Algorithmique des:

- **Listes**
- **Arbres**
- **Graphes**

I.S.I.A.

COURS 2

Franck.Nielsen@sophia.inria.fr

Année 1995-1996

INRIA SOPHIA ANTIPOLIS

Introduction

– Algorithmes+Structures de données=Programme

Un programme informatique est constitué d'un algorithme et de structures de données manipulées par l'algorithme.

– **Structure de données**: Organiser les données.

– →Choix crucial pour l'analyse de la complexité.

– Ce cours étudie:

  – Les listes ou structures linéaires.

  – Les arbres ou structures arborescentes.

  – Les graphes ou structures relationnelles.

– Deux algorithmes sur les graphes:

  1. Recherche du plus court chemin
  2. Tri topologique (Extension linéaire)

Structure de données&type abstrait

→Représentation organisée de données **typées** (scalaire, produit, union). Types primitifs (Entier, Booleen, ...)
→Valeur prise dans un ensemble totalement défini de possibilités.
On caractérise une structure de donnée par:

- Une variable la décrivant (notation).
- Un constructeur qui structure des données de type plus "simple".
- Opérations accessibles sur cette structure de données.
- Propriétés logiques de ces opérations.

→Type de données **abstrait** permettant la généricité.

Exercice 1 *Décrire le type* PILE *qui permet de structurer des éléments de type* E. *Les opérations sont* **Vide**, **Empiler**, **Dépiler**, **Nombre_Elements**, ...

Les listes

→Forme simple (linéaire) d'organiser les données. Utilisée dans la vie quotidienne (exemples?)
D'un point de vue formel, une liste est une suite finie d'éléments d'un même ensemble (de même type construit).
→Application de **N** dans **E**

Exercice 2 *Donner des listes et leur fonction de* $\mathbf{E}^{\mathbf{N}}$. *Quelles sont les opérations de base que l'on souhaiterait avoir sur les listes?*

Le type abstrait LISTE

Soit $\mathcal{L}$ l'ensemble des listes à éléments de $\mathcal{E}$.
→LISP est un langage fonctionnel reposant sur la manipulation des listes.

- On note $\emptyset$ la liste de $\mathcal{L}$ vide.
- On classe les opérations sur les listes en deux catégories: celles qui laissent inchangée $L \in \mathcal{L}$ et celles qui modifient L:

  Opérations de consultation. Longueur, **Accés** qui accéde à un élément de $\mathcal{E}$: **Accés**(L, i).

  $$\text{Accés} : \mathcal{L} \times \mathbf{N}^* \rightarrow \mathcal{E}$$

  Opérations de modification. Insérer $(\mathcal{L} \times \mathbf{N}^* \times \mathcal{E} \rightarrow \mathcal{L})$, **Supprimer**$(\mathcal{L} \times \mathbf{N}^* \rightarrow L)$.

Propriétés logiques d'un opérateur: par exemple l'opérateur **Longueur**.

Axiomes de Longueur

$$\forall L \in \mathcal{L}\backslash\{\emptyset\}, \forall k, 1 \leq k \leq Longueur(L),$$
$$Longueur(Supprimer(L,k)) = Longueur(L) - 1$$

$$\forall L \in \mathcal{L}, \forall E \in \mathcal{E}, \forall k, 1 \leq k \leq Longueur(L) + 1,$$
$$Longueur(\text{Insérer}(L,k,E)) = Longueur(L) + 1$$

Exercice 3 *Donner Les axiomes caractérisant les fonctions* **Accés** *et* **Insérer**.

Axiomatique

$$\forall L \in \mathcal{L}, \forall E \in \mathcal{E}, \forall k, 1 \leq k \leq Longueur(L) + 1,$$

on a:

$$\forall i, 1 \leq i \leq k, \text{Accés}(\text{Insérer}(L, k, E), i) = \text{Accés}(L, i)$$

$$\text{Accés}(\text{Insérer}(L, k, E), k) = E$$

$$\forall i, k < i \leq Longueur(L) + 1, \text{Accés}(\text{Insérer}(L, k, E), i) = \text{Accés}(L, i - 1)$$

$$\forall L \in \mathcal{L}, \forall k, 1 \leq k \leq Longueur(L),$$

on a:

$$\forall i, 1 \leq i \leq k, \text{Accés}(\text{Supprimer}(L, k), i) = \text{Accés}(L, i)$$

$$\forall i, k \leq i \leq Longueur(L) - 1, \text{Accés}(\text{Supprimer}(L, k), i) = \text{Accés}(L, i + 1)$$

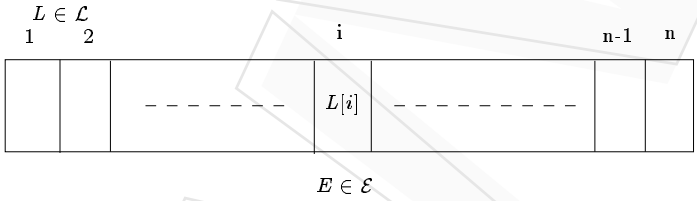
$\rightarrow$Type abstrait algébrique

Représentation des Listes

→nombreuses façons de représenter les listes sur une machine. Les 2 plus répandues sont:

– par un tableau (tableau éventuellement circulaire):

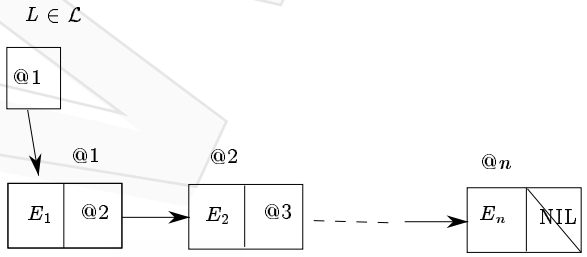
type liste= **array**[1..n] **of** Element;



– par une liste chaînée:

type liste= ↑cellule;

cellule=**record**
val:Element;
suivant:liste
end;



Comparatifs

	Tableau	Liste Chaînée
Avantages	– Meilleure utilisation de la mémoire (→stockage contigue) – Accés direct au i-ième élément $L[i]$ – Parcours séquentiel grâce à l'indice i – Algorithmique du tableau basé sur la manipulation des indices.	– Place allouée en fonction de $Longueur(L)$. – Insertion/Suppression facile de $E \in \mathcal{L}$.
Inconvénients	– La taille du tableau doit être connue à l'avance – Suppression ou insertion d'un élément coûteux	– Utilisation de pointeurs – L'insertion d'un élément demande une place mémoire – Accés au i-ième élément se fait par $i-1$ chainages

Taxonomie des Listes

→Il existe plusieurs variantes des listes:

- Mixte: On chaine des tableaux (l'élément d'une cellule est un tableau)
- Double chaînage: une cellule pointe sur son prédecesseur et sur son successeur. Permet d'obtenir deux sens de parcours.
- Chaînage circulaire: La dernière cellule pointe sur la première cellule. Plus d'élément **tête** et **queue**. →il faut un compteur pour comptabiliser le nombre d'éléments. Permet de parcourir aisément les listes.

Exercice 4 *Donner une représentation graphique de ses structures de données ainsi que leur déclaration en tant que type générique (le type abstrait algébrique étant toujours celui de* Liste*).*

Implantation

```
function Longueur(L:liste):integer;
begin
if L=nil then Longueur:=0
         else Longueur:=1+Longueur(L↑suivant);
end;
```

```
function Accés(L:liste;i:integer):Element;
begin
if (i = 1 or L = nil) then
                          if L=nil then writeln('Erreur');
                                   else Accés:=L↑val;
                    else Accés:=Accés(L↑suivant,i-1);
end;
```

Exercice 5 *Dérecursivez les fonctions ci-dessus et donner une implantation de Insérer(L, i, E) et Supprimer(L, i).*

Gestion d'une PILE

→On implante les opérations élémentaires de la Liste $L \in \mathcal{L}$ par une pile (**Last In First Out** – **LIFO**).
→Les opérations **Insérer** et **Supprimer** sont restreintes.

Les opérations élémentaires de la pile sont **Empiler**, **Dépiler** et **Sommet**. Soit P une pile de $\mathcal{P}$ et $E \in \mathcal{E}$ un élément. Les axiomes correspondant au type algébrique PILE sont les suivants:

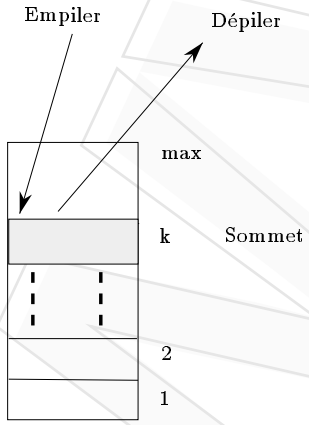
$$\text{Sommet}(\text{Empiler}(P, E)) = E$$

$$\text{Dépiler}(\text{Empiler}(P, E)) = P$$

En langage de type PASCAL, on peut décrire le type PILE comme suit:

```
type pile=record
            s:integer;
            P:array[1..Max] of Element
            end;
```


Implantation




```
procedure Sommet(p:PILE;var E:element);
begin
if p.s = 0 then writeln('Pile vide');
           else E := p.P[s];
end;
```

Implantation(suite)

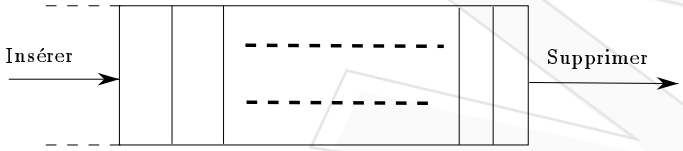
```
procedure Empiler(var p:PILE; E:element);
begin
if p.s = Max then writeln('Pile pleine');
                 else
                      begin
                      p.s := p.s + 1;
                      p.P[p.s] := E;
                      end;
end;
```

```
procedure Dépiler(var p:PILE);
begin
if p.s = 0 then writeln('Pile vide');
           else
           p.s := p.s - 1;
end;
```


```
#include <iostream.h>
// Fichier en-tete pour utiliser flux I/O


#define max 500

#define OK 2
#define PILE_VIDE 1
#define PILE_PLEINE 0

template <class Element> class pile{
private:
int n;
Element P[max];


public:
    pile(){n=-1;};
    ~pile(){delete P;}
int   Sommet(Element &E){if (n>-1) {E=P[n];return OK;} else
 return PILE_VIDE; }
int Depiler(){if (n>=0) {n--;return OK;} else return PILE_VIDE;}
int Empiler(Element E){if (n<max-1) {P[++n]=E;return OK;} else
 return PILE_PLEINE;}
};

int main()
{
pile<int> P;
int E;
int i,nombre;

cout <<"J'ai cree une pile d'entier"<<endl;
for(i=0;i<max;i++) (void)P.Empiler(i);
cout <<"Je depile combien d'elements?";
cin >>nombre;
for(i=0;i<nombre;i++) (void)P.Depiler();
if (P.Sommet(E)==PILE_VIDE) {cout<<"Pile vide!"<<endl;}
else {
    cout <<"Le Sommet est "<<E<<endl;}
return 0;
}
```


Gestion d'une FILE

→Structure **FIFO** (*First In First Out*). On insére à une extrêmité de la liste et on supprime à l'autre extrêmité.



Soit $F \in \mathcal{F}$ une file et $E \in \mathcal{E}$ un élément.
Opérations élémentaires:

Ajouter. Ajoute E à la queue de F.

Retirer. Retire E de le tête de F.

Premier. Accéde à l'élément de tête de F.

Les axiomes principaux sont:

- Si $F \neq \emptyset$ alors
  - Premier(Ajouter(F, E)) = Premier(F)
  - Retirer(Ajouter(F, E)) = Ajouter(Retirer(F), E)
- Si $F = \emptyset$ alors
  - Premier(Ajouter(F, E)) = E
  - Retirer(Ajouter(F, E)) = $\emptyset$

Implantation

Exercice 6 *Implanter une* FILE *à l'aide d'un tableau, d'une liste chainée. Donner des problèmes où les piles&files aident à implanter efficacement une solution.*

```
program File;
(* Programme implantant une FILE grace a une liste
 doublement chainee *)

const n=500;

type Element=integer;

type liste=^cellule;

cellule=record
   val    : Element;
   suivant,precedant : liste;
end;

var
   tete,queue : liste;


(* par defaut renvoit 0 si la file est vide *)
function Premier:Element;
begin
   if (tete<>nil) then Premier:=tete^.val; else Premier:=0;
end; { Premier }
```

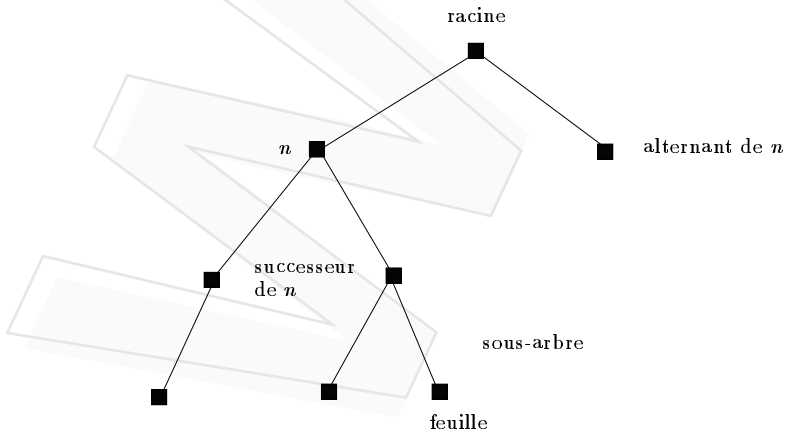
```
procedure Ajouter(E : Element);
var
   c : liste;
begin
   new(c); (* allocation memoire *)
   c^.val:=E;
   if (tete=nil) then begin
      c^.suivant:=nil;c^.precedant:=nil;
      tete:=c;
      queue:=c;
   end
   else
   begin
      c^.precedant:=nil;
      c^.suivant:=queue;
      queue^.precedant:=c;
     queue:=c;
   end;

end; { Ajouter }


procedure Retirer;
var
   atete : liste;
begin
   if (tete<>nil) then
   begin
      atete:=tete^.precedant;
      dispose(tete);
      if (atete<>nil) then atete^.suivant:=nil;
      tete:=atete;
   end;
end; { Retirer }
```

Les arbres – ARBRE

→Structure de données extrêmement utilisée qui permet de *rechercher*, *insérer* ou *supprimer* des éléments en $O(\log n)$ où n est le nombre d'éléments stockés dans les feuilles.
Un arbre est un ensemble d'éléments appelés nœuds organisés de façon hiérarchique à partir d'un nœud distingué appelé *racine*. C'est un *graphe connexe acyclique maximal*.

- La **racine** ne posséde pas de père.
- Un **nœud interne** est un nœud possédant des fils.
- Une **feuille** est un nœud sans fils.

Soit n un nœud interne. On appelle **successeurs** ses fils et **alternants** ses **frères**.



Arbre binaire - n-aire

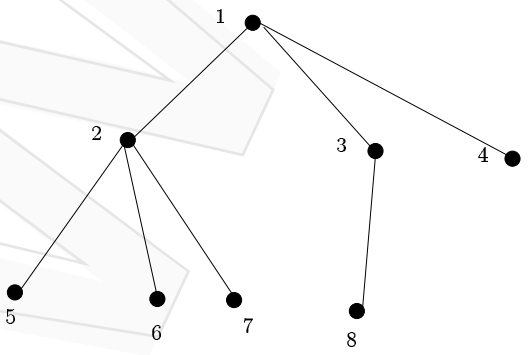
Un **sous-arbre** est un arbre comprenant la hiérarchie à partir d'un nœud.
Le degré sortant d'un nœud est son nombre de fils. L'arité d'un arbre est le maximum des degrés sortants des nœuds le composant.
→Arbre binaire (d'arité 2), n-aire.

Si les fils d'un nœud sont ordonnés, on appelle *successeur* le fils le plus à gauche et *alternant* le frére le plus à gauche. On peut définir un tel ordre pour tout arbre.

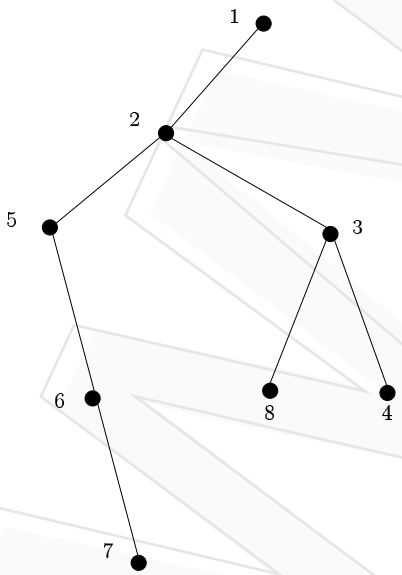
Bijection entre les arbres binaires et n-aires [Knuth]

- Le fils gauche de n est le successeur de n.
- Le fils droit de n est l'alternant de n.

Arbre ternaire



Bijection



Exercice 7 *Décrire une structure de donnée qui code les arbres binaires, n-aires. Écrire un algorithme qui convertit un arbre binaire en arbre n-aire et réciproquement. Quel est le coût de votre algorithme? Est-il asymptotiquement optimal?*

→On priviligie l'étude des arbres binaires dans la suite.

Arbres Binaires

→Définition récursive ensembliste des arbres binaires $\mathcal{B}$.

$$\mathcal{B} = \{\emptyset\} \bigcup (\cup_n < n, \mathcal{B}, \mathcal{B} >),$$

avec n un nœud et $\emptyset$ l'arbre vide. $< n, \mathcal{B}_g, \mathcal{B}_d >$ est l'arbre de racine n avec comme sous-arbre gauche l'arbre $\mathcal{B}_g$, et comme sous-arbre droit l'arbre $\mathcal{B}_d$.
La *hauteur* d'un nœud x est le nombre d'arêtes de l'unique chemin liant la racine à x. La hauteur d'un arbre est le maximum de la hauteur de ses nœuds.
On a:

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Un arbre est **strictement binaire** ssi tous les nœuds internes ont exactement deux fils.
→Un arbre strictement binaire de n nœuds internes à $n+1$ feuilles.
Le nombre b_n d'arbres binaires à n noeuds est

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

n-ième nombre de Catalan.

Preuve par les séries génératrices

On considère $\mathcal{B} =< r, \mathcal{B}_g, \mathcal{B}_d >$ avec $|\mathcal{B}| = n$ et $\mathcal{B}_g = k$, $\mathcal{B}_d = n - 1 - k$.

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-k-1}$$

Utilisons la série génératrice $b(z) = \Sigma_{n\geq 0}\, b_n z^n$. Alors,

$$b(z) = 1 + \sum_{n\geq 1} \left(\sum_{k=0}^{n-1} b_k b_{n-1-k} \right) z^n$$

$$b(z) = 1 + z \sum_{n\geq 1} \left(\sum_{k=0}^{n-1} b_k b_{n-1-k} \right) z^{n-1}$$

mais $\Sigma_{n\geq 1} \left(\Sigma_{k=0}^{n-1}\, b_k b_{n-1-k} \right) z^{n-1} = b^2(z)$.

$$b(z) = 1 + z b^2(z)$$

On a $b(z) = \frac{1-\sqrt{1-4z}}{2z}$
On utilise alors l'égalité:

$$(1+x)^\alpha = 1 + \sum_{k\geq 1} \frac{\alpha(\alpha-1)\ldots(\alpha-k+1)}{k!} x^k$$

Ce qui méne au résultat □

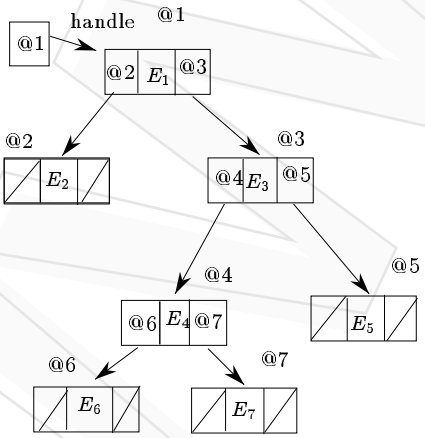
Représentation des arbres binaires

Plusieurs représentations possibles. Les plus utilisées sont:

- Les cellules chaînées grâce aux pointeurs.
- Le tableau

```
type arbre=↑noeud;
nœud=record
        val:Element;
        gauche,droite: arbre;
        end;
```

```
type nœud=record
            val:Element;
            gauche,droite: integer;
            end;
arbre=array[1..n] of noeud;
```



	Valeur	gauche	droite
1			
2			
3			3
4	E_4	2	
5			

Parcours d'arbres binaires

Parcourir c'est visiter tous les nœuds de l'arbre suivant une stratégie. On distingue: le parcours *préfixe*, *suffixe* et *infixe*.
Soit Visite($\mathcal{B}$) une procédure de visite de l'arbre binaire $\mathcal{B} =< r, \mathcal{B}_g, \mathcal{B}_d >$ et Affichage($\mathcal{B}$) une procédure d'affichage de r.
Dans l'ordre d'execution des procédures, on a:

Préfixe. Affichage($\mathcal{B}.r$), Visite($\mathcal{B}_g$), Visite($\mathcal{B}_d$)

Infixe. Visite($\mathcal{B}_g$), Affichage($\mathcal{B}.r$), Visite($\mathcal{B}_d$)

Postfixe. Visite($\mathcal{B}_g$), Visite($\mathcal{B}_d$), Affichage($\mathcal{B}.r$)

Exercice 8 *Écrire les procédures de parcours d'un arbre binaire. Executer vos algorithmes sur l'arbre* $\langle -, \langle *, \langle +, 2, 3\rangle, 6\rangle, 2\rangle$ *qui code l'expression arithmétique* $(2+3)*6-2$. *En déduire un algorithme pour évaluer une expression arithmétique à partir de l'arbre la codant.*


Arbre n-aire&Parcours

→Utilisée de manière pratique par les programmes. Deux types de parcours:

Parcours en largeur. On visite, profondeur par profondeur, l'arbre $\mathcal{B}$.

Parcours en profondeur. On essaye de s'enfoncer dans l'arbre le plus profond et le plus à gauche possible.

type arbre=↑noeud;
nœud=**record**
val:Element;
fils: **array**[1..n] **of** arbre;
end;

La récursivié permet d'écrire simplement la procédure de parcours en profondeur.
→La procédure de parcours en largeur peut se faire grâce à l'utilisation d'une FILE:


Parcours en largeur

début
$F \leftarrow \emptyset$;
Ajouter$(F, Racine(A))$;
tant que $F \neq \emptyset$ **faire**
    début
    $t \leftarrow$ Premier(F);
    Affichage(t);
    Retirer(F);
    pour i de 1 **à** n **faire**
        si $t.fils[i] \neq nil$ **alors** Ajouter$(F, t.fils[i])$ **fsi**;
    fin;
fin;

Exercice 9 *Ecrire le parcours en profondeur et comparer les méthodes de visite.*

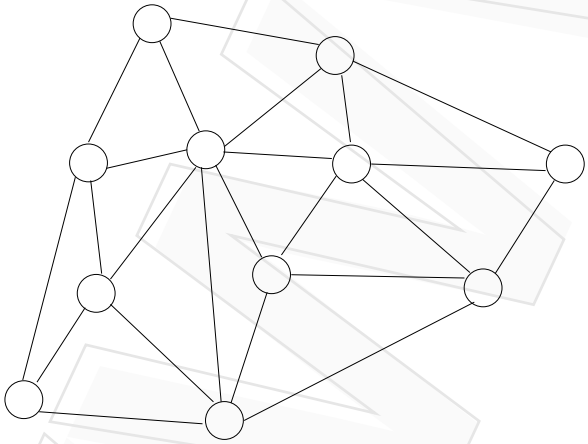
Les graphes

C'est une structure relationnelle qui modélise les interactions avec des objets. Par exemple: les réseaux routiers, téléphoniques...
Soit $\mathcal{S}$ un ensemble de **sommets** et $\mathcal{A}$ une relation booléenne binaire sur $\mathcal{S} \times \mathcal{S}$. Un graphe $\mathcal{G}$ est défini par le couple $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ où les éléments de $\mathcal{A}$ sont appelés **arcs**.
Si la relation binaire est symétrique alors le graphe est **non-orienté**. Un graphe $\mathcal{G}$ est valué sur $\mathcal{E}$ s'il existe une application de l'ensemble de ses arcs $\mathcal{A}$ dans $\mathcal{E}$ (réseau routier valué dans $\mathbf{R}$).
Soit $(x, y) \in \mathcal{A}$ une arête orientée alors y est un **successeur** de x et x est un **prédecesseur** de y. Une arête de la forme (S, S) est appelée **boucle**. Un graphe sans boucle est dit **simple**.

- degré entrant/sortant
- chaine et chemin (orientation)
- circuit (chemin dont les extrêmités coïncident)
- graphe connexe, fortement connexe
- graphe complet, clique, stable, ...

Représentation des graphes

Exemple de graphe (un graphe planaire provenant d'une triangulation):



Essentiellement, trois types de représentation des graphes:

- matrice d'adjacence ($\Omega(n^2)$ arêtes)
- par pointeurs ($O(n)$ arêtes)
- format mixte

Matrice d'adjacence

$\mathcal{G} = (\mathcal{S}, \mathcal{A})$ un graphe à n sommets. On code la relation binaire $\mathcal{A}$ (arêtes) dans une matrice $M = (m_{ij})_{ij}$ booléenne (n, n) $(M \in M_{\{0,1\}}(n, n))$:
$m_{ij} = 1$ ssi $(S_i, S_j) \in \mathcal{A}$ et $m_{ij} = 0$ ssi $(S_i, S_j) \notin \mathcal{A}$.

type graphe=**array**[1..n] **of boolean**;

Avantages

- Existence d'un arc en temps constant.
- Accés facile aux arcs. Programmes simples

Inconvénient.

- Stockage en $\Omega(n^2)$ même si $o(n)$ arêtes.
- Pour accéder aux successeurs d'un nœud, il faut parcourir les n cases. Ne dépend pas du degré du nœud.

Représentation mixte

type graphe=**array**[1..n] **of** sommet;
sommet=↑cellule;
cellule=**record**
numero:1..n;
suivant:sommet;
end;

Avantages

- Meilleure utilisation de l'espace mémoire
- Parcours des successeurs d'un nœud en temps proportionnel à son degré sortant.

Inconvénient.

- Test d'existence d'un arc en $O(n)$.
- Pas d'accés facile aux prédecesseurs.
- Utilisation plus délicate du chainage dans les algorithmes.

Parcours en profondeur

On part d'un sommet r et on suit un chemin le plus longtemps possible; Quand on rencontre une extrêmité, on retourne en arrière pour redescendre. Les sommets explorés sont **marqués** afin d'explorer toutes les composantes connexes du graphe.
Soit Successeur$(s) = \{s' \in \mathcal{S} | (s, s') \in \mathcal{A}\}$ l'ensemble des successeurs de s. Le parcours d'une composante connexe se fait comme suit:

procédure parcours_profondeur(s:sommet);
var
s':sommet;
début
$marque(s) \leftarrow vrai$;
$Affiche(s)$;
pour tout$s' \in$ Successeur(s) **faire**
si non $marque(s')$ **alors**
parcours_profondeur(s');
fin;

Exercice 10 *Donner l'algorithme final qui explore toutes les composantes connexes et faites tourner l'algorithme sur un graphe. Analyser la complexité de cette algorithme.*

Parcours en largeur

On parcourt niveau par niveau le graphe à partir d'un sommet $s \in \mathcal{S}$ arbitraire. On utilise une FILE. Complexité?

procédure parcours_largeur(s:sommet);
var
F:FILE;
s', s'':sommet;
début
$F \leftarrow \emptyset$;
$marque(s) \leftarrow vrai$;
$Ajouter(F, s)$;
$Affiche(s)$;
tant que $F \neq \emptyset$ **faire**
    $s' \leftarrow$ Premier(F);
    Retirer(F);
    pour tout $s'' \in$ Successeur(s') **faire**
        si non $marque(s'')$ **alors**
        début
        $marque(s'')$;
        $Affiche(s'')$;
        $Ajouter(F, s'')$;
        fin;
        ftq;
fin;

Le plus court chemin

Soit S_d, S_f deux sommets de $\mathcal{G}$ et v une valuation dans $\mathbf{R}$. On pose $v(S_i, S_j) = +\infty$ ssi $(S_i, S_j) \notin \mathcal{A}$. Soit C un chemin reliant S_d à S_f:

$$C = S_d S_1 ... S_k S_f$$

On associe au chemin la fonction coût $c(\dot{})$:

$$c = v(S_d, S_1) + (\sum_{i=1}^{k-1} v(S_i, S_{i+1})) + v(S_k, S_f)$$

Exemple: fonction kilométrique sur un réseau routier, ...

On cherche à minimiser le coût d'un chemin liant S_d à S_f.

- Calculer tous les plus courts chemins de S_d à $\mathcal{S}$.
- Calculer tous les plus courts chemins entre deux sommet quelconques de $\mathcal{G}$.

Les algorithmes connus résolvent également un des deux problèmes ci-dessus.

Algorithme de Dijkstra

Pas toujours une solution pour le plus court chemin: non connexité, cycle de poids négatif, ... On *suppose* dans la suite q'il n'existe pas de cycle à coût négatif.
L'algorithme de Dijkstra ne prend que des valuations de $\mathbf{R}^+$.
$\rightarrow$construction incrémentale

On calcule tous les plus courts chemin entre S_1 et les sommets de S. On utilise deux tableaux:
$C(i)$ coût d'un plus court chemin entre S_1 et S_i.
$PRED(i)$ prédecesseur de S_i dans le plus court chemin de S_1 à S_i

$PRED(\cdot)$ permet de lire à l'envers les plus courts chemins de S_1 à S_i.

Idée:
Prendre parmi les sommets non-pris le sommet S_i qui a le coût minimum de S_1 à S_i. Regarder tous les sommets atteignables de S_i parmi les sommets S_j non-pris, et mettre à jour le coût actuel de S_i à S_j.

procédure Dijsktra;
var ...
début
pour i **de** 1 **à** n **faire**
début
$C[i] \leftarrow v(S_1, S_i)$;
$PRED[i] \leftarrow 1$;
fin;
$termine \leftarrow faux$;
$E \leftarrow \{S_1\}$;
tant que non $termine$ **et** $\mathcal{S} \backslash E \neq \emptyset$ **faire**
début
$j \leftarrow \min_{\mathcal{S}\backslash\{E\}}\{C, 0\}$;
$termine \leftarrow (C[j] = +\infty$ ou $j = 0)$;
si non $termine$ **alors**
$E \leftarrow E \cup \{S_j\}$;
pour tout $S_k \in$ Successeur$(S_j)_{|\mathcal{S}\backslash\{E\}}$ **faire**
début
$c \leftarrow C[j] + v(S_j, S_k)$;
si $c < C[k]$ **alors**
$C[k] \leftarrow c$;
$PRED[k] \leftarrow j$;
fsi;
fin;
ftq;

Déroulement et Correction totale

$\rightarrow$Terminaison: à une étape donnée soit on ajoute un sommet soit *termine* devient vrai.

Soit E_k l'ensemble des sommets E à la k-ième itération,
...
On montre par récurrence sur k:

- Si S_i est un sommet de E_k alors $C_k[i]$ est le coût d'un plus court chemin de S_1 à S_i et $PRED_k[i]$ est l'indice du prédecesseur de S_i dans ce plus court chemin (E_k-chemin).
- Si S_i n'est pas un sommet de E_k alors si S_i n'est pas E_k atteignable alors $C_k[i] = +\infty$, sinon $C_k[i]$ est le coût d'un plus court E_k chemin de S_1 à S_i et $PRED_k[i]$ est l'indice du prédecesseur de S_i dans ce plus court E_k-chemin.

Exercice 11 *Démontrer par récurrence la correction totale*

Complexité de l'algorithme

→La recherche du minimum à chaque itération demande un temps linéaire en la taille de l'ensemble. Temps en $O(n^2)$ ($n-1$ itérations).
→Complexité de mise à jour de $C[]$ et $PRED[]$ est proportionnelle au nombre de successeurs visités, donc majoré par le nombre d'arêtes $|\mathcal{A}| = O(n^2)$.
→Si le graphe est connexe alors $\Omega(n^2)$ réalisé par cet algorithme.
→si on organise l'ensemble $S\mathcal{E}$ comme un tas alors la complexité devient $O(n \log n + |\mathcal{A}|)$ (avantage pour les graphes planaires).

Algorithme de Floyd: tous les plus courts chemins

→valuation dans **R** (mais pas de cycle de poids négatif)
→calcule tous les plus courts chemins
→Complexité cubique!

Principe de l'algorithme

Processus itératif. A l'étape k on a calculé tous les p.c.c. de $\mathcal{S}_k = \{S_1, ..., S_k\}$. A l'étape $k+1$, on examine si on peut trouver des meilleurs chemins avec S_{k+1} et on calcule les plus courts chemins de S_i à S_{k+1}.

Structure de données

- $C[i,j]$ tableau bidimensionnel qui donne le coût $c(S_i, S_j)$.
- $PRED[i,j]$ tableau bidimensionnel qui contient l'indice du sommet précedant S_i dans un plus court chemin de S_i à S_j.

Algorithme de Floyd

procédure FLOYD;
var ...
début
pour i **de** 1 **à** n **faire**
    pour$j$ **de** 1 **à** n **faire**
        début
        $C[i,j] \leftarrow v(S_i, S_j)$;
        $PRED[i,j] \leftarrow i$;
        fin;
pour k **de** 1 **à** n **faire**
    pour i **de** 1 **à** n **faire**
        pour j **de** 1 **à** n **faire**
            début
            $r \leftarrow C[i,k] + C[k,j]$;
            si $r < C[i,j]$ **alors**
              $C[i,j] \leftarrow r$;
              $PRED[i,j] \leftarrow PRED[k,j]$;
              fsi;
    fin;
fin;

Correction&Complexité de l'algorithme

On prouve par récurrence la propriété suivante:

- Si S_i et S_j sont k-joignables, $C_k[i,j]$ est le plus court k-chemin de S_i à S_j et $PRED_k[i,j]$ est l'indice du prédecesseur de S_j pour ce chemin.
- Si S_i et S_j ne sont pas k-joignables alors $C_k[i,j] = +\infty$

$\rightarrow$Trois boucles imbriquées: $O(n^3)$.
$\rightarrow$mauvaise complexité mais facilité d'implantation!

Tri topologique

Un graphe $\mathcal{G}$ sans cycle peut modéliser un ordre partiel. On veut parcourir les sommets de $\mathcal{G}$ des plus petits éléments aux plus grands en respectant l'ordre partiel Successeur(). On obtient une **extension linéaire** $>$ de Successeur(). $\rightarrow$Graphe de tâches, parallélisme, connaissance, ...

Soit $PRED(S_i) = \{S \in \mathcal{S} | (S, S_i) \in \mathcal{A}\}$ l'ensemble des prédecesseurs de S_i.

procédure TRI_TOPOLOGIQUE;
var ...
début
$\mathcal{T} \leftarrow \emptyset$;
pour i **de** 1 **à** n **faire**
début
Chercher $S \in \mathcal{S} \backslash \{\mathcal{T}\}$ tel que $PRED(S) \subset \mathcal{T}$;
Ajouter S dans $\mathcal{T}$;
fin;
fin;

Exercice 12 *Montrer la correction totale de l'algorithme. Donner une version de l'algorithme en utilisant la fonction* **successeur**.

Tri topologique&Complexité

On utilise une représentation par pointeurs du graphe $\mathcal{G}$.
On prouve le thérème suivant:

- On peut toujours trouver $x \in \mathcal{S} \backslash \{\mathcal{T}\}$ tel que $PRED(x) \subset \mathcal{T}$.
- Invariant de boucle:

$$\forall S_i, S_j \in \mathcal{T},\ (S_i, S_j) \in \mathcal{A} \Rightarrow S_i < S_j \text{ dans } \mathcal{T}$$

Exercice 13 *Détailler l'algorithme de tri topologique (le choix de S) et montrer que l'on peut obtenir une complexité linéaire en $O(n + |\mathcal{A}|)$.*