Exercice 1

Un graphe non orienté est *biparti* si l'on peut partager l'ensemble de ses sommets en deux sous-ensembles tels qu'aucune arête du graphe relie deux sommets appartenant au même sous-ensemble.

1. Montrer que si un graphe est biparti, alors il ne peut pas contenir de cycle impair.
2. Soit G un graphe biparti. Soit u et v deux sommets de G reliés par une chaîne. Montrer que u et v sont dans la même partition si et seulement si la longueur du chemin les reliant est paire.
3. Soit G un graphe connexe. Montrer que G contient un cycle impair (un cycle ayant un nombre impair d'arêtes) si et seulement si quelque soit $u \in V(G)$ il existe $v_1, v_2 \in V(G)$ tels que $v_1v_2 \in E(G)$ et $\text{dist}(u, v_1) = \text{dist}(u, v_2)$.
4. En s'appuyant sur les questions précédentes, proposer un algorithme permettant de déterminer si un graphe connexe est biparti.

Exercice 2

```
0   PP(G)
1     pour chaque sommet u de V faire
2         couleur[u] <- BLANC
3         pere[u] <- nil
4     temps <- 0
5     pour chaque sommet u de V faire
6         si couleur[u] = BLANC
7             alors Visiter_PP(u)
```

```
0   Visiter_PP(u)
1     couleur[u] <- GRIS
2     d[u] <- temps <- temps + 1
3     pour chaque v de Adj[u] faire
4         si couleur[v] = BLANC
5             alors pere[v] <- u
6                   Visiter_PP(v)
7     couleur[u] <- NOIR
8     f[u] <- temps <- temps + 1
```

Appliquer l'algorithme de parcours en profondeur `PP` au graphe G_1 de la Fig. 1 (on conviendra que dans les listes d'adjacence les sommets sont rangés dans l'ordre croissant de leur numéro) :

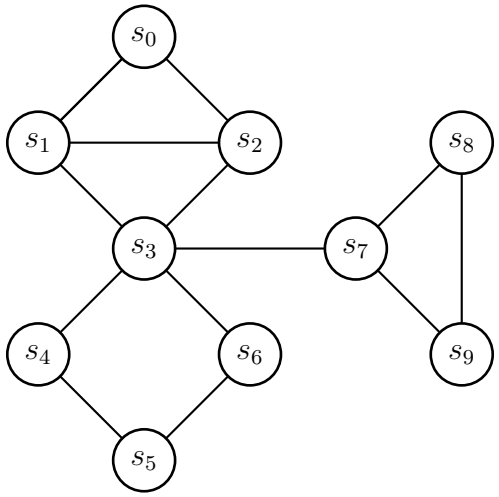


Figure 1 – G_1

Exercice 3

Pour les deux graphes non orientés H_1 et H_2 ci-dessous dire si T_1 et T_2 peuvent respectivement être des arbres couvrants obtenus par un parcours en profondeur − Justifier les réponses en spécifiant l'éventuel sommet de départ et l'ordre des sommets dans les listes d'adjacence.

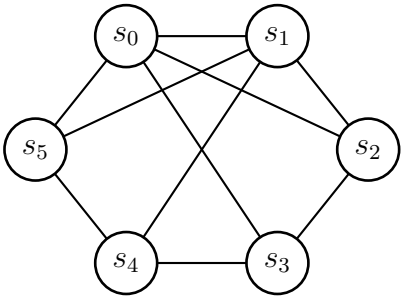


FIGURE 2 – H_1

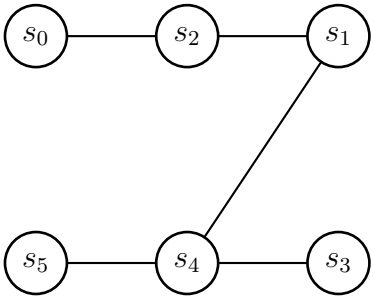


FIGURE 3 – T_1

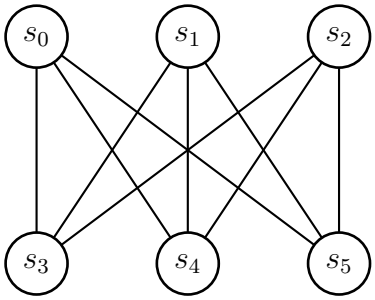


FIGURE 4 – H_2

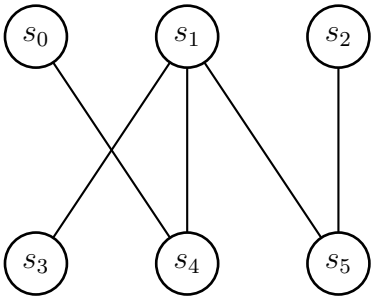
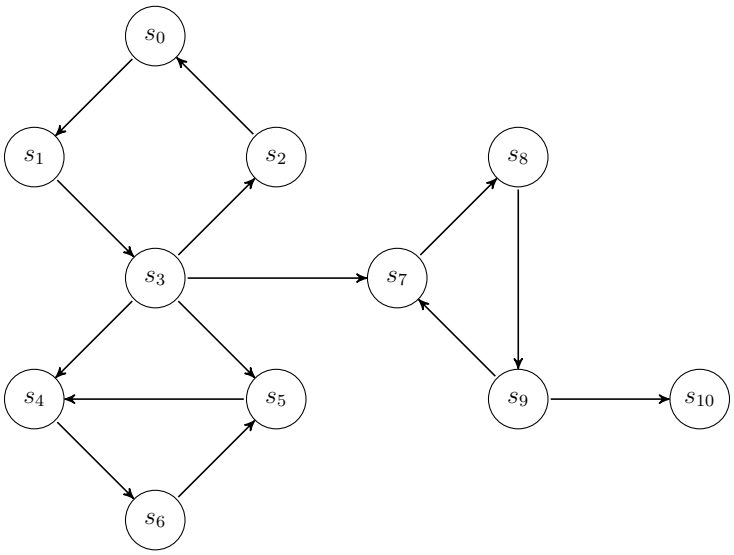


FIGURE 5 – T_2

Exercice 4

1. Appliquer l'algorithme vu en cours pour déterminer les *composantes fortement connexes* du graphe G_1 ci-dessous :



2. Étant donné une représentation par listes d'adjacence, calculer la complexité de l'algorithme du calcul des composantes fortement connexes.

3. En quoi le nombre de composantes fortement connexes d'un graphe peut-il être modifié par l'ajout d'un nouvel arc ?

Rappel :

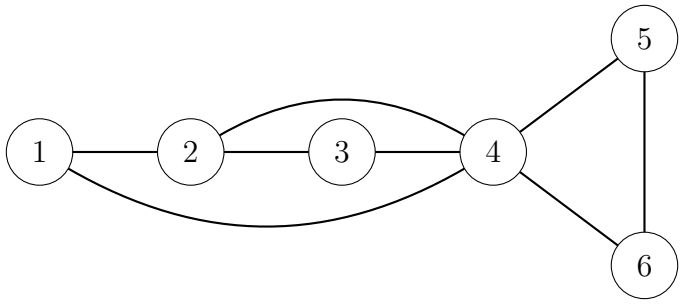
`Composantes_fortement_connexes(`G`)`

1 `PP(`G`)`
2 `calculer` G^{-1}
3 `PP(`G^{-1}`) en considérant les sommets dans l'ordre décroissant des f(u) obtenus par PP(`G`) à l'étape 1`
4 `Afficher les arborescences de la forêt obtenue à l'étape 3`

Exercice 5

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 :



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.

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.

4. Modifier l'algorithme `PP` pour calculer `l[v]` pour tout sommet `v` d'un graphe G.

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]`.

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.

7. Tester l'algorithme modifié sur le graphe G ci-dessus et sur le graphe obtenu en ajoutant à G l'arête `{6,3}`.
8. Tester l'algorithme sur le graphe H ci-dessous, en partant du sommet `r=0` et en suivant l'ordre croissant des sommets.