.: CONSEILS :.

Application d'un parcours profondeur ou largeur.

Tout d'abord, il faut déterminer ce qui servira de tableau de marques :

- Un simple tableau de booléens, ou alors :
- Si la forêt couvrante doit être récupérée, le tableau des pères (dans ce cas, lors d'un parcours en profondeur, on marque le sommet juste avant d'appeler le parcours récursif dessus).
- Lorsque l'ordre préfixe de rencontre est nécessaire (pour classifier les arcs, ou lorsqu'on cherche à savoir si un sommet peut remonter plus haut dans l'arborescence, voir les points d'articulations, ou les composantes fortement connexes).
- Des cas particuliers : une double marque (-1, 1 par exemple) dans le test de bipartisme, les hauteurs des sommets dans le calcul de la longueur de cycles...

Dans tous les cas, le vecteur en question devra être initialisé dans l'algorithme d'appel.

L'algo d'appel sera différent selon les problèmes. Il faut déterminer :

- Si un sommet de départ particulier est nécessaire pour le premier appel : recherche d'un chemin à partir d'un sommet donné par exemple.
- Si le parcours du graphe doit être complet (il doit être relancé sur tous les sommets non marqués) : recherche des composantes connexes, test d'une propriété sur tout le graphe...

Ensuite, selon les problèmes, le parcours pourra être interrompu ou pas : lorsque la réponse la question est trouvée, inutile de continuer ! Ci-dessous deux exemples d'application du parcours profondeur. pour le parcours largeur, voir le td et les partiels.

.: ARBRES :.

1. Les affirmations équivalentes :
   a. *G est un arbre*
   b. *G est sans cycle avec N-1 arêtes*

   ---

   c. G est connexe et sans cycle
   d. G est connexe avec N-1 arêtes
   e. Deux sommets quelconques de G sont reliés par une chaîne élémentaire unique.
   f. G est sans cycle, mais si une arête quelconque est ajoutée à A, le graphe résultant contient un cycle.
   g. G est connexe mais si on enlève une arête, il ne l'est plus.
2. **Principe** :

   La définition utilisée ici est : ''connexe sans cycle``. On réalise un parcours en profondeur du graphe qui recherche s'il y a des arcs retour : un successeur déjà marqué différent du père. Dans ce cas le parcours est interrompu. De plus pendant le parcours on compte les sommets rencontrés. Le graphe sera un arbre s'il n'admet aucun cycle et si à la suite du parcours tous les sommets ont été rencontrés (il est connexe).

   Spécifications :

   La fonction `est_arbre (t_graphe_s G)` détermine si le graphe *G* est un arbre.
   La fonction `cycle (t graphe s G, entier s, pere, t vect booleens M, entier nb_som)` réalise le parcours en profondeur du graphe *G* à partir du sommet *s*, *pere* étant son père dans le parcours, le tableau*M* sert de marque. *nb_som* permet de compter les sommets rencontrés. La fonction retourne vrai dès qu'un cycle est détecté.

Le parcours en profondeur :

```
algorithme fonction cycle : booleen
   parametres locaux
      t_graphe_s    G
      entier        s, pere
   parametres globaux
      t_vect_booleens    M
      entier             nb_som

   variables
      entier    i
debut
   M[s] <- vrai
   nb_som <- nb_som + 1
   pour i <- 1 jusqu'à G.ordre faire
      si G.adj[s,i] <> 0 alors
         si non M[i] alors
            si cycle (G, i, s, M, nb_som) alors
               retourne (vrai)
            fin si
         sinon
            si i <> pere alors
               retourne (vrai)
            fin si
         fin si
      fin si
   fin pour
   retourne (faux)
fin algorithme fonction cycle
```

La fonction principale : Elle fait les initialisations et lance le parcours en profondeur. Inutile de relancer sur les sommets non marqués : si le graphe n'est pas connexe, ce n'est pas un arbre !

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

   variables
      entier          i
      t_vect_booleen    M
debut
   pour i <- 1 jusqu'à G.ordre faire
      M[i] <- faux
   fin pour
   i <- 0
   retourne (non cycle (G, 1, -1, M, i) et (i = G.ordre))
fin algorithme fonction est_arbre
```

.: LONGUEUR DE CYCLE :.

Le but est ici de calculer la longueur du plus long cycle élémentaire d'un graphe non orienté.

Rappels :
Un cycle est élémentaire s'il ne passe pas plusieurs fois par un même sommet.

Méthode :

- Quel parcours ? Les arcs retours permettent de détecter les cycles, ce sera donc un parcours en profondeur.
- Appel ? Il faut parcourir le graphe en entier : l'algo d'appel devra relancer sur tous les sommets. Il n'y a pas de sommet de départ particulier.
- Comment ? Chaque arc retour représente un cycle élémentaire, il ne reste plus qu'à trouver sa longueur. On marque les sommets avec leur profondeur dans l'arborescence : la longueur d'un cycle repéré par l'arc retour *(x,y)* sera la différence des profondeurs de *x* et de *y* (le nombre d'arcs couvrants entre *y* et *x*), plus un pour l'arc retour.
  Il suffira de garder le maximum des longueur ainsi calculées.
  Rappel : deux emplacements pour marquer un sommet lors du parcours en profondeur :
  - au début du parcours à partir de ce sommet (au début de l'algo récursif du parcours à partir de *s*)
  - juste avant de lancer l'appel dessus (juste avant l'appel récursif sur *s*.

  Ici, on utilisera la deuxième méthode : on marque un sommet avec sa profondeur dans l'arborescence, c'est à dire pour chaque successeur de *s*, sa marque est celle de *s* + 1. (Si on veut utiliser la première méthode, il suffit d'ajouter un paramètre supplémentaire représentant la profondeur du sommet actuel : on rappelle avec la profondeur + 1.)

  Attention Lors du parcours en profondeur d'un graphe non orienté à partir du sommet *s* : si un successeur *i*est déjà marqué, *(s,i)* n'est pas forcément un arc retour. Il faut vérifier que *i* n'est pas le père de *s* (qu'on n'a pas pris l'arc couvrant *(i,s)* dans l'autre sens !).

Algo récursif :

```
algorithme fonction plc_rec : entier
   parametres locaux
      t_graphe_s    g
      entier    s, pere
   parametres globaux
      t_vect_entiers    prof

   variables
      entier    i, long_max
debut
   long_max <- 0
   pour i <- 1 jusqu'a g.ordre faire
      si g.adj[s,i] alors    /* i est un successeur de s */
         si prof[i] = -1 alors
            prof[i] <- prof[s]+1    /* on marque i avec sa profondeur */
            long_max <- max (long_max, plc_rec (g, i, s, prof))
         sinon
            si i <> pere alors    /* (s,i) arc retour */
               long_max <- max (long_max, prof[s]-prof[i]+1)
            fin si
         fin si
      fin si
   fin pour
   retourne (long_max)
fin algorithme fonction plc_rec
```

Algo d'appel :

```
algorithme fonction plus_long_cycle : entier
   parametres locaux
      t_graphe_s    g

   variables
      t_vect_entiers    prof
      entier    i, long_max
debut
   pour i <- 1 jusqu'a g.ordre faire
      prof[i] <- -1
   fin pour
   long_max <- 0
   pour i <- 1 jusqu'a g.ordre faire
      si prof[i] = -1 alors
         prof[i] <- 0
         long_max <- max (long_max, plc_rec (g, i, -1, prof))
      fin si
   fin pour
   retourne (long_max)
fin algorithme fonction plus_long_cycle
```