Structures de données et algorithmes

Olivier Roques

2016-2017

1 Structures de données

Piles Traitement **LIFO**. Opérations de base possibles :

- Test de pile vide
- Empiler un élément au début (*push*)
- Dépiler un élément et le renvoyer (*pop*)

Files Traitement **FILO**. Opérations de base possibles :

- Test de file vide
- Enfiler un élément à la fin
- Défiler un élément au début et le renvoyer
- Récupérer le nombre d'éléments dans la file

Parcours d'un arbre On prend en compte le nœud :

- **Préordre** (ou préfixe) : au premier passage
- **Symétrique** (ou infixe) : au second passage (dans un arbre binaire complet)
- **Postfixe** : au dernier passage

Définition 1.1. Un *arbre binaire complet* est un arbre dans lequel tout nœud interne a exactement deux fils.

Définition 1.2. Un *arbre binaire parfait* est arbre dans lequel tous les étages de l'arbre sont remplis sauf éventuellement le dernier qui est rempli en partant de la gauche.

Définition 1.3. Un *arbre binaire équilibré* est un arbre tel que pour tout nœud, les sous-arbres gauche et droit ont des hauteurs qui diffèrent au plus de 1. Un arbre binaire parfait est *équilibré*.

2 Recherche et tri

Dans cette section, T est un tableau de taille $n \in \mathbb{N}^*$.

2.1 Recherche dichotomique dans un tableau trié

Algorithme Complexité en $O(\log_2 n)$. On cherche clé dans le tableau trié T :

```
gauche = 0
droite = n-1
trouve = faux
TANT QUE (non trouve) ET (gauche ≤ droite), FAIRE:
    milieu = (gauche + droite) // 2
    SI (cle < T[milieu]), ALORS: droite = milieu - 1
    SINON SI (cle > T[milieu]), ALORS: gauche = milieu + 1
    SINON: trouve = vrai
RENVOYER trouve
```

2.2 Tri sélection

Algorithme Complexité en $O(n^2)$ dans tous les cas.

```
POUR i VARIANT DE 0 À n, FAIRE:
    indicePetit = i
    min = T[i]
    POUR j VARIANT DE i+1 À n+1, FAIRE:
        SI T[j] < min, ALORS:
            indicePetit = j
            min = T[j]
    echanger(T, i, indicePetit)
```

2.3 Tri insertion

Algorithme Complexité en $O(n)$ dans le cas optimal, $O(n^2)$ en moyenne et dans le pire des cas.

```
POUR i VARIANT DE 1 À n+1:
    p = i
    cle = T[p]
    TANT QUE p  ≥ 1 ET T[p-1] > cle, FAIRE:
        T[p] = T[p-1]
        p = p - 1
    T[p] = cle
```

2.4 Tri rapide

Algorithme Complexité en $O(n \ln n)$ dans le cas optimal et en moyenne, $O(n^2)$ dans le pire des cas.

Fonction partition(g, d) Cette fonction prend en paramètres g et d qui sont les indices des bornes du sous-tableau de T qu'on manipule.

```
pivot = g
cle = t[g]
POUR i VARIANT DE g+1 À d, FAIRE:
    SI T[i] < cle, ALORS:
        pivot = pivot + 1
        echanger(T, i, pivot)
echanger(T, g, pivot)
RENVOYER pivot
```

Procédure triRapide(g, d)

```
SI g < d, ALORS:
    m = partition(g, d)
    tri_rapide(g, m)
    tri_rapide(m+1, d)
```

Il suffit alors d'appeler cette procédure avec les paramètres (0, n) pour trier le tableau T.

2.5 Tri par arbre binaire de recherche

Définition 2.1. Un *arbre binaire de recherche* est un arbre tel que, pour tout nœud interne a, les données contenues dans le sous-arbre gauche de a sont inférieures ou égales à la donnée contenue dans a, qui est elle-même inférieure ou égale aux données contenues dans le sous-arbre droit de a.

Algorithme Un parcours en ordre symétrique de l'arbre donne la liste triée des données que contient l'arbre. Complexité en $O(n \ln n)$ dans le meilleur des cas, en $O(n^2)$ dans le pire des cas.

2.6 Algorithme du tri tas

Définition 2.2. On appelle *tas* un arbre binaire parfait dans lequel l'élément stocké en un nœud quelconque est plus grand que les éléments stockés dans son ou ses nœuds fils.

Algorithme Complexité en $O(n \ln n)$ dans tous les cas.

Procédure montée(p) On dispose d'un tas de $p-1$ nœuds et on ajoute un nœud d'indice p.

```
i = p
cle = T[p]
TANT QUE i ≥ 1 ET cle > T[(i-1) // 2], FAIRE:
    T[i] = T[(i-1) // 2]
    i = (i-1) // 2
T[i] = cle
```

Procédure descente(q, p) Ici, p est la taille des données et q est l'indice de l'élément diminuée.

```
trouve = faux
i = q
cle = T[q]
TANT QUE (non trouve) ET (2*i + 1  ≤ p), FAIRE:
    SI 2*i + 1 = p, ALORS: indice_grand = p
    SINON:
        SI T[2*i+1] ≥ T[2*i+2], ALORS: indice_grand = 2*i + 1
        SINON: indice_grand = 2*i + 2
    SI cle < T[indice_grand], ALORS:
        T[i] = T[indice_grand]
        i = indice_grand
    SINON: trouve = vrai
T[i] = cle
```

Procédure triTas()

```
POUR p VARIANT DE 2 À n+1, FAIRE: montee(p)
POUR p VARIANT DE n À 1, FAIRE:
    echanger(T, 0, p)
    descente(0, p-1)
```

3 Le hachage

Définition 3.1. Une *fonction de hachage* est une méthode permettant de caractériser une information, une donnée. En faisant subir une suite de traitement à une entrée, elle génère une empreinte servant à identifier la donnée initiale.

4 L'algorithme de Huffman

Problème On code les caractères d'un texte à l'aide de suites de 0 et de 1. On souhaite alors déterminer un *codage optimal* tel que le texte codé soit le plus court possible.

Description de l'algorithme Complexité en $O(n \ln n)$ où n est le nombre de caractère à coder.

1. n nœuds (avec caractère + occurrences) rangés par ordre décroissant de nombres d'occurrences dans un tableau A.
2. Soient x et y les caractères contenus dans les nœuds A[n-2] et A[n-1]. On construit un nouveau nœud N contenant $x + y$ et $occ(x) + occ(y)$.
3. On donne à N A[n-2] comme fils gauche et A[n-1] comme fils droit.
4. On remplace A[n-2] par N et on supprime le nœud suivants
5. On déplace le nœud N dans ce tableau de tel sorte qu'il reste trié.
6. On itère jusqu'à ce qu'il n'y ait plus qu'un seul arbre dans la liste. On a alors obtenu un *codage optimal* en attribuant 0 à une descente à gauche et 1 à une descente à droite.

5 Graphes

Définition 5.1. $G = (V, E)$ est un graphe où V est l'ensemble des *sommets* et E l'ensemble des *arêtes* (ou *arcs* pour un graphe orienté). $|V|$ est appelé l'*ordre* du graphe et $|E|$ la *taille* du graphe.

Définition 5.2. Un *graphe complet* est un graphe dans lequel deux sommets quelconques sont adjacents. Un *graphe partiel* de $G = (V, E)$ est un graphe $G_p = (V, F)$ où F est une partie de E.

Définition 5.3. Le *degré* $d(x)$ d'un sommet x est le nombre d'arêtes incidentes à x. Dans un graphe orienté, le *degré sortant* $d_+(x)$ est le nombre d'arcs d'origine x, et le *degré entrant* $d^-(x)$ est le nombre d'arcs d'extrémité x.

Définition 5.4. Soit $G = (V, E)$ un graphe non orienté. On appelle *chaîne* (ou *chemin* pour un graphe orienté) une suite $x_1 e_1 x_2 \dots x_{k-1} e k-1 x_k$ avec $x_i \in V$ pour $1 \leq i \leq k$, $e_j \in E$ et $e_j = \{x_j, x_{j+1}\}$ pour $1 \leq j \leq k-1$. Une chaîne est dite *élémentaire* si tous les sommets la constituant sont deux à deux distincts.

Définition 5.5. Un *cycle* (ou *circuit* pour un graphe orienté) est une chaîne dont les deux extrémités coïncident. Un cycle est dit *élémentaire* si tous les sommets la constituant sont deux à deux distincts (excepté l'origine et l'extrémité de la chaîne).

Définition 5.6. Un *arbre* est un graphe connexe sans cycle.

Définition 5.7. Une *racine* d'un graphe orienté G est un sommet r tel qu'il existe un chemin de r à tout sommet de G. On appelle *arborescence* de racine r un graphe orienté tel que :

- le graphe non orienté sous-jacent est un arbre ;
- pour tout sommet x l'unique chaîne entre r et x du graphe non orienté sous-jacent correspond à un chemin de r vers x dans l'arborescence.

6 Arbre couvrant de poids minimum

Problème On cherche un graphe partiel d'un graphe orienté valué qui soit un arbre et qui soit de coût minimum. Puisqu'il s'agit d'un graphe partiel, cet arbre a pour ensemble de sommets l'ensemble de tous les sommets du graphe initial ; aussi dit-on qu'il s'agit d'un arbre couvrant.

Algorithme de Kruksal Cet algorithme a une complexité en $O\big(n^2 + m \log_2 m\big)$ (n sommets, m arêtes). Il se déroule en deux phases :

1. Trier les arêtes par ordre des poids croissants ;
2. Tant que l'on n'a pas retenu $n-1$ arêtes, procéder aux opérations suivantes : considérer, dans l'ordre du tri, la première arête non examinée ; si elle forme un cycle avec les arêtes précédemment choisies, la rejeter, sinon la garder.

Algorithme de Prim Complexité en $O\big(n^2\big)$. On étend de proche en proche un arbre couvrant en atteignant un sommet supplémentaire à chaque étape, et en prenant, à chaque étape, l'arête la plus légère parmi celles qui joignent l'ensemble des sommets déjà couverts à l'ensemble des sommets non encore couverts.

7 Problème des plus courts chemins

Problème Étant donné un sommet r, on veut trouver un plus court chemin de r à tous les sommets du graphe.

Algorithme de Djikstra Cet algorithme de complexité $O(n^2)$ ne concerne que les graphes à valuations positives. Ici, A est l'ensemble des sommets couverts à une étape donnée, $\pi(x)$ donne la distance de r à x et pere(x) donne le prédécesseur de x dans le plus court chemin de r à x.

```
A = {r}
pivot = r
π(r) = 0
Pour tout sommet x différent de r, faire: π(x) = +∞
Pour j variant de 0 à n, faire:
    Pour tout sommet y non dans A et successeur de pivot, faire:
        Si π(pivot) + poids(pivot, y) < π(y), alors:
            π(y) = π(pivot) + poids(pivot, y)
            pere(y) = pivot
    Chercher un sommet y non dans A tel que π(y) soit minimum
    pivot = y
    A = A ∪ {y}
```

Définition 7.1. Une *numérotation topologique* est une numérotation des sommets d'un graphe telle que tous les prédécesseurs d'un sommet i aient un numéro inférieurs à j.

Déterminer une numérotation topologique

```
Pour i variant de 1 à n, faire:
    Choisir un sommet x de G de degré entrant nul
    num(x) = i
    G = G \ {x}
```

Algorithme de Bellman Cet algorithme de complexité $O(n^2)$ et concerne les graphes à valuations positives et négatives mais ne possédant pas de circuit.

```
Déterminer une numérotation num topologique du graphe
π(r) = 0
Pour tout sommet x différent de r, faire: π(x) = +∞
Pour i variant de 2 à n+1, faire:
    Soit x le sommet tel que num(x) = i
    π(x) = min{ π(y) + poids(y, x) | y tel que (y, x) arc }
    Si le minimum est réalisé grâce au sommet y, alors: pere(x) = y
```

8 Parcours de graphe

Définition 8.1. *Marquer un sommet* : passer son état de "non marqué" à "marqué".
Traverser un arc (x, y) : Lorsque x est marqué, on regarde si y est ou n'est pas marqué. L'arc prend alors l'état "traversé".

8.1 Parcours "marquer-examiner"

Ce parcours s'applique à un graphe orienté. On dispose ici d'une liste d'attente L.

Définition 8.2. *Examiner un sommet*, c'est :

```
Pour tout arc (x, y):
    Traverser l'arc (x, y)
    Si y n'est pas marqué, alors:
        Marquer y
        pere(y) = x
        Mettre y dans L
```

Algorithme Un parcours du type "marquer-examiner" s'énonce alors par :

```
Marquer r et le mettre dans L
Tant que L n'est pas vide:
    Retirer un sommet x
    Examiner x
```

Si la liste d'attente est gérée en file, le parcours est appelé *parcours en largeur*.

8.2 Parcours en profondeur

Définition 8.3. On attribue le *numéro préfixe* d'un sommet au moment où on marque ce sommet. On attribue le *numéro postfixe* d'un sommet juste avant de reculer de ce sommet.

Algorithme Un parcours en profondeur à partir d'un sommet r est noté $DFS(r)$. L'algorithme itératif est :

```
Tant que l'algorithme n'est pas terminé:
    Noter x le sommet courant
    S'il existe un arc (x, y) non traversé, alors:
        Traverser (x, y)
        Si y n'est pas encore marqué, alors:
            Marquer y
            pere(y) = x
            Poser y comme sommet courant
    Sinon:
        Si x != r, alors: poser pere(x) comme sommet courant
        Sinon: l'algorithme est terminé
```

L'algorithme récursif, avec les numérotations préfixe et postfixe, est :

```
Marquer le sommet x
Attribuer à x sa numérotation préfixe
Tant qu'il existe un arc (x, y) avec y non marqué:
    Poser pere(y) = x
    Appeler DFS(y)
Attribuer à x son numéro postfixe
```

L'algorithme est encore valable dans le cas d'un graphe non orienté.

8.3 Application : détermination des composantes fortement connexes

Définition 8.4. Étant donné un graphe orienté $G = (V, E)$, on définit une relation d'équivalence $\mathcal{R}$ sur V par : $x\mathcal{R}y \iff$ il existe un chemin de x à y et il existe un chemin de y à x. Les classes d'équivalence de x pour cette relation s'appellent les composantes fortement connexes de G.

Algorithme Complexité en $O(n^2)$ pour une représentation en matrice d'adjacence ou en $O(n+m)$ pour une représentation en liste des successeurs. Il se déroule en 2 phases :

1. **Phase** 1 :
   (a) Choisir un sommet initial, lancer DFS à partir de ce sommet et numéroter les sommets en ordre postfixe.
   (b) S'il reste des sommets non numérotés, relancer DFS à partir de tels sommets.
   (c) Recommencer jusqu'à ce que tous les sommets soient numérotés.
2. **Phase** 2 :
   (a) Retirer toutes les marques et inverser l'orientation de tous les arcs.
   (b) Relancer DFS en commençant par le sommet de plus grand numéro.
   (c) S'il reste des sommets non numérotés, relancer DFS à partir du sommet non marqué de plus grand numéro.
   (d) Recommencer tant que tous les sommets n'ont pas été atteints.

Alors chaque application de DFS pendant la phase 2 détermine une composante fortement connexe de G.

9 Théorie des flots

Définition 9.1. Un *réseau* est un un graphe orienté $G = (V, E)$ pour lequel on a défini une application *capacité*, notée c, de E dans $]0, +\infty[$, et choisi deux sommets privilégiés, s (*source*) et p (*puits*).

Définition 9.2. Une *coupe* $(S, \overline{S})$ est l'ensemble des arcs d'origine dans S, qui est un ensemble de sommets contenant s et ne contenant pas p, et d'extrémité dans $\overline{S}$, qui est le complémentaire de S dans V.

Définition 9.3. On définit la *capacité d'une coupe* $(S, \overline{S})$ par : $c(S, \overline{S}) = \sum_{u \in (S, \overline{S})} c(u)$.

Définition 9.4. On appelle *flot* une application f de E dans $\mathbb{R}_+$ qui vérifie les deux propriétés suivantes :

- Pour tout arc $u \in E$, $0 \leq f(u) \leq c(u)$.
- Pour tout sommet x autre que p et s, il y a *conservation du flot* en x : le flot total entrant en x (somme des valeurs de $f(u)$ pour tous les arcs u d'extrémité x) est égal au flot total sortant de x (somme des valeurs de $f(u)$ pour tous les arcs u d'origine x).

On appelle *flux* d'un arc u la quantité $f(u)$.

Définition 9.5. On définit la *valeur du flot* f, noté $val(f)$, par l'une des trois expressions suivantes :

- $val(f)$ = flot total quittant s moins flot total entrant en s.
- $val(f)$ = flot total entrant en p moins flot total quittant p.
- $val(f)$ = somme des flux sur les arcs de $(S, \overline{S})$ (noté $f(S, \overline{S})$) moins la somme des flux sur les arcs de $(\overline{S}, S)$.

Théorème 9.1. Soit f un flot et $(S, \overline{S})$ une coupe. On a alors :

- $val(f) \leq c(S, \overline{S})$
- Si $val(f) = c(S, \overline{S})$, alors f est de valeur maximal et $(S, \overline{S})$ est de capacité minimum.
- $val(f) = c(S, \overline{S})$ si et seulement si :
  (i) Pour tout arc u de $(S, \overline{S})$, $f(u) = c(u)$.
  (ii) Pour tout arc u de $(\overline{S}, S)$, $f(u) = 0$.

Algorithme de Ford et Fulkerson Cet algorithme détermine un flot de valeur maximum, ainsi qu'une coupe de capacité minimum (obtenu en isolant les sommets marqués lors de la dernière application de l'algorithme de marquage). Cet algorithme est de complexité $O(mnc_{\max})$ où $c_{\max}$ désigne la capacité maximum des arcs.

Première étape C'est un "algorithme de marquage" destiné à exhiber une chaîne pour laquelle le flot peut être augmenté.

```
Marquer s par (Delta, +∞)
Pour tout sommet x != s, considérer x comme non marqué
Considérer qu'aucun sommet n'est examiné
Tant que p non marqué et qu'un sommet marqué non examiné x existe, faire:
    Soit α la valeur absolue du second paramètre de la marque de x
    Pour tout successeur y de x non marqué, faire:
        Si c(x, y) > f(x, y) alors:
            β  = min{ α, c(x, y) - f(x, y) }
            Marquer y par (x, +β)
    Pour tout prédécesseur z de x qui n'est pas marqué, faire:
        Si f(z, x) > 0, alors:
            β = min{ α, f(z, x) }
            Marquer z par (x, -β)
    Considérer x comme examiné
```

Seconde étape On exploite la chaîne trouvée précédemment pour améliorer le flot courant.

```
Définir un flot f, éventuellement le flot nul
Faire:
    Appliquer l'algorithme de marquage
    Si p est marqué alors:
        Soit C la chaîne augmentante de s à p
        Soit α la valeur absolue du second paramètre de la marque p
        Pour tout arc (x, y) de C parcouru à l'endroit, faire:
            f(x, y) = f(x, y) + α
        Pour tout arc (x, y) de C parcouru à l'envers, faire:
            f(x, y) = f(x, y) - α
Jusqu'à ce que p ne soit plus marqué
```

10 Complexité d'un problème

Définition 10.1. Un problème est dit *polynomial* s'il existe un algorithme de complexité polynomiale permettant de répondre à la question posée dans ce problème.. La *classe* P est l'ensemble de tous les problèmes de reconnaissance polynomiaux.

Définition 10.2. Un problème de reconnaissance est dans la *classe NP* si, pour toute instance de ce problème, on peut vérifier qu'une solution proposée ou devinée convient pour cette instance en un temps polynomial par rapport à la taille de l'instance. On a $P \subset NP$.

Définition 10.3. Étant donnés deux problèmes de décision D_1 et D_2, on écrit $D_1 \prec D_2$ si les conditions suivantes sont réalisées :

- On peut *transformer polynomialement* D_1 en D_2 : Il existe une application f qui transforme toute instance I de D_1 en une instance $f(I)$ de D_2, et un algorithme polynomial, par rapport à la taille de I, pour calculer $f(I)$.
- Il y a équivalence entre les deux énoncés "D_1 admet la réponse 'oui' pour l'instance I" et "D_2 admet la réponse 'oui' pour l'instance $f(I)$".

Définition 10.4. Un problème Q est dit *NP-complet* s'il est dans la classe NP et si, pour tout problème Q' de la classe NP, on a $Q' \prec Q$.

11 Séparation et évaluation

Définition 11.1. Un *problème d'optimisation linéaire en nombres entiers* est un problème de la forme :

$$\text{Maximiser } c^T x \text{ tel que } Ax \leq b \text{ et } x \in \mathbb{N}^n$$

où c et b sont des vecteurs et A une matrice à valeurs entières.

Algorithme de Séparation et Évaluation La technique de *Séparation et Évaluation* est une méthode algorithmique pour résoudre un problème d'optimisation linéaire, particulièrement utilisée pour résoudre des problèmes NP-complets. Il s'agit de rechercher une solution optimale dans un ensemble dénombrable de solutions possibles. Elle se déroule en 2 étapes :

Séparation La *séparation* consiste à diviser le problème en sous-problèmes qui ont chacun leur ensemble de solutions réalisables de telle sorte que tous ces ensembles forment un recouvrement de l'ensemble des solutions. Ainsi, en résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial.

Évaluation L'*évaluation* d'un nœud de l'arbre de recherche a pour but de déterminer l'optimum de l'ensemble des solutions réalisables associé au nœud en question ou, au contraire, de prouver mathématiquement que cet ensemble ne contient pas de solution intéressante pour la résolution du problème.

Pour déterminer qu'un ensemble de solutions réalisables ne contient pas de solution optimale, la méthode la plus générale consiste à déterminer un minorant du coût des solutions contenues dans l'ensemble (s'il s'agit d'un problème de minimisation). Si on arrive à trouver un minorant qui est supérieur au coût de la meilleure solution trouvée jusqu'à présent, on a alors l'assurance que le sous-ensemble ne contient pas l'optimum.