

Cours 9

Exploration Programmation à objets

Jean-Jacques.Levy@inria.fr

<http://jeanjacqueslevy.net>

tel: 01 39 63 56 89

secrétariat de l'enseignement:

Catherine Bensoussan

cb@lix.polytechnique.fr

Aile 00, LIX

tel: 01 69 33 34 67

<http://www.enseignement.polytechnique.fr/informatique/>

Plan

1. Algorithmes gloutons
2. Programmation dynamique
3. Énumération d'arborescences
4. Modularité en Java
5. Programmation à objets

Exploration

- algorithmes gloutons
la solution locale \Rightarrow la solution globale
complexité en général linéaire
- programmation dynamique
tabulation des solutions partielles \Rightarrow la solution globale
complexité souvent en $O(n^3)$
- énumération
essayer toutes les solutions
(avec retours en arrière [*backtracking*]
et optimisations éventuelles [*pruning*])
complexité en général exponentielle

Exemple 1 d'algorithme glouton

Dans un graphe valué non orienté trouver l'arbre de recouvrement minimal. Application: minimiser la longueur de fil pour alimenter un réseau.

Lemme Pour toute partition \mathcal{N} et \mathcal{N}' de l'ensemble V des noeuds d'un graphe valué non-orienté $G = (V, E, \delta)$, un arbre de recouvrement minimal contient l'arc de plus petite valeur reliant un noeud de \mathcal{N} à un noeud de \mathcal{N}' .

Algorithme de Prim:

- on démarre d'un noeud n quelconque, qui sera racine de l'arbre de recouvrement minimal.
- on rajoute à l'arbre tout arc de longueur minimal vers un noeud qui n'est pas dans l'arbre.

Exercice 1 Implémentation? Complexité?

Exemple 2 d'algorithme glouton

Marche du cavalier sur un échiquier: on choisit le mouvement qui déplace le cavalier sur la case où il attaque le moins de cases.

```
static int[] x = {2, 1, -1, -2, -2, -1, 1, 2};
static int[] y = {1, 2, 2, 1, -1, -2, -2, -1};

static void marche (int[][] m, int i, int j) {
    int n = 0; int i0, j0;
    do {
        m[i][j] = n++;
        i0 = i; j0 = j;
        int min = Integer.MAX_VALUE;
        for (int d = 0; d < x.length; ++d) {
            int na = nAttaques (m, i0+x[d], j0+y[d]);
            if (min > na) {
                i = i0+x[d]; j = j0+y[d];
                min = na;
            }
        }
    } while (i != i0 || j != j0);
}
```

Marche du cavalier – suite

```
static boolean dansEchiquier(int[][] m, int i, int j) {
    return 0 <= i && i < m.length && 0 <= j && j < m[0].length;
}

static int nAttaques (int[][] m, int i, int j) {
    if ( !(dansEchiquier (m, i, j) && m[i][j] == -1) )
        return Integer.MAX_VALUE;
    else {
        int res = 0;
        for (int d = 0; d < x.length; ++d) {
            int i_n = i+x[d], j_n = j+y[d];
            if ( dansEchiquier (m, i_n, j_n) && m[i_n][j_n] == -1 )
                ++res;
        }
        return res;
    }
}
```

Exemple 1 de programmation dynamique

Plus courts chemins entre tous les noeuds d'un graphe par l'algorithme de Floyd en modifiant l'algorithme de Warshall.

$$d_{-1}(i, j) = m(i, j)$$
$$d_k(i, j) = \min\{d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j)\}$$

```
static Graphe plusCourtsChemins (Graphe g) {
    int n = g.m.length;
    Graphe c = copieGraphe (g);
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                c.m[i][j] <- Math.min (c.m[i][j], c.m[i][k] + c.m[k][j]);
    return c;
}
```

Exercice 2 Complexité?

Exercice 3 Comparer à l'algorithme de Dijkstra

Exemple 2 de programmation dynamique

Les plus longues sous-séquences communes entre deux chaînes de caractères.

$$L(i, j) = \begin{cases} 1 + L(i-1, j-1) & \text{si } u_i = v_j \\ \max(L(i, j-1), L(i-1, j)) & \text{sinon.} \end{cases}$$

```
static int[][] plusLongueSSC (String u, String v) {
    int n = u.length(); int m = v.length();
    int longueur[][] = new int[n][m]; int pred[][] = new int[n][m];
    for (int i = 0; i < n; ++i) longueur[i][0] = 0;
    for (int j = 0; j < m; ++j) longueur[0][j] = 0;
    for (int i = 1; i < n; ++i)
        for (int j = 1; j < m; ++j)
            if (u.charAt(i) == v.charAt(j)) {
                longueur[i][j] = 1 + longueur[i-1][j-1]; pred[i][j] = 0;
            } else if (longueur[i][j-1] > longueur[i-1][j]) {
                longueur[i][j] = longueur[i][j-1]; pred[i][j] = 1;
            } else {
                longueur[i][j] = longueur[i-1][j]; pred[i][j] = 2;
            }
    return pred;
}
```

Programmation dynamique – suite

Exercice 4 Complexité de SSC?

Exercice 5 Comment implémenter la commande `diff` du système Unix?

Exercice 6 L'associativité de la multiplication de matrices permet de multiplier n matrices $M_0 M_1 \dots M_{n-1}$ sans préciser le parenthésage. Pourtant, cela peut faire varier le nombre de multiplications scalaires à effectuer. Donner un algorithme qui utilise la programmation dynamique pour trouver l'ordre optimal.

Exemple 1 d'énumération

Mettre 8 reines sur un échiquier sans qu'elles soient en prise.

```
static boolean conflit (int i1, int j1, int i2, int j2) {
    return (i1 == i2) || (j1 == j2) ||
           (Math.abs (i1 - i2) == Math.abs (j1 - j2));
}

static boolean compatible (int[] pos, int i, int j) {
    for (int k = 0; k < i; ++k)
        if (conflit (i, j, k, pos[k])) return false;
    return true;
}

static void reines (int[] pos, int i) {
    if (i >= pos.length) imprimerEchiquier(pos);
    else
        for (int j = 0; j < pos.length; ++j)
            if (compatible (pos, i, j)) { pos[i] = j; reines (pos, i+1); }
}

static void nReines (int n) {
    int[] pos = new int[n]; reines (pos, 0);
}
```

Exemple 2 d'énumération

Enumération de tous les chemins dans un graphe.

```
static void enumeration (Graphe g, int x) {
    num[x] = ++id;
    for (int ls = g.succ[x]; ls != null; ls = ls.suivant) {
        int y = ls.val;
        if (num[y] == -1)
            enumeration(g, y);
    }
    --id; num[x] = -1;
}
```

- En fait une légère modification sur le parcours en profondeur d'abord.
- Certains problèmes (circuit hamiltonien, voyageur de commerce, ...) n'ont pas de meilleure solution (pour l'instant).
- On peut s'arrêter dès qu'on trouve une solution. Auquel cas, on dit que la solution a été trouvée après plusieurs retours en arrière (*backtracking*).

Modularité

- modularité en général
- modularité en Java

Programmation à objets

- jusqu'à présent: programmation procédurale \simeq Pascal
- programmation dirigée par les données = programmation à objets

D'où

	modification données	modification programme
prog. procédurale	changement global	changement local
prog. objets	changement local	changement global

Dans un univers où toutes les données sont proches les unes des autres (i.e. dérivent d'un même modèle), la programmation à objets peut être avantageuse. L'exemple est une boîte à outils graphique, où on ne code principalement que des **incréments**.

Programmation à objets

Deux autres arguments sont utilisés par les pro-objets:

- la **modularité**: les classes regroupent la définition des objets et des méthodes \Rightarrow la programmation est modulaire.

Mais modularité \neq programmation à objets.

La modularité provient aussi de l'accessibilité des champs définis dans chaque classe et de sa compositionnalité (un module est construit à partir d'autres modules).

- l'**héritage**: une classe peut être sous-classe d'une autre classe. Ce qui permet d'ajouter ou de redéfinir des champs ou méthodes. Quand une méthode est appelée, on utilise la méthode de la plus grande sous-classe contenant l'objet.

En Java, l'héritage est **simple**. Une classe n'est sous-classe que d'une seule classe. Donc pas d'ambiguïté sur la méthode à appeler. Mais, la **lecture** d'un programme peut être **délicate**, puisqu'elle suppose comprise la hiérarchie des classes.

Exemple de programmation à objets

Dans une classe abstraite, les champs ne sont pas tous définis.

```
abstract class List {  
    abstract boolean empty ();  
    abstract List append (List x);  
    abstract boolean equals (List x);  
    abstract public String toString ();  
}
```

Une instance: les listes vides.

```
class Nil extends List {  
    Nil () { }  
  
    boolean empty () {return true; }  
    List append (List x) { return x; }  
    boolean equals (List x) { return x.empty(); }  
    public String toString () { return ""; }  
}
```

Exemple de programmation à objets

Une autre instance: les listes non vides.

```
class Cons extends List {
  int hd; List tl;
  Cons (int v, List x) { hd = v; tl = x; }

  boolean empty () {return false; }

  List append (List x) { return new Cons(hd, tl.append(x)); }

  boolean equals (List x) { return
    !x.empty() && hd == ((Cons)x).hd && tl.equals(((Cons)x).tl);
  }

  public String toString () {
    return "[" + hd + (tl.empty() ? "]" : "; " + tl.toString() + "]");
  }
}
```

Remarquer la conversion de type obligatoire dans equals. C'est le problème dit du typage des méthodes binaires.

Cf. le cours sur la programmation objet et la modularité en Majeure 1 pour plus de détails.

Exemple de programmation à objets

On peut par la suite se servir de ces fonctions comme suit:

```
public static void main (String[] args) {  
    List x = new Cons (3, new Cons (4, new Nil()));  
    List y = new Cons (5, new Nil());  
    System.out.println (x.append(y));  
    System.out.println (x.equals(y));  
    System.out.println (x.equals(x));  
}
```

Autre Exemple de programmation à objets: les termes

```
abstract class Terme {
    abstract public String toString ();
}

class Add extends Terme {
    Terme a1, a2;
    Add (Terme x, Terme y) {a1 = x; a2 = y; }
    public String toString () { return "(" + a1 + " + " + a2 + ")"; }
}

class Mul extends Terme {
    Terme a1, a2;
    Mul (Terme x, Terme y) {a1 = x; a2 = y; }
    public String toString () { return "(" + a1 + " * " + a2 + ")"; }
}

class Const extends Terme {
    int valeur;
    Const (int n) { valeur = n; }
    public String toString () { return valeur + ""; }
}
```

Terme – bis

Et on utilise les termes en notation orientée-objet (OO):

```
public static void main (String[] args) {
    Terme x = new Mul (new Add (new Const(4), new Const(5)),
                      new Const (1));
    System.out.println (x);
}
```

Pour rajouter de nouveaux constructeurs, il suffit de rajouter de nouvelles sous-classes (programmation incrémentale):

```
class Var extends Terme {
    String nom;
    Var (String s) { nom = s; }
    public String toString () { return nom; }
}

class Sub extends Terme {
    Terme a1, a2;
    Sub (Terme x, Terme y) {a1 = x; a2 = y; }
    public String toString () { return "(" + a1 + " - " + a2 + ")"; }
}
```

Terme – ter

Si on rajoute maintenant une méthode dans les termes, le changement est global.

```
abstract class Terme {
    abstract int eval (Env e);
    abstract public String toString ();
}

class Add extends Terme {
    Terme a1, a2;
    Add (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Env e) { return a1.eval(e) + a2.eval(e); }
    public String toString () { return "(" + a1 + " + " + a2 + ")"; }
}

class Mul extends Terme {
    Terme a1, a2;
    Mul (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Env e) { return a1.eval(e) * a2.eval(e); }
    public String toString () { return "(" + a1 + " * " + a2 + ")"; }
}
```

```
class Const extends Terme {
    int valeur;
    Const (int n) { valeur = n; }
    int eval (Env e) { return valeur; }
    public String toString () { return valeur + ""; }
}

class Var extends Terme {
    String nom;
    Var (String s) { nom = s; }
    int eval (Env e) { return Env.assoc(nom, e); }
    public String toString () { return nom; }
}

class Sub extends Terme {
    Terme a1, a2;
    Sub (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Env e) { return a1.eval(e) - a2.eval(e); }
    public String toString () { return "(" + a1 + " - " + a2 + " "; }
}
```

Procédural ou Objets?

- dépend de la compréhension. Vient du contrôle ou des données.
- le style de programmation diffère entre petits programmes et gros programmes ($> 10^4$ lignes).
- dépend de la stratégie de modification.
- en dernière analyse, c'est affaire de goût.
- objets \neq modularité.
- programmation à objets utilisée dans les boîtes à outils graphiques (look commun à toutes les fenêtres) et réseau (données et méthodes se déplacent simultanément).