ses ancêtres dans l'arbre précédent (par convention une boucle sera considérée comme un arc retour).
- les arcs dits transverses : les autres.

Dans le cas orienté, on a aussi généralement une variable globale *temps* qui commence à 0 et qui sert à noter pour chaque sommet u deux temps $pre(u)$ et $post(u)$ qui notent les début et fin de visite. La pre-visite (respectivement post-visite) consiste a incrémenter *temps* et à l'affecter à $pre(u)$ (resp. $post(u)$).

On rappelle que pour deux sommets distincts u et v, les deux intervalles $[pre(u); post(u)]$ et $[pre(v); post(v)]$ ne peuvent se chevaucher (soit ils ne s'intersectent pas, soit l'un est inclue dans l'autre). De plus pour tout arc uv on a nécessairement $pre(v) < post(u)$ (à cause de l'arc on ne peut avoir fini de traiter u si on a pas commencé a traiter à v). Enfin un résultat du cours permet de caractériser exactement le type d'un arc en fonction de ces valeurs :

- Si $pre(u) < pre(v) < post(v) < post(u)$ alors uv est un arc avant
- Si $pre(v) < pre(u) < post(u) < post(v)$ alors uv est un arc retour
- Si $pre(v) < post(v) < pre(u) < post(u)$, alors uv est un arc transverse.

Exercice 2 : Version itérative et temps de visite

On a donné dans les rappels le pseudocode pour la version récursive de l'algorithme du parcours en profondeur. On a aussi expliqué comment obtenir les tableaux *pre* et *post* avec les pre et post visite. Ecrire la version itérative (il faut utiliser une pile) en indiquant comment obtenir ces temps en sortie de l'algorithme.

Exercice 3 : Parcours en profondeur de graphe orientés

1. Appliquer au graphe de la Figure 1 l'algorithme de parcours en profondeur à partir du sommet 0 en considérant que le graphe est présenté par liste d'adjacence *avec sommets triés par étiquette croissante*. Dessiner la forêt du parcours, donner les dates *pre* et *post*, et étiqueter chaque arc par son type "avant", "retour" ou "transverse".

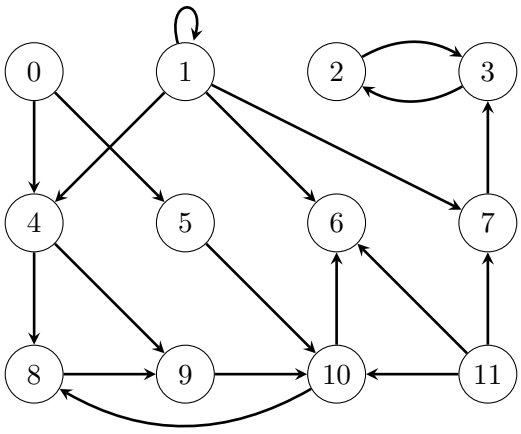


FIGURE 1 – Un graphe orienté.

2. Ici on ne suppose pas nécessairement que les listes d'adjacence sont triées par ordre de sommet croissant ; est-il possible que l'exécution de l'algorithme de parcours en profondeur pour le graphe de la Figure 1 donne les types d'arcs indiqués par la Figure 2 ? Justifiez votre réponse.

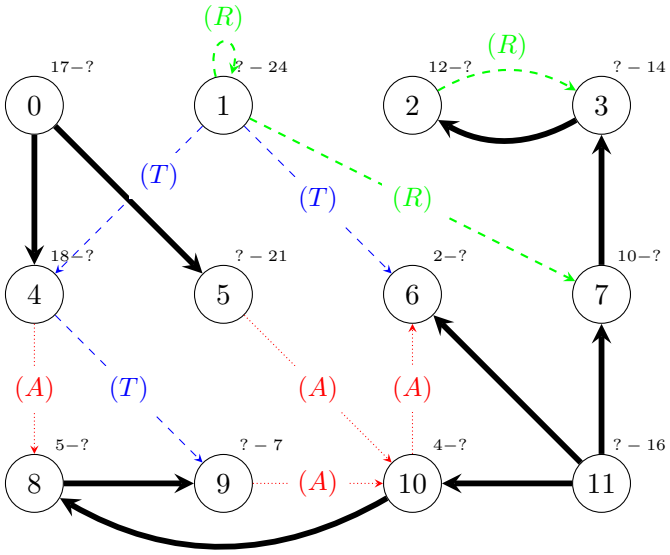


FIGURE 2 – Résultat possible de l'exécution du `PPC` ?

3. Même question si dans la Figure 2, on ne garde que les arcs de parcours (en gras). Si la réponse est «oui», indiquer pour chaque arc son type lors d'une exécution valide de `PPC` et remplir les dates manquantes de coloriage en gris et en noir.

4. Existe-il une exécution de l'algorithme de parcours en profondeur pour le graphe de la Figure 1 tel que les dates de coloriages en gris et en noir soient celles indiquées par les Figures 3 et respectivement 4 ci-dessous ? Justifiez votre réponse. Si oui, indiquer le type de chaque arc.

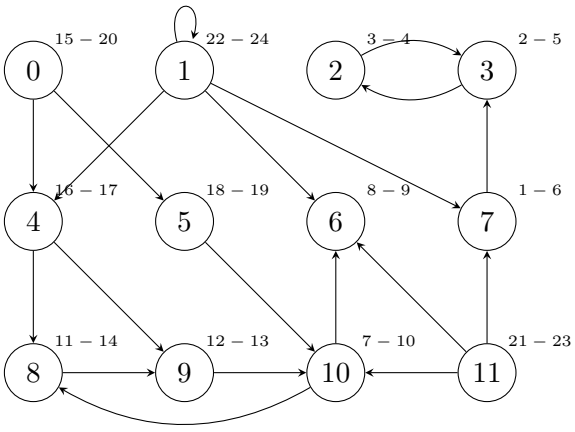


FIGURE 3 – Valeurs possibles de **pre** et **post** ?

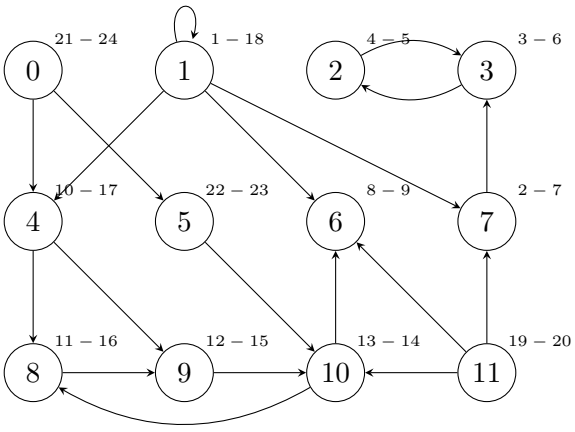


FIGURE 4 – Valeurs possibles de **pre** et **post** ?

Exercice 4 : Graphes Orientés Acycliques et Tri Topologique

Un graphe orienté acyclique (en anglais Directed Acyclic Graph, ou DAG) est un graphe orienté qui ne contient aucun cycle orienté.

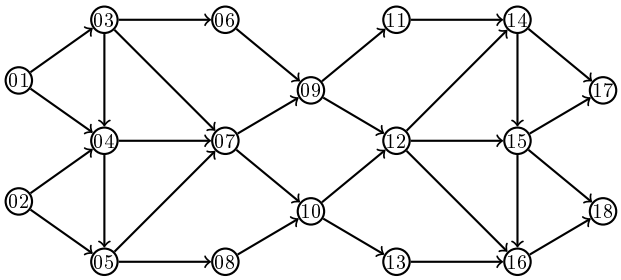
1. Montrer que si C est un cycle dans un graphe orienté G, alors dans toute exécution de parcours en profondeur de G, un des arcs de C sera un arc retour (on pourra utiliser la caractérisation des arcs retour par les fonctions `pre` et `post`). En déduire une modification de l'algorithme de parcours en profondeur qui permet de décider si un graphe orienté est un DAG.

2. Un *tri topologique* d'un graphe orienté $G = (V, E)$ est une énumération $(s_1, s_2, \ldots, s_n)$ des éléments de V telle que, pour tous i, j, on ait $(s_i, s_j) \in E \Rightarrow i < j$. Montrer que si un graphe orienté admet un tri topologique, alors il est nécessairement acyclique.

3. Montrer la réciproque en prouvant que si on effectue un parcours en profondeur d'un DAG et qu'on trie les sommets par ordre décroissant de la valeur de la fonction $post(v)$, on obtient un tri topologique du graphe.

4. Ceci fournit donc un algorithme pour construire le tri topologique d'un DAG. Quelle est sa complexité ?

5. Appliquer l'algorithme sur le graphe suivant.



6. Montrer que si un graphe orienté est tel que tout sommet a au moins un voisin entrant, alors il contient un cycle.

7. On vient donc de montrer qu'un DAG possède nécessairement une source (un sommet de degré entrant 0). En se basant sur cette observation proposer un autre algorithme permettant de construire le tri topologique d'un DAG. Quelle est sa complexité (on suppose que le graphe est représenté par liste d'adjacence) ?

8. On aurait aussi pu montrer que que tout DAG possède nécessairement un puits (degré sortant 0) et en déduire un algorithme pour le tri topologique qui construit l'ordre par la fin. Aurait-on obtenu une meilleure complexité ?

9. Améliorer la complexité de l'algorithme de la question 7 en construisant, puis en maintenant une liste de tous les sommets qui peuvent être la prochaine source choisie. Quelle est la nouvelle complexité ?