CONTRÔLE HORS-CLASSEMENT
ÉCOLE POLYTECHNIQUE
INFORMATIQUE – COURS INF 431

GUILLAUME HANROT ET JEAN-JACQUES LÉVY

On se propose de résoudre le problème de l'affectation de k tâches à n employés ($k > 0$, $n > 0$). Un employé se voit attribuer une tâche prise parmi une liste de tâches pour lesquelles il est compétent. On cherche à trouver une affectation optimale qui maximise le nombre de tâches qui peuvent être effectuées simultanément.

Ce problème sera modélisé en utilisant un graphe. La première partie définit un certain nombre de classes utiles dans la modélisation et le traitement du problème ; la seconde partie calcule l'affectation en utilisant le graphe construit.

1. Modélisation du problème

Les employés ont chacun un nom représenté par une chaîne de caractères ; les tâches sont repérées par leurs identifiants qui sont des entiers naturels tous différents. On considère un graphe avec $n+k+2$ sommets ; chaque sommet est soit un employé, soit une tâche, soit un des deux sommets spéciaux `Debut` et `Fin`. Il y a un arc allant d'un employé vers une tâche pour laquelle cet employé est compétent. Il y a des arcs entre le sommet `Debut` et tous les employés ; il y a également des arcs entre toutes les tâches et le sommet `Fin` (cf. la figure 1).

Dans l'exemple de la figure 1, on a $n = k = 3$; le graphe a 3 sommets « employés » avec pour nom `"Pierre"`, `"Paul"`, `"Jean"`, et 3 sommets « tâches » t_1, t_2, t_3. En outre, `"Pierre"` est compétent pour les tâches t_2 et t_3 ; `"Paul"` est compétent pour t_1 ; `"Jean"` est compétent pour t_3. Une affectation optimale affecte `"Pierre"` à t_2, `"Paul"` à t_1 et `"Jean"` à t_3, permettant d'effectuer 3 tâches en parallèle, alors que l'affectation de `"Pierre"` à t_3 empêche `"Jean"` de travailler et ne laisse que la possibilité à `"Paul"` de faire t_1.

Les quatres sortes de sommets sont représentées par un type disjonctif à partir d'une classe abstraite `Sommet`, de telle manière que chaque sommet a un entier naturel distinct comme numéro (champ `num`), une

Date: 27 avril 2004.

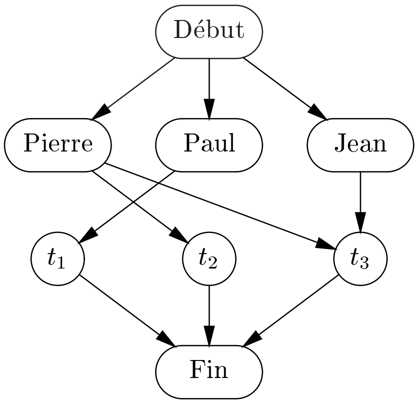

Fig. 1 – Graphe des employés et des tâches

liste d'entiers qui désigne les numéros de ses successeurs dans le graphe (champ `succ`). La classe `Employe` a les champs supplémentaires `nom` pour le nom de l'employé et `aFaire` indiquant la tâche finalement affectée à l'employé ; la classe `Tache` a un champ entier `id` donnant l'identifiant i de la tâche t_i ; les classes `Debut` et `Fin` n'ont pas de champs supplémentaires. On utilisera également la classe suivante pour manipuler les listes d'entiers :

```
class Liste {
  int val; Liste suivant;
  Liste (int x, Liste ls) { val = x; suivant = ls; }

  static Liste enlever (int x, Liste ls) {
    if (ls == null) return null;
    else if (ls.val == x) return ls.suivant;
    else return new Liste (ls.val, enlever (x, ls.suivant));
} }
```

Question 1 Définir les classes `Sommet`, `Employe`, `Tache`, `Debut`, `Fin` en donnant un constructeur dans chaque classe.

Question 2 Écrire une méthode `estEmploye`, sans argument, qui renvoie la valeur `true` si le sommet est un employé, et `false` dans les autres cas.

Question 3 Écrire une méthode `toString` permettant d'afficher tout sommet. Pour un employé, on écrira son nom et la tâche à effectuer si elle existe ; pour une tâche, son identifiant précédé de la lettre "t" ; pour les sommets spéciaux de début et de fin, les chaînes de caractères `"début"` et `"fin"`.

Le graphe est représenté par un tableau `sommet` de $n + k + 2$ sommets. Ainsi le graphe de l'exemple précédent est représenté par le tableau de 8 éléments comme indiqué sur la figure 2.

Question 4 Définir la classe `Graphe` avec un constructeur prenant en argument son nombre de sommets.

Question 5 Écrire la méthode `ajouterSommet`(x, s) qui ajoute le sommet s dans le graphe en lui donnant le numéro x.

Question 6 Écrire les méthodes `ajouterArc`(x, y) et `enleverArc`(x, y), qui ajoute (ou supprime) un arc entre le sommet de numéro x et le sommet de numéro y.

2. Résolution du problème

On utilise un graphe des employés et des tâches comme décrit précédemment avec les deux sommets distingués, dorénavant appelés d pour l'objet de la classe `Debut` et f pour l'objet de la classe `Fin`.

L'algorithme pour construire l'affectation optimale fonctionne par itération du calcul suivant : à chaque itération, on cherche un chemin quelconque reliant d à f dans le graphe. Si on trouve un tel chemin, on modifie le graphe en retournant les arcs qui composent ce chemin dans le graphe et on passe à l'itération suivante. Si un tel chemin n'existe pas, l'algorithme est terminé. Sur la figure 3, on voit les différentes étapes de cet algorithme, inspiré d'un algorithme plus général sur les flots dans les graphes par Ford et Fulkerson.

L'affectation optimale est donnée en consultant les arcs reliant un sommet « tâche » à un sommet « employé » pour tous les sommets « employés » accessibles depuis le sommet f à la fin de l'algorithme. Sur l'exemple de la figure 3(f), les trois sommets t_1, t_2 et t_3 sont accessibles depuis le sommet de fin f ; ils n'ont alors qu'un seul successeur dans le graphe (on peut démontrer que c'est toujours le cas). Ce sommet est par ailleurs forcément un « employé » et une affectation optimale donne donc `Paul` pour t_1, `Pierre` pour t_2, `Jean` pour t_3.

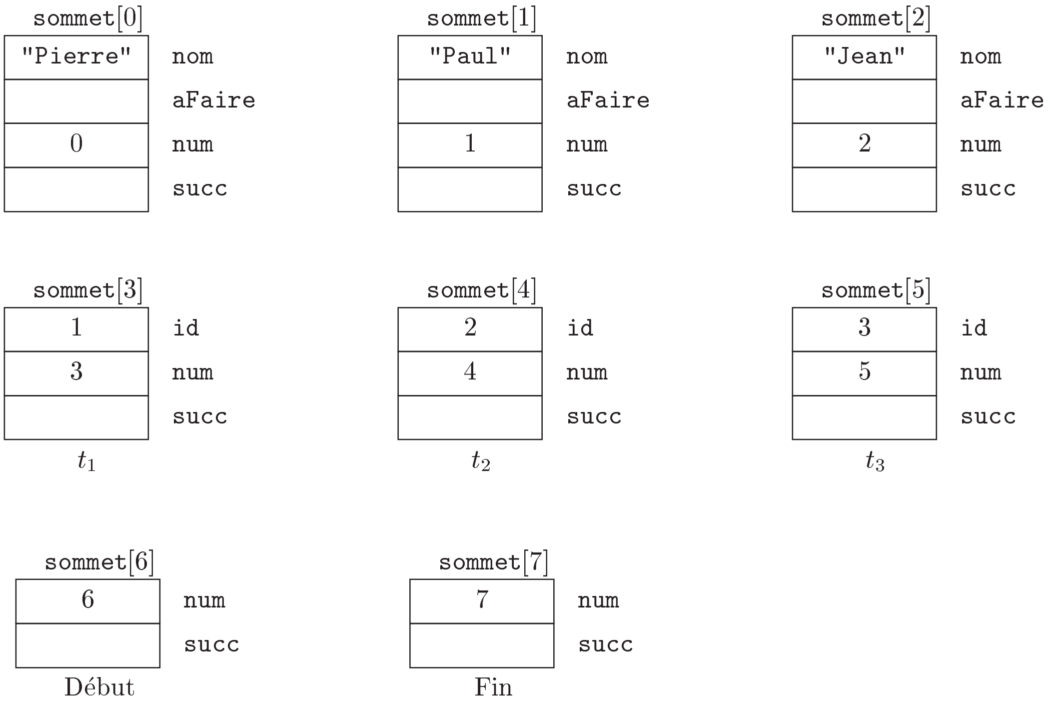

FIG. 2 – Implémentation du graphe des employés et des tâches : les champs `aFaire` n'ont pas encore de valeur ; les champs `succ`, non complétés sur la figure, indiquent les extrémités des arcs issus de chaque sommet. Ainsi `sommet[0].succ` vaut la liste $\langle 4, 5\rangle$, `sommet[1].succ` vaut la liste $\langle 3\rangle$, `sommet[2].succ` vaut la liste $\langle 5\rangle$, `sommet[3].succ` vaut la liste $\langle 7\rangle$, `sommet[4].succ` vaut la liste $\langle 7\rangle$, `sommet[5].succ` vaut la liste $\langle 7\rangle$, `sommet[6].succ` vaut la liste $\langle 0, 1, 2\rangle$, `sommet[7].succ` vaut `null`.

On représente un chemin dans le graphe par la liste des numéros de tous les sommets qui le composent (sommets de départ et d'arrivée compris) dans l'ordre où ils sont rencontrés sur le chemin.

Question 7 Écrire la méthode `chemin`(x, y) qui renvoie comme résultat un chemin allant du sommet de numéro x au sommet de numéro y dans le graphe. Si le chemin n'existe pas, cette méthode renvoie `null`. Quelle est la complexité en temps de cette méthode ?

Question 8 Écrire la méthode `retournerLesArcsDe`(ℓ) qui modifie le graphe en retournant les arcs rencontrés sur le chemin ℓ.

Question 9 Écrire la méthode `calculerResultat`(f) qui prend en argument le sommet f de fin, et qui affecte, en fin de l'algorithme, les champs `aFaire` des employés par la tâche qu'ils devront faire dans l'affectation optimale trouvée par l'algorithme.

Question 10 Écrire la méthode `fordFulkerson`(d, f) qui trouve une affectation optimale à partir des deux sommets d et f de début et de fin. La méthode ne renvoie pas de résultat, mais modifie les champs `aFaire` des employés contenus dans le graphe. Quelle est la complexité en temps de cette méthode ?

Question 11 Écrire la méthode `afficherResultat`() qui imprime pour tous les employés leur tâche affectée.

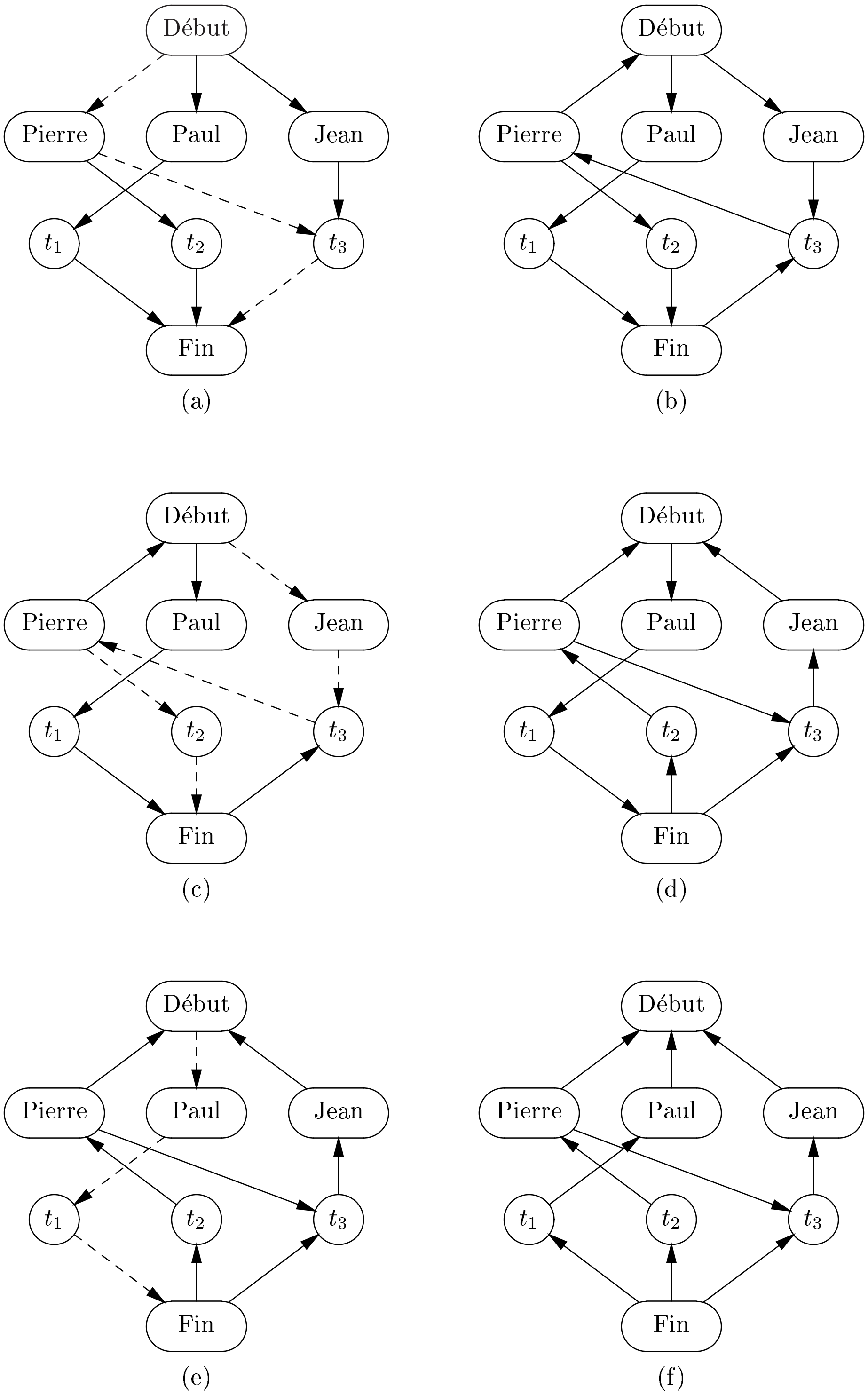

FIG. 3 – Algorithme d'affectation : les 6 étapes successives ; les chemins trouvés entre les sommets Début et Fin sont en traits discontinus ; ses arcs sont retournés à l'étape suivante (remarquons que le graphe devient cyclique à l'étape (b)). A la fin de l'algorithme, sur la figure (f), les 3 tâches t_1, t_2 et t_3 sont des successeurs de Fin dans le graphe ; l'affectation optimale est donnée par les arcs reliant t_1 à Paul, t_2 à Pierre et t_3 à Jean.

Question 1

```
abstract class Sommet {
  int num;
  Liste succ;
}

class Employe extends Sommet {
  String nom;
  Tache aFaire;
}

class Tache extends Sommet {
  int id;
  Tache (int x) { id = x; }
}

class Debut extends Sommet { }
class Fin extends Sommet { }
```

Question 2

```
abstract class Sommet { ...
  abstract boolean estEmploye();
}

class Employe extends Sommet { ...
  boolean estEmploye() { return true; }
}

class Tache extends Sommet { ...
  boolean estEmploye() { return false; }
}

class Debut extends Sommet {
  boolean estEmploye() { return false; }
}

class Fin extends Sommet {
  boolean estEmploye() { return false; }
}
```

Question 3

```
class Employe extends Sommet { ...
  public String toString() {
    if (aFaire == null) return nom;
    else return nom + " " + aFaire;
  }
}

class Tache extends Sommet { ...
  public String toString() { return "t" + id; }
}

class Debut extends Sommet { ...
  public String toString() { return "Debut"; }
```

```
}

class Fin extends Sommet { ...
  public String toString() { return "Fin"; }
}
```

Question 4

```
class Graphe {
  Sommet[ ] sommet;
  Graphe (int n) { sommet = new Sommet[n]; }
}
```

Question 5

```
void ajouterSommet (int x, Sommet s) { sommet[x] = s; s.num = x; }
```

Question 6

```
void ajouterArc (int x, int y) {
  sommet[x].succ = new Liste (y, sommet[x].succ);
}

void supprimerArc (int x, int y) {
  sommet[x].succ = Liste.enlever (y, sommet[x].succ);
}
```

Question 7 La fonction suivante prend un temps $O(n + k)$.

```
Liste chemin (Sommet x, Sommet y) {
  boolean[ ] dejaVu = new boolean[sommet.length];
  return dfs (x.num, y.num, dejaVu);
}

Liste dfs (int d, int f, boolean[ ] dejaVu) {
  dejaVu[d] = true;
  if (d == f)
    return new Liste (d, null);
  for (Liste ls = sommet[d].succ; ls != null; ls = ls.suivant) {
    int x = ls.val;
    if ( !dejaVu[x] ) {
      Liste r = dfs (x, f, dejaVu);
      if (r != null)
        return new Liste (d, r);
    }
  }
  return null;
}
```

Question 8

```
void retournerLesArcsDe (Liste ls) {
  for (int x = -1; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (x >= 0) {
      supprimerArc (x, y);
      ajouterArc (y, x);
    }
    x = y;
  }
}
```

Question 9

```
void calculerResultat (Fin f) {
  for (Liste ls = f.succ ; ls != null; ls = ls.suivant) {
    int x = ls.val;
    int y = sommet[x].succ.val;
    sommet[y].aFaire = (Tache) sommet[x];
  }
}
```

Question 10

```
void fordFulkerson(Debut d, Fin f) {
  Liste ls;
  while ((ls = chemin(d, f)) != null)
    retournerLesArcsDe (ls);
  calculerResultat (f);
}
```

À chaque étape, on trouve un chemin de d vers f ; ce chemin passe une seule fois par f, donc contient un arc d'extrémité f et aucun arc d'origine f. Par suite, à chaque execution de `chemin` puis `retournerLesArcsDe`, le nombre d'arcs d'extrémité f décroît d'une unité. Comme initialement le nombre d'arcs d'extrémité f vaut k, en au plus k étapes, on peut garantir qu'il n'y a plus d'arcs d'extrémité f, donc a fortiori plus de chemin de d à f. On effectue donc $O(k)$ appels à `chemin`, qui est de complexité $O(n + k)$, et $O(k)$ appels à `retournerLesArcsDe`. Cette dernière méthode parcourt les listes de successeurs des sommets du chemin. Dans ces parcours, un arc est rencontré au plus une fois, donc la complexité est dominée par le nombre d'arcs du graphe, soit $O(nk)$. Enfin, `calculerResultat` a un coût proportionnel au nombre d'arcs issus de fin, soit au plus $O(k)$. La complexité totale est donc $O(nk^2)$ (le terme principal est celui provenant de `retournerLesArcsDe`), que l'on peut ramener à $O(nk \min(n, k))$ en remarquant que le même argument de décroissance s'applique aux arcs sortant de d.

Question 11

```
void afficherResultat() {
  for (int i = 0; i < sommet.length; ++i) {
    Sommet s = sommet[i];
    if (s.estEmploye())
      System.out.println (s);
  }
}
```

La correction et l'optimisation des algorithmes de flots dans les graphes sont étudiées dans le cours « Conception des Algorithmes et Optimisation » de la majeure 2 d'Informatique.