Algorithmique
Partiel n° 1

Info-Spé – Epita

D.S. 312009.3 BW (4 jan 2011 - 10 :00)

Consignes (à lire) :

- ☐ Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - Aucune réponse au crayon de papier ne sera corrigée.
- ☐ La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- ☐ **Les algorithmes :**
  - Tout algorithme doit être écrit dans le langage Algo (pas de C, Caml ou autre).
  - Tout code Algo non indenté ne sera pas corrigé.
  - Tout ce dont vous avez besoin (types, routines) est indiqué en **annexe** (dernière page) !
- ☐ Durée : 2h00

Exercice 1 (Graphes : Court cours – *4 points*)

1. Comment utiliser/modifier ***le plus simplement possible*** l'algorithme de parcours en profondeur d'un graphe non orienté pour déterminer si ce dernier est connexe ?

2. Peut-on appliquer cette méthode à un graphe orienté pour déterminer si celui-ci est fortement connexe ? Justifier.

3. Si *pref[i]* retourne le Numéro d'ordre préfixe de rencontre du sommet i, dans la forêt couvrante associée au parcours en profondeur d'un graphe orienté G, les arcs $x \rightarrow y$ tels que *pref[y]* est inférieur à *pref[x]* dans la forêt sont appelés ? Comment peut-on les différencier ?

Exercice 2 (Graphes : dessiner c'est gagner – *4 points*)

Soit le graphe `G=<S,A>` orienté avec :

```
   S={1,2,3,4,5,6,7,8,9,10}
et A={(1,2),(1,6),(1,7),(2,3),(2,6),(3,1),(3,5),(4,3),(4,8),(4,9),(4,10),(5,1),(7,6),
       (8,5),(8,10),(10,9)}
```

1. Représenter graphiquement le graphe correspondant à `G`.

2. Donner le tableau `DemiDegréIntérieur` tel que $\forall\ i \in [1, Card(S)]$, `DemiDegréIntérieur[i]` soit égal au demi-degré intérieur de `i` dans `G`.

3. Représenter (dessiner) la forêt couvrante associée au parcours en profondeur du graphe `G`. *Ajouter aussi les autres arcs en les qualifiant à l'aide d'une légende explicite.* On considérera le sommet 1 comme base du parcours, les sommets devant être choisis en ordre numérique croissant.

Exercice 3 (ARN : question d'équilibre – *5 points*)

Rappels :

◇ Un *arbre bicolore* (*arbre rouge-noir*) est un arbre binaire de recherche dont les nœuds portent une information supplémentaire : ils sont rouges ou noirs (ou blancs sur un tableau noir !). C'est une représentation des arbres 2-3-4.

◇ L'arbre est "équilibré" à condition que les propriétés suivantes soient respectées :
  - Un nœud rouge ne peut pas avoir de fils rouge.
  - La couleur de la racine est noire.
  - Le nombre de nœuds noirs sur tous les chemins de la racine aux feuilles est le même (hauteur noire = hauteur arbre 2-3-4).

On désire vérifier si un arbre bicolore est bien équilibré. Il faut donc vérifier que les nœuds rouges et noirs respectent bien les propriétés rappelées ci-dessus.

L'algorithme à écrire ici devra vérifier :

- Qu'il n'y a pas deux nœuds rouges qui se suivent : pour cela la fonction retournera un entier égal à 0 ou 1 selon que la racine de l'arbre parcouru est rouge ou pas.
- Que les hauteurs noires en chaque branche sont identiques : pour cela, la fonction prendra un paramètre global *hauteur* qui contiendra la hauteur de l'arbre parcouru.

Dans la cas où l'arbre n'est pas équilibré, la fonction retournera un entier négatif.

La vérification que la racine est noire sera faite par la fonction d'appel, donnée ci-dessous.

```
algorithme fonction est_arn : booleen
    parametres locaux
        t_arn    A

    variables
        entier    hauteur
debut
    retourne (A = NUL) ou (non A↑.rouge et (test_arn (A, hauteur) <> -1))
fin algorithme fonction est_arn
```

Compléter la fonction récursive `test_arn (t_arn` A`)` (vous pouvez ajouter des variables si cela vous semble nécessaire).

Le type `t_arn` est rappelé en annexe.

Exercice 4 (Poids cumulé d'un arbre couvrant – *7 points*)

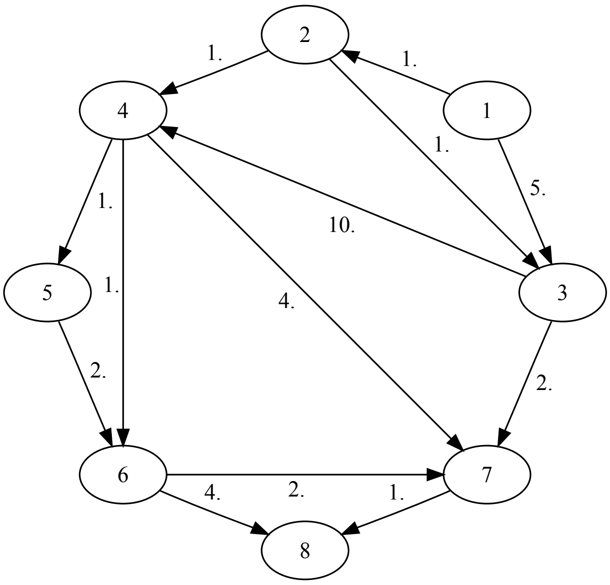

FIGURE 1 – Graphe orienté et valué

Dans cet exercice nous travaillerons avec des graphes **orientés** et **valués** en représentation dynamique.

On définit le poids cumulé d'un sommet dans un arbre couvrant (issu d'un parcours profondeur) comme la somme des poids des *fils* de sommet dans l'arbre plus le coût des arcs couvrant joignant ces sommets, c'est à dire : $p[s] = \sum_i(\texttt{cout}(s, s_i) + p[s_i])$ où s_i est un *fils* du sommet s dans l'arbre couvrant.

On se propose d'écrire un algorithme qui construit l'arbre couvrant du parcours **profondeur** d'un graphe et calcule le poids cumulé des sommets de cet arbre.

Par exemple, pour le graphe de la figure 1 en partant du sommet 2 (en rencontrant les sommets dans l'odre croissant) on obtiendra le vecteur de poids (présenté ici avec le vecteur de pères représentant l'arbre couvrant) suivant :

	1	2	3	4	5	6	7	8
pere	0	-1	2	3	4	5	6	7
poids	∞	17.	16.	6.	5.	3.	1.	0.

Remarque : les feuilles de l'arbre couvrant ont un poids de zéro et les sommets non atteints par le parcours ont un poids de $+\infty$.

Pour la suite, on considère que le principe d'un parcours profondeur est acquis (vous n'avez pas à décrire le principe du parcours, seulement ce qui est spécifique à l'algorithme.)

1. Écrire la fonction `cumul(ps,pere,poids)` qui effectue le parcours profondeur depuis le sommet pointé par `ps` et remplit le vecteur `pere` représentant l'arbre couvrant ainsi que le vecteur (réel) `poids` contenant les poids des sommets atteints par le parcours. La fonction renverra le poids cumulé du sommet `ps`.

2. Écrire la fonction (appel de l'algorithme précédant) `poids_cumul(s,g,pere,poids)` qui lance le parcours en profondeur sur le sommet `s` dans le graphe `g`.

Annexes

Représentation des arbres bicolores

```
types
    /* déclaration du type t_element */
    t_arn = ↑ t_noeud_arn

    t_noeud_arn = enregistrement
        t_element    cle
        booleen      rouge
        t_arn        fg, fd
    fin enregistrement t_noeud_arn
```

Représentation dynamique des graphes

```
types
    t_listsom = ↑ s_som
    t_listadj = ↑ s_ladj
    s_som     = enregistrement
       entier      som
       t_listadj   succ
       t_listadj   pred
       t_listsom   suiv
    fin enregistrement s_som
    s_ladj      = enregistrement
       t_listsom   vsom
       reel        cout
       t_listadj   suiv
    fin enregistrement s_ladj
    t_graphe_d  = enregistrement
       entier     ordre
       booleen    orient
       t_listsom  lsom
    fin enregistrement t_graphe_d
```

Autres types utiles

```
constantes
    Max = /* une valeur suffisante !*/
types
    t_vect_entiers  = Max entier
    t_vect_booleens = Max booleen
    t_vect_reels = Max reel
```

Routines autorisées

Files

Toutes les opérations sur les files peuvent être utilisées à condition de préciser le type des éléments.

- `file_vide (): t_file` : initialise la file
- `est_vide (t_file` f`) : booleen` : indique si f est vide
- `enfiler (t_elt_file` e`, t_file` f`): t_file` : enfile e dans f
- `defiler (t_file` f`): t_elt_file` : défile et retourne le premier élément de f

Autres

Les fonctions `max`, `min`, `abs`, ainsi que les valeurs ∞ et $-\infty$ sont aussi autorisées. De même pour la fonction `recherche (entier` s`, t_graphe_d` G`)` qui retourne le pointeur sur le sommet s dans G (de type `t_listsom`).