Implémentation et parcours de graphe en OcamL

par Humbert Florent (millie)

Date de publication : 14/11/2006

Dernière mise à jour : 14/11/2006

Dans ce cours, nous allons voir comment implémenter des graphes en camL afin d'effectuer des parcours sur ceux-ci de manière efficace.

I - Introduction et remerciements

Les graphes représentent un outil mathématique et informatique puissants. Ils permettent de modéliser de nombreux problèmes, certains sont très connus :

- Recherche du plus court chemin d'un point à un autre
- Problème du flot maximal
- Problème du couplage maximal
- Coloration de cartes
- Planification de travaux
- Modélisation de système distribué

La plupart de ces problèmes utilisent la notion de parcours de graphe. Ce qui est une bonne nouvelle car cet algorithme est rapide et se prête très bien à une implémentation récursive, nous allons le voir dans le langage fonctionnel OCaml.

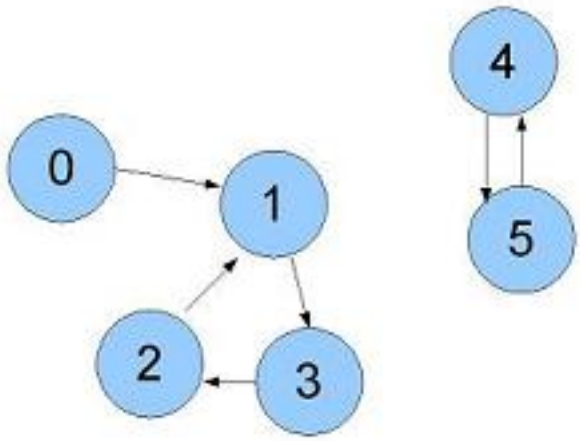
I-A - Remerciements

Je tiens à remercier pcaboche pour les conseils qu'il a pu me donner pour la rédaction de ce petit cours. Je remercie également Miles pour sa relecture minutieuse.

II - Notion de graphes orientés

II-A - Qu'est-ce qu'un graphe ?

Dans ce cours, nous allons uniquement utiliser la notion de graphe orienté. Un graphe est de manière peu formelle un ensemble de sommet relié par des arêtes. Une représentation d'un tel objet ressemble à l'image suivante :



Représentation d'un graphe

On supposera de plus que le graphe est simple (bien que l'algorithme ne change pas, on verra après pourquoi) : c'est à dire qu'il n'est pas possible qu'il y ait deux arêtes qui aillent du même sommet à un autre.

Les sommets auxquels on peut directement accéder à partir d'un sommet en n'empruntant qu'une seule arête s'appellent les voisins. Par exemple le voisin de 0 est 1, le voisin de 1 est 3. Il peut y avoir plusieurs voisins si plusieurs arêtes partent d'un sommet.

II-B - Formalisation

Un graphe orienté simple peut être défini comme suit :

Un graphe G est un couple (V,E) où V (vertices) est l'ensemble des sommets (nombre fini) et E (edges) l'ensemble des arêtes (ce sont des couples de sommets de V).

Par exemple, dans notre exemple, V = {0,1,2,3,4,5} et E = {(0,1), (1,3), (3,2) , (2,1),(4,5), (5,4)}.

On pourrait représenter le graphe en camL directement sous cette forme, mais cela n'est pas efficace pour les parcours (profondeur et largeur) de graphe. C'est ce que nous allons voir.

III - Parcours et algorithme

Nous allons dans cette partie expliquer ce qu'est un parcours et voir les algorithmes de parcours en profondeur et largeur.

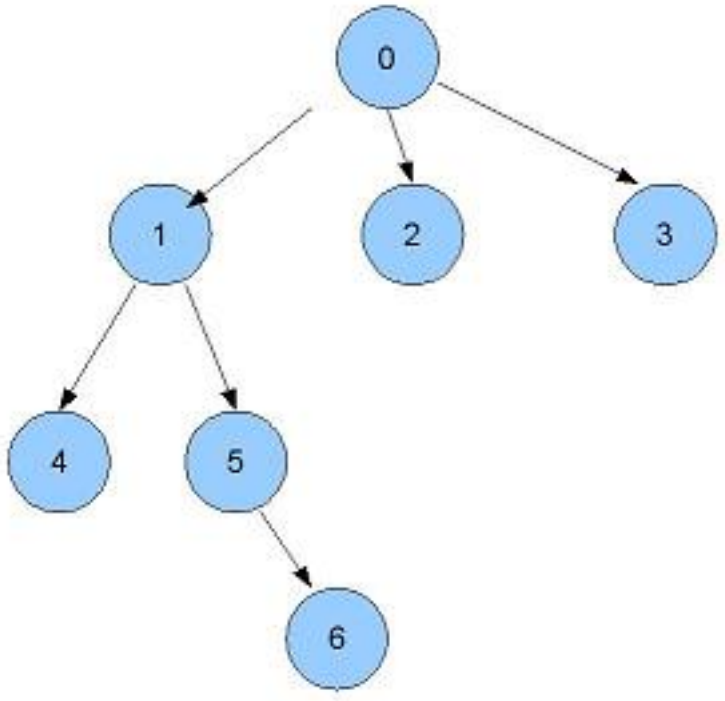
III-A - Qu'est ce qu'un parcours ?

Un parcours est une procédure qui permet d'aller chercher l'ensemble des sommets accessibles à partir d'un sommet choisi et d'effectuer une opération dessus. Par exemple, dans la première figure, l'ensemble des sommets accessibles à partir de 0 sont : 1, 2 et 3. Mais l'ensemble des sommets accessibles à partir de 3 sont uniquement : 3, 2 et 1. En effet, on ne peut pas retourner au sommet 0 car il n'existe pas d'arête allant vers 0.

En pratique, on travaille récursivement sur les voisins du sommet. Dans le parcours en largeur, on parcourt les successeurs du sommet de départ par rapport à sa profondeur. On regarde tout d'abord, les successeurs séparés par une arête. Ensuite les successeurs séparés par deux arêtes et ainsi de suite. Dans le parcours en profondeur, on parcourt récursivement les voisins du sommet en allant "en profondeur". Ces deux algorithmes sont en fait très similaires et ne dépendent que de l'ordre dans lequel on choisit le futur sommet à visiter.

III-B - Exemple

Afin de mieux comprendre la notion de parcours, nous allons voir un exemple sur la figure suivante.

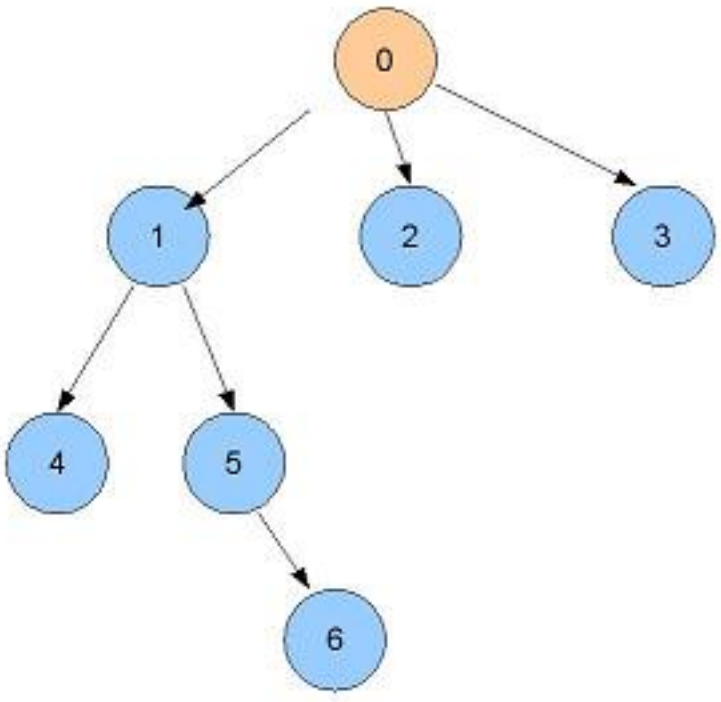


Graphe 3

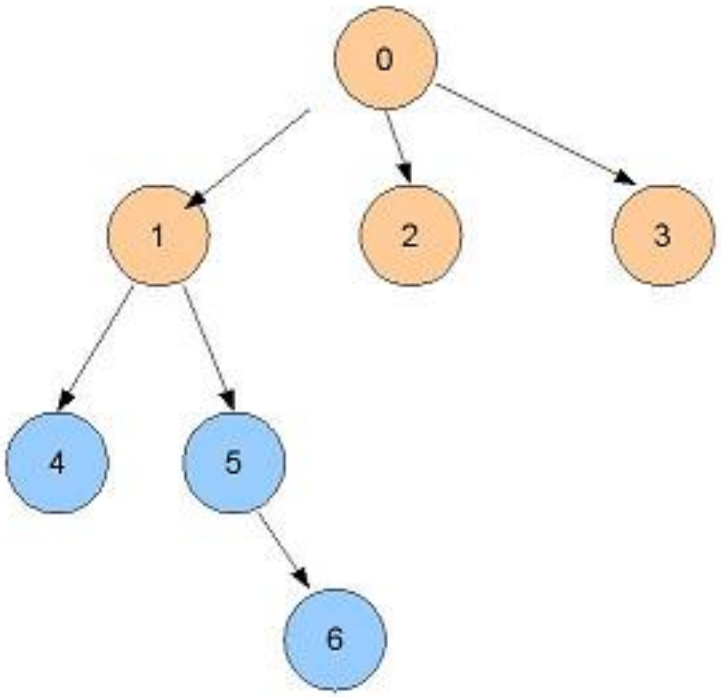
III-C - Exemple du parcours en largeur

En partant du sommet 0, le parcours en largeur donne successivement.

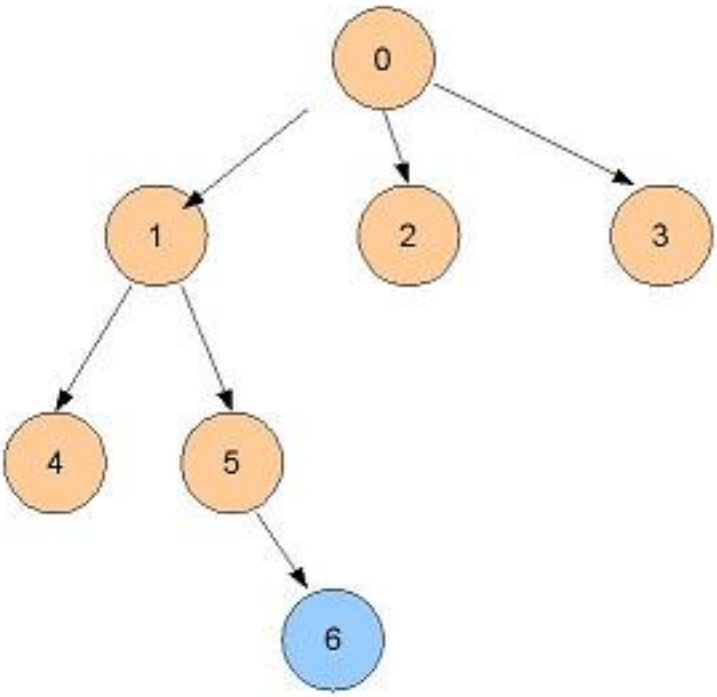
- On visite le sommet 0



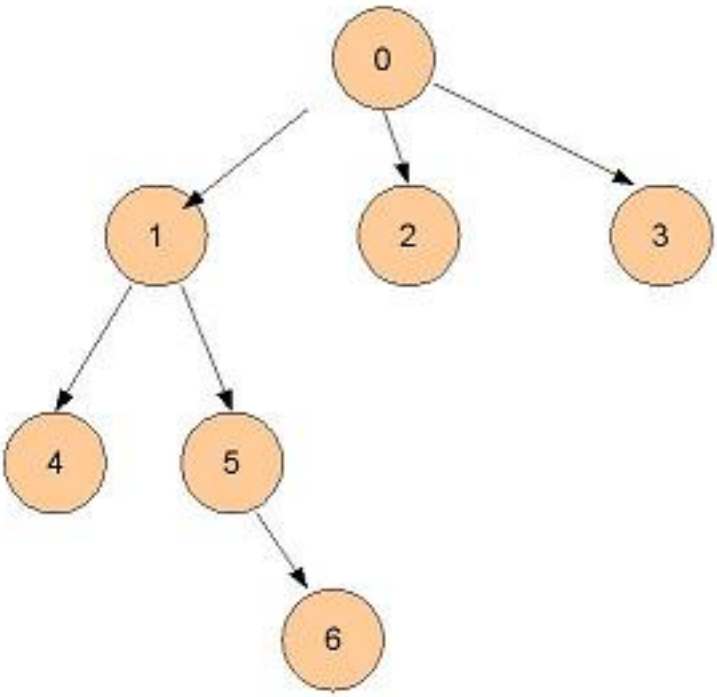
- Les voisins de distance 1 : 1, 2, 3



- Les voisins de distance 2 : 4, 5



- Puis les voisins de distance 3 : 6



- On a parcouru tout le graphe, l'algorithme s'arrête

On va récapituler le procédure en mettant dans une liste les éléments restant à visiter.

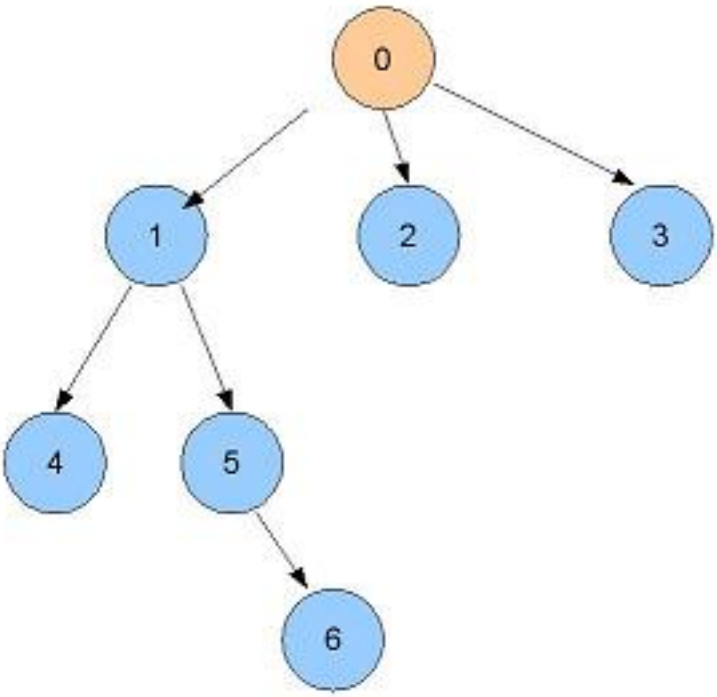
- On visite 0, on place dans la liste les éléments restant à visiter dans l'ordre : 1, 2, 3
- On visite le premier élément de la liste 1, ses voisins sont 4 et 5, il reste à visiter 2, 3, 4, 5 (il faut bien que l'on met les éléments à la fin de la liste)
- On visite 2 qui n'a pas de voisins, il reste à visiter 3, 4, 5
- On visite 3 qui n'a pas de voisins, il reste à visiter 4, 5
- On visite 4 qui n'a pas de voisin, il reste à visiter 5
- 5 a un unique voisin, il reste donc à visiter 6
- Puis on visite le dernier sommet

Lorsque l'on ajoute les voisins à la liste des éléments à ajouter, on les place à la fin, et on lit le premier élément de la liste, cela correspond à une structure de **file**.

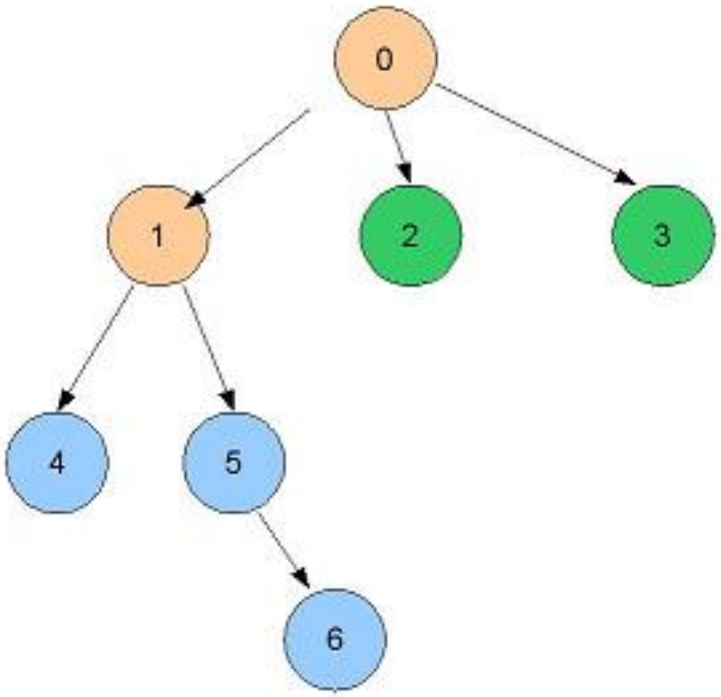
III-D - Exemple du parcours en profondeur

Le parcours en profondeur peut donner (mais ce n'est pas unique) :

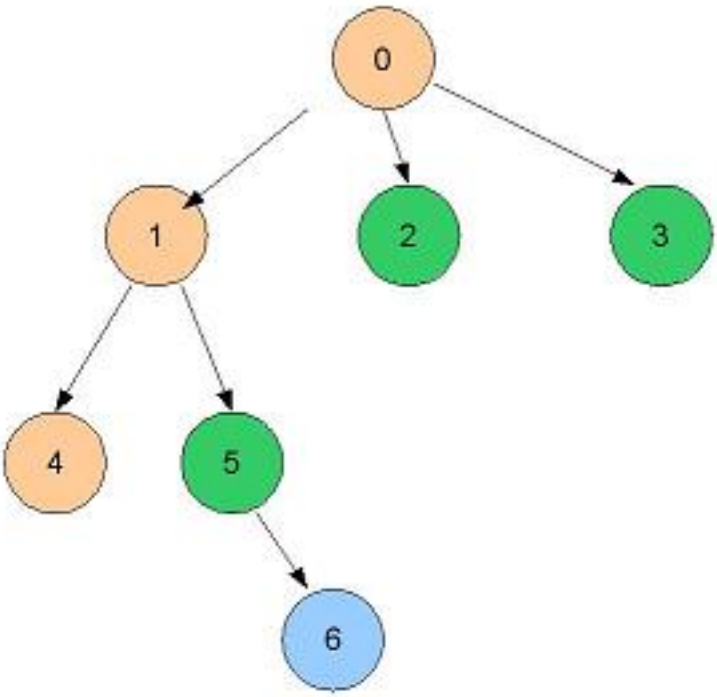
- On visite le sommet 0



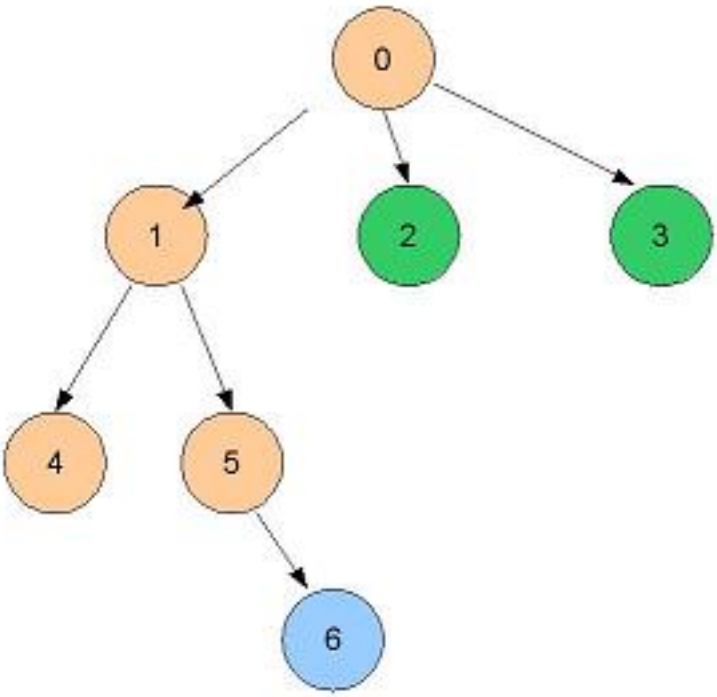
- Voisins : 1, 2, 3, on visite 1 (reste à visiter 2,3)



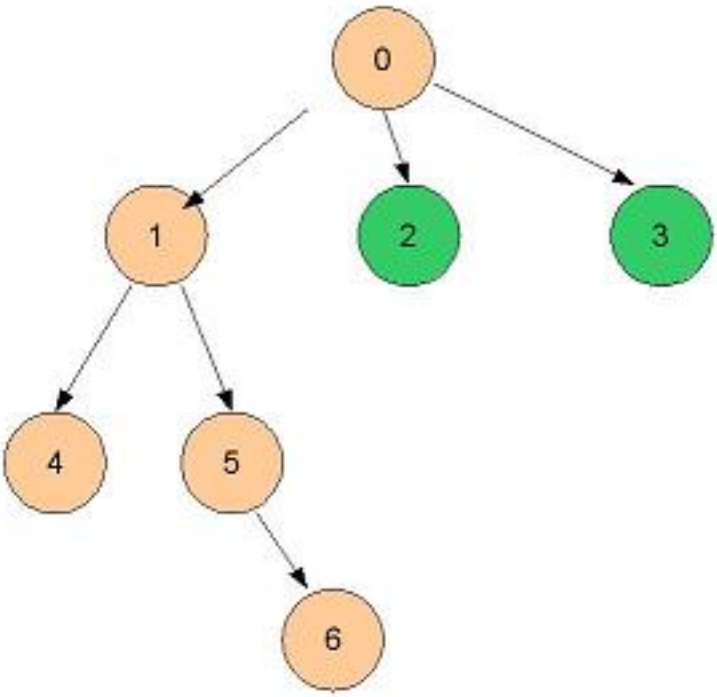
- Voisins de 1 : 4 et 5, on visite 4 (reste à visiter 5, 2, 3)



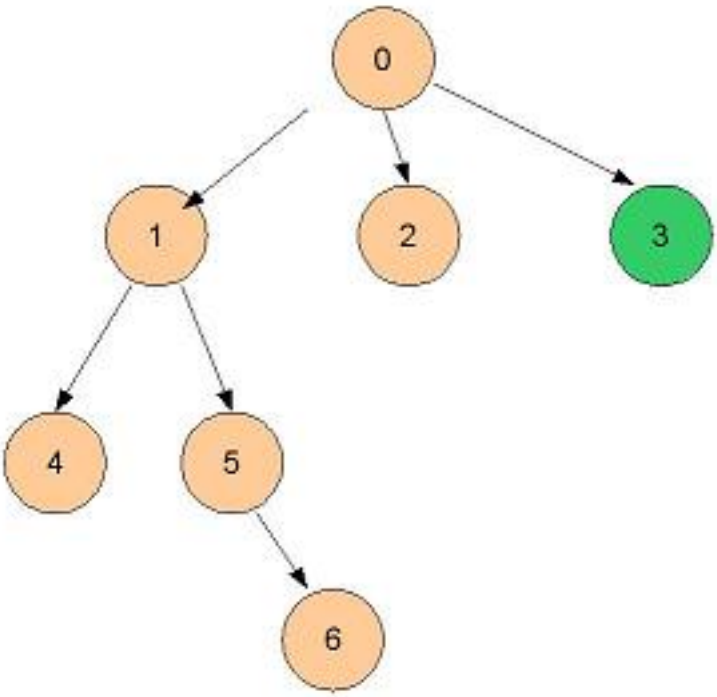
- 4 n'a plus de voisin, on visite 5 (reste à visiter 2,3)



- Voisins de 5 : 6, on visite cet unique sommet (reste à visiter 2,3)



- 6 n'a pas de voisin, on retourne à 2 (reste à visiter 3)



- 2 n'a pas de voisin, on retourne à 3 (il ne reste rien à visiter)

On va récapituler le procédure en mettant dans une liste les éléments restant à visiter.

- On visite 0, on place dans la liste les éléments restant à visiter dans l'ordre : 1, 2, 3
- On visite le premier élément de la liste 1, ses voisins sont 4 et 5, il reste à visiter 4, 5, 2, 3 (il faut bien que l'on met les éléments au début de la liste)
- On visite 4 qui n'a pas de voisins, il reste à visiter 5, 2, 3
- On visite 5 qui a un voisin, il reste à visiter 6, 2, 3
- On visite 6 qui n'a pas de voisin, il reste à visiter 2 et 3
- 2 n'a pas de voisin, il reste donc à visiter 3
- 3 n'a pas de voisin, le parcours est terminé

Lorsque l'on ajoute les voisins à la liste des éléments à ajouter, on les place au début, et on lit le premier élément de la liste, cela correspond à une structure de **pile**.

III-B - Algorithme

Lorsque l'on détermine les voisins d'un sommet. On sait que l'on doit tous les visiter. Une idée serait donc de mettre dans un ensemble les sommets qu'il reste à visiter. Nous n'allons pas expliciter l'implémentation d'un tel ensemble (qui correspondra à une file ou à une pile, mais nous généralisons le procédé dans un premier temps), on supposera qu'il dispose des opérations suivantes :

- [] : représente l'ensemble vide
- ensembleChoisir (Ensemble) -> element : retourne un élément de l'ensemble
- ensembleRetirer (Ensemble) -> Ensemble : retire l'élément que l'on obtient avec ensembleChoisir

- ensembleUnion(Ensemble 1, Ensemble 2) -> Ensemble : détermine la réunion des deux ensembles (on peut supposer qu'il existe des doublons)
- element appartient Ensemble -> Booléen est vrai si l'élément appartient à l'ensemble et faux sinon

À un moment du parcours, on suppose que l'on dispose des 2 ensembles suivants :

- avisiter : l'ensemble des sommets qu'il reste à visiter
- aetevisite : l'ensemble des sommets qui ont déjà été visité

L'algorithme peut s'écrire comme suit :

```
fonction Parcours g avisiter aetevisite=
  Si avisiter = [] alors /*Si il ne reste rien à visiter, on a fini*/
     fin;
  Sinon
     sommetcourant = ensembleChoisir(avisiter) /*sommet que l'on traite*/
     resteavisiter = ensembleRetirer(avisiter)
     Si sommetcourant appartient à aetevisite alors /*Si l'élément a déjà été visité*/
          Parcours g aetevisite resteavisiter /*on visite les sommets suivants*/
     Sinon
       /* on ajoute au élément à visiter les voisins du sommet courant,
          et on ajoute le sommetcourant au élément qui ont été visité*/
          Parcours g ensembleUnion(resteavisiter, voisin(g, sommetcourant))
ensembleUnion(aetevisite, sommetcourant)
```

III-C - Explicitation du type Ensemble

Nous allons chercher à expliciter le type ensemble, cela nous permettra de définir l'implémentation du graphe afin d'optimiser le parcours.

Il faut noter que même si l'ensemble admet des doublons, l'algorithme est bon. Cela nous permet ainsi d'envisager un type liste pour l'ensemble **avisiter**. L'opération **ensembleChoisir** consistera juste à prendre le premier élément de la liste. et l'opération **ensembleRetirer** consistera à prendre la queue de la liste.

Ce qui devient intéressant, c'est le choix d'implémentation de l'opération **ensembleUnion**. Si l'on choisit d'ajouter les éléments à la queue de la liste, l'algorithme devient un parcours en largeur et si l'on choisit d'ajouter les éléments à la tête de la liste, cela devient un parcours en profondeur.

Ainsi, dans le cas du parcours en largeur, comme on ajoute les éléments à la queue de la liste, cela revient à utiliser comme implémentation du type **Ensemble** une **file**. Dans le parcours en profondeur, cela revient à utiliser une **pile**.

IV - Implémentation

À chaque fois que l'on ajoute une liste à une autre liste, c'est la liste des voisins d'un sommet. Ceci est très important, car cela suggère d'utiliser des listes de voisins d'un sommet. Ainsi, pour chaque sommet S, on créait un couple *(S, Voisins)* où S est le sommet et *Voisins* est la liste des voisins. Ainsi, pour représenter l'ensemble du graphe, il suffit de lister tous ces couples.

IV-A - Récapitulation

On implémente l'ensemble des arêtes du graphe par une liste de liste des successeurs.

Par exemple, sur le graphe 3, on a : [(0,[1,2,3]), (1,[4,5]), (2,[]), (3,[]), (4,[]), (5,[6]), (6,[])]

Pour le premier graphe, on a : [(0,[1]), (1,[3]), (2,[1]), (3,[2]), (4,[5]), (5,[4])]

Si par exemple, on souhaite déterminer les voisins du sommet 3, il suffit d'aller chercher le couple dont le premier élément est 3, puis de renvoyer la liste des voisins (le deuxième élément du couple).

V - Programmation et sources

V-A - Type graphe

Le graphe sera définie comme suit en camL

Définition du type
```
type graphe  = (int * int list) list;;
```

On a simplifié en supposant que les sommets étaient représentés par des entiers, mais il est possible d'utiliser une définition plus générale.

Pour créer un graphe, il suffit par exemple de faire :

Création
```
let gtest =[(1,[2;4]); (2,[1;3]); (3,[]); (4,[])];;
```

V-B - Détermination des voisins d'un sommet

Afin de simplifier le code du parcours, nous allons utiliser une fonction permettant de déterminer la liste des voisins d'un sommet. Nous avons déjà précisé précédemment la manière dont on pouvait s'y prendre pour réaliser cela.

Récuperation des voisins
```
let rec getVoisins g sommet =
  match g with
   [] -> failwith "Accès à un sommet non existant"
  |(s, voisins)::reste -> if s = sommet then
                            voisins
                          else
                            getVoisins reste sommet;;
```

V-C - Appartenance à une liste

Nous allons définir une fonction permettant de savoir si un élément est dans une liste. Pour cela, nous utiliserons la fonction **mem** du module List. Cette fonction servira notamment à déterminer si un élément a déjà été visité ou non.

Appartenance à une liste
```
let isinListe liste element= List.mem element liste;;
```

V-D - Programme de parcours

Nous allons écrire à présent la fonction principale de parcours (qui est écrite de manière générale) qui dépend d'une fonction d'ajout : **fAjout**. Pour faire le lien avec la partie précédente, cette fonction est la fonction **ensembleUnion**. La différence par rapport à l'algorithme est que l'on passe en argument une fonction *fTraitement* qui réalise un traitement sur un sommet.

Parcours
```
(* Le parcours *)
 (*
    g correspond au graphe,
```

Parcours
```
      aetevisite est la liste des sommets visité
      avisiter sont les sommets à visiter
      fAjout la fonction
      fTraitement la fonction qui traite le sommet
   *)
   let rec parcours g aetevisite avisiter fAjout fTraitement =
      match avisiter with
        [] -> ()
        |sommetCourant::reste ->
               if (isinListe aetevisite sommetCourant) then
                   parcours g aetevisite reste fAjout fTraitement
               else
                begin
                 fTraitement sommetCourant;

                 let voisins = getVoisins g sommetCourant
                 in

                 let resteavisiter = fAjout reste voisins
                 in

                 let cequiaetevisite = sommetCourant::aetevisite
                 in

                    parcours g cequiaetevisite resteavisiter fAjout fTraitement
                end
     ;;
```

V-E - Parcours en largeur

Comme nous l'avons vu dans la partie précédente, le parcours en largeur se fait à l'aide d'une file. Cela nous incite ainsi à utiliser comme fonction **fAjout** une fonction enfilant les éléments à la fin de la liste. Le code est le suivant :

Fonction d'ajout de sommet
```
   let enfiler file listelement=
     file@listelement;;
```

À présent, nous pouvons simplement écrire la fonction de parcours en largeur.

Parcours en largeur
```
   let parcoursLargeur g sommet fTraitement =
     parcours g [] [sommet] enfiler fTraitement;;
```

V-F - Parcours en profondeur

De la même manière que précédemment, comme la fonction fAjout est implémentée par une pile, nous allons créer une fonction ajoutant une liste d'élément en début de liste.

Fonction d'ajout de sommet
```
   let empiler pile listelement=
     listelement@pile;;
```

Il ne nous reste plus que la fonction principale.

Parcours en profondeur
```
   let parcoursProfondeur g sommet fTraitement=
    parcours g [] [sommet]  empiler fTraitement;;
```

VI - Sources complètes

Sources, version OcamL

```
type graphe  = (int * int list) list;;

let rec getVoisins g sommet =
  match g with
  [] -> failwith "Accès à un sommet non existant"
 |(s, voisins)::reste -> if s = sommet then
                            voisins
                                                 else
                                        getVoisins reste sommet;;


let isinListe liste element= List.mem element liste;;

let rec parcours g aetevisite avisiter fAjout fTraitement =
   match avisiter with
     [] -> ()
     |sommetCourant::reste ->
            if (isinListe aetevisite sommetCourant) then
                parcours g aetevisite reste fAjout fTraitement
            else
             begin
              fTraitement sommetCourant;

                         let voisins = getVoisins g sommetCourant
                         in

                         let resteavisiter = fAjout reste voisins
                         in

                         let cequiaetevisite = sommetCourant::aetevisite
                         in

                            parcours g cequiaetevisite resteavisiter fAjout fTraitement
             end
   ;;


let enfiler file listelement=
  file@listelement;;
let empiler pile listelement=
  listelement@pile;;

let parcoursProfondeur g sommet fTraitement=
 parcours g [] [sommet]  empiler fTraitement;;

let parcoursLargeur g sommet fTraitement =
  parcours g [] [sommet] enfiler fTraitement;;
```

Le code source est également disponible pour camL Light : source caml Light

VII - Exemple d'application

Par exemple, l'exécution sur le graphe 3 donne le résultat suivant.

Exemple

```
let graphe3  = [(0, [1;2;3]); (1, [4;5]); (2, []); (3, []); (4, []);(5, [6]); (6,[])];;

parcoursLargeur graphe3 0 print_int;;
#0123456- : unit = ()

parcoursProfondeur graphe3 0 print_int;;
#0145623- : unit = ()
```

VIII - Conclusion

Au sein de ce tutorial, nous avons pu voir une application intéressante de l'utilisation de la notion de graphe. Les algorithmes sur ces objets étant en général récursifs, le langage camL permet une implémentation simple et efficace de ceux-ci. Nous avons utilisé plusieurs particularités de camL :

- les listes se révèlent extrêmement intéressantes dans notre algorithme
- les algorithmes peuvent s'écrire sous forme récursive terminale, ce qui permet une très bonne optimisation de camL
- l'utilisation de *let* permet de rendre plus lisible le code, cette instruction n'influe pas sur le temps d'exécution du programme, en effet, les valeurs sont remplacées directement lors de la compilation
- le passage en paramètre de fonction permet de rendre plus général le code

En réfléchissant un peu, nous pourrions programmer assez facilement des fonctions supplémentaires. Notamment la détermination de cycle ou encore la recherche de forte connexité.

Pour toute remarque ou suggestion, vous pouvez m'envoyer un message privé.