Les Graphes
Représentations et explorations
Un bout de 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_sommets = 100
types
   t_mat_adj  = Max_sommets × Max_sommets entier
   t_mat_cout = Max_sommets × Max_sommets reel

   t_graphe_s = enregistrement
       booleen     orient entier
       ordre t_mat_adj   adj
       t_mat_cout  cout
   fin enregistrement t_graphe_s
```

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
```

Solution 1.3 **Demi-degrés)**

1. Le **degré** d'un sommet x ($d^o(x)$) est le nombre d'arcs ou d'arêtes dont x est une extrémité. Les boucles comptent double !
   Dans un **graphe orienté**, le **demi-degré extérieur** (resp. **intérieur**), noté $d^{o+}(x)$ (resp. $d^{o-}(x)$) est le nombre d'arcs ayant leur extrémité initiale (resp. terminale) en x.

2. Degrés et demi-degrés dans les graphes G_1^* et G_2^* :

		1	2	3	4	5	6	7	8	9
graphe G_1^*	d^{o+}	5	2	2	1	1	4	2	3	0
	d^{o-}	0	3	2	4	1	3	4	0	3
	d^o	5	5	4	5	2	7	6	3	3
graphe G_2^*	d^o	4	5	1	2	3	3	2	6	2

3. *Représentation statique :*

   Spécifications :

   La procedure `demi_degres (t_graphe_s` *g*`), t_vect_entiers` *ddi*, *dde* calcule les demi-degrés intérieurs (*ddi*) et extérieurs (*dde*) de tous les sommets du graphe G orienté.

```
algorithme procedure demi_degres
   parametres locaux
      t_graphe_s     G
   parametres globaux
      t_vect_entiers ddi, dde

   variables
      entier    i, j
debut
   pour i ← 1 jusqu'a G.ordre faire
      ddi[i] ← 0
      dde[i] ← 0
   fin pour

   pour i ← 1 jusqu'a g.ordre faire
      pour j ← 1 jusqu'a G.ordre faire
         dde[i] ← dde[i] + G.adj[i,j]
         ddi[j] ← ddi[j] + G.adj[i,j]    /* ou : ddi[i] ← ddi[i] + G.adj[j,i] */
      fin pour
   fin pour
fin algorithme procedure demi_degres
```

Représentation dynamique :

Spécifications :

La procedure `demi_degres (t_graphe_d` *g*`), t_vect_entiers` *ddi*, *dde* calcule les demi-degrés intérieurs (*ddi*) et extérieurs (*dde*) de tous les sommets du graphe *G* orienté.

```
algorithme procedure demi_degres
   parametres locaux
      t_graphe_d     G
   parametres globaux
      t_vect_entiers ddi, dde

   variables
      entier      s, sadj
      t_listsom   ps
      t_listadj   pa
debut
   pour i ← 1 jusqu'a G.ordre faire
      ddi[i] ← 0
      dde[i] ← 0
   fin pour

   ps ← G.lsom
   tant que ps <> NUL faire
      s ← ps↑.som
      pa ← ps↑.succ
      tant que pa <> NUL faire
         sadj ← pa↑.vsom↑.som
         dde[s] ← dde[s] + pa↑.nbliens
         ddi[sadj] ← ddi[sadj] + pa↑.nbliens
         pa ← pa↑.suiv
      fin tant que
      ps ← ps↑.suiv
   fin tant que
fin algorithme procedure demi_degres
```

2 Parcours

Solution 2.1 **(Parcours en largeur)**

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

Nous utiliserons de plus le type suivant :

```
constantes
    Max_sommets = ...
types
    t_vect_entiers = Max_sommets entier
```

Représentation statique :

Spécifications :
La procédure `largeur_stat (t_graphe_s` 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 et sert de marques (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`.
Les éléments de la file sont ici des entiers.

```
algorithme procedure largeur_stat
    parametres locaux
        t_graphe_s      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.ordre faire
            si G.adj[s,i] <> 0 alors       /* i est un successeur de s */
                si pere[i] = 0 alors
                    pere[i] ← s            /* i est non marqué */
                    f ← enfiler (i, f)
                fin si
            fin si
        fin pour
    tant que non est_vide (f)
fin algorithme procedure largeur_stat
```

Spécifications :

La procédure `parcours_largeur_stat (t_graphe_s` 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 et serrt de marques.

```
algorithme procedure parcours_largeur_stat
    parametres locaux
        t_graphe_s       G
        entier           s
    parametres globaux
        t_vect_entiers   pere

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

    largeur_stat (G, s, pere)

    pour s ← 1 jusqu'a g.ordre 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 3.1)

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 :**

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

Spécifications :

La procédure `prof_rec (t_graphe_s` 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 et sert de marque (toutes les cases sont à 0 pour les sommets non encore visités).

Remarque : Lorsque c'est le vecteur *pere* qui sert de marque comme ici, les sommets sont marqués avant de lancer le parcours récursif (juste avant l'appel). La plupart du temps, les sommets sont marqués au début du parcours récursif (voir le parcours d'un graphe orienté).

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

    variables
        entier   i
debut
    pour i ← 1 jusqu'a G.ordre 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_graphe_s` G, `entier` s, `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_graphe_s       G
        entier           s
    parametres globaux
        t_vect_entiers   pere

    variables
        entier   i
debut
    pour i ← 1 jusqu'a G.ordre faire
        pere[i] ← 0
    fin pour
    pour i ← 1 jusqu'a G.ordre 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`.

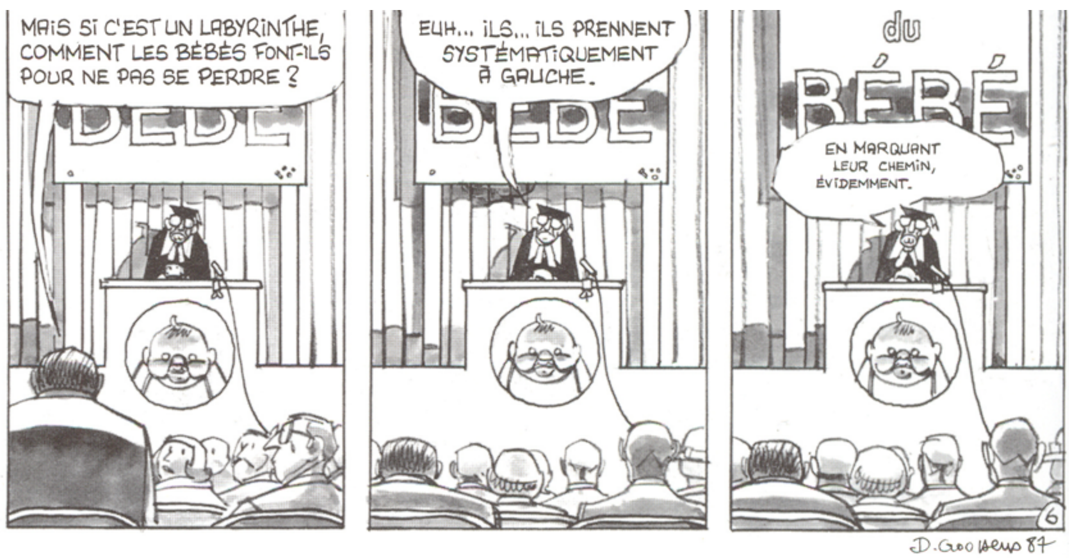
```
algorithme procedure prof_rec_dyn
    parametres locaux
        t_listsom       ps
    parametres globaux
        entier          cpt
        t_vect_entiers  pere, op, os    /* op sert aussi de marque */

    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_graphe_s        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.ordre 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 3.1 **(Graphes bipartis –** ***Partiel décembre 2009*****)**

1. G_3 Le premier graphe n'est pas biparti.
   G_4 Le deuxième est biparti avec $S_1 = \{1, 4, 5, 9\}$ et $S_2 = \{2, 3, 6, 7, 8\}$.
   G_5 Le troisième ne l'est pas, sauf si on enlève la boucle ! Dans ce cas une possibilité est $S_1 = \{1, 3, 6, 8, 9\}$ et $S_2 = \{2, 4, 5, 7\}$.

2. Le parcours est ici fait en largeur.

   Spécifications :
   La fonction `test_partiel (t_listsom` *ps*, `t_vect_entiers` *marque*) retourne un booléen indiquant si le sous-graphe parcouru à partir du sommet pointé par *ps* est biparti.

   Les éléments de la file sont ici des pointeurs de type `t_listsom`.

```
algorithme fonction test_partiel : booleen
    parametres locaux
        t_listsom        ps
    parametres globaux
        t_vect_entiers    marque

    variables
        t_listadj    pa
        entier       s, sadj
        t_file       f          /* la file contient des pointeurs sur sommets */
        booleen      test
debut
    marque [ps↑.som] ← 1
    f ← file_vide()
    f ← enfiler (ps, f)
    test ← vrai
    tant que test et non est_vide (f) faire
        ps ← defiler (f)
        pa ← ps↑.succ
        s ← ps↑.som
        tant que test et (pa <> NUL) faire
            sadj ← pa↑.vsom↑.som

            si marque[sadj] = 0 alors
                marque[sadj] ← - marque[s]
                f ← enfiler (pa↑.vsom, f)
            sinon
                si marque[sadj] = marque[s] alors
                    test ← faux
                fin si
            fin si

            pa ← pa↑.suiv
        fin tant que
    fin tant que

    vide_file (f)
    retourne (test)
fin algorithme fonction test_partiel
```

Spécifications :
La fonction `biparti (t_graphe_d` G`)` retourne un booléen indiquant si le graphe non orienté G est biparti.

```
algorithme fonction biparti : booleen
    parametres locaux
        t_graphe_d        G

    variables
        t_vect_entiers    marque
        entier            i
        t_listsom         ps
debut
    pour i ← 1 jusqu'a G.ordre faire
        marque[i] ← 0
    fin pour

    ps ← G.lsom
    tant que ps <> NUL faire
        si marque[ps↑.som] = 0 alors
            si non test_partiel (ps, marque) alors
                retourne (faux)
            fin si
        fin si
        ps ← ps↑.suiv
    fin tant que

    retourne (vrai)
fin algorithme fonction biparti
```

Voir le partiel de décembre 2009 pour une version avec parcours profondeur.

Solution 3.2 **(Poids cumulé d'un arbre couvrant –** ***partiel janvier 2011*****)**

Voir correction du partiel en ligne

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

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

3. (a) Montrons que dans un graphe sans circuit $\forall(u, v) \in A$, $os[u] > os[v]$. Lors du parcours des graphes orientés 4 types d'arcs (u,v) :

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

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

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

   La propriété est donc démontrée.

   (b) **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_graphe_s` 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.

```
algorithme procedure tri_rec
    parametres locaux
        t_graphe_s       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.ordre 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_graphe_s` G) affiche une solution de tri topologique pour le graphe sans circuit G.

```
algorithme procedure tri_topo
    parametres locaux
        t_graphe_s        G

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

    pour s ← 1 jusqu'a G.ordre 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
```

(c) Pour vérifier l'existence d'une solution, il suffit de rechercher les éventuels arcs en arrière (voir page 6) : 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.