Connexités
Correction

1 Connexité

Solution 1.1 **(Algernon et le labyrinthe)**

Remarque : le labyrinthe donné ici ne fonctionne pas, il faut y ajouter une porte entre 33 et 24 !

Traduction des questions en termes de graphes :

1. Est-ce qu'il existe une chaîne (des chaînes différentes) entre la source et la sortie ?
2. Est-ce que le graphe est connexe .
3. Est-ce que le graphe est biconnexe ?

Solution 1.2 **(Réseau de routeurs)**

2. Graphe connexe (*connected graph*) :

   (c) **Spécifications :**
   La fonction `connexite (t_graph_stat` G`) : booleen` indique si le graphe G est connexe.

```
algorithme procedure prof_rec
   parametres locaux            /* pas demandé dans le td */
      t_graph_stat    G
      entier          s, no
   parametres globaux
      t_vect_booleens   M
      entier            nbsom

   variables
      entier    i
debut
   nbsom ← nbsom + 1
   M[s] ← vrai
   pour i ← 1 jusqu'a G.ordre faire
      si G.adj[s,i] <> 0 alors
         si non M[i] alors
            prof_rec (G, i, M, nbsom)
         fin si
      fin si
   fin pour
fin algorithme procedure prof_rec
```

```
algorithme fonction connexite : booleen
   parametres locaux
      t_graph_stat    G

   variables
      entier    i
      t_vect_booleens   M
debut
   pour i ← 1 jusqu'a G.ordre faire
      M[i] ← faux
   fin pour
   i ← 0
   prof_rec (G, 1, M, i)
   retourne (i = G.ordre)
fin algorithme fonction connexite
```

Solution 1.3 **(Minimiser les liaisons)**

1. Pour obtenir un réseau où tous les routeurs sont reliés mais en utilisant un minimum de liaisons, il suffit d'éliminer les liaisons inutiles, tout en conservant la connexité : en éliminant les cycles.

2. Un tel graphe est un *arbre*.
   (a) Si on ajoute une arête quelconque à un *arbre*, on ajoute un cycle.
   (b) Si on enlève une arête quelconque à un *arbre*, celui-ci n'est plus connexe.
   (c) Les trois propriétés d'un graphe qui est un *arbre* :
   — Connexe
   — Sans cycle
   — $N - 1$ arêtes si N sommets
   Rappel : Un *cycle* est une chaîne $(s_0, s_1, ...s_\lambda)$ dont les λ arêtes sont toutes distinctes deux à deux et tel que les 2 sommets aux extrémités de la chaîne coïncident.
   Un cycle $(s_0, s_1, ...s_\lambda)$ est élémentaire s'il ne contient pas plusieurs fois le même sommet.

3. Algorithme :
   (a) Pour déterminer si un graphe est un *arbre*, il suffit de vérifier deux des trois propriétés ci-dessus.
   (b) Donc un graphe à N sommets est un *arbre* si :
   — Il est connexe sans cycle ;
   — il est connexe avec $N - 1$ arêtes ;
   — il est sans cycle avec $N - 1$ arêtes.

   Pour vérifier chacune des propriétés :
   — La connexité, il suffit de compter le nombre de sommets (voir exercice 1.2).
   — Pour le nombre d'arêtes, il suffit d'ajouter un compteur incrémenté à chaque successeur (attention, chaque arête est rencontrée deux fois lors du parcours).
   — La présence d'un cycle dans un graphe fait apparaître lors du parcours en profondeur au moins un arc retour (voir l'exercice suivant).

(c) Pour vérifier si un graphe est un *arbre*, nous allons utiliser la définition "connexe sans cycle", en utilisant donc un parcours profondeur.

L'algorithme récursif :

La fonction `test_rec (t_listsom` *ps*`, entier` *pere*`, t_vect_booleens` *M*`, entier` *nb_som*`)` réalise le parcours en profondeur à partir du sommet pointé par *ps*, *pere* étant son père dans le parcours. Le tableau *M* sert de marque, *nb_som* permet de compter les sommets rencontrés. Elle retourne le booléen *faux* si le graphe n'est pas un arbre.

```
algorithme fonction test_rec : booleen
   parametres locaux
      t_listsom   ps
      entier    pere
   parametres globaux
      t_vect_booleens M
      entier nb_som

   variables
      t_listadj   pa
debut
   M[ps↑.som] ← vrai
   nb_som ← nb_som + 1
   pa ← ps↑.succ
   tant que pa <> NUL faire
      si non M[pa↑.vsom↑.som] alors
         si non test_rec(pa↑.vsom, ps↑.som, M, nb_som) alors
            retourne faux
         fin si
      sinon
         si pa↑.vsom↑.som <> pere alors
            retourne faux          /* arc retour */
         fin si
      fin si
      pa ← pa↑.suiv
   fin tant que
   retourne vrai
fin algorithme fonction test_rec
```

L'algorithme d'appel :

La fonction `est_arbre (t_graph_dyn,` *G*`)` détermine si le graphe *G* est un arbre. Elle utilise la fonction `test_rec` (voir page précédente).

```
algorithme fonction est_arbre : booleen
   parametres locaux
      t_graph_dyn G

   variables
      entier i
      t_vect_booleens M
debut
   pour i ← 1 jusqu'a G.ordre faire
      M[i] ← faux
   fin pour
   i ← 0
   retourne (test_rec (G.lsom, -1, M, i) et (i = G.ordre))
fin algorithme fonction est_arbre
```

Voir en ligne pour une version détaillée (dans les deux représentations).

Solution 1.4 **(I want to be a tree –** ***Contrôle 2012*****)**

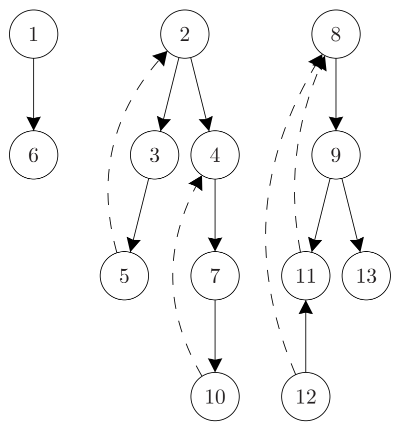

FIGURE 1 – Parcours profondeur du graphe "Not a tree yet"

1. Lors du parcours profondeur du graphe :
   (a) Les arcs retours sont inutiles pour la connexité.
   (b) L'arc (x, y) est un arc retour si y est un successeur de x déjà marqué tel que y a été rencontré avant x et y n'est pas le père de x.
   (c) *La liste des arêtes du graphe "Not a tree yet" supprimées :* 2–5, 4–10, 8–11 et 8–12.

2. Une fois le parcours terminé, si on a attribué à chaque sommet un numéro de composante connexe :
   (a) S'il y a k composantes connexes, il suffit d'ajouter $k - 1$ arêtes.
   (b) Pour rendre le graphe connexe, on peut par exemple ajouter une arête du premier sommet choisi pour le parcours vers tous les sommets racines des autres arbres : les sommets sur lesquels on relance le parcours dans l'algo d'appel parce que non marqués.
   (c) *Le tableau des composantes connexes du graphe "Not a tree yet"* :

	1	2	3	4	5	6	7	8	9	10	11	12	13
cc	1	*2*	*2*	*2*	*2*	1	*2*	**3**	**3**	*2*	**3**	**3**	**3**

3. **L'algorithme :**
   Pendant le parcours profondeur :
   — Construire le vecteur des composantes connexes : on marque les sommets avec le numéro de composante, qui augmente à chaque appel du parcours récursif depuis l'algo d'appel.
   — Ajouter au graphe les arêtes qui permettent de le rendre connexe (dans l'algo d'appel) : après le premier appel sur le sommet 1, on ajoute l'arête $(1, y)$ pour chaque sommet non marqué (sur lequel on relance le parcours).
   — Supprimer du graphe les arêtes "inutiles" : lors du parcours depuis x, dès que l'on rencontre un successeur y déjà marqué tel que y n'est pas le père de x.

L'algorithme d'appel :

La procédure `make_me_tree (t_graph_stat` G, `t_vect_entiers` cc) transforme G en arbre et remplit cc, vecteur des composantes connexes du graphe de départ.

L'algorithme récursif :

La procédure `prof_rec (entier` s, $pere$, no_cc, `t_graph_stat` G, `t_vect_entiers` cc) effectue le parcours profondeur du graphe G à partir du sommet s. $pere$ est le sommet père de s dans la forêt couvrante, no_cc est le numéro de la composante connexe actuelle et cc, qui sert de marque, est le vecteur de composantes.

```
algorithme procedure prof_rec
    parametres locaux
        entier      s, pere
        entier      no_cc
    parametres globaux
        t_graph_stat     G
        t_vect_entiers cc

    variables
        entier    s_adj
debut
   cc[s] ← no_cc
   pour s_adj ← 1 jusqu'a G.ordre faire
      si G.adj[s,s_adj] <> 0 alors
         si cc[s] = 0 alors
            prof_rec (s_adj, s, no_cc, G, cc)
         sinon
            si s_adj <> pere alors
               G.adj[s,s_adj] ← 0
            fin si
         fin si
      fin si
   fin pour
fin algorithme prof_rec
                                        /* algo d'appel */
 algorithme procedure make_me_tree
    parametres globaux
        t_graph_stat    G

    variables
        t_vect_entiers    cc
        entier            no_cc, x, y
 debut
    pour x ← 1 jusqu'a G.ordre faire
        cc[x] ← 0
    fin pour
    no_cc ← 1
    prof_rec(1, -1, no_cc, G, cc)
    x ← 1
    pour y ← 2 jusqu'a G.ordre faire
        si cc[y] = 0 alors
           no_cc ← no_cc + 1
           prof_rec (y, -1, no_cc, G, cc)
           G.adj[x,y] ← 1
           G.adj[y,x] ← 1
           x ← y     /* optionnel ! */
        fin si
    fin pour
 fin algorithme procedure make_me_tree
```

2 Forte connexité

Solution 2.2 **(Circulation à sens unique)**

2. **Méthode 1 :** deux parcours

   (a) Une composante fortement connexe peut contenir plusieurs circuits. Si tous les arcs du graphe sont retournés, les circuits demeurent les mêmes donc le graphe inverse de G contient les même composantes fortement connexes que G.

   (b) **Principe avec 2 parcours :** il faut déterminer l'ordre suffixe des sommets lors d'un parcours en profondeur (premier parcours) ; puis, effectuer un parcours en profondeur sur le graphe transposé (parcourir les liens prédécesseurs) en choisissant les sommets en ordre suffixe inverse (du premier parcours). Les arborescences obtenues sont les composantes fortement connexes.

   (c) Voir TD.

   (d) **Spécifications :** La fonction `comp_fortes (t_graph_dyn` G`, t_vect_entiers` cfc`) : entier` retourne le nombre de composantes fortement connexes du graphe non orienté G. Pour chaque sommet s, $cfc[s]$ indique le numéro de la composante à laquelle il appartient.

```
algorithme procedure calcul_os
  parametres locaux
    t_listsom          ps
  parametres globaux
    t_vect_entiers     M
    t_pile             p
  variables
    t_listadj          pa
debut
  M[ps↑.som] ← -1
  pa ← ps↑.succ
  tant que pa <> NUL faire
    si M[pa↑.vsom↑.som]=0 alors
      calcul_os (pa↑.vsom, M, p)
    fin si
    pa ← pa↑.suiv
  fin tant que
  p ← empiler (ps, p)
fin algorithme procedure calcul_os
```

```
algorithme procedure prof_inverse
  parametres locaux
    t_listsom          ps
  parametres globaux
    t_vect_entiers     cfc
    entier             no
  variables
    t_listadj          pa
debut
  cfc[ps↑.som] ← no
  pa ← ps↑.pred
  tant que pa <> NUL faire
    si cfc[pa↑.vsom↑.som] = -1 alors
      prof_inverse (pa↑.vsom, cfc, no)
    fin si
    pa ← pa↑.suiv
  fin tant que
fin algorithme prof_inverse
```

```
algorithme fonction comp_fortes : entier
  parametres locaux
    t_graph_dyn         G
  parametres globaux
    t_vect_entiers      cfc
  variables
    t_pile              p
    entier              i
    t_listsom           ps
debut
  pour i ← 1 jusqu'a G.ordre faire
    cfc[i] ← 0
  fin pour
  p ← pile_vide ()
  ps ← G.lsom
  tant que ps <> NUL faire
    si cfc[ps↑.som] = 0 alors
      calcul_os (ps, cfc, p)
    fin si
    ps ← ps↑.suiv
  fin tant que
  i ← 0
  tant que non est-vide (p) faire
    ps ← sommet (p)
    p ← depiler (p)
    si cfc[ps↑.som] = -1 alors
      i ← i+1
      prof_inverse(ps, cfc, i)
    fin si
  fin tant que
  retourne (i)
fin algorithme fonction comp_fortes
```

3. **Méthode 2 :** Tarjan, le vrai...
   (a) **Principe de Tarjan :** (voir en ligne pour plus de détails)
   On utilise un **parcours profondeur**, dans lequel on numérote les sommets en ordre préfixe de rencontre ($op[x]$ qui sert ausi de marque). Afin de récupérer plus tard les sommet, ceux-ci sont également empilés en préfixe.
   Pour chaque sommet x on calcule $retour[x] = min\{op[x], retour[y], op[z]\}$ pour tout
   — (x, y) arc couvrant
   — (x, z) arc retour, ou arc croisé si la racine de la composante à laquelle appartient z est un ancêtre de x.
   En suffixe, on vérifie si la valeur de retour du sommet courant x est toujours identique à sa valeur d'ordre préfixe. Si c'est le cas, le sommet x est alors une racine de composante. Il suffit de dépiler tous les sommets jusqu'à retrouver x pour constituer la composante fortement connexe.
   (c) **Spécifications :** La fonction `Tarjan (t_graph_dyn` G`, t_vect_entiers` cfc`) : entier` retourne le nombre de composantes fortement connexes du graphe non orienté G. Pour chaque sommet s, $cfc[s]$ indique le numéro de la composante à laquelle il appartient.

```
algorithme fonction zane : entier
   parametres locaux
      t_listsom      ps
   parametres globaux
      t_vect_entiers op, cfc
      entier         cpt, ncfc
      t_pile         p

   variables
      t_listadj   pa
      entier      x, y
      entier      retour_x, retour_y
debut
   x ← ps↑.som
   cpt ← cpt + 1
   op[x] ← cpt
   retour_x ← op[x]
   p ← empiler (x, p)

   pa ← ps↑.succ
   tant que pa <> NUL faire
      y ← pa↑.vsom↑.som
      si op[y] = 0 alors
         retour_x ← min (retour_x, zane (pa↑.vsom, op, cfc, cpt, ncfc, p))
      sinon
         retour_x ← min (retour_x, op[y])
      fin si
      pa ← pa↑.suiv
   fin tant que

   si retour_x = op[x] alors
      ncfc ← ncfc + 1
      faire
         y ← depiler(p)
         cfc[y] ← ncfc
         op[y] ← ∞
      tant que (y <> x)
   fin si

   retourne retour_x
fin algorithme fonction zane
```

```
algorithme fonction Tarjan : entier
   parametres locaux
      t_graph_dyn        g
   parametres globaux
      t_vect_entiers    cfc

   variables
      t_vect_entiers op
      entier         i, ncfc, cpt
      t_listsom      ps
      t_pile         p
debut
   pour i ← 1 jusqu'a g.ordre faire
      op[i] ← 0
   fin pour

   cpt ← 0
   ncfc ← 0
   ps ← g.lsom
   tant que ps <> NUL faire
      si op[ps↑.som] = 0 alors
         i ← zane (ps, op, cfc, cpt, ncfc, p)
      fin si
      ps ← ps↑.suiv
   fin tant que

   retourne ncfc
fin algorithme fonction Tarjan
```

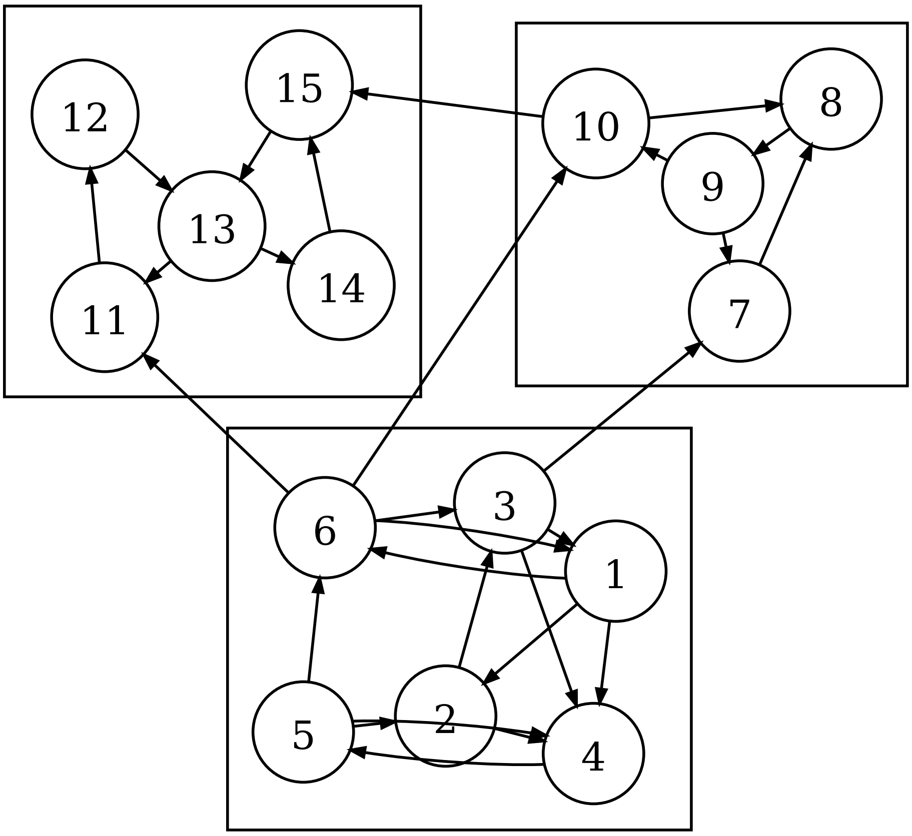

FIGURE 2 – Composantes fortement connexes

Solution 2.3 **(Forte connexité : élimination des inutiles –** ***Contrôle 2014*****)**

1. Les arcs qui peuvent être enlevés sont les arcs avants.

2. Lorsque les sommets sont numérotés en ordre préfixe de rencontre, ce sont les arcs $x \to y$ tels que $pref[x] < pref[y]$ qui ne sont pas des arcs couvrants.

3. *Les arcs à enlever du graphe G_2 :* $7 \to 10, 9 \to 10, 4 \to 10, 2 \to 5$.

4.
```
algorithme procedure prof_rec
    parametres locaux
        entier          s
    parametres globaux
        t_graph_stat     G
        t_vect_entiers  pref
        entier          cpt

    variables
        entier         sadj
debut
    cpt ← cpt + 1
    pref[s] ← cpt
    pour sadj ← 1 jusqu'a G.ordre faire
        si G.adj[s,sadj] <> 0 alors
            si pref[sadj] = 0 alors
                prof_rec (sadj, G, pref, cpt)
            sinon
                si pref[sadj] > pref[s] alors
                    G.adj[s,sadj] ← 0
                fin si
            fin si
        fin si
    fin pour
fin algorithme procedure prof_rec


algorithme procedure enleve_avants
    parametres globaux
        t_graph_stat   G

    variables
        entier   s, cpt
        t_vect_entiers   pref
debut
    pour s ← 1 jusqu'a G.ordre faire
        pref[s] ← 0
    fin pour

    cpt ← 0
    pour s ← 1 jusqu'a G.ordre faire
        si pref[s] = 0 alors
            parc_prof (s, G, pref, cpt)
        fin si
    fin pour
fin algorithme procedure enleve_avants
```

5. Bonus :

Il faut ajouter des arcs entre les racines des arbres (pour former un circuit de racines). Au sein de chaque arbre, on peut ajouter des arcs retours depuis toutes les feuilles vers la racine.

Solution 2.4 **(Graphe réduit)**

Cette technique de décomposition en composantes fortement connexes puis de calcul du graphe réduit permet de classer les états d'un système et de mettre en évidence les états dits *terminaux*, que le système ne peut quitter une fois atteints...

Spécifications :
La procédure `graphe_reduit (t_graph_stat` G, `t_vect_entiers` cfc, `entier` nb_cfc, `t_graph_stat` Gr`)` construit Gr graphe réduit de G à partir du vecteur des composantes fortement connexes de G, cfc, ainsi que leur nombre nb_cfc.

```
algorithme procedure graphe_reduit
    parametres locaux
        t_graph_stat      G
        t_vect_entiers  cfc       /* le vecteur des composantes fortement connexes */
        entier          nb_cfc     /* le nombre de composantes */
    parametres globaux
        t_graph_stat      Gr

    variables
        entier    x, y
  debut
    Gr.orient ← vrai
    Gr.ordre ← nb_cfc
    pour x ←  jusqu'a Gr.ordre faire
        pour y ← Gr.ordre faire
            Gr.adj[x,y] ← 0
        fin pour
    fin pour

    pour x ← 1 jusqu'a G.ordre faire
        pour y ← 1 jusqu'a G.ordre faire
            si G.adj[x,y] <> 0 alors
                si cfc[x] <> cfc [y] alors
                    Gr.adj[cfc[x], cfc[y]] ← 1
                fin si
            fin si
        fin pour
    fin pour
  fin algorithme procedure graphe_reduit
```