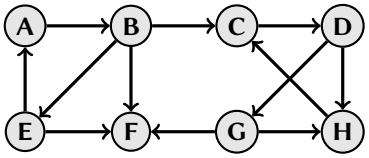
Outils logiques et algorithmiques – TD 5 – Parcours et terminaison

Exercice 1 (Parcours en profondeur et détection de cycles) Voici un algorithme de parcours en profondeur pour des graphes orientés. L'algorithme manipule une pile avec les notations suivantes : `[]` désigne la pile vide, `p.empiler(s)` place l'élément `s` au sommet de la pile `p`, et `p.depiler()` retire l'élément au sommet de la pile `p`. Notez que dans sa forme présentée ici l'algorithme ne consulte jamais la pile : il se contente d'y enregistrer certaines informations.

```
 1   exploration(s):
 2       en_cours := []
 3       visiter(s)

 4   visiter(s):
 5       si s est déjà marqué:
 6           ne rien faire
 7       sinon:
 8           en_cours.empiler(s)
 9           marquer s
10           pour chaque successeur s' de s:
11               visiter(s')
12           en_cours.depiler()
```

1. Exécuter l'algorithme `exploration` sur le sommet A du graphe suivant. On pourra supposer que toute énumération de sommets est faite par ordre alphabétique.



2. Décrire l'effet de `visiter(s)` sur la pile `en_cours`, et notamment la relation entre l'état de la pile en entrée et l'état de la pile en sortie. Justifier que cette spécification est correcte et dire quel est le sommet retiré du sommet de la pile par l'instruction (ligne 12)

   ```
   en_cours.depiler()
   ```

3. Montrer qu'à tout moment de l'exécution de `exploration(s)`, les sommets contenus dans la pile `en_cours` décrivent un chemin dans le graphe exploré.
4. On propose la technique suivante pour tester la présence d'un cycle :

   > *Si au cours de la visite d'un sommet* `s` *le test de la ligne 5 s'évalue à vrai, alors on déclare que le graphe contient un cycle passant par* `s`*. Si en revanche le parcours se termine sans que ce cas se soit produit alors on déclare que la graphe ne contient pas de cycle.*

   Montrer que ce critère est invalide et le corriger.

□

Exercice 2 (Terminaison de parcours de graphe) Voici un algorithme de parcours de graphe.

```
explorer(G, s)
    marquer s
    à_explorer = {s}
    tant que à_explorer n'est pas vide
        retirer un sommet s de à_explorer
        ...
        pour chaque voisin v de s
            si v n'est pas marqué alors
                marquer v
                ajouter v à à_explorer
```

1. Lors d'un tour de la boucle `tant que`, comment évoluent le nombre de sommets dans l'ensemble `à_explorer` et le nombre de sommets marqués ?
2. Justifier que l'exploration termine pour tout graphe `G` fini.

□

Exercice 3 (Ordres) On définit une relation sur les fonctions de $\mathbb{N} \longrightarrow \mathbb{N}$:

$f < g \triangleq$ il existe $l \in \mathbb{N}$ tel que $f(l) < g(l)$ et $(\forall k < l, f(k) = g(k))$

1. Comparer selon cette relation les fonctions $f(k) = k$, $g(k) = (k\%3)$ et $h(k) = (k\%6)$. On précisera à chaque fois l'entier l de la définition.
   On rappelle la définition de l'opération modulo : $(k\%p)$ représente le reste de la division euclidienne de k par p.
2. Montrer que cette relation transitive.
   Comme la relation est en outre irréflexice, on dira qu'il s'agit d'un ordre strict.
3. Est-ce un ordre total ?
4. Pour tout n, on considère la fonction f_n telle que $f_n(k) = 0$ si $k < n$ et $f_n(k) = 1$ si $n \leq k$. Montrer que $f_{n+1} < f_n$ pour tout n.
5. L'ordre proposé est-il bien fondé ?

□

Exercice 4 (Fonction de Ackermann) Soient $(A, <_A)$ et $(B, <_B)$ des ensembles, chacun muni d'un ordre strict bien fondé. On définit une relation binaire $<$ sur $A \times B$ par la condition suivante :

$(a_1, b_1) < (a_2, b_2)$ si et seulement si $a_1 <_A a_2$ ou $a_1 = a_2 \wedge b_1 <_B b_2$

1. En admettant que cette relation est un ordre strict, montrer qu'il est bien fondé. Au fait, reconnaissez-vous cet ordre ?

Voici le code d'une fonction java.

```
static int ack(int m, int n) {
  if (m == 0)      return n+1;
  else if (n == 0) return ack(m-1, 1);
  else             return ack(m-1, ack(m, n-1));
}
```

2. Montrer que, pour tous entiers positifs m et n, l'appel $\mathsf{ack}(m, n)$ termine.

□