ALGORITHMIQUE FIIFO1

2000-2001

MÉRÉ Aurélien

TABLE DE MATIERES

Chapitre 1 : Introduction

Bibliographie

Aho, Hopcroft, Ullman « Structures de données et algorithmes », Intereditions 1993
Carrez, « Des structures aux bases de données », Dunoe 1990
Fournier J-P. « Passeport pour l'algorithmique objet », Vuibert 1998
Froidevaux, Gaudel, Soria, « Types de données et algorithmes », McGramHill 1990

Partie 1 : Algorithmique

Algorithme : Ensemble de règles opératoires, d'actions élémentaires à exécuter, que l'ordinateur appliquera sur des données jusqu'à l'obtention de la solution cherchée en un nombre fini d'opérations pour résoudre un problème.

Algorithmique : Problème du choix de l'algorithme et étude préliminaire de l'algorithme avant sa mise en forme sous forme de programme

Etude préliminaire : Le but est de préciser les actions à entreprendre sans entrer dans les détails de l'implémentation, et de vérifier la correction vis-à-vis du problème, c.a.d qu'il donne bien la solution attendue, en un temps fini. On étudie alors la complexité en temps et en espace mémoire

Spécification : Décrire ce que fait l'algorithme sans détailler la manière employée. On précise les valeurs d'entrée et de sortie.

Tester : Proposer des jeux de tests en justifiant le choix des exemples.

Partie 2 : Complexité

Complexité en temps

Ce qui nous intéresse, c'est l'ordre de grandeur du temps d'exécution d'un algorithme et son évolution en fonction de la taille des données. On veut comparer les algorithmes résolvant un même problème, pour des données de grande taille, indépendamment du langage de programmation et de la machine. Il faut déterminer les opérations fondamentales qui interviennent dans ces algorithmes et qui sont telles que le temps d'exécution est proportionnel au nombre d'opérations fondamentales.

Il y a trois cas :

- Cas pire : on maximise le nombre d 'opérations fondamentales
- Cas optimal : on minimise le nombre d'opérations fondamentales
- Moyenne : Fait appel à un modèle probabiliste

Complexité en espace

On va évaluer la place mémoire nécessaire pour l'exécution de l'algorithme (pour stocker les données sans prendre en compte l'espace occupé par les instructions du programme).

Chapitre 2 : Structures séquentielles

Listes

Une liste linéaire est une suite finie, éventuellement vide, d'éléments repérés selon leur rang dans la liste. L=<e_1, e_2, …, e_n>. On note aussi L=(e_1, e_2, …, e_n). On parle de n-uplets.

L'ordre sur les places des éléments est fondamental, c'est un ordre total (deux éléments quelconques peuvent être comparés). Il existe une fonction successeur qui permet d'atteindre n'importe quelle place à partir de la première. La première place s'appelle la tête. La $n^{ième}$ place s'adresse par $succ^{k-1}$(tete(L)) soit l'itérée (k-1) de la fonction succ. Le nombre de places de L est la longueur de L.

Schéma de traitement usuel : Examen séquentiel dans l'ordre des places de toutes les places d'une liste pour faire un même traitement sur chaque élément. (Remarque : l'élément est contenu dans la place).

```
X := tete(L) ;
Traiter (X) ;
Pour i de 1 jusqu'à Longueur(L) faire
     X :=succ(X) ;
     Traiter (X) ;
Fin Pour
```

Représentation des listes

- Représentation contiguë

La liste est représentée par un tableau dont la i-ème case est la i-ème place de la liste. On sur-dimensionne le tableau : on réserve un nombre de cases supérieur à la longueur de L. Pour savoir quels sont les éléments de la liste, il faut connaître explicitement la longueur de la liste.

- Pointeurs : Gestion dynamique de la mémoire

Bien souvent, on a besoin de gérer dynamiquement l'espace mémoire : certains objets, particulièrement encombrants, ne sont plus nécessaires à la fin d'une procédure et on a envie de s'en débarrasser. Dans tous les langages de programmation, il y a possibilité de réserver (allouer) des emplacements de mémoire vierge, et la capacité de les rendre (libérer).

Les objets gérés par le programmeurs sont anonymes. Pour savoir les utiliser (modifier leurs valeurs, etc…) il faut savoir où ils sont rangés. Pour cela, on dispose d'objets appelés pointeurs, qui fournissent les adresses où sont rangés les objets.

- Représentation chaînée d'une liste séquentielle

On utilise des pointeurs pour chaîner entre eux les différents éléments de la liste et la liste est déterminée par l'adresse du premier élément. L est une variable de type pointeur qui pointe sur un enregistrement à deux champs : le premier champ contient l'élément, et le second contient l'adresse de l'élément suivant (pointeur).

Exemple ADA

```
Type CELLULE ;
Type LISTE is access CELLULE ;
Type CELLULE is record
      Valeur : ELEMENT ;
      Suivant : LISTE ;
End Record ;
```

1- Définition en avant du type CELLULE nécessaire
2- Définition du type LISTE récursif

Les objets accédés (ici les cellules) sont créés par l'exécution d'un allocateur qui peut fournir éventuellement une valeur initiale. Par défaut le champ pointeur vaut le pointeur NULL noté **Ø**

Allocation dynamique en ADA

```
L := new CELLULE'(37, NULL) ;
```

- Algorithme itératif d'insertion d'un élément à la k^{ième} place dans une liste chaînée

```
Procedure Inserer ( L: liste{ES}; k: entier{E}; x: élément{E};
                    erreur: booléen{ES})

var     P, C, PP : LISTE ;
        i : entier ;

début
     si (k=1) alors
          erreur := faux ;
          c := allouer(CELLULE) ;
          c^.valeur := x ;
          c^.suivant := L ;
          L := C ;
     Sinon
          I := 1 ; erreur := faux ;
          Tant que (i < k) et non(erreur) faire
                Si (P=NULL) alors
                      Erreur := VRAI ;
                Sinon
                      PP := P ;
                      P := P^.suivant ;
                      I := i + 1 ;
                Fin si
          Fin Tant que
          Si non(erreur) alors
                C := allouer (CELLULE) ;
                C^.valeur := x ;
                C^.suivant := P ;
                PP^.suivant := C ;
          Fin si
     Fin si
Fin
```

On compte opération fondamentale l'instruction P := P^.Suivant.

- Autres types de représentation chaînée des listes

Les listes circulaires : On remplace dans la dernière place de la liste le pointeur NULL par un pointeur vers la tête de la liste. On utilise un pointeur dernier qui permet de récupérer le dernier élément, et en conséquence le premier avec le lien suivant.

Les listes doublement chaînées : On utilise deux pointeurs, l'un pointant sur l'élément suivant, et l'autre pointant sur l'élément précédent.

- Comparaison entre deux types de représentation

1- Espace mémoire
2- Complexité des procédures d'insertion et suppression
3- Longueur
4- Facilité d'accès au $k^{ième}$ élément

- Type liste considéré de façon récursive

Une liste est alors considérée comme une tête de liste suivie d'une liste plus petite appelée corps de liste. Une liste réduite à un élément est composée d'une tête de liste contenant n suivi d'un corps de liste qui est une liste vide. On a ainsi :
Tete(L) : Accès au premier élément de L
Corps(L) : Liste L dans laquelle on a supprimé le premier élément.
L'accès au $k^{ième}$ élément de L correspond donc à : tete(corps$^{(k-1)}$(L))

Piles et files

Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont les accès, insertions et suppressions aux extrémités.

- Piles

Les insertions et suppressions se font à une seule et même extrémité, appelée sommet de la pile. Une pile a une structure LIFO (Last-in First-out). Les opérations élémentaires sur les piles sont :

1- Tester si une pile est vide
2- Empiler
3- Récupérer le sommet
4- Dépiler

- Files

On fait les adjonctions à une extrémité et les suppressions à l'autre extrémité. On peut faire une analogie avec les files d'attentes. Une file a une structure FIFO (First-in First-out).

Chapitre 3 : Récursivité

Dans la plupart des langages de programmation, on peut écrire des fonctions récursives. Si on veut gérer la récursivité comme un compilateur, il faut gérer des piles contenant :

- les valeurs courantes des paramètres de la procédure
- les valeurs courantes des variables de la procédure
- une indication de l'adresse de retour

Algo récursif vs Algo itératif

Pour certains problèmes la version récursive est plus simple à concevoir, ce qui ne veut pas dire de meilleure complexité. C'est le cas de certaines fonctions mathématiques qui se définissent naturellement récursives : factorielle n, suite de Fibonacci.

Les algorithmes récursifs ont un certain coût car il faut construire un nouvel espace de travail à chaque appel récursif. Si dans l'espace de travail on ne manipule que peu de variables, le coût sera réduit.

Tous les langages de programmation n'acceptent pas la récursivité. Ils permettent tous d'écrire des programmes itératifs qui permettent de résoudre tous les problèmes.

Les algorithmes récursifs donnent toute la mesure de leur efficacité lorsqu'ils sont employés avec des structures de données récursives, telle la liste chaînée.

Conditions de terminaison

La condition de terminaison doit être prouvée. Deuxième exemple avec l'algorithme de recherche dichotomique d'un élément X dans une liste représentée par un tableau t entre les bornes g et d. La liste doit être triée au préalable.

```
Procedure Dicho (X : ELEMENT {E}, T : TABLEAU {E}, g, d : ENTIERS {E})
Variable m :entier ;
Début
        Si (g <= d) alors début
              M := (g+d) div 2 ;
              Si (X=T[M]) alors retourne M ;
              Sinon
                    Si (X<T[M]) alors retourne dicho(X, t, g, m-1) ;
                                sinon retourne dicho(X, t, m+1, d) ;
                    fin si
              fin si
        fin ;
        retourne 0 ;
fin ;
```

Montrons que la procédure se termine toujours : il y a toujours un nombre fini d'appels récursifs. On termine soit avec un retour différent de 0, cas où la recherche aboutit, soit avec un résultat nul, cas où la recherche échoue.

Au premier appel, les bornes de l'intervalle sur lequel on travaille sont g=g0 et d=d0. On va montrer que s'il y a appel récursif, on va travailler sur [g1, d1] inclus dans [g0, d0].

Si $g_k > d_k$, il n'y a pas d'appel récursif et le résultat est nul. Dans le cas contraire, on a soit $X=T[m_k]$ où $m_k = (g_k + d_k)$ div 2, auquel cas il n'y a pas d'appel récursif, soit on a $X<T[m_k\}$ auquel cas, il y a appel récursif. On travaille alors après cet appel sur l'intervalle $[g_{k+1}, d_{k+1}]$ où $g_{k+1} = gk$ et $d_{k+1}=m_{k-1}$

On en déduit que $d_{k+1} - g_{k+1} < d_n - g_k$.

Dérécursification

On appelle récursion finale ou terminale le cas où l'appel récursif dans l'algorithme n'est suivi d'aucune instruction. Il n'y a pas de traitement à faire après chacun des appels.

U : Liste de paramètres
C : Condition portant sur U
T : Traitement de base portant sur U

Schéma récursif

```
Procedure P(U)
Début
        Si C alors début
                D ;
                P(f(u))
                Fin
        Sinon T;
        Fin si;
Fin
```

Schéma itératif correspondant

```
Procedure P(U)
Variable v;
Début
        v := u ;
        TantQue C faire
        début
                D ;
                V:=f(v)
        Fin
Fin
```

Chapitre 4 : Structures arborescentes

Définition

Un arbre binaire B est soit vide (noté Ø), soit de la forme B=<O, B_1, B_2>O si B_1 et B_2 sont deux arbres binaires et O est un nœud appelé racine. B_1 est le sous-arbre gauche et B_2 le sous-arbre droit.

Lorsque l'on met des valeurs aux nœuds, on parle d'arbres binaires étiquetés.

Représentation chaînée des arbres binaires

```
Type NŒUD ;
Type ARBRE : pointeur (NŒUD) ;
Type NŒUD : enreg(val : ELEMENT, g, d : ARBRE)
```

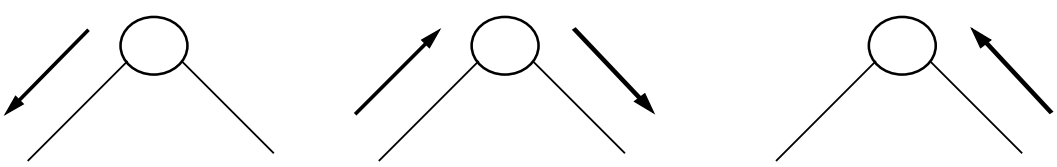
Le fils gauche d'un nœud est la racine de son sous-arbre gauche. On dit qu'il y a un lien gauche du nœud vers sont fils gauche. Si le nœud n'a pour fils gauche n, il n'a pour père n.

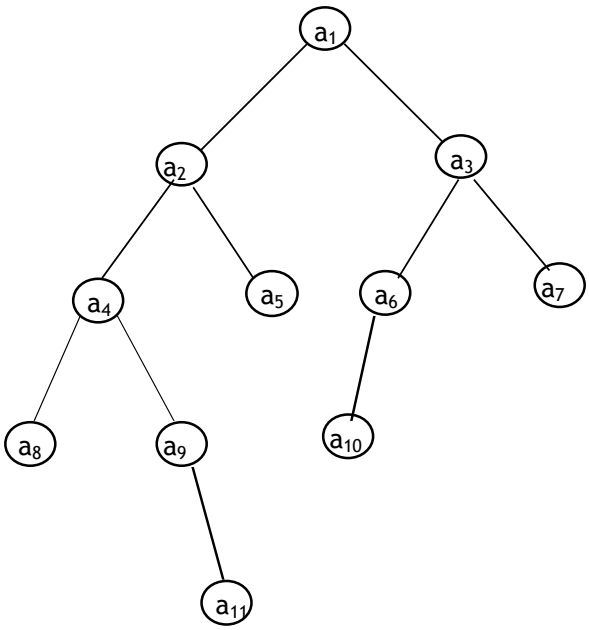
Un nœud a un seul père mais peut avoir 0, 1 ou 2 fils. Un nœud sans fils est appelé feuille. Un nœud qui a deux fils et appelé nœud interne. Un nœud qui a un seul fils est un nœud simple. Un chemin dans un arbre binaire est une suite de nœuds consécutifs. On appelle branche tout chemin de la racine à une feuille. La hauteur d'un nœud n est le nombre de liens sur l'unique chemin qui va de la racine à n.

Parcours d'un arbre binaire

Le but est de visiter tous les nœuds pour effectuer un certain traitement sur chaque nœud. On appelle ce parcours « parcours à main gauche » : on visite chaque nœud trois fois. Le cas terminal implique un traitement spécifique appelé TERM.

```
Procedure Parcours (A : Arbre {E})
Début
       Si A=Ø alors TERM;
       Sinon Début
             T1 ;
             Parcours (sg(a)) ;
             T2 ;
             Parcours (sd(a)) ;
             T3 ;
             Fin ;
       Fin Si ;
Fin ;
```





Le parcours de cet arbre complet, si l'on considère que T1, T2 et T3 affichent le nœud en cours de traitement, et si TERM affiche le caractère #, on obtient à l'écran la chaîne suivante :

$a_1a_2a_4a_8\#a_8\#a_8a_4a_9\#a_9a_{11}\#a_{11}\#a_{11}a_9a_4a_2a_5\#a_5\#a_5a_2a_1a_3a_6a_{10}\#a_{10}\#a_{10}a_6\#a_6a_3a_7\#a_7\#a_7a_3a_1$

Si l'on effectue uniquement le traitement T1, le parcours se fait par ordre préfixe : la liste des étiquettes des nœuds visités sont rangés par ordre préfixe.

$a_1a_2a_4a_8a_9a_{11}a_5a_3a_6a_{10}a_7$

Si l'on effectue uniquement le traitement T2, le parcours est effectué par ordre symétrique : $a_8a_4a_9a_{11}a_2a_5a_1a_{10}a_6a_3a_7$
Si l'on effectue uniquement le traitement T3, le parcours est réalisé par ordre post-fixe : $a_8a_{11}a_9a_4a_5a_2a_{10}a_6a_7a_3a_1$

Propriétés mathématiques des arbres binaires

Soit A un arbre binaire de n nœuds. Soit h la hauteur de cet arbre.

Arbre très compact <= h <= n-1 : Arbre filiforme dégénéré

$\text{Log}_2\ n <= h$

Un arbre binaire localement complet est un arbre binaire tel que tout nœud a 0 ou 2 fils.

Proposition : Un arbre binaire localement complet à n nœuds internes a (n+1) feuilles.

Chapitre 5 : Tries

On cherche une structure de données pour stocker un ensemble de « mots », ici des chaînes de caractères. On voudra effectuer de façon économique des opérations de base (recherche, adjonction, suppression).

Exemple : Chaîne de chiffres

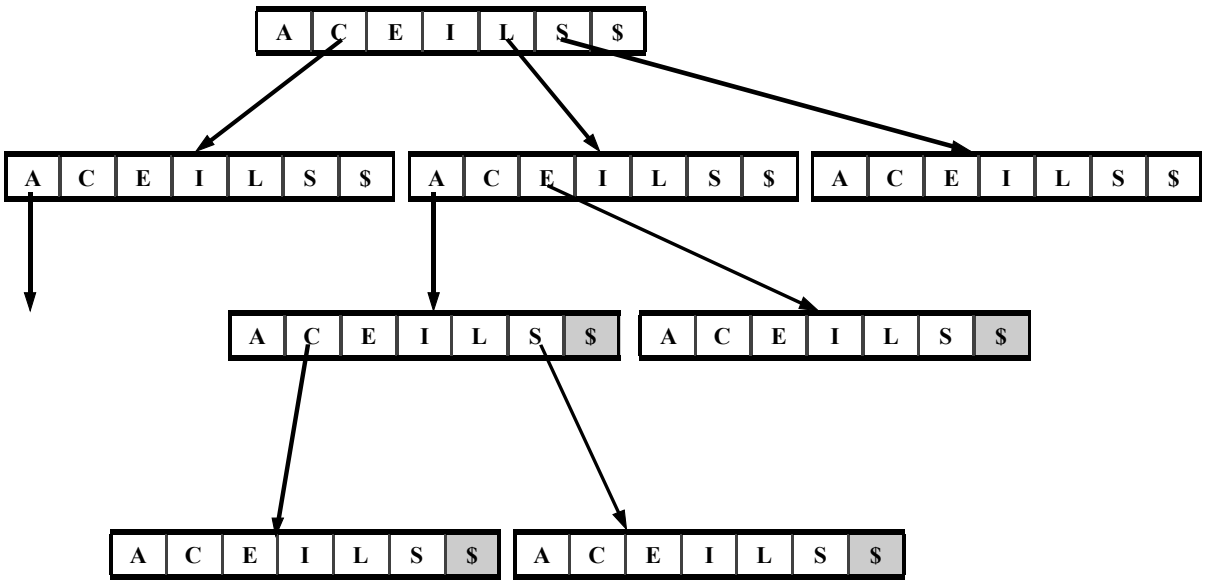
$$\Sigma=\{0,1,2,...,9,',',\}$$

$$S=\left\{\sqrt{2};\ 1/\ln 2;\ \frac{1+\sqrt{5}}{2};\ln 10;e;\pi\right\}=\{1{,}414;1{,}44;1{,}6;2{,}3;2{,}718;3{,}14\}$$

On utilise un symbole spécial pour indiquer la fin d'un mot. Pour une chaîne de caractères, on utilisera le symbole $.

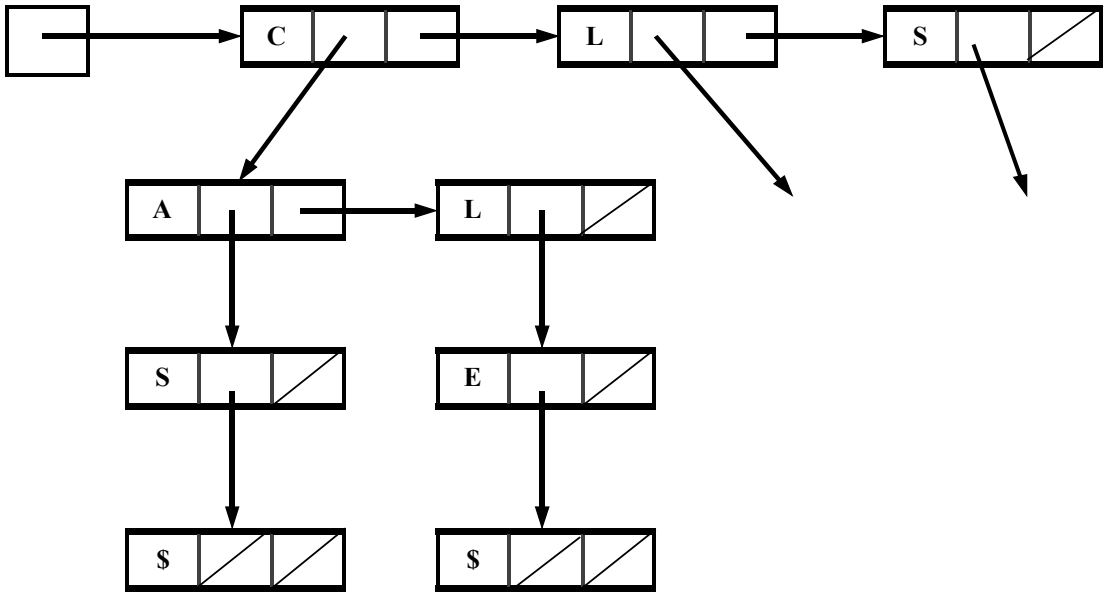
1ère représentation : Tableau indexé par Epsilon, de pointeurs sur des tableaux.

Exemple : $\Sigma=\{A,C,E,I,L,S,\$\}$, $S=\{LE,CAS,CLE,SAC,SEL,LAC,LA,lAS,SIC,SI\}$



On utilise des tableaux de 7 cases ayant pour valeurs des pointeurs sur des tableaux. Beaucoup d'espace est utilisé si les mots ont peu de préfixes communs. Ainsi, avec 10 mots de longueur inférieure ou égale à 4, on a au maximum 40 cases.

2ème représentation : On chaîne entre eux les sommets et on utilise un arbre n-aire (représentation frère droit, fils ainé). Toute liste contient tous les nœuds qui ont même père.




```
type Dico : pointeur(triplet) ;
type Triplet : enreg(c : char, fd : dico, fa : dico) ;
type Mot : tableau(1..lgmax, char) ;
```

On va écrire une procédure récursive d'insertion d'un mot représenté par le tableau T dans le dictionnaire.

```
Procedure insert (D : dico {E/S}, T : mot {E}, i : entier {E})
Début
   Si D=NULL alors
         D := Allouer (triplet)
         D^.c := T[i]; D^.fa := NULL; D^.fd := NULL;
         Si T[i]<>'$' alors insert(D^.fa, T, i+1) ;
   Sinon
         Si T[i]=D^.c alors
               Si T[i]<>'$' alors insert (D^.fa, T, i+1) ;
         Sinon
               Insert(D^.fd, T, i) ;
         Fin si ;
   Fin Si ;
Fin ;
```

Chapitre 6 : Graphes

Définition sur les graphes

Objets : sommets, arcs (cas orienté), arètes (cas non orienté)

R relation binaire sur S : R <= S*S $(x,y)\in\Re,x\in S,y\in S$

Propriétés d'une relation binaire R sur S*S

1) réflexive : $\forall x\in S,xRx$
2) symétrique : $\forall x\in S,\forall y\in S,[xRy\Rightarrow yRx]$
3) transitive : $\forall u\in S,\forall v\in S,\forall w\in S,[uRv,vRw]\Rightarrow uRw$

Une relation binaire qui est réflexive symétrique et transitive est appelée relation d'équivalence.

Un graphe orienté G est un couple <S, A> où S est un ensemble fini de sommets. A est un ensemble fini de paires ordonnées d'éléments de S. A <= S*S

Soit $(u,v)\in A\leq S\times S$. On note $u\rightarrow v$. On dit que u est l'origine de l'arc, v l'extrémité. V est un successeur de U. U est un prédécesseur de V.

Dans un graphe, un sommet donné peut avoir plusieurs successeurs et plusieurs prédécesseurs. On appelle chemin dans un graphe G orienté G=<S, A> une suite de sommets : s1…sn tel que s_1->s_2,…, s_{k-1}->$s_{k.}$ sont les arcs de k.

Un graphe non orienté G=<S, A> est un graphe orienté tel que sa relation binaire est symétrique.

Un graphe orienté valvé G=<S, A, C> est un triplet avec S l'ensemble des sommets, A l'ensemble des arcs et C une fonction de coût.

Représentation des graphes

Représentation matricielle

Soit G=<S, A> un graphe orienté de n sommets et p arcs. (0 <= p <= n^2). On représente par une matrice carrée M de taille n*n.

1) Cas non valvé

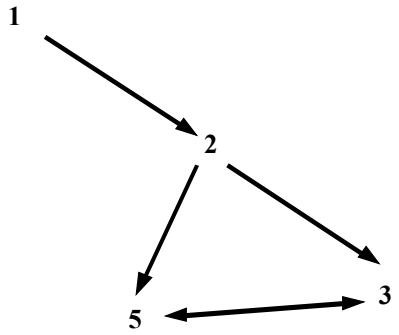
On utilise une matrice de booléens M[i, j]. M[i, j]=faux s'il n'y a pas d'arcs et M[i, j]=vrai s'il y a un arc.

2) Cas valvé

M[i, j]=cout i->j si l'arc existe M[i, j]=∞ si une valeur spécifique ne peut pas être le coût d'un arbre. M est une matrice carrée de taille n*n à valeurs réelles.

Représentation par tableau de sommets et listes chaînées de successeurs

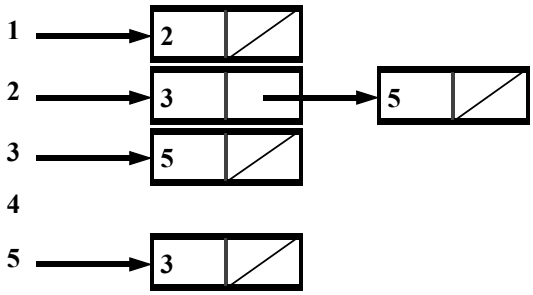
A chaque sommet, on associe la liste de ses successeurs. G=<S, A>. S={1,2,3,4,5} et A={(1,2), (2,3), (3,5), (2,5), (5,3)}.



Matrice booléenne

F	V	F	F	F
F	F	V	F	V
F	F	F	F	V
F	F	F	F	F
F	F	V	F	F

Représentation en tableau de sommets et liste de successeurs



Remarque :

signifie que le sommet 2 a deux successeurs, le sommet 3 et le sommet 5. On a représenté les deux arcs 2->3 et 2->5.

```
Type sommet ;
Type liste_sommet : ptr(sommet) ;
Type sommet : enreg(val (1..n), suivant : liste_sommet) ;
Type tab_sommet : tableau(n, liste_sommet) ;
```