Annexe 1 : Version itérative du parcours en profondeur *(Depth First Search)*

Structures de données utilisées :

– On utilise une pile P, pour laquelle on suppose définies les opérations *init_pile(P)* qui initialise la pile P à vide, *empile(P,s)* qui ajoute s au sommet de la pile P, *est_vide(P)* qui retourne vrai si la pile P est vide et faux sinon, *sommet(P)* qui retourne le sommet s au sommet de la pile P, et *depile(P,s)* qui enlève s du sommet de la pile P.

– On utilise, comme pour le parcours en largeur, un tableau π qui associe à chaque sommet le sommet qui l'a fait entrer dans la pile, et un tableau *couleur* qui associe à chaque sommet sa couleur (blanc, gris ou noir).

– On va en plus mémoriser pour chaque sommet s_i :
- $dec[s_i]$ = date de découverte de s_i (passage en gris)
- $fin[s_i]$ = date de fin de traitement de s_i (passage en noir)

où l'unité de temps est une itération. La date courante est mémorisée dans la variable *tps*.

```
Algorithme de parcours en profondeur des sommets accessibles
    init_pile(P)
    pour tout sommet s_i ∈ S faire
        π[s_i] ← nil
        couleur[s_i] ← blanc
    fin pour
    tps ← 0
    dec[s_0] ← tps
    empile(P, s_0)
    couleur[s_0] ← gris
    tant que est_vide(P) = faux faire
        tps ← tps + 1
        s_i ← sommet(P)
        si ∃s_j ∈ succ(s_i) tel que couleur[s_j]=blanc alors
            empile(P, s_j)
            couleur[s_j] ← gris
            π[s_j] ← s_i
            dec[s_j] ← tps
        sinon /* tous les successeurs de s_i sont gris ou noirs */
            depile(P, s_i)
            couleur[s_i] ← noir

            fin[s_i] ← tps
        finsi
    fin tant
fin DFS
```

Complexité.

Chaque sommet (accessible depuis s0) est mis, puis enlevé, une fois et une seul dans la pile. A chaque fois qu'on enlève un sommet de la pile, on parcourt tous ses successeurs ; chaque arc (ou arête) du graphe sera utilisé une fois et une seule dans l'algorithme. Si le graphe contient n sommets et m arcs/arêtes, alors :

- pour une représentation par matrice d'adjacence *O(n^2)*
- pour une représentation par listes de successeurs en $O(n + m)$

Annexe 2 : Version itérative du parcours en largeur *(Breath First Search)*

Structures de données utilisées :

- On utilise une file F, pour laquelle on suppose définies les opérations *init_file(F)* qui initialise la file F à vide, *ajoute_fin_file(F,s)* qui ajoute le sommet s à la fin de la file F, *est_vide(F)* qui retourne vrai si la file F est vide et faux sinon, et *enleve_debut_file(F,s)* qui enlève le sommet s au début de la file F.
- On utilise un tableau π qui associe à chaque sommet le sommet qui l'a fait entrer dans la file, et un tableau *couleur* qui associe à chaque sommet sa couleur (blanc, gris ou noir).
- On va en plus utiliser un tableau d qui associe à chaque sommet son niveau de profondeur par rapport au sommet de départ s_0 (autrement dit, $d[s_i]$ est la longueur du chemin dans l'arborescence π de la racine s_0 jusque s_i). Ce tableau sera utilisé plus tard, cf. 7.3.

```
Algorithme de parcours en largeur (BFS)(S, A, s0)
    init_file(F)

      pour tout sommet si ∈ S faire
          π[si] ← nil
          d[si] ← ∞
          couleur[si] ← blanc
      fin pour
      d[s0] ← 0
      ajoute_fin_file(F, s0)
      couleur[s0] ← gris
      tant que est_vide(F) = faux faire
          enleve_debut_file(F, si)
          pour tout sj ∈ succ(si) faire
              si couleur[sj] = blanc alors
                  ajoute_fin_file(F, sj)
                  couleur[sj] ← gris
                  π[sj] ← si
                  d[sj] ← d[si] + 1
              finsi
          fin pour
          couleur[si] ← noir
      fin tant
  fin BFS
```

Complexité.

Le raisonnement est analogue à celui tenu pour le parcours en profondeur. Chaque sommet (accessible depuis s0) est mis, puis enlevé, une fois et une seul dans la file. A chaque fois qu'on enlève un sommet de la file, on parcourt tous ses successeurs ; chaque arc (ou arête) du graphe sera utilisé une fois et une seule dans l'algorithme. Si le graphe contient *n* sommets et *m* arcs/arêtes, alors :

- pour une représentation par matrice d'adjacence *O(n^2)*
- pour une représentation par listes de successeurs en *O(n + m)*