Graphes (Graphs)
Représentations et parcours
Correction

1 Représentations

Solution 1.1 **(Représentation statique)**

1. La représentation statique d'un graphe est la représentation par **matrice d'adjacence**.

2. Dans cette représentation, ce sont les arcs qui sont donnés dans une matrice carré : les lignes et les colonnes représentant les sommets (leur numéro), à chaque case i, j se trouve le nombre d'arcs ou d'arêtes entre i et j.

5. Afin de pouvoir représenter à la fois les graphes à liaisons multiples et les graphes simples ou 1-graphes éventuellement valués, nous utilisons deux matrices distinctes : une d'entiers pour les liaisons, une de réels pour les coûts.
   Le graphe sera donc représenté par 4 informations : les deux matrices, l'ordre du graphe et un booléen indiquant le caractère orienté du graphe.

```
constantes
   Max = 100
types
   t_mat_adj    = Max × Max  entier
   t_mat_weight = Max × Max  reel
   t_graph_stat = enregistrement
       booleen      orient
       entier      order
       t_mat_adj    adj
       t_mat_weight cout
   fin enregistrement t_graph_stat
```

Solution 1.2 **(Représentation dynamique)**

1. L'autre manière de représenter les graphes utilise les **listes d'adjacence** : à chaque sommet est associée la liste de ses successeurs. Le graphe est alors représenté par l'ensemble des sommets sous forme d'une liste.

5. Le graphe est représenté par :
   – l'ordre du graphe : *ordre*
   – *orient* : booléen indiquant s'il est orienté
   – *lsom* : la liste chaînée des sommets

   Chaque sommet est :
   – *som* : son "numéro"
   – *succ* : la liste chaînée de ses successeurs
   – *pred* : la liste chaînée de ses prédécesseurs (à NUL si le graphe est non orienté)

   Un élément de la liste d'adjacence (successeurs ou prédécesseurs) sera :
   – *vsom* : un pointeur vers le sommet adjacent dans la liste de sommets du graphe
   – *cout* : le coût de la liaison
   – *nbliens* : le nombre de liaisons

Dans chaque liste chaînée, le champ *suiv* représente le lien vers l'élément suivant.

```
types
    t_listsom = ↑ s_som        /* liste des sommets */

    t_listadj = ↑ s_ladj       /* liste d'adjacence */

    s_som      = enregistrement        /* un sommet */
       entier       som
       t_listadj    succ
       t_listadj    pred
       t_listsom    suiv
    fin enregistrement s_som

    s_ladj      = enregistrement       /* un successeur (ou prédécesseur) */
       t_listsom    vsom
       entier       nbliens
       reel         cout
       t_listadj    suiv
    fin enregistrement s_ladj

    t_graphe_d  = enregistrement       /* le graphe */
       entier       ordre
       booleen      orient
       t_listsom    lsom
    fin enregistrement t_graphe_d
```

2. **Spécifications :**
   La fonction `recherche (entier` s`, t_graphe_d` G`)` retourne le pointeur vers le sommet numéro s dans le graphe G. $1 \leq s \leq ordre(G)$: le sommet existe forcément.

```
algorithme fonction recherche : t_listsom
    parametres locaux
        entier         s
        t_graphe_d     G

    variables
        t_listsom      ps
debut
    ps ← G.lsom
    tant que ps↑.som <> s faire
        ps ← ps↑.suiv
    fin tant que
    retourne (ps)
fin algorithme fonction recherche
```

2 Parcours

Solution 2.1 **(Parcours en largeur)**

4. **Les algorithmes de parcours en largeur :**

Pour ces deux algorithmes, nous supposons que les routines sur les files sont implémentées. Nous utiliserons de plus le type suivant :

```
constantes
    Max = ...
types
    t_vect_entiers  = Max entier
```

Représentation statique :

Spécifications :

La procédure `largeur_stat (t_graph_stat` g, `entier` s, `t_vect_entiers` $pere$) effectue le parcours en largeur du graphe g à partir du sommet s. Le vecteur $pere$ contient la forêt couvrante associée (toutes les cases sont à 0 pour les sommets non encore visités).

Remarques :

Dans cette procédure, le parcours ne se fait que sur les descendants de s. Le parcours complet sera effectué par la procédure `parcours_largeur_stat`.

```
algorithme procedure largeur_stat
    parametres locaux
        t_graph_stat        g
        entier              s
    parametres globaux
        t_vect_entiers      pere      /* sert aussi de marque */

    variables
        t_file    f            /* Les éléments de la file sont ici des entiers */
        entier    i
debut
    pere[s] ← -1
    f ← file_vide ()
    f ← enfiler (s, f)
    faire
        s ← defiler (f)
        pour i ← 1 jusqu'a g.order faire
            si (g.adj[s,i] <> 0) et (pere[i] = 0) alors
                pere[i] ← s
                f ← enfiler (i, f)
            fin si
        fin pour
    tant que non est_vide (f)
fin algorithme procedure largeur_stat
```

Spécifications :

La procédure `parcours_largeur_stat (t_graph_stat` g, `entier` s, `t_vect_entiers` $pere$) effectue le parcours en largeur **complet** du graphe g à partir du sommet s. Le vecteur $pere$ contient la forêt couvrante associée.

```
algorithme procedure parcours_largeur_stat
    parametres locaux
        t_graph_stat        g
        entier              s
    parametres globaux
        t_vect_entiers      pere

    variables
        entier      i
debut
    pour i ← 1 jusqu'a g.order faire
        pere[i] ← 0
    fin pour
    largeur_stat (g, s, pere)

    pour s ← 1 jusqu'a g.order faire
        si pere[s] = 0 alors
            largeur_stat (g, s, pere)
        fin si
    fin pour
fin algorithme procedure parcours_largeur_stat
```

Représentation dynamique (voir l'exercice biparti, partiel janvier 2012).

Solution 2.2 **(Parcours en profondeur)**

4. Conditions pour classer les arcs lors du parcours d'un **graphe non orienté**, $\forall(i,j) \in A$:

couvrants	$i = pere[j]$
en arrière	$j \neq pere[i]$ et i est un descendant de j.

Le parcours d'un graphe orienté :

On numérote les sommets en ordre préfixe (op), en ordre suffixe (os), avec un seul et unique compteur !

Conditions pour classer les arcs lors du parcours d'un **graphe orienté**, $\forall(i,j) \in A$:

couvrants	$i = pere[j]$
en avant	$op[i] < op[j] < os[j] < os[i]$ et $i \neq pere[j]$
retours	$op[j] < op[i] < os[i] < os[j]$
croisés	$op[j] < os[j] < op[i] < op[j]$

5. **Les algorithmes de parcours en profondeur :**

Remarque : Lorsque c'est le vecteur pere qui sert de marque comme dans le premier algorithme, les sommets sont marqués avant de lancer le parcours récursif (juste avant l'appel). Les sommets peuvent aussi être marqués au début du parcours récursif (voir le deuxième algorithme).

(a) *Le graphe est non orienté et représenté par une matrice d'adjacence.*

Spécifications :

La procédure `prof_rec (t_graph_stat` *g*, `entier` *s*, `t_vect_entiers` *pere*) effectue le parcours en profondeur du graphe non orienté *g* à partir du sommet *s*. Le vecteur *pere* contient la forêt couvrante associée (toutes les cases sont à 0 pour les sommets non encore visités).

Remarques :

Dans cette procédure, le parcours ne se fait que sur les descendants de *s*. Le parcours complet sera effectué par la procédure `parcours_profondeur`.

```
algorithme procedure prof_rec
    parametres locaux
        t_graph_stat       g
        entier             s
    parametres globaux
        t_vect_entiers     pere      /* sert aussi de marque */

    variables
        entier    i
debut
    pour i ← 1 jusqu'a g.order faire
        si g.adj[s,i] <> 0 alors
            si pere[i] = 0 alors
                pere[i] ← s              /* arc (s,i) couvrant */
                prof_rec (g, i, pere)
            sinon
                si i <> pere[s] alors
                    /* arc (s,i) retour sauf si arc (i,s) retour ! */
                fin si
            fin si
        fin si
    fin pour
fin algorithme procedure prof_rec
```

Spécifications :

La procédure `parcours_profondeur (t_graph_stat` *g*, `t_vect_entiers` *pere*) effectue le parcours en profondeur **complet** du graphe non orienté *g*. Le vecteur *pere* contient la forêt couvrante associée.

```
algorithme procedure parcours_profondeur
    parametres locaux
        t_graph_stat       g
    parametres globaux
        t_vect_entiers     pere

    variables
        entier    i
debut
    pour i ← 1 jusqu'a g.order faire
        pere[i] ← 0
    fin pour

    pour i ← 1 jusqu'a g.order faire
        si pere[i] = 0 alors
            pere[i] ← -1
            prof_rec (g, i, pere)
        fin si
    fin pour
fin algorithme procedure parcours_profondeur
```

(b) *Le graphe est orienté et représenté par listes d'adjacence.*

Spécifications :

La procédure `prof_rec_dyn (t_listsom` *ps*, `t_vect_entiers` *pere*, *op*, *os*, `entier` *cpt*) effectue le parcours en profondeur à partir du sommet *s* pointé par *ps* du graphe orienté contenant ce sommet. Le vecteur *pere* contient la forêt couvrante associée. Les sommets sont numérotés à l'aide du compteur *cpt* lors de leur rencontre en préfixe (dans *op*) et suffixe (dans *os*).

Les vecteurs *op* et *os* contiennent la valeur 0 pour tous les sommets non encore visités.

Remarques :

Dans cette procédure, le parcours ne se fait que sur les descendants de *s*. Le parcours complet sera effectué par la procédure `parcours_profondeur_dyn`.

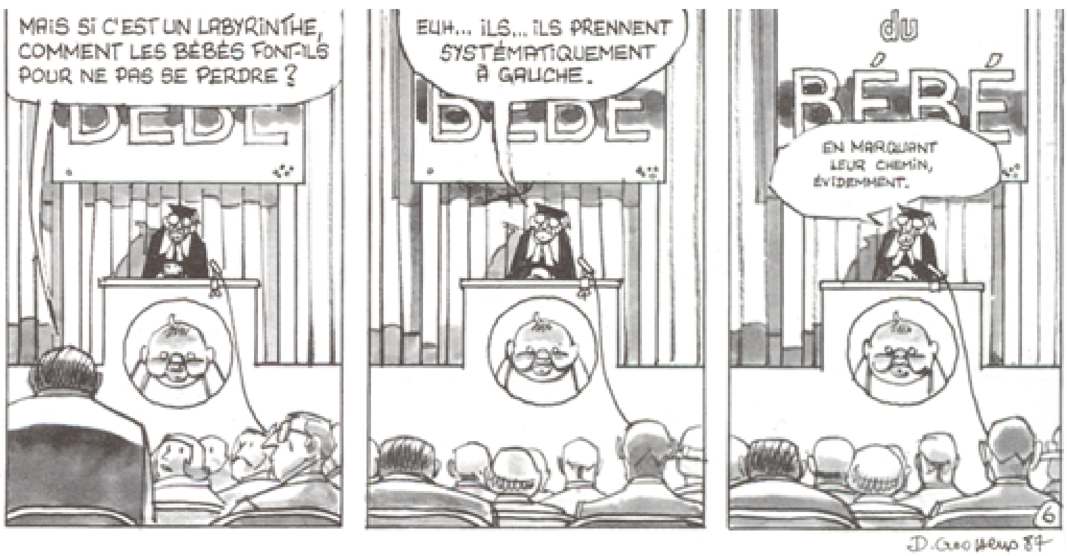
```
algorithme procedure prof_rec_dyn
    parametres locaux
        t_listsom       ps
    parametres globaux
        entier           cpt
        t_vect_entiers   pere, op, os      /* op sert aussi de marque */

    variables
        t_listadj    pa          /* pointeur sur sommet adjacent */
        entier        s, sadj     /* sommet courant, sommet adjacent */
debut
    s ← ps↑.som
    cpt ← cpt+1
    op[s] ← cpt

    pa ← ps↑.succ
    tant que pa <> NUL faire
        sadj ← pa↑.vsom↑.som
        si op[sadj] = 0 alors
            pere[sadj] ← s
                /* arc (s,sadj) couvrant */
            prof_rec_dyn (pa↑.vsom, cpt, pere, op, os)
        sinon
            si op[s] < op[sadj] alors
                /* arc (s,sadj) en avant */
            sinon
                si os[sadj] = 0 alors
                    /* arc (s,sadj) retour */
                sinon
                    /* arc (s,sadj) croisé */
                fin si
            fin si
        fin si

        pa ← pa↑.suiv
    fin tant que

    cpt ← cpt+1
    os[s] ← cpt
fin algorithme procedure prof_rec_dyn
```

Spécifications :

La procédure `parcours_profondeur_dyn (t_graphe_d` *g*`, entier` *s*`, t_vect_entiers` *pere*`)` effectue le parcours en profondeur **complet** du graphe non orienté g à partir du sommet s. Le vecteur *pere* contient la forêt couvrante associée.

```
algorithme procedure parcours_profondeur_dyn
    parametres locaux
        t_graph_stat        g
        entier              s
    parametres globaux
        t_vect_entiers      pere

    variables
        t_vect_entiers     op, os
        entier             cpt, i
        t_listsom          ps
debut

    pour i ← 1 jusqu'a g.order faire
        op[i] ← 0
        os[i] ← 0
    fin pour
    cpt ← 0

    pere[s] ← -1
    ps ← recherche (s, g)
    prof_rec_dyn (ps, cpt, pere, op, os)

    ps ← g.lsom
    tant que ps <> NUL faire
        si op[ps↑.som] = 0 alors
            pere[ps↑.som] ← -1
            prof_rec_dyn (ps, cpt, pere, op, os)
        fin si
        ps ← ps↑.suiv
    fin tant que
fin algorithme procedure parcours_profondeur_dyn
```

2. Principe du parcours en profondeur :



3 Applications

Solution 3.3 **(Compilation, cuisine...)**

1. Par exemple, voici deux ordres possibles d'exécution :
   $5, 4, 8, 7, 1, 3, 2, 6, 9$
   $5, 8, 4, 7, 1, 3, 6, 9, 2$

   Le graphe représentant le problème (figure 1) : les sommets représentent les instructions, chaque arc $(i,\ j)$ indique qu'il faut avoir exécuté l'instruction i pour pouvoir exécuter la j. On parle de graphe de dépendance ou précédence.

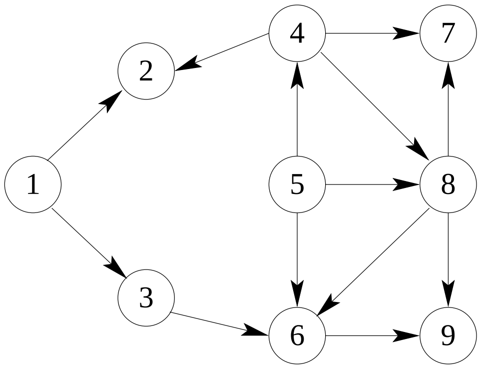


FIGURE 1 – Graphe de dépendance

2. Une solution de tri ne peut exister que s'il n'y a pas de circuit dans le graphe.

3. Montrons que $\forall (u, v) \in A$, $os[u] > os[v]$. Les graphes orientés possèdent 4 types d'arcs (u,v) :

   couvrants et avants : si $op[u] < op[v] < os[v] < os[u]$

   retours : n'existent pas, le graphe est sans circuit

   croisés : si $op[v] < os[v] < op[u] < os[u]$

   La propriété est donc démontrée.

4. **Principe :**
   Lors du parcours en profondeur du graphe, les sommets sont ajoutés à une pile lors de leur rencontre en suffixe. Une fois le parcours de tout le graphe terminé, le contenu de la pile est affiché.

   Spécifications :
   La procédure `tri_rec (t_graph_stat` G`, entier` s`, t_vect_booleens` $marque$`, t_pile` tri`)` effectue le parcours en profondeur à partir du sommet s du graphe orienté G. Le vecteur $marque$ contient $vrai$ pour tous les sommets déjà visités, $faux$ pour les autres. Les sommets sont empilés en ordre suffixe de rencontre dans la pile tri.

   Remarques :
   Dans cette procédure, le parcours ne se fait que sur les descendants de s. Le parcours complet sera effectué par la procedure `tri_topo`.

```
algorithme procedure tri_rec
    parametres locaux
        t_graph_stat      G
        entier            s
    parametres globaux
        t_vect_booleens   marque
        t_pile            tri

    variables
        entier        i
debut
    marque[s] ← vrai
    pour i ← 1 jusqu'a G.order faire
        si (G.adj[s,i] <> 0) et non marque[i] alors
            tri_rec (G, i, marque, tri)
        fin si
    fin pour
    tri ← empiler (s, tri)
fin algorithme procedure tri_rec
```

Spécifications :
La procédure `tri_topo (t_graph_stat` G`)` affiche une solution de tri topologique pour le graphe sans circuit G.

```
algorithme procedure tri_topo
    parametres locaux
        t_graph_stat       G

    variables
        t_pile            tri
        t_vect_booleens   marque
        entier            s
debut
    pour s ← 1 jusqu'a G.order faire
        marque[s] ← faux
    fin pour

    pour s ← 1 jusqu'a G.order faire
        si non marque[s] alors
            tri_rec (G, s, marque, tri)
        fin si
    fin pour

    tant que non est_vide (tri) faire      /* on affiche tous les éléments de la pile */
        ecrire (sommet (tri))
        tri ← depiler (tri)
    fin tant que
fin algorithme procedure tri_topo
```

5. Pour vérifier l'existence d'une solution, il suffit de rechercher les éventuels arcs en arrière (voir **??**) : si un arc en arrière est trouvé alors c'est qu'il existe un circuit dans le graphe, et donc il n'y a pas de solution de tri topologique.