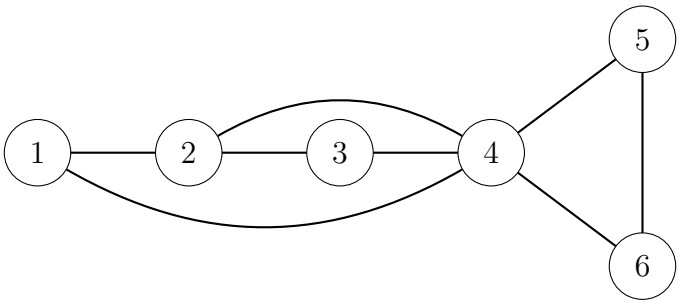
Dans cet exercice nous souhaitons utiliser l'algorithme `PP` du parcours en profondeur pour déterminer les points d'articulation d'un graphe non-orienté. Un *point d'articulation* est un sommet dont la suppression augmente le nombre de composantes connexes du graphe.

1. Déterminer les points d'articulation du graphe G suivant :



Le graphe G possède un seul point d'articulation qui est le sommet s_4.

2. Modifier `PP(G)` en `NB_CC(G)` afin de compter le nombre de composantes connexes du graphe G. En utilisant `NB_CC(G)` écrire un algorithme calculant les points d'articulation du graphe G. Calculer sa complexité.

Voici maintenant un algorithme pour calculer le nombre de composantes connexes :

```
NB_CC(G)
1  nb_composantes_connexes <- 0
2  pour chaque sommet u de G
3      faire couleur[u] <- BLANC
4           pere[u] <- nil
5  temps <- 0
6  pour chaque sommet u de G
7      faire si couleur[u] = BLANC
8          alors nb_composantes_connexes++
9               Visiter_PP(u)
10 return nb_composantes_connexes
```

Voici l'algorithme qui permet de calculer les points d'articulation d'un graphe :

```
ENSEMBLE_PA(G)
1  ens_pa <- ensemble_vide
2  nb_cc_reference <- NB_CC(G)
3  pour chaque sommet u de G faire
4    si NB_CC(G \ {u}) > nb_cc_reference
5      alors ens_pa <- ens_pa U {u}
6  return ens_pa
```

La complexité de `NB_CC(G)` est la même que celle de `PP(G)` soit $\mathcal{O}(n+m)$. La complexité de `ENSEMBLE_PA(G)` est donc $\mathcal{O}\left(n(n+m)\right)$.

Dans la suite, nous allons mettre en place un algorithme plus efficace de calcul des points d'articulation. Pour cela, on fixe un sommet `r` et on considère l'arborescence de liaison `T(r)` définie par le parcours en profondeur de G à partir de `r`. Les arcs de *liaisons* sont les arcs `(u,v)`

de G où `u`= `pere(v)` c'est-à-dire les arcs de `T(r)`, tandis que les arcs de *retour* sont les arcs `(u,v)` de G qui ne sont pas de liaison et où `v` est un ancêtre de `u` dans `T(r)`. On peut montrer que dans un `PP` sur un graphe non-orienté, les arcs sont soit de liaison soit de retour.

Pour tout sommet `v`, on définit `l[v]` comme la plus petite valeur de `d[u]` où `u` est soit égal à `v`, soit l'extrémité d'un arc retour `(w,u)` avec `w` descendant de `v` (`w`=`v` possible).

En d'autres termes, `l[v]` est la plus petite valeur de `d[u]` atteignable en utilisant au plus un arc de retour.

3. Calculer l'arbre `T(r)` pour G en prennant `r=1`. Distinguer les arcs de retour, et calculer `l[u]` pour tous les sommets `u` de G.
   L'arbre $T(1)$ de G est représenté dans le graphe de la Figure 1 ci-dessous, les arcs de liaisons étant en noir et ceux de retour en gris.
   Les valeurs de d, f et l sont données par le tableau suivant :

v	1	2	3	4	5	6
$d[v]$	1	2	3	4	5	6
$f[v]$	12	11	10	9	8	7
$l[v]$	1	1	1	1	4	4

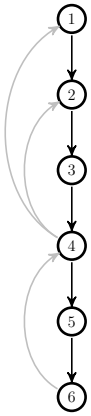


FIGURE 1 – PP(G)

4. Modifier l'algorithme `PP` pour calculer `l[v]` pour tout sommet `v` d'un graphe G.
   Propriétés de `l[]` : on remarque que `l[u]` est le minimum des trois valeurs suivantes : `d[u]`, le minimum du `l[]` de ses fils, `d[v]` si u possède un arc de retour (u, v).

   On rappelle qu'au cours de l'exécution de l'algorithme, un arc de retour (u, v) est arc vers un voisin GRIS v qui n'est pas le père de u.

   Pour calculer la table `l[]`, on conserve `PP(G)` mais on modifie comme suit `Visiter_PP(u)`.

```
Visiter_PP(u)
  1  couleur[u] <- GRIS
* 2  l[u] <- d[u] <- temps <- temps + 1
  3  pour chaque v de Adj[u] faire
  4        si couleur[v] = BLANC alors
  5              pere[v] <- u
  6              Visiter_PP(v)
* 7              si l[v] < l[u] alors
* 8                    l[u] <- l[v]
* 9        sinon si d[v] < l[u] et pere[u] <> v alors // arc retour
*10                    l[u] <- d[v]
 11  couleur[u] <- NOIR
 12  f[u] <- temps <- temps+1
```

5. Établir que `v` est un point d'articulation si et seulement si :
   (a) soit `v` $=$ `r` et `v` a au moins deux fils dans `T(r)`,
   (b) soit `v` $\neq$ `r` et `v` a au moins un fils `w` dans `T(r)` tel que `l[w]` $\geq$ `d[v]`.

   Preuve du théorème de la Question 5 :

Cas 1 : `v` $=$ `r` est un p.a. $\Leftrightarrow$ `v` a au moins deux fils dans `T(r)`,

Cas 1.1 : `v` $=$ `r` est un p.a. $\Rightarrow$ `v` a au moins deux fils dans `T(r)`
Si v est un p.a. alors $G \setminus \{v\}$ possède au moins deux composantes. Il est clair que tout arbre couvrant G enraciné en r possède deux fils au moins, en particulier $T(r)$.

Cas 1.2 : `v` $=$ `r` a au moins deux fils dans `T(r)` $\Rightarrow$ `v` est un p.a.
Si r a deux fils, disons f_1 et f_2, et r n'est pas un p.a., alors il existe une arête (x, y) de $E[G] \setminus E[T(r)]$ avec x descendant de f_1 et y descendant de f_2. Une telle arête ne peut exister, puisque si $d[f_1] < d[f_2]$, alors y aurait été visité par x (qui aurait été BLANC puisque visité après f_2). Alors y aurait été un descendant de f_1 : contradiction. Donc r est un p.a.

Cas 2 : `v` $\neq$ `r` est un p.a. $\Leftrightarrow$ `v` a au moins un fils `w` dans `T(r)` tel que `l[w]` $\geq$ `d[v]`

Cas 2.1 : `v` $\neq$ `r` est un p.a. $\Rightarrow$ `v` a au moins un fils `w` dans `T(r)` tel que `l[w]` $\geq$ `d[v]`
Si v est un p.a. alors $G \setminus \{v\}$ possède au moins deux composantes, disons G_1 et G_2. Supposons $r \in G_1$. Tous les sommets u de G_2 ont une date de visite $d[u] > d[v]$. Sans perte de généralité, on suppose que la composante G_2 est choisie de sorte que les sommets sont visités juste après ceux de G_1. Soit $w \in G_2$ le premier sommet visité après v, donc avec $d[w] = d[v] + 1$. Notons que w est un fils de v. Si $l[w] < d[v]$ alors il existe un arc de retour entre un sommet de G_2 (descendant d'un des fils de v dans G_2) et un sommet visité avant v (donc dans G_1) : contradiction car v est un p.a., donc il n'y a pas de tel arc de retour. Donc $l[w] \geq d[v]$.

Cas 2.2 : `v` $\neq$ `r` a au moins un fils `w` dans `T(r)` tel que `l[w]` $\geq$ `d[v]` $\Rightarrow$ `v` est un p.a.
Supposons que v a un fils w avec $l[w] \geq d[v]$. Soit G_w le sous-graphe de G induit par w et ses descendants dans $T(r)$, et soit $G_0 = G \setminus (G_w \cup \{v\})$. On observe que $w \in G_w$ et que `pere`$[v] \in G_0$ donc G_w et G_0 sont deux sous-graphes non vides. Comme $l[w] \geq d[v]$, il n'y a pas d'arc retour (et donc d'arête de G) entre G_w et G_0. Donc v est un p.a.

6. Modifier l'algorithme `PP` pour calculer, pour tout sommet `v` d'un graphe, `pa[v]` qui vaut `vrai` si et seulement si `v` est un point d'articulation. Comparer la complexité de cet algorithme avec celle de l'algorithme de la question 2.

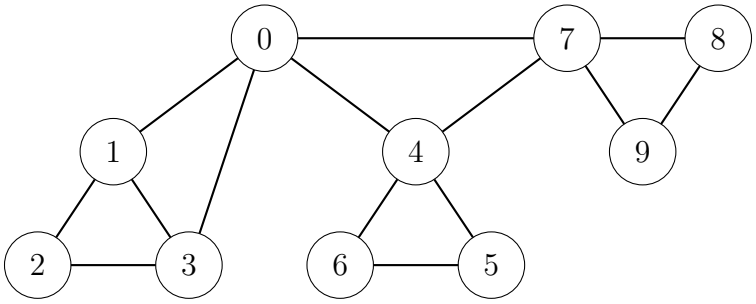
   Pour calculer la table `pa[]` on modifie `PP(G)` pour initialiser toutes les valeurs de `pa[]` à faux, en rajoutant en 3' (derrière pere[u] <- nil) pa[u] <- faux. Puis on modifie `Visiter_PP` de la manière suivante :

```
Visiter_PP(u)
 1  couleur[u] <- GRIS
 2  l[u] <- d[u] <- temps <- temps + 1
 3  pour tout v de Adj[u] faire
 4      si couleur[v] = BLANC alors
 5      |   pere[v] <- u
 6      |   si pere[u] = nil et temps > d[u] alors   //condition 1
 7      |       pa[u] <- vrai
 8      |   Visiter_PP(v)
 9      |   si l[v] < l[u] alors
10      |   |    l[u] <- l[v]
11      |   sinon si l[v] >= d[u] et pere[u] <> nil alors  //condition 2
12      |       pa[u] <- vrai
13      sinon si couleur[v] = Gris et v <> pere[u] et d[v] < l[u] alors
14              l[u] <- d[v]
15  couleur[u] <- Noir
16  f[u] <- temps <- temps+1
```

   La complexité de cet algorithme est la même que celle de `PP(G)` c'est-à-dire $\mathcal{O}(n+m)$, ce qui est meilleur que $\mathcal{O}(n(n+m))$.

7. Tester l'algorithme modifié sur le graphe G ci-dessus et sur le graphe obtenu en ajoutant à G l'arête `{6,3}`.
   Sur le graphe G, on obtient bien `pa[4] = true` car `l[5] = 4 = d[4]`.
   Si on rajoute l'arête $(6,3)$, les valeurs de `l[5]` et `l[6]` changent pour passer à 3, ce qui est strictement inférieur à `d[4]` et donc il n'y a plus de point d'articulation.

8. Tester l'algorithme sur le graphe H ci-dessous, en partant du sommet `r=0` et en suivant l'ordre croissant des sommets.



L'arbre du parcours en profondeur est donné par la Figure 2 et les valeurs de `d[]`, `f[]`, `l[]` et `pa[]` par le tableau ci-dessous.

v	0	1	2	3	4	5	6	7	8	9
$d[v]$	1	2	3	4	8	9	10	13	14	15
$f[v]$	20	7	6	5	19	12	11	18	17	16
l	1	1	1	1	1	8	8	1	13	13
pa	$true$	$false$	$false$	$false$	$true$	$false$	$false$	$true$	$false$	$false$

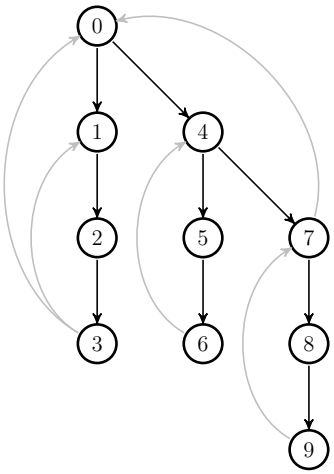


FIGURE 2 – PP(H)