



F. Morain

INSTITUT NATIONAL
DE RECHERCHES
EN INFORMATIQUE
ET EN AUTOMATIQUE



UNIVERSITÉ DE TECHNOLOGIE RAUZY - LEZ-DE-FRANCE

Amphi 10 : modélisation

4 juillet 2011

- Amphi 1 : introduction.
- Amphi 2 : tableaux/String.
- Amphi 3 : récursivité.
- Amphi 4 : classes, objets.
- Amphi 5 : consolidation.
- Amphi 6 : listes.
- Amphi 7 : introduction au génie logiciel.
- Amphi 8 : arbres.
- Amphi 9 : recherche d'information.
- Amphi 10 : modélisation.

Amphi 10 : modélisation

- I. Modélisation.
- II. Graphes.
- III. Exemple : réseau social (suite et fin ?).
- IV. Conclusions sur le cours.
- V. L'invité du jour : A. Rauzy.



I. Modélisation

Briques de base :

- **tableau** : informations sans corrélation;
- **liste** : ordre sur les informations;
- **arbre** : rapport hiérarchique entre les éléments.

Pb. que faire si les rapports sont plus complexes?

Principes

Modèle :

- Information = données + relations entre les données.
- Éventuellement : contraintes et opérations possibles.

Réalisations : souvent plusieurs possibles et on peut même les faire cohabiter.

⇒ séparer modèle et réalisation.

En Java : à peu près équivalent à interface + implémentation.

- Identifier les données : facile ; recenser relations et contraintes : plus difficile.
- Relier son problème à un modèle type ; permet de prendre les bons composants sur étagère.

Multiplets

Un multiplet (élément d'un produit cartésien) permet d'associer des données de types disparates mais connus et en nombre fini.

Traduction en Java : classe dont chaque objet peut représenter un multiplet particulier. Mais il n'y a pas correspondance directe.

Ex. $z \in \mathbb{C}$, $z = x + iy = \rho \exp(i\theta)$. Une implémentation est

```
public class Complexe{
    double re, im, rho, theta;
}
```

Mais : \pm plus faciles en coordonnées cartésiennes ; $\times, /$ plus faciles en coordonnées polaires. Fonctions pénibles à écrire.

Multiplets (2/3)

Plus souple :

```
public interface Complexe{
    public double getRe();
    public double getIm();
    public double getRho();
    public double getTheta();
}
```

Multiplets (3/3)

```
public class Cartesien implements Complexe{
    private double x, y;

    // requis par Complexe
    public double getRe(){ return this.x; }
    public double getIm(){ return this.y; }
    public double getRho(){
        return Math.sqrt(this.x*this.x+this.y*this.y);
    }
    public double getTheta(){
        if(this.x == 0)
            return (this.y > 0 ? Math.PI/2 : -Math.PI/2);
        else
            return Math.atan(-this.y/this.x);
    }
    // nouveaux
    public void setRe(double x){ this.x = x; }
    public void setIm(double y){ this.y = y; }
}
```

B) Conteneurs, collections et ensembles

But : association de données de même type.

Conteneur : type abstrait très général pour lequel sont définis

- ajout et suppression d'un élément ;
- test d'appartenance d'un élément ;
- itération (accès à tous les éléments à tour de rôle) ;
- souvent : nombre d'éléments ; vider le conteneur.

Collection : type de conteneur pouvant contenir la même donnée plusieurs fois (et donc énumérée plusieurs fois).

Ensemble : conteneur avec contrainte d'unicité \Rightarrow prise en compte par l'ajout, qui peut devenir coûteux (tableau, liste), $O(1)$ par hachage.

Conteneurs ordonnés

Ordre : soit par l'ordre ou la position de leur ajout, soit par une relation d'ordre définie sur les données.

Collections séquentielles : ordre défini principalement par l'ordre des ajouts.

Restrictions supplémentaires : accès et suppression possibles que sur le dernier ou le premier ajouté \Rightarrow piles, files.

Collections ordonnées : requiert une relation d'ordre total sur les éléments \Rightarrow structures d'arbres équilibrés ; ou tableau trié si statique (accès en $O(n \log n)$).

C) Associations

Déf. table d'association = ensemble de couples (clef, données) avec une fonction particulière qui consiste à retrouver les données associées à une clef.

Sans ordre : hachage organisé suivant les clefs.

Avec ordre : arbre binaire de recherche sur les clefs.

Exemple : file de priorité

Pb. Les données arrivent avec une *priorité* de traitement.

Ex. file d'impression, ordonnanceur de système, etc.

Opérations demandées : stocker un nouvel élément avec sa priorité ; traiter la tâche prioritaire suivante.

Cas statique : le nombre de tâches est fixé une fois pour toutes \Rightarrow tri des tâches en fonction de leur priorité.

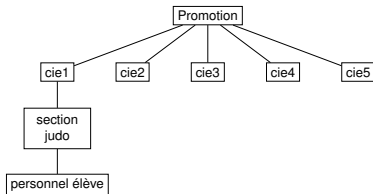
Cas dynamique :

```
public interface FileDePriorite{
    public boolean estVide();
    public void ajouter(String x, int p);
    public int tacheSuivante();
}
```

Plusieurs solutions : tableau ($O(n^2)$), tas ($O(n \log n)$) : cf. poly).

D) Quand les relations sont elles-mêmes des données

Retour à la promotion :

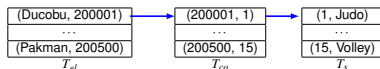


Rem. Peu adapté à la recherche d'un élève particulier, d'un cadre particulier.

À la base des systèmes de gestion de base de données relationnelles.

- une table T_{el} pour les élèves, contenant un identifiant unique (ici le matricule, appelé **clef primaire**) ;
- une table T_s pour les sections (avec identifiant_section) ;
- une table T_{co} contenant les couples (élève, section) ou (matricule, identifiant_section).

⇔ On sépare données et relations entre données.

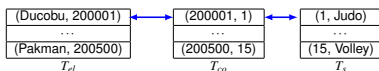


Quelques requêtes :

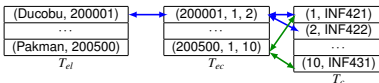
- lister les élèves : affichage T_{el} ;
- chercher un élève et sa section : chercher dans T_{el} , en déduire la clef primaire, puis la section avec T_{co} et T_s ;
- lister les sections avec leurs élèves : chercher dans T_{co} .

Entités-relations (3/3)

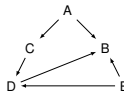
Cas 1:1



Marche également pour le cas N:M, par exemple élèves ↔ coursA2 :



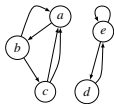
Graphes



- modélisation de réseaux de toutes sortes : plan du métro, train, liaisons électriques, gazoducs, spatiales, INTERNET ;
- représentations de liens logiques : écosystèmes ; graphe de dépendance entre fichiers source, diagramme d'héritage, réseaux sociaux ;
- modélisation de l'état d'un système ;
- allocations de ressources, chemin de coûts minimaux, etc.

II. Graphes

A) Définition

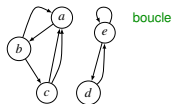


Un **graphe** $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est donné par un ensemble \mathcal{S} de **sommets** et un ensemble \mathcal{A} d'**arcs**, $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$.

Ex. $\mathcal{S} = \{a, b, c, d, e\}$, $\mathcal{A} = \{(a, b), (b, a), (b, c), (c, a), (d, e), (e, d)\}$.

Rem. La complexité des algorithmes sera fonction de $n = |\mathcal{S}|$ et souvent également de $m = |\mathcal{A}|$ (avec $m \leq n^2$).

Terminologie

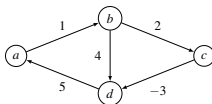


- $\alpha = (a, b)$ est **orienté** de a vers b ;
- α a pour **origine** a et **destination** b ;
- α est **incident** à a ainsi qu'à b ;
- a est un **prédécesseur** de b , et b un **successeur** de a ;
- a et b sont **adjacents**.

Graphe **simple** s'il ne comporte pas de boucles ou d'arcs multiples;
multigraphe s'il comporte des arcs multiples (mais pas de boucles).

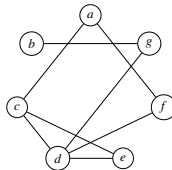
Graphe valué

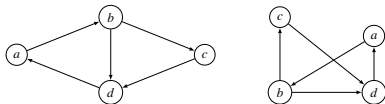
Ex. Carte des distances, etc.



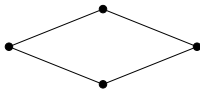
Graphe non orienté (ou symétrique)

Cas particulier (fréquent) où $(i, j) \in \mathcal{A} \Leftrightarrow (j, i) \in \mathcal{A}$; on parle alors plutôt d'ensemble d'**arêtes**.





Grphe abstrait:



B) Représentations des graphes

Comment manipuler un graphe?

- On doit d'abord modéliser le problème et trouver un codage adéquat. Cela peut se faire par acquisition, etc.
- **Attention: il n'y a pas de représentation canonique (dessin) d'un graphe...**
- Une fois cela fait, on peut passer au codage de l'information.

- Accessibilité, connexité, plus courts chemins :

$$a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_p$$

avec $(a_i, a_{i+1}) \in \mathcal{A}$.

- Planarité.
- Coloration (th. des 4 couleurs ; attribution des fréquences).
- Dessins : comment dessiner un graphe plan (circuit électronique) ?
- Visualisation : comment comprendre un graphe avec 10^6 sommets ?
- Optimisation combinatoire : voyageur de commerce, etc.

Quelles structures?

Abstraction: $\mathcal{G} = (\mathcal{S}, \mathcal{A})$.

Codage de \mathcal{S} : ensemble de sommets, par exemple $[0..n-1]$.

Codage de \mathcal{A} :

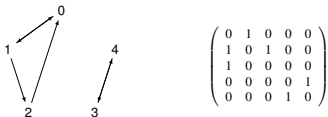
- **graphe simple** (pas de boucle ni arête multiple): matrice, tableau de listes (ou autre structure garantissant l'unicité).
- **multigraphe**: tableau de listes (ou multi-ensembles).

Matrice d'adjacence

Pour \mathcal{G} simple, matrice $n \times n$ telle que:

$$M_{ij} = \begin{cases} 1 & \text{si } (i,j) \in \mathcal{A}, \\ 0 & \text{sinon.} \end{cases}$$

Ex.



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Grphe valué: $M_{ij} = v(i,j)$.

Représentation par listes de successeurs

On associe à chaque sommet i la liste des sommets j tels que $(i,j) \in \mathcal{A}$.

Ex.

$$\begin{aligned} L[0] &= (1) \\ L[1] &= (0, 2) \\ L[2] &= (0) \\ L[3] &= (4) \\ L[4] &= (3) \end{aligned}$$

Comparaisons des deux représentations

On manipulera les deux représentations en fonction des contextes. On pourra aussi abstraire une partie des propriétés.

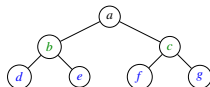
type	mémoire	utilisation
matrice	$O(n^2)$	graphe dense
listes	$O(\mathcal{A} + n)$	graphe creux

- Listes vs. matrices: on gagne quand $|\mathcal{A}| \ll n^2$.
- Matrices mal adaptées aux arcs multiples; très adaptées aux petits graphes.

C) Parcours en largeur d'abord

Idee: (*Breadth-first search*) les voisins de nos voisins sont nos voisins. On procède ainsi par "cercles" concentriques.

Exemple avec un arbre (déjà vu):



On utilise une file d'attente pour gérer la liste des sommets non encore explorés.

En pseudocode :

```
fonction bfs(G, s)
1. F ← {s};
2. L ← {s};
3. tantque F n'est pas vide faire
   t ← tête(F);
   L ← L ∪ t; // rajout en queue
   pour u voisin de t faire
     F ← F ∪ u;
```

...mais va boucler sur



Première solution

```
fonction bfs(G, s)
1. F ← {s};
2. L ← {s};
3. tantque F n'est pas vide faire
   t ← tête(F);
   L ← L ∪ t;
   pour u voisin de t faire
     si u n'a pas déjà été vu alors
       F ← F ∪ u;
```

Programmation: on utilise un tableau pour gérer l'état d'un sommet.

Le pseudocode

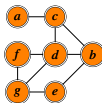
```
fonction bfs(G, s)
0. pourtout sommet t faire
   etat[t] ← inexploré;
1. F ← {s}; L ← {s}; etat[s] ← encours;
2. tantque F ≠ ∅
   // tous les sommets de F ont
   // pour état encours
   t ← tête(F);
   L ← L ∪ t;
   pour u voisin de t faire
     si etat[u] == inexploré alors
       etat[u] ← encours;
       F ← F ∪ u;
   etat[t] ← exploré;
```

Prop. La complexité de bfs est $O(|V| + |E|)$.

Ex. Traduire ce pseudocode en Java.

Exemple

```
b c d e
c a d b
a c
d c f g b
e b g
f d g
g d f e
```

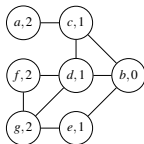


$F = \emptyset$ $F = (b)$ $F = (c)$ $F = (c, d)$ $F = (c, d, e)$ $F = \dots$

$L = \emptyset$ $L = (b)$ $L = (b, c)$ $L = (b, c, d)$ $L = (b, c, d, e)$ $L = \dots$

Propriétés du parcours BFS

Le parcours crée des couches successives $C_0 = \{b\}$, C_1, \dots :



Prop. Pour un parcours BFS(s), C_i contient les sommets à distance i de s .

Rem. On peut se servir d'un parcours BFS(s) pour trouver un chemin entre s et t (il y a d'autres algorithmes si besoin est).

Le loup, la chèvre et le chou

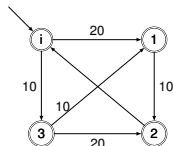
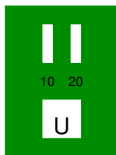
Pb. un passeur (P) doit faire passer un loup (L), une chèvre (C) et un chou (H) d'une rive à l'autre d'une rivière. Il n'y a qu'une place dans sa barque et on ne peut laisser L avec C ou C avec H.

Modélisation : graphe dont les sommets sont des états (D, A) avec D (resp. A) les personnages sur la rive de départ (resp. d'arrivée).

Principes :

- on calcule tous les cas permis (parmi tous les sous-ensembles de $\{P, L, C, H\}$, donc parmi $2^4 = 16$);
- on énumère tous les arcs possibles (par énumération des sous-ensembles des sommets);
- on cherche un chemin entre l'état initial (PLCH,) et final (, PLCH).

D) Application : modélisation des états d'un système

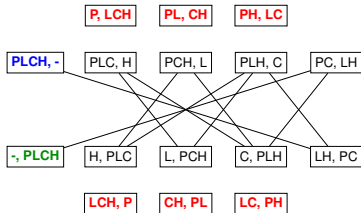


Pièces: 10, 10, 10.

↔ **automate fini.** On le représente comme un graphe.

Jeu : on lit une suite de lettres et on fait avancer son pion sur la nouvelle case. On gagne si à la fin, on est sur un état sortant.

Le loup, la chèvre, le chou (2/2)



Problème : 2^n états !

III. Exemple : réseau social (suite et fin ?)

Que demander au réseau ? D'abord de pouvoir rajouter de nouveaux membres. Un nouvel arrivant peut s'incrimer comme membre isolé, ou bien arriver comme étant ami d'un membre. On considère que l'adresse électronique est suffisante pour identifier un membre de façon unique.

Identification unique des membres (avec l'adresse électronique) ⇒ hachage.

Gestion de liens entre membres ⇒ graphe ; graphe creux car peu d'amis en moyenne, donc tableau de listes.

Affichage des amis de mes amis, etc. ⇒ parcours en profondeur d'abord.

Est-ce tout pour devenir riche ?

Il reste à gérer :

- les aspects systèmes ;
- les aspects réseaux (mail, communication, téléchargements, etc.) ;
- la partie web : design, etc. cf. [modal web](#) !
- le business plan. . .

Et encore : tourner un film sur votre vie, participer au prochain EG8, etc.

IV. Conclusions sur le cours

- Introduction à la programmation moderne (objets) : cela demande de la rigueur, prend du temps, il faut que ça mûrisse et que vous fassiez un ou plusieurs projets (Modals, 431, etc.).
- Travail de l'ingénieur TTA : utiliser des bibliothèques prédéfinies (de préférence pas au hasard), utilisant par exemple les génériques :

```
LinkedList<Abonne> l =
    new LinkedList<Abonne>();
l.add(new Abone("paul", "0607082811"));
l.add(new Abone("roger", "0607084501"));
Abonne a = l.getFirst();
for (Abonne a : l)
    TC.println(a.enChaine());
```

- Vous n'aurez aucun problème à utiliser un autre langage : PHP, Objective C, C++.

Un peu de C++

```
#include <iostream>
#include <cstdlib>
using namespace std;

class Jour{
public:
    // constructeur explicite
    Jour(int jour, int mois);
    // méthode d'objet
    void afficher(void);
    // méthode de classe
    static void calendrier(void);
private:
    int j, m;
    // méthode d'objet
    void tester();
};
```

```
Jour::Jour(int jour, int mois){
    j = jour;
    m = mois;
}

void Jour::afficher(){
    string str = "jour";
    cout << str << " " << m << "/" << j << endl;
}

int main(){
    Jour jm(12, 1);
    jm.afficher();
}
```

Quelques défis à relever

- Intégrer l'informatique dans votre vision du monde!
- Déploiement sans précédent de l'informatique embarquée (voitures).
- Sécurité au sens large.
- Loi de Moore.

L'informatique doit servir...

... pas asservir.

Quelques domaines de l'informatique :

- architecture des processeurs et ordinateurs (machines parallèles, vectorielles, multiprocesseurs); systèmes d'exploitation; réseaux; programmation distribuée et temps-réel;
- langages (conception, compilation, etc.); logique mathématique et preuves mécaniques; certification des programmes;
- algorithmique (conception, analyse); calcul formel; protection des données (crypto); bases de données; images (analyse, synthèse, vision); géométrie algorithmique, etc.); robotique; etc.

Votre formation à la seule science utile ne fait que commencer !

Derniers mots

Prochains rendez-vous : TD10.

Examen (pale) Hors Classement : lundi 11 juillet de 9h00 à 11h00.
Pour mesurer l'acquis, par pour vous classer ! Nous serons tolérants sur la syntaxe, du moment que vos programmes sont compréhensibles (n'exagérez pas).

Note finale: 1/3 pale machine + 2/3 pale papier (sur 18 points) + 2 points de note de participation aux TDs.

Important pour les transcrits pour aller aux US dans 2 ans !

Consignes pour l'examen Hors Classement

Arrivez à l'heure en sachant où vous allez (T5 ou T6 – il faut courrir vite en cas d'erreur); **tous documents fournis dans le cadre du cours + notes personnelles**; les EV2 auront 30 minutes de plus.

Si vous ratez, faites-vous rattraper! Important pour aller aux US après...

Jedis' tricks:

- lisez l'énoncé jusqu'au bout pour noter par avance ce que vous pensez savoir faire facilement, d'autant plus que les exercices sont indépendants;
- si vous ne savez pas faire une question, abandonnez la pour en essayer une autre, quitte à y revenir plus tard.