Représentation mémoire d'un « Graphe »

Représentations d'un graphe en mémoire

Ce document est une discussion sur les représentations les plus « classiques » d'un graphe en mémoire.
On notera G = (S , A) un graphe.
On suppose que S, l'ensemble des sommets, est l'ensemble des entiers compris entre 0 et N-1 pour un graphe ayant N sommets.
Les extraits de code fournis sont exprimés en C++, en se limitant volontairement à des types ou constructeurs simples, à l'exclusion de `class`, `map`, `list`, … .

Matrice d'adjacence sommets / sommets ➔ Tableaux

Graphe orienté non valué – Tableau dynamique

La traduction immédiate est un tableau à 2 dimensions :

```
bool ** MAdj ;
```

avec

`MAdj[x][y]` a pour valeur '`true`' si et seulement s'il existe un arc de x vers y.

Remarque : si le type `bool` n'existe pas, on définira un tableau de type `int` en restreignant les valeurs possibles à '0' ou '1'. Mieux, on pourra utiliser un type défini avec `enum` (avec 2 valeurs possibles).

Nombre de sommets

L'utilisation de ce genre de tableau implique une allocation dynamique qui ne peut être faite que lorsque le nombre de sommets est connu.
Nous supposons qu'il est stocké dans une variable spécifique, par exemple

```
int nbSommets ;
```

Création et Initialisation

Dès que le nombre de sommets est connu, il faut bien évidemment créer la matrice :

```
MAdj = new bool * [ nbSommets ] ;
for ( int ligne = 0 ; ligne < nbSommets ; ligne ++ ) {
     MAdj [ ligne ] = new bool [ nbSommets ] ;
} ;
```

Il est préférable, juste après cette création, d'initialiser l'ensemble des cases de ce tableau avec la valeur 'FALSE' (correspondant à un graphe sans arcs) :

```
for ( int ligne = 0 ; ligne < nbSommets ; ligne ++ ) {
     for ( int colonne = 9 ; colonne < nbSommets ; colonne ++ ) {
          MAdj [ ligne ] [ colonne ] = FALSE ;
     } ;
} ;
```

Graphe non orienté

En cas de graphe non orienté, pour chaque couple de sommets x et y, `MAdj[x][y]` et `MAdj[y][x]` auront la même valeur.

Graphe valué

La solution la plus immédiate est de disposer d'un tableau similaire, par exemple si les valeurs sont de type entier :

```
int ** MVal
```

La valeur stockée dans une case de `MVal` n'a de sens que lorsque la case correspondante de `MAdj` a la valeur '`true`'.
La création de MVal est similaire à celle de MAdj.

Type « graphe »

On peut regrouper toutes ces données dans une structure :

```
typedef struct {
     bool **    MAdj ;
     int **     MVal ;
     int        nbSommets ;
} type_graphe ;
type_graphe G ;
```

On utilise G.MAdj, G.MVal, G.nbSommets.

Alternatives

En cas de graphe valué, il est possible de remplacer la matrice MVal de plusieurs façons :

- Définition de MAdj comme une matrice de pointeurs vers les valeurs :
  `int *** MAdj`
  Lorsque MAdj[x][y] = FALSE, l'arc n'existe pas.
  Si MAdj[x][y] est initialisé, il contient l'adresse de la valeur de l'arc.

- Matrice d'entiers avec une valeur particulière pour l'absence d'arc :
  int ** MAdj
  Si MAdj[x][y] a cette valeur particulière, l'arc n'existe pas ; si l'arc existe alors la matrice contient directement la valeur.
  Cette technique impose donc qu'une valeur ne soit jamais présente comme valeur d'arc. Ce peut être utilisé par exemple si les arcs ont tous une valeur positive ou nulle : on prend alors '-1' comme valeur particulière.

Tableau statique surdimensionné

On peut également utiliser un tableau de taille fixe surdimensionné, dans lequel la zone significative réellement utilisée est définie par la variable `nbSommets`. Dans ce cas, un nombre maximum de sommets est défini, par exemple avec `const int maxSommets = 100`, et les tableaux déclarés en utilisant cette valeur, par exemple `bool MAdj[maxSommets][maxSommets]`. La valeur de `nbSommets` permet alors de ne rechercher des informations que dans la partie significative du tableau.

Matrice d'incidence sommets / arcs ➔ Tableaux

Note : cette matrice n'a de sens que dans le cas de graphe orienté
Il est également délicat lors de la présence de boucles, car il faut alors trouver un artifice pour remplacer les valeurs '+1' et '-1' qui se confondent dans la même case. Une solution simple consiste alors à utiliser par exemple la valeur '2'.

Graphe orienté non valué

La matrice d'incidence associé les valeurs '+1', '-1' ou '0' à chaque couple identifiant un sommet et un arc.
La traduction immédiate est un tableau à 2 dimensions

```
int ** MInc
```

et la valeur d'une case `Minc[s][a]` dépend de la relation entre le sommet 's' et l'arc 'a'.

On peut limiter l'espace mémoire utilisé en remplaçant le type 'int' par un 'enum' à 3 valeurs.

Nombres de sommets et d'arcs

Cette définition implique une allocation dynamique qui ne peut être faite que lorsque le nombre de sommets et le nombre d'arcs sont connus. Comme pour la matrice d'adjacence, on préférera sauvegarder ces valeurs dans 2 variables :

```
int nbSommets ;
int nbArcs;
```

Création

La création est similaire à celle de la matrice d'adjacence, avec cependant deux tailles différentes : nombre de sommets pour la 1ère dimension ; nombre d'arcs pour la 2nde.

Graphe valué

En cas de graphe valué, il suffit de sauvegarder les valeurs dans un tableau spécifique contenant une valeur par arc, par exemple

```
int * MVal ;
```

Ce tableau est créé lorsque le nombre d'arcs est connu, et est indépendant du nombre de sommets.

La valeur contenue dans la case 'a' de ce tableau correspond à l'arc désigné par l'indice 'a' dans la 2nde dimension du tableau Minc.

Tableau statique surdimensionné

Cf. matrice d'adjacence. Le principe est le même pour le nombre d'arcs.

Listes chaînées des prédécesseurs

Liste des prédécesseurs

L'idée est de représenter l'ensemble des prédécesseurs d'un sommet dans une liste chaînée. On définit alors chaque maillon de cette chaîne sous la forme :

```
typedef struct eltLstPred {
      int sommet ;
      eltLstPred * predecesseurSuivant } eltLstpred ;
```

Il suffit d'avoir un tableau (1 case par sommet) qui contient un pointeur vers le premier élément de sa liste des prédécesseurs. Mais on retombe alors dans les problèmes évoqués précédemment relatifs au nombre de sommet (allocation dynamique ou surdimensionnement).

Liste de listes de prédécesseurs

Tout comme la liste chaînée n'impose aucune contrainte sur le nombre de prédécesseurs associés à un sommet, il est possible d'avoir une structure similaire pour contenir la liste des listes de prédécesseurs. On constitue ainsi une liste de sommets ; chaque maillon de cette liste représente un sommet et pointe vers le premier élément de sa liste de prédécesseurs :

```
typedef struct eltLstSommets {
      int sommet ;
      eltLstPred * predecesseurs ;
      eltLstSommets * sommetSuivant } eltLstSommets ;
```

Remarque : la valeur 'NULL' associée au champ `predecesseurs` d'un élément de cette liste indique que le sommet n'a pas de prédécesseur.
Chaque arc est représenté par l'appartenance de son extrémité initiale à une liste de prédécesseurs.

Il suffit alors d'une variable de type « liste des sommets » pour représenter le graphe :

```
eltLstSommets * monGraphe ;
```

Graphe valué

En cas de graphe valué, il suffit d'ajouter un champ, par exemple `int valeur`, dans la structure `eltLstPred`.

Listes chaînées des successeurs

Similaire à la SdD précédente, mais listes des successeurs au lieu de liste des prédécesseurs.

Listes chaînes des prédécesseurs et successeurs

Dans certain cas, il peut être opportun de disposer des deux listes simultanément, selon les algorithmes exécutés. Par exemple, Dijkstra fait appel aux successeurs d'un sommet à chaque itération tandis que Bellman fait appel aux

prédécesseurs de tous les sommets à chaque itération. Le code et l'exécution de chaque algorithme sera optimisé en choisissant la représentation correspondante. On peut imaginer des cas où à la fois le parcours des prédécesseurs d'un sommet et celui de ses successeurs sont nécessaires. On représentera ainsi simultanément les 2 listes.
La liste des sommets devient alors :

```
typedef struct eltLstSommets {
      int sommet ;
      eltLstPred * predecesseurs ;
      eltLstSucc * successeurs ;
      eltLstSommets * sommetSuivant } eltLstSommets ;
```

Remarque : une telle SdD impose de la rigueur dans son utilisation pour respecter la cohérence entre les listes des prédécesseurs et les listes des successeurs !

Listes chaînées des voisins

Dans le cas de graphe non orienté, on ne fait pas de différence entre prédécesseur et successeur. On pourra ainsi utilise rune SdD proche des listes des prédécesseurs (ou successeurs), en renommant le champ 'voisins'. Tout comme pour le cas d'une SdD contenant les listes des prédécesseurs et celles des successeurs, il faut ici faire attention à la cohérence des données.

Tableau des arcs

Une SdD simple (en apparence) est d'avoir un tableau de triplets extrémité initiale / extrémité terminale / valeur, par exemple en définissant une structure représentant un arc :

```
typedef struct {
      int extInitiale ;
      int extTerminale ;
      int valeur } unArc ;
```

et en en constituant un tableau `unArc * tabArcs`.
La taille du tableau dépend du nombre d'arcs.

Cette SdD est simple quant à sa conception, mais est particulièrement lourde lors de l'exécution d'un programme puisqu'elle nécessite souvent un parcours complet du tableau à la recherche de l'information voulue.
Elle pose également le problème de représentation des points isolés (extrémité d'aucun arc). Il faut donc lui ajouter une SdD contenant leur identification. Dans le cas où les sommets sont numérotés de '`0`' à '`N-1`' pour un graphe de N sommets, il suffit d'avoir une variable `int nbSommets` comme utilisé précédemment.

Il existe bien d'autres façons de représenter un graphe en mémoire. Comme pour toute représentation, il n'y a pas de limite à l'imagination, si ce n'est la facilité et l'efficacité du code qui l'utilise !