Notes de cours Algorithmique Avancée: Master 1 Bioinformatique, Université Paris VII

Michel Habib,
email: habib@liafa.jussieu.fr
http: www.liafa.jussieu.fr/ habib

18 décembre 2007

Table des matières

1 Complexité d'un algorithme, d'un problème **7**
- 1.1 Algorithmes . . . 7
- 1.2 Modèles de machine . . . 8
- 1.3 Quelques problèmes de nature algorithmique . . . 10
- 1.4 Quelques paradigmes utiles pour construire un algorithme efficace . . . 12
- 1.5 Complexité d'un algorithme . . . 13
- 1.6 Complexité d'un problème . . . 14
- 1.7 Notations . . . 15
  - 1.7.1 Parties entières : . . . 15
  - 1.7.2 $\mathbf{O}, \mathbf{\Omega}, \mathbf{\Theta}$ . . . 15
  - 1.7.3 Rappels . . . 15
  - 1.7.4 Exemples d'analyses de programmes . . . 17
- 1.8 Schémas d'algorithmes récursifs . . . 19
- 1.9 Équations de récurrences . . . 20
- 1.10 Exemples d'application . . . 22
- 1.11 Bornes inférieures de complexité . . . 23

2 Preuves d'algorithmes et de programmes **25**
- 2.1 Trois exemples d'algorithmes simples dont l'analyse ne l'est pas 25

3 Structures de données **29**
- 3.1 Arborescences binaires de recherche . . . 29

4 Algorithmes de graphes **35**
- 4.1 Exercices . . . 38
- 4.2 Dualité cycle-cocycle . . . 38
- 4.3 Espaces vectoriels des cycles-coycles . . . 39

5 Arbre recouvrant de poids minimum et algorithmes gloutons 41
5.1 Quelques variantes 46
5.1.1 Ordre lexicographique sur les arbres 46

6 Graphes orientés 49
6.1 Définitions de base sur les graphes orientés 49
6.2 Arborescences 50

7 Parcours de graphes 53
7.1 Parcours générique 53
7.1.1 Parcours en largeur 55
7.1.2 Parcours en profondeur 55

8 Algorithmes de plus courts chemins 61
8.1 Introduction 61
8.2 Les différents problèmes 61
8.2.1 Algorithme de Dijkstra [16] 62
8.2.2 Algorithme de Bellman - Ford 64
8.2.3 L'algorithme A* 65

9 Fermeture transitive 67
9.1 Calcul via des produits de matrices 67
9.2 Algèbres max, plus 68

10 Algorithme d'Euclide 3 siècles avant J.C. 69
10.1 69

11 Cryptographie à clef publique 71
11.1 RSA 71
11.1.1 Le travail préliminaire de B 72
11.1.2 L'émission d'un message crypté par A 72
11.1.3 Justifications 72
11.2 Quelques notions utiles d'arithmétique 73
11.2.1 Cryptanalyse de RSA 73

12 Une fonction qui croit énormément 75
12.1 Fonction d'Ackermann 75

13 Recueil d'examens **77**
13.1 Janvier 2000 77
13.1.1 Première partie : complexité 77
13.1.2 Deuxième partie : algorithmes 78
13.1.3 Question de cours 79
13.2 Test 79
13.3 Janvier 2001 80
13.3.1 Complexité 80
13.3.2 Arbres recouvrants optimaux 81
13.3.3 Applications 82
13.3.4 Spécifications 82
13.4 Janvier 2002 82
13.4.1 Première partie : complexité 82
13.4.2 Deuxième partie : algorithmes 83
13.4.3 Question de cours 84
13.5 Corrigé janvier 2002 85
13.5.1 Complexité 85
13.5.2 Algorithmes 85
13.6 Janvier 2003 86
13.6.1 Compréhension et Analyse d'algorithmes 86
13.6.2 Algorithmes 88
13.6.3 Question de cours 88
13.7 Janvier 2004 89
13.7.1 Algorithmes et Complexité 89
13.7.2 Structures de données 89
13.7.3 Applications du cours 90
13.8 Janvier 2005 91
13.8.1 Algorithmes et Complexité 91
13.8.2 Conception d'algorithmes 91
13.8.3 Algorithmes gloutons et algorithmes de graphes 92

[1]

[1] Merci de me faire part de toute remarque, concernant ce texte, erreurs, passages trop rapides ...

Chapitre 1

Complexité d'un algorithme, d'un problème

Un calcul peut se voir comme la construction d'un signe symbolique (le résultat) qui peut s'interpréter comme des propriétés d'un autre signe symbolique (la donnée).

Un signe symbolique est un état d'un objet physique et un calcul est un processus physique réalisé par une machine physique (un ordinateur par exemple, ou encore un boulier).

Ainsi dès que l'on se donne une règle pour interpréter des états physiques comme signes symboliques, tout processus physique portant sur ces états peut être considéré comme un calcul.

1.1 Algorithmes

On appelle **calcul** une séquence d'opérations élémentaires qui est :

(i) finie

(ii) déterministe, i.e. à chaque étape la nouvelle opération à effectuer est entièrement déterminée par les précédentes).

(iii) réalisable (chaque opération élémentaire peut être réalisée en un temps fini)

(iv) complète (les informations requises pour une étape de l'algorithme proviennent uniquement des étapes précédentes.

La définition ci-dessus n'est pas très formelle, mais elle est suffisante pour cette première approche. Pour une définition formelle d'un calcul, il faut se ramener à un modèle de calcul plus formel tel celui de la machine de Turing.

Lorsqu'un calcul s'arrête en un temps fini et que le résultat final fournit la réponse au problème on dit alors que ce calcul est un **algorithme**.

Lorsqu'il s'arrête en un temps fini mais que l'on n'a pas réussi à démontrer que le résultat final est le bon, on dit alors que ce calcul est une **heuristique**.

On cite souvent l'exemple d'une recette de cuisine comme exemple d'algorithme primitif. Au regard des définitions précédentes une recette de cuisine est certainement un calcul, mais la preuve formelle de la validité du résultat est difficile (surtout quand je suis au fourneau !), et donc il faut les ranger dans la catégorie des heuristiques.

En résumé avant d'analyser un algorithme il faut d'abord vérifier sa preuve, i.e. qu'il s'arrête en un temps fini et que son résultat est le bon.

La preuve de l'arrêt est en général simple sur les algorithmes vus en cours, cependant ce n'est pas toujours le cas, cf. la fonction Tak décrite au chapitre 2. Dans le cas général ce problème est indécidable (i.e. n'admet pas de solution algorithmique).

Les informaticiens ont soigneusement défini les notions d'efficacité algorithmique en essayant d'éviter toute référence aux machines physiques, en particulier en définissant des modèles abstraits de calcul, comme nous allons le voir.

1.2 Modèles de machine

Les modèles que nous allons considérer dans ce qui suit ne prennent pas en compte le parallélisme. L'intuition sous-jacente est celle d'un ordinateur monoprocesseur classique.

Random Access Machine (ou modèle RAM)

Il s'agit d'une machine monoprocesseur selon [3] qui possède une mémoire à accès direct (organisée comme un grand tableau) cette mémoire est constituée d'un ensemble de registres r_0, ..., r_i, ...

Le registre r_0 joue un rôle particulier, c'est avec lui que toutes les opérations se font.

Chaque registre mot mémoire peut contenir un entier et l'on fait l'hypothèse que tous les mots de la mémoire sont accessibles dans le même temps.

De même on suppose que les entiers manipulés dans nos exercices tiennent en codage binaire sur un registre de la mémoire. Le programme d'une telle machine est une séquence numérotée finie d'instructions élémentaires définies ci-dessous :

- Lire[i] : Lire le contenu d'un mot mémoire d'adresse i et mettre cette valeur dans le registre r_0.
- Ecrire[i] : Ecrire le contenu du registre r_0 à l'adresse i.
- Faire des opérations arithmétiques +, -, *, ./. Par exemple :
  Plus[i] : $r_0 \leftarrow r_0 + r_i$
- Faire des tests logiques simples, par exemple Zero[j] : la prochaine instruction exécutée est l'instruction de numéro j si le contenu de r_0 est ≥ 0).
- Une instruction de Saut :
  Goto[j] : la prochaine instruction exécutée est l'instruction j.
  ainsi qu'une instruction d'arrêt :
  Stop.

La complexité de l'exécution d'un algorithme sur une RAM, sera le nombre d'opérations élémentaires (comme définies ci-dessus), dans certains cas les opérations *, ./. ne seront pas considérées comme élémentaires.

On peut remarquer que dans ce modèle le programme n'est pas en mémoire et qu'il correspond grosso modo à une calculette munie d'un langage d'assembleur.

Autres modèles de machine

Il existe bien d'autres modèles de machine :

- RASP ou Random Access Stored Program machine dans la quelle le programm est stocké en mémoire et donc modifiable en cours d'exécution par programme.
- Les machines RAM ou RASP à coût logarithmique afin de tenir compte de la taille des entiers manipulés. Ce modèle revient à prendre pour opération élémentaire une opération sur un bit. Il parait plus réaliste que le modèle RAM classique.
- Les machines à pointeurs qui permettent de modéliser les machines LISP.
- La machine de Turing.
- Le λ-calcul.
- Les fonctions récursives.

– Les programmes **tant que**
– ...

Selon la fameuse thèse de Church (vérifée jusqu'à présent), tous les modèles de machines réalistes sont équivalents (i.e. ils définissent le même ensemble de fonctions : **les fonctions calculables**).

Cependant d'un modèle à l'autre il peut y avoir des différences d'efficacité, il est donc intéressant d'analyser en détail les simulations d'un modèle par un autre [29].

Ainsi une variante couramment énoncée de la thèse de Church : *tous les modèles de machines réalistes sont polynomialement équivalents* vient d'être infirmée par les modèles de calcul quantiques.

Enfin pour mieux appréhender les problèmes algorithmiques issus du parallélisme, il a été inventé le modèle **PRAM** ou parallèle RAM.

Exercice : Analyser suivant les modèles de machine RAM et RAM à coût logarithmique des algorithmes de calcul du produit de deux entiers n et m ainsi que le calcul de $n!$.

1.3 Quelques problèmes de nature algorithmique

Voici une liste de problèmes algorithmiques classiques :

1. **Nom** (*Inférieur à la demi-somme*)
   Données: une liste de n entiers $a_1, \dots a_n$ positifs ou nuls
   Résultat: i tel que $a_i \leq \frac{1}{2}\Sigma_{j=1}^{j=n} a_j$
2. **Nom** (*Tri*)
   Données: un tableau Tab[1 :n] contenant n entiers positifs
   Résultat: le tableau trié par ordre croissant
3. **Nom** (*Occurences multiples*)
   Données: un tableau Tab[1 :n] contenant n entiers positifs
   Résultat: la liste des valeurs apparaissant plusieurs fois dans le tableau
4. **Nom** (*Sous-vecteur Max*)
   Données: un tableau Tab[1 :n] contenant n entiers positifs ou négatifs
   Résultat: le sous-tableau Tab[i :j] avec $1 \leq i \leq j \leq n$ dont la somme des éléments est maximale
5. **Nom** (*Elément médian*)
   Données: un tableau Tab[1 :n] contenant n entiers positifs
   Résultat: l'élément médian du tableau

6. **Nom** (*Enveloppe Convexe*)
Données: un ensemble de n points du plan, définis par leurs coordonnées
Résultat: l'enveloppe convexe de ces n points, définie par la liste circulaire de ces points

7. **Nom** (*Calcul de X^n*)
Données: une loi $*$ associative et un entier n
Résultat: Un schéma de calcul de $\overbrace{X * \ldots * X}^{n}$

8. **Nom** (*Factorisation*)
Données: n un entier
Résultat: les facteurs premiers de n

9. **Nom** (*Isomorphisme*)
Données: Deux graphes $G_1 = (X_1, E_1)$, $G_2 = (X_2, E_2)$
Résultat: Existe-t-il un isomorphisme entre G_1 et G_2 ?

10. **Nom** (*Voyageur de commerce*)
Données: n villes et la matrice associée des distances,
Résultat: la tournée passant une fois et une seule par chaque ville et dont la longueur totale est minimale

11. **Nom** (*Permutations*)
Données: un entier n
Résultat: Faire la liste de toutes les permutations sur [1 :n]

12. **Nom** (*Go*)
Données: un Go-ban, i.e. une grille 19x19
Problème (*Faire une partie de Go contre un joueur moyen*)

13. **Nom** (*Pavage du plan*)
Données: n polygones
Résultat: Peut-on paver le plan avec ces n polygones ?

Commentaires : Pour les problèmes de 1 à 7, nous avons à notre disposition de bons algorithmes. La question est alors d'utiliser le meilleur possible. Pour les problèmes 8, 9 les algorithmes connus ne sont pas satisfaisants. Par contre en 2002, un résultat théorique important [2] lire ausi http ://www.cse.iitk.ac.in/news/primality.html, assure l'existence d'un algorithme polynomial qui vérifie si un nombre est premier. Cela ne résout pas directement le problème de la factorisation d'un entier en ses facteurs premiers mais cela donne des raisons d'espérer l'existence d'un algorithme efficcace de factorisation. Rappelons que le problème 8 est au centre des questions liées

à la cryptographie, tel le code RSA ([28] inventé par Rivest, Shamir et Adelman) basé sur des propriétés de la factorisation des entiers. Pour le problème du voyageur de commerce il est possible qu'il n'existe pas de bon algorithme. Quant au problème 11, il est par nature exponentiel, en effet dès que $n \geq 20$ le résultat risque de saturer la mémoire. Le problème 12 semble très difficile. Enfin il a été prouvé que problème 13 n'admet pas d'algorithme (problème dit **indécidable**).

1.4 Quelques paradigmes utiles pour construire un algorithme efficace

Un paradigme est une méthode générique qui s'applique dans plusieurs situations algorithmiques. Voici une liste de quelques paradigmes algorithmiques :

(0) **L'induction**
Ce paradigme donne lieu a des processus itératifs.

(i) **Diviser pour régner**
Exemples : procédés dichotomiques, mais aussi d'une manière plus générale tous les algorithmes récursifs.

(ii) **Trouver un ordre "optimal" sur les opérations à effectuer**
Exemples : les parcours en profondeur dans les graphes, les utilisations des ordres lexicographiques.

(iii) **Utiliser une structure de données ad hoc**
Cette structure de données doit permettre de réaliser efficacement les opérations de base de l'algorithme. Exemples : arborescences binaires ordonnées de recherche, arborescences bicolorées, B-arborescences, files de priorité.

(iv) **Stocker des résultats des calculs intermédiaires**.
Ici intervient le compromis calcul/mémoire. Par exemple pour calculer un sinus on peut utiliser un calcul (dévelopement en série) ou aller chercher une valeur dans une table (si le calcul a déjà été fait).

(v) **Utiliser une méthode de balayage**
Exemple : drapeau Hollandais, sous-vecteur maximum, et de nombreux problèmes de géométrie algorithmique.

(vi) **Utiliser un procédé stochastique** (à base de tirages aléatoires).
Les algorithmes *randomisés* ou probabilistes sont souvent très simples

à mettre en oeuvre et l'on peut montrer qu'ils donnent le bon résultat avec une forte probabilité [25].

(vii) **Programmation dynamique**
Sorte d'énumération implicite et efficace de l'ensemble des solutions d'un problème afin de choisir la meilleure.

(viii) **Structure discrète des données**
Dans certains problèmes on peu tirer partie du fait que les données appartiennent à un domaine restreint, par exemple les algortihmes de tri Radix sort ou Bucket sort.

Dans le cadre des problèmes d'optimisation, d'autres paradigmes apparaissent et qui vont servir à construire des algorithmes ou des heuristiques.

(i) **gloutonnerie**
Algorithmes de résolution pour le problème de l'arbre de poids minimum.

(ii) **Améliorations par échanges**
Couplages de cardinal maximum dans un graphe biparti et chaîne améliorante.

1.5 Complexité d'un algorithme

L'analyse d'algorithmes se veut un outil d'aide à la comparaison des performances des algorithmes. Nous utiliserons ci-après le mot **complexité** à son sens premier (du latin *complexus enlacé, embrassé* signifiant un objet constitué de plusieurs éléments ou plusieurs parties).

Les outils et méthodes que nous allons développer dans ce cours devrait vous convaincre que la complexité des algorithmes n'est pas compliquée (qui est hélas le sens commun actuel du mot complexité) !

Dans le cadre de l'analyse d'algorithmes, il s'agit de mesurer les ressources critiques (coûteuses) utilisées par les algorithmes. Parmi les ressources fréquemment étudiées on trouve : le temps d'exécution ou de programmation, la mémoire, les processeurs, les messages, mais on pourrait tout aussi bien étudier sur les implémentations matérielles d'un algorithme la surface du circuit ou le nombre de portes logiques, l'énergie consommée, les radiations émises ... ou encore le nombre de prions libérés (informatique biologique).

D'ailleurs lors de la conception de certains ordinateurs mobiles le pro-

blème crucial est celui de l'énergie (i.e. la consommation électrique) induite par l'émission réception. Le calcul proprement dit engendrant une consommation négligeable. Il faut donc trouver des protocoles où l'on émet-écoute le moins longtemps possible (à coup sûr).

Il s'agit d'évaluer les performances de l'algorithme en fonction de cette ressource, i.e. la taille de cette ressource utilisée par l'algorithme.

Définition 1 *On appelle complexité de l'algorithme A pour la ressource R la fonction :*

$T(A, R, n) = max\{$ *ressource R utilisée par A sur l'ensemble des données de taille n*$\}$

Définition 2 *La taille d'une donnée est simplement la taille d'un* ***bon*** *codage en mémoire de cette donnée exprimée en nombres de bits.*

Ainsi la taille d'un codage d'un entier n sera $\lceil log_2(n+1) \rceil$, et non pas n. On préfère les codages binaires aux codages unaires des entiers.

Lorsqu'il n'y a pas d'ambiguïté, on notera $T(A, R, n)$ simplement par $T(n)$. Cette mesure est appelée **la mesure du plus mauvais cas**, car elle garantit que tout comportement de l'algorithme A utilisera au plus $T(A, R, n)$ éléments de la ressource R.

1.6 Complexité d'un problème

Une question naturelle lorsqu'on sait analyser un algorithme consiste à vouloir exprimer quelques chose sur le problème associé.

On peut par exemple chercher la complexité du meilleur algorithme capable de traiter ce problème.

Mais on peut aussi se demander si pour résoudre un problème tout algorithme n'a pas une complexité minimale. C'est la questions des bornes inférieures de complexité. C'est une question très difficile. On dispose de bien peu de résultats de ce type.

Rabin [30] fut le premier à se poser des questions de complexité d'un problème, il raisonnait par analogie à la physique.

En général les calculs des fonctions de complexité sont difficiles à faire de manière exacte et on procède par approximations pour ce faire on utilise des notations bien pratiques préconisées par Knuth en 76 [21] et présentées ci-après.

1.7 Notations

1.7.1 Parties entières :

À tout nombre réel $x \geq 0$ nous associerons, $\lfloor x \rfloor$ la partie entière inférieure de x, i.e. le plus grand entier inférieur à x. De même $\lceil x \rceil$ représentera l'entier immédiatement supérieur à x.

1.7.2 $\mathbf{O}, \mathbf{\Omega}, \mathbf{\Theta}$

O(f)=$\{g : N \to N | \exists c > 0, \exists n_0 \in N$ tels que :$\forall n \geq n_0, g(n) \leq cf(n)\}$
$\Omega(f) = \{g : N \to N | \exists c > 0, \exists n_0 \in N$ tels que : $\forall n \geq n_0, g(n) \geq cf(n)\}$
et enfin $\Theta(f) = O(f) \cap \Omega(f)$ [1].

Le plus généralement (et en tout cas dans ce qui suit), on considère le temps d'exécution que l'on va confondre avec le nombre d'instructions élémentaires utilisées par l'algorithme.

1.7.3 Rappels

Pour un entier positif m on notera $\lceil log_2(m) \rceil$ la partie entière supérieure de $log_2(m)$, c'est de manière équivalente l'entier unique t qui vérifie les inégalités :
$2^{t-1} < m \leq 2^t$.

Propriété 1 *Un entier m peut se coder à l'aide de* $\lceil log_2(m+1) \rceil$ *bits.*

Preuve: Il suffit d'utiliser un codage binaire de m. Un bit de plus serait nécessaire si le codage doit accepter des entiers positifs et négatifs. □

Propriété 2 (Stirling) $\sqrt{2\pi n}(\frac{n}{e})^n < n! < \sqrt{2\pi n}(\frac{n}{e})^n e^{\frac{1}{4n}}$

Corollaire 1 $log_2(n!) \in \Theta(nlogn)$.

Preuve: Cette formule de Stirling nous permet d'en déduire qu'il existe un constante $c > 0$ telle que :
$|\lceil log_2(n!) \rceil - nlog_2(n) + nlog_2(e) - \frac{1}{2}log_2(n)| \leq c$
et donc $log_2(n!) \in \Theta(nlogn)$. □

[1] **Attention ce sont des ensembles de fonctions de** $N \to N$

Définition 3 *On appelle* ***arborescence binaire****, une arborescence dans laquelle tout sommet non feuille admet au plus deux fils.*

Définition 4 *On appelle* ***arborescence binaire saturée****, une arborescence binaire dans laquelle tout sommet non feuille admet exactement deux fils.*

Propriété 3 *Soit A une arborescence binaire saturée ayant n sommets, et posons a (resp. f) le nombre de sommets internes (resp. de feuilles) de A. Alors $a = f - 1$.*

Preuve: Par définition, $n = a + f$. La preuve se fait par induction sur n. Lorsque $n = 1$, alors par convention $a = 0$ et $f = 1$. Dès que $n \geq 3$, il suffit d'enlever deux feuilles ayant le même père pour obtenir le résultat. □

Propriété 4 *Une arborescence binaire ayant f feuilles est de hauteur au moins $\lceil log_2(f) \rceil$.*

Preuve: Il suffit de remarquer que l'arborescence de hauteur h ayant le plus grand nombre de feuilles est l'arborescence complète qui admet exactement 2^h feuilles. □

Notons L_a (resp. L_f) la somme des longueurs des chemins de la racine à tous les sommets internes (resp. à toutes les feuilles).

Propriété 5 $L_f = L_a + 2a$

Preuve: La preuve se fait par induction sur a. Pour $a = 1$, $L_a = 0$ et $L_f = 2$ et la propriété est bien vérifiée. Pour une arborescence ayant $a > 1$ noeuds internes, il suffit d'en considérer un x dont les deux fils sont des feuilles f_1 et f_2. Si l'on enlève ses deux feuilles à l'arborescence, nous obtenons une nouvelle arborescence ayant un noeud interne de moins. Il est facile de vérifier :

$L_f = L'_f + 2(k+1) - k$ et $L_a = L'_a + k$ (où k représente la longueur du chemin de la racine au sommet x).

Par hypothèse d'induction : $L'_f = L'_a + 2(a-1)$ d'où l'on en déduit : $L_f = L_a + 2a$.□

Propriété 6 $L_a \geq \Sigma_{i=1}^{i=a} \lfloor log(i) \rfloor$

Preuve: Un arborescence binaire saturée admet au plus 2^k sommets à distance k de la racine. Afin d'otenir cette formule, il suffit de numéroter les a sommets internes par niveaux à partir de la racine. La racine ayant le numéro 1. Pour le sommet de numero i, la longueur du chemin de 1 à i est supérieure à $\lfloor log(i) \rfloor$. □

Il est facile de voir que les arborescences pour lesquelles il y a égalité dans la formule précédente sont les arborescences quasi-complètes, i.e. toutes les feuilles sont à distance h ou h+1 de la racine.

Propriété 7 $L_a \geq (a+1)\lfloor log(a+1) \rfloor - 2^{\lfloor Log(a+1) \rfloor + 1} + 2$

Preuve:

On peut retrouver cette formule en majorant la somme de l'expression de la proposition précédente par l'intégrale associée (on peut même fournir un encadrement). Mais on peut aussi utiliser l'identité :

$\Sigma_{i=1}^{i=n} a_i = n.a_n - \Sigma_{i=1}^{i=n-1} i(a_{i+1} - a_i)$. □

1.7.4 Exemples d'analyses de programmes

On commencera par décomposer le programme en blocs élémentaires d'instructions.

L'itération : B_1

pour *i=1 à n* **faire**
 ⌊ A(i)

Dont la complexité temporelle est :
$T(B_1) = \sum_{i=1}^{i=n} T(A(i))$

Notons que dans ce cas nous avons la valeur exacte.

Le bloc condition : B_2

si *Condition* **alors**
 ⌊ faire A
sinon
 ⌊ faire B

Si on ne sait pas quelle est la réponse au test booléen *Condition* :

soit : $T(B_2) = T(Condition) + T(A)$

soit : $T(B_2) = T(Condition) + T(B)$

Il y a donc deux valeurs possibles et dans le cadre de la complexité dans le plus mauvais cas, on écrira :

$T(B_2) \leq T(Condition) + Max\{T(A), T(B)\}$.

ou encore :

$T(B_2) \in O(T(Condition) + Max\{T(A), T(B)\})$.

$T(B_2) \in \Omega(T(Condition) + Min\{T(A), T(B)\})$.

La boucle **tant que** : B_3

$I \leftarrow 1$
tant que *Condition et* $I < n + 1$ **faire**
  A(i)
  $I \leftarrow I + 1$

S'il est difficile de prévoir en fonction de la donnée du programme quand la *Condition* est vérifiée, nous ne pouvons déduire que :

$T(B_3) = \sum_{i=1}^{i=k} T(A(i) + kT(Condition))$ pour un certain k, $1 \leq k \leq n$.

Ce que l'on écrit :

$\forall n, T(B_3)(n) \in [0, \sum_{i=1}^{i=n} T(A(i)) + n.T(Condition)]$ ou encore :

$T(B_3) \in O(\sum_{i=1}^{i=n} T(A(i))] + n.T(Condition))$ et

$T(B_3) \in \Omega(1)$

En conclusion comme tout programme d'un langage de programmation est décomposable en blocs élémentaires, cette méthode permet d'obtenir des expressions qui sont une majoration (resp. une minoration) de la complexité temporelle du programme.

Cependant ces formules ne sont pas toujours significatives (par exemple lorsque le programme à analyser contient trop de blocs conditionnels) et il faut donc utiliser d'autres méthodes de calcul si l'on veut une évaluation moins grossière de la complexité du calcul.

Nous allons voir ci-après comment traiter les programmes récursifs, illustration du paradigme de programmation : Diviser pour régner.

1.8 Schémas d'algorithmes récursifs

Résoudre-Pb(A) :
if *taille(A)<taille-critique* **then**
└ Arrêt
begin
découper le problème
résoudre-Pb(A_1)
résoudre-Pb(A_2)
...
Assembler-résultats
end

Résoudre-Pb(A) :
if *taille(A)<taille-critique* **then**
└ Arrêt
begin
découper le problème
sélectionner un sous-problème A_i
résoudre-Pb(A_i)
end

Ces deux algorithmes récursifs ont des comportements différents, le premier est un schéma récursif classique, le deuxième s'apparente à la dichotomie. À chacun de ces schémas, il est possible d'associer les équations que vérifient les complexités temporelles.

Pour le premier algorithme nous avons les équations :

$$\begin{cases} T(taille - critique) = c \\ \text{et pour une donnée A, telle que : } |A| > Taille - critique, \\ T(|A|) = g(|A|) + f(|A|) + \sum_{i=1}^{i=k} T(|A_i|) \end{cases}$$

Où g et f représentent respectivement les complexités des opérations de découpe en sous-problèmes et d'assemblage de résultats.

Pour le deuxième schéma nous avons :

$$\begin{cases} T(taille - critique) = c \\ T(|A|) = g(|A|) + f(|A|) + max\{T(|A_1|), \ldots T(|A_k|)\} \end{cases}$$

Où g et f représentent respectivement les complexités des opérations de découpe en sous-problèmes et de sélection du sous-problème intéressant.

La résolution de ces systèmes d'équations permet de mesurer l'efficacité de l'algorithme (avant son implémentation). Il n'est pas toujours facile de résoudre de telles équations, cependant nous disposons d'un ensemble de résultats.

1.9 Équations de récurrences

Les résultats suivants seront admis sans preuve.

Théorème 1 *Les équations de récurrence :*

$$\begin{cases} T(1) = c, \\ n \geq 2, T(n) = aT(n/b) + cn \end{cases}$$

avec $a, b, c > 0$ et où n/b représente soit $\lfloor n/b \rfloor$ soit $\lceil n/b \rceil$, ont pour solution :

$a < b \implies T(n) \in \Theta(n)$

$a = b \implies T(n) \in \Theta(nlog(n))$

$a > b \implies T(n) \in \Theta(n^{log_b(a)})$

Théorème 2 *Les équations de récurrence :*

$$\begin{cases} T(1) = c \\ n \geq 2, T(n) = aT(n/b) + c \end{cases}$$

avec $a \geq 1, b > 1, c > 0$ et où n/b représente soit $\lfloor n/b \rfloor$ soit $\lceil n/b \rceil$ ont pour solution :

$a < b$ *si* $a = 1 \implies T(n) \in \Theta(log(n))$

si $a > 1 \implies T(n) \in \Theta(n^{log_b(a)})$

$a = b \implies T(n) \in \Theta(n)$

$a > b \implies T(n) \in \Theta(n^{log_b(a)})$

Remarque 1 *Le cas où a=1 et b=2 correspond au processus dichotomique classique. Pour démontrer les deux théorèmes précédents, on peut commencer par considérer les entiers $n = b^t$.*

Le théorème suivant est une sorte de généralisation des deux précédents [10] :

Théorème 3 *Les équations de récurrence :*

$$\begin{cases} T(1) = c \\ n \geq 2, T(n) = aT(n/b) + f(n) \end{cases}$$

avec $a \geq 1, b > 1$ *et où* n/b *représente soit* $\lfloor n/b \rfloor$ *soit* $\lceil n/b \rceil$ *ont pour solution :*

si $f(n) \in O(n^{log_b(a)-\epsilon})$ *pour* $\epsilon > 0 \Longrightarrow T(n) \in \Theta(n^{log_b(a)})$

si $f(n) \in \Theta(n^{log_b(a)}) \Longrightarrow T(n) \in \Theta(n^{log_b(a)} log(n)) = \Theta(f(n)log(n))$

si $f(n) \in \Omega(n^{log_b(a)+\epsilon})$ *pour* $\epsilon > 0$ *et si* $af(n/b) \leq df(n)$ *avec* $d < 1$, $\Longrightarrow T(n) \in \Theta(f(n))$

Remarque 2 *Ce théorème ne couvre pas tous les cas possibles de la fonction* f*. Enfin terminons par les équations d'autres schémas récursifs.*

Théorème 4 *Les équations de récurrence :*

$$\begin{cases} T(1) = a \\ n \geq 2, T(n) = bT(n-1) + a \end{cases}$$

avec $a > 0, b \geq 1$ *ont pour solution :*

$b = 1 \Longrightarrow T(n) = an \in \Theta(n)$

$b \geq 1 \Longrightarrow T(n) \in \Theta(b^{n-1})$

Remarque 3 *Le problème des tours de Hanoï correspond au cas où* $a = 1, b = 2$*, et* $T(n) \in \Theta(2^{n-1})$*, donc un algorithme récursif exponentiel.*

Théorème 5 *Les équations de récurrence :*

$$\begin{cases} T(1) = a \\ n \geq 2, T(n) = bT(n-1) + an \end{cases}$$

avec $a > 0, b \geq 1$ *ont pour solution :*

$b = 1 \Longrightarrow T(n) \in \Theta(n^2)$

$b \geq 1 \Longrightarrow T(n) \in \Theta(b^{n-1})$

Remarque 4 *Le plus mauvais cas de l'algorithme Quicksort correspond au cas où* $b = 1, a = 1$*, cet algorithme a donc un comportement quadratique dans le pire des cas.*

En guise de conclusion considérons les équations de récurrence :

$$\begin{cases} T(1) = a \\ n \geq 2, T(n) = T(n/5) + T(3n/10) + an \end{cases}$$

Ces équations ne sont pas exactement du type de celles présentées ci-dessus, elles correspondent à l'algorithme linéaire proposé par Tarjan pour le calcul d'un élément médian d'un tableau. Faisons donc l'hypothèse d'un schéma récursif linéaire.

Propriété 8 $T(n) \in O(n)$.

Preuve: Nous allons le montrer par induction.

Vrai pour n=1. Supposons que la propriété soit vraie jusqu'à l'ordre n. Il existe donc un réel c tel que pour tout $n' < n : T(n') \leq cn'$.

$T(n) = T(n/5) + T(3n/10) + an \leq cn/5 + 3cn/10 + an$

donc $T(n) \leq cn/2 + an$

et nous avons bien $T(n) \leq cn$,

dès que $cn/2 + an \leq cn$ i.e. $c \geq 2a$. □

Généralisons un peu en considérant les équations de récurrence :

$$\begin{cases} T(1) = a \\ n \geq 2, T(n) = \sum_{i=1}^{i=k} a_i T(\frac{n}{b_i}) + an \end{cases}$$

Propriété 9 *Si* $\sum_{i=1}^{i=k} \frac{a_i}{b_i} < 1$ *alors* $T(n) \in O(n)$.

Preuve identique à celle de la propriété précédente.

1.10 Exemples d'application

Appliqués au problème du tri ces paradigmes permettent de produire :

(i) donne les algorithmes tri par insertion, tri par fusion (Merge sort) ou Quicksort suivant la stratégie de partitionnement choisie.

(iii) donne l'algorithme Heapsort.

(v) donne l'algorithme tri par bulles.

Ainsi sur le problème du tri les paradigmes (i) et (iii) sont les plus efficaces.

Exercices

1. Illustrer ces paradigmes en considérant le problème classique Sous-vecteur Max dû à Bentley in Programming Pearls, et comparer les complexités des algorithmes obtenus (cf. TD No 2, Algo IUP1 95/96) (on peut montrer sur cet exemple que le paradigme gagnant est le (iv)).
2. Même question sur le problème de l'enveloppe convexe.
3. Soient deux vecteurs de nombres entiers, $a = (a_1, a_2, \ldots a_k)$ et $b = (b_1, b_2, \ldots b_k)$

   On considère l'ordre produit classique sur les vecteurs, i.e. $a \leq b$ ssi $\forall i a_i \leq b_i$.

   On notera p_j le jième nombre premier (par convention $p_1 = 2$)

   (a) On notera $A = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$ et $B = p_1^{b_1} p_2^{b_2} \ldots p_k^{b_k}$.
   Montrer que $a \leq b$ ssi A divise B.

   (b) En déduire un algorithme en $O(1)$ pour tester $a \leq b$ lorsqu'on dispose de A et B.

   (c) Est-ce vraiment raisonnable ? Comparer l'algorithme en O(1) avec un algorithme normal qui fait k comparaisons d'un a_i avec un b_i. En particulier on évaluera la complexité du calcul de A et B.

1.11 Bornes inférieures de complexité

Considérons le problème suivant :
Nom (*Maximum*)
Données: n entiers $a_1, \ldots a_n$
Résultat: l'élement maximum des a_i
Il existe un algorithme très simple :
Maximum :
$M = a_1$
for $i = 2$ *à* n **do**
$\lfloor$ $M = max\{M, a_i\}$

Cet algorithme utilise n-1 appels à la fonction max qui retourne le maximum de 2 entiers. Il est donc en $O(n)$. Il est facile de voir que cet algorithme est optimal en O, car il faut bien lire toute la donnée pour trouver le maximum.

Exercices

1. Écrire un algorithme qui calcule les minimum et maximum d'un ensemble de n entiers.
2. Montrer que pour déterminer le 2ème élément d'un tour de tennis (du type Wimbledon ou Roland Garos), il faut faire $log_2(n) - 1$ nouveaux matchs.

Sur des problèmes plus complexes, il est plus difficile d'obtenir des bornes inférieures. Pour ce faire il est souvent nécessaire de faire des hypothèses sur l'algorithme, en particulier restreindre les opérations élémentaires qu'il peut utiliser. Citons le résultat classique suivant :

Théorème 6 *Un algorithme de tri d'un ensemble de n nombres entiers n'utilisant que des comparaisons nécessite au moins* $\Omega(nlogn)$ *comparaisons.*

Chapitre 2

Preuves d'algorithmes et de programmes

Par définition un algorithme doit fournir un résultat correct à la fin de son exécution. Pour montrer qu'une séquence de calculs est un algorithme, il faut donc prouver deux choses distinctes :

- La terminaison, i.e. le calcul se termine en un temps fini. Pour ce faire il faut trouver un paramètre qui varie de manière strictement monotone au cours du calcul (cf. exemple 3 suivant).
- Le résultat obtenu est le bon. Pour ce faire comme souvent les calculs sont structurés à l'aide de boucles (tant que), il faut identifier les propriétés qui sont invariantes d'une itération à l'autre (cf. la preuve des algorithmes sur le problème de l'arbre de poids minimal). Cette méthode dite des invariants permet de bien comprendre l'algorithme (en particulier la preuve des invariants doit dépendre de toutes les étapes élémentaires du calcul, dans le cas contraire soit la preuve est fausse, soit l'algorithme fait des calculs inutiles).

Uniquement lorsque les réponses à ces deux questions sont réglées on peut s'intéresser à la complexité de l'algorithme.

2.1 Trois exemples d'algorithmes simples dont l'analyse ne l'est pas

Pour le premier exemple la question de la terminaison n'est pas simple. Quant au deuxième, sa terminaison est conjecturée mais il n'en existe aucune

preuve à ma connaissance ! Pour le troisième la terminaison est prouvée, mais elle donne une borne énorme sur le temps de calcul. .

1. Pour i,j, k entiers on définit la fonction Tak proposée par Takeuchi pour le test des implémentations de la récursivité dans des langages fonctionnels de type Lisp (cf. R.P. Gabriel) :

   Tak(i, j, k) **:**
   si $i \leq j$ **alors**
   | **retourner** j
   sinon
   | $Tak(Tak(i-1, j, k), Tak(j-1, k, i), Tak(k-1, i, j))$

   Montrer que cet algorithme doublement récursif s'arrête pour tout triplet d'entiers (i, j, k) et trouver ce qu'il calcule.

2. Un autre exemple célèbre, le problème de Syracuse :

   f(n) **:**
   si $n = 1$ **alors**
   | **retourner** 1
   sinon
   | **si** n *est pair* **alors**
   | | **retourner** $f(n/2)$
   | **sinon**
   | | **retourner** $f(3n+1)$

   Expérimentalement (i.e. en programmant cette fonction) pour tout entier n fixé, $f(n) = 1$ (c.a.d. le calcul s'arrête en un temps fini).
   Il est conjecturé que ce programme s'arrête pour toute valeur de n.
   Ce problème dont l'énoncé est très simple, a été posé en 1930 par Lothar Collatz, puis transmis par Helmutt Hasse puis enfin par Stanislas Ulam et a passionné tous les mathématiciens qui l'ont considéré. On le connaît aussi sous le nom de la conjecture $3n + 1$.

3. On considère une permutation σ sur $[1, n]$ et la procédure suivante :

   tant que $\sigma(1) \neq 1$ **faire**
   | retourner (inverser) les $\sigma(1)$ premiers éléments de σ

   Proposition 1 *Pour toute permutation σ la procédure s'arrête en au plus 2^n étapes.*

Preuve: : il suffit de considérer le vecteur caractéristique des éléments de σ qui vérifient $\sigma(i) = i$. D'une itération à l'autre ce vecteur ne peut qu'augmenter lexicographiquement (vers la droite). Comme il y a 2^n vecteurs différents la procédure va s'arrêter en au plus 2^n étapes.

Manifestement c'est une majoration grossière, voici une conjecture transmise par V. Chvatal :

Conjecture : La procédure s'arrête en au plus n^2 étapes.

On peut en déduire un algorithme de tri :

pour $i = 1$ *à* n **faire**
  tant que $\sigma(i) \neq i$ **faire**
    retourner (inverser) les éléments de i à $\sigma(i)$

La seule chose connue est que cet algorithme utilise au plus 2^n opérations élémentaires pour trier n nombres !

En fait ce problème peut se voir comme un cas particulier des problèmes de tri de crêpes.

4. Problèmes de tas de crêpes.

Ces quelques exemples montrent qu'il existe des fonctions simples à programmer, mais difficiles à analyser.

Chapitre 3

Structures de données

Quand il s'agit de construire une structure de données efficace pour un algorithme, on doit commencer par une phase de spécification des opérations qui sont à effectuer sur cette structure de données. Pour ce faire on peut utiliser un langage de spécifications formelles de type **Z** ou **B** [1].

Imaginons avoir à concevoir un *dictionnaire informatique*, i.e. une structure de données qui permette de rechercher un élément, d'insérer un nouvel élément et de détruire un élément.

Dans certaines applications il sera nécessaire de munir les éléments du dictionnaire d'une relation d'ordre et de savoir retrouver les éléments maximum et minimum ainsi que le successeur et/ou le prédécesseur d'un élément donné.

Enfin il peut être utile de fusionner ou partitionner ces dictionnaires.

3.1 Arborescences binaires de recherche

On considère des arborescences binaires vérifiant : chaque noeud de l'arborescence possède une clé, trois pointeurs : fg (pour fils-gauche), fd (pour fils-droit) et père, ainsi qu'un enregistrement (contenant les données). Par convention l'absence d'un pointeur sera noté **NIL**. En outre dans ce qui suit une arborescence sera confondue avec sa racine.

Définition 5 *Une arborescence binaire ordonnée est une arborescence binaire A dont les clés vérifient :*

$\forall x \in A$, $\forall y \in fg(x)$, $\forall z \in fd(x)$, $cle(y) \leq cle(x) \leq cle(z)$

Par souci de simplification, on supposera que deux éléments du dictionnaire ne peuvent pas posséder une même clé.

Rechercher(x, k) **:**
si $x = NIL$ *ou* $k = cle(x)$ **alors**
  retourner x
si $k < cle(x)$ **alors**
  Rechercher(fg(x), k)
sinon
  Rechercher(fd(x), k)

Les calculs de Minimum et Maximum reviennent à calculer la feuille la plus à gauche (resp. à droite).

Minimum(x) **:**
tant que $fg(x) \neq NIL$ **faire**
  $x \leftarrow fg(x)$
retourner x

La fonction Maximum s'écrit dualement.

Successeur(x) **:**
si $fd(x) \neq NIL$ **alors**
  retourner $Minimum(fd(x))$
$y \leftarrow pere(x)$
tant que $y \neq NIL$ *et* $x = fd(y)$ **faire**
  $x \leftarrow y$
  $y \leftarrow pere(y)$
retourner y

Remarquons que les implémentations ci-dessus des fonctions Minimum et Successeur ne demandent aucune comparaison de clés, elle se font uniquement sur la structure de l'arborescence.

Inserer(z, A) **:**
Hypothèse : on suppose $z \notin A$
$y \leftarrow NIL$
$x \leftarrow racine(A)$
tant que $x \neq NIL$ **faire**
  $y \leftarrow x$
  si $cle(z) < cle(x)$ **alors**
    $x \leftarrow fg(x)$
  sinon
    $x \leftarrow fd(x)$
$pere(z) \leftarrow y$
si $y = NIL$ **alors**
  $racine(A) = z$
sinon
  si $cle(z) < cle(y)$ **alors**
    $fg(y) \leftarrow z$
  sinon
    $fd(y) \leftarrow z$

Pour terminer une fonction un peu plus compliquée :

Detruire(z, A) **:**
Hypothèse : on suppose $z \in A$
si $fg(z) = NIL$ *ou* $fd(z) = NIL$ **alors**
  $y \leftarrow z$
sinon
  $y \leftarrow Successeur(z)$
si $fg(y) \neq NIL$ **alors**
  $x \leftarrow fg(y)$
sinon
  $x \leftarrow fd(y)$
si $x \neq NIL$ **alors**
  $pere(x) \leftarrow pere(y)$
si $pere(y) = NIL$ **alors**
  $racine(A) \leftarrow x$
sinon
  si $y = fg(pere(y))$ **alors**
    $fg(pere(y)) \leftarrow x$
  sinon
    $fd(pere(y)) \leftarrow x$
si $y \neq z$ **alors**
  $cle(z) \leftarrow cle(y)$
  Si y a d'autres champs les copier
retourner y (la place est libérée)

Les fonctions de fusion, partition sont laissées en exercice.

Théorème 7 *Il est possible de réaliser les opérations Rechercher, Min, Max, Successeur, Prédécesseur, Inserer, Detruire en* $O(hauteur(A))$ *sur une arborescence binaire ordonnée* A.

Preuve: Il suffit de remarquer que tous les algorithmes précédents ont une complexité en $O(hauteur(A))$.

Rappelons que pour certaines arborescences binaires ordonnées :
$hauteur(A) \in O(|A|)$.

Afin de concrétiser cette valeur, considérons une arborescence binaire ordonnée contenant des identifiants (par exemple, numéro de Sécurité Sociale) de la population d'un grand pays (approx 200 millions d'habitants). En remarquant que $2^{28} = 268435456$. Dans le pire des cas, i.e. une arborescence

telle que : $hauteur(A) \in O(|A|)$, la recherche de certains éléments dans l'arborescence peut demander jusqu'à 2^{27} tests. Par contre si A est équilibrée, il suffit d'au plus 28 tests.

Afin d'améliorer le résultat du théorème ci-dessus, il faut construire et maintenir des arborescences binaire ordonnées A vérifiant :

$hauteur(A) \in O(log_2(|A|))$.

Toute la difficulté va être de maintenir dynamiquement cette relation lors des opérations d'Insertion et de Destruction.

Pour ce faire plusieurs structures ont été proposées : les arborescences AVL ou arborescences quasi-équilibrées, les arborescences bicolorées, les B-arborescences (très utilisées dans le domaine des bases de données).

Chapitre 4

Algorithmes de graphes

Nous noterons un graphe $G = (X, E)$, où X est un ensemble **fini** de sommets et $E \subseteq X^2$ un ensemble d'arêtes. Une arête est donc constituée de deux sommets. Les graphes non orientés considérés dans ce cours seront, sauf mention contraire, **simples** : sans boucle (arête de type xx) et sans arête multiple (plusieurs arêtes entre deux sommets).

$H = (Y, F)$ est un **sous-graphe induit** de $G = (X, E)$, si $Y \subseteq X$ et si $F = \{e \in E |$ les deux extrémités de e sont dans $Y\}$.

$H = (Y, F)$ est un **sous-graphe partiel** de $G = (X, E)$, si $Y \subseteq X$ et si $F \subseteq E \cap Y^2$.

En général $|X| = n$ et $|E| = m$. Le **degré** d'un sommet noté $d(x)$ est le nombre d'arêtes adjacentes à x. Un sommet **pendant** dans un graphe, est un sommet x tel que : $d(x) = 1$.

Une **chaîne** de longueur k est un graphe :

$P = (\{x_0, x_1, \ldots, x_k\}, \{x_0x_1, x_1x_2, \ldots x_{k-1}x_k\})$ ayant $k + 1$ sommets et k arêtes, noté P_k. Ce graphe s'appelle une chaîne de longueur k joignant x_0 à x_k.

On confond souvent la chaîne avec l'une des deux séquences : $[x_0, x_1, \ldots, x_k]$ ou $[x_k, x_{k-1}, \ldots, x_0]$. Un graphe réduit à un sommet et sans arête est une chaîne de longueur 0.

Un **cycle** de longueur k est un graphe :

$C = (\{x_1, \ldots, x_k\}, \{x_1x_2, x_2x_3, \ldots x_{k-1}x_k, x_kx_1\})$ ayant k sommets et k arêtes, noté C_k. On confond souvent le cycle avec l'une des séquences : $[x_1, x_2, \ldots, x_k, x_1]$ (à une permutation circulaire près).

Avec de telles définitions, une chaîne (resp. un cycle) ne passe qu'une fois par un sommet, cela correspond à une chaîne (resp. cycle) élémentaire chez

certains auteurs.

On appelle **marche** dans $G = (X, E)$ une séquence non nulle $M = x_0 e_1 x_1 e_2 \dots e_k x_k$ alternant sommets et arêtes telle que :

$\forall 1 \leq i \leq k$, les extrémités de e_i sont x_{i-1} et x_i. On dit que la marche est de longueur k et qu'elle joint x_0 à x_k.

Lorsque $x_0 = x_k$ on dit que la marche est **fermée** ou un **pseudocycle**. Si le graphe est simple, nous pouvons omettre les arêtes et noter la marche par $[x_0, x_1, \dots, x_k]$.

Lorsque toutes les arêtes de la marche sont différentes on dit que la marche est un **sentier**[1], lorsque tous les sommets sont distincts la marche est une chaîne.

Proposition 2 *S'il existe une marche finie joignant x à y dans G, alors il existe une chaîne joignant x à y dans G, en outre les sommets et les arêtes de cette chaîne appartiennent à la marche.*

Preuve: Soit M la marche de G joignant x à y. Il existe donc une marche de longueur minimum joignant x à y dans G. Notons M', cette marche. Si elle n'est pas élémentaire, on peut extraire une marche de longueur strictement plus petite, d'où la contradiction. □

N.B. Cette preuve repose sur la finitude du graphe, car si M est de longueur infinie, il n'est pas sûr que M' soit de longueur strictement inférieure.

Proposition 3 *S'il existe un pseudo-cycle de longueur impaire dans G, alors il existe un cycle impair dans G.*

Un graphe $G = (X, E)$ est **connexe**, si $\forall x, y \in X$, il existe une chaîne allant de x à y.

Proposition 4 *On considère un graphe $G = (X, E)$ connexe, si $\forall x \in X$, $d(x) \leq 2$ alors G est soit une chaîne, soit un cycle.*

Preuve: Si G n'a aucune arête, vu l'hypothèse de connexité, G est réduit à un sommet et est donc une chaîne de longeur 0. Sinon G possède au moins une arête, donc au moins une chaîne et soit C une chaîne de longueur maximale de G. Seules les extrémités x, y de la chaîne peuvent être adjacentes à des

[1] Ces définitions de chaînes, marches, sentiers et autres cycles ne sont pas standard dans la littérature

arêtes n'appartenant pas à la chaîne (car les autres sommets de la chaîne sont déjà de degré 2). Mais si ces arêtes ont une extrémité hors de la chaîne, celle-ci ne serait pas de longueur maximale. Donc soit $d(x) = d(y) = 1$ et G est une chaîne, soit $xy \in E$ et G est un cycle. □

Théorème 8 *Les 6 conditions suivantes sont équivalentes et caractérisent les arbres :*

1. *G est connexe minimal (si on enlève une arête, le graphe n'est plus connexe)*
2. *G est sans cycle maximal (si on ajoute une arête, le graphe admet un cycle)*
3. *G est connexe sans cycle*
4. *G est connexe avec $n-1$ arêtes*
5. *G est sans cycle avec $n-1$ arêtes*
6. *$\forall x, y \in X$, il existe une chaîne unique joignant x à y.*

Preuve: Il est facile de vérifier que les conditions 1, 2, 3 et 6 sont équivalentes.

Montrons l'équivalence avec les conditions 4 et 5. Pour ce faire nous allons commencer par deux lemmes.

Lemme 1 *Un graphe connexe ayant $n-1$ arêtes possède nécessairement un sommet pendant.*

Preuve: G étant connexe, $\forall x \in X$, $d(x) \geq 1$ (il n'y a pas de sommet isolé). S'il n'existe pas de sommet pendant, alors $\forall x \in X$, $d(x) \geq 2$ et donc $\Sigma_{x \in X} d(x) \geq 2n > 2m = 2n - 2$, d'où la contradiction. □

Lemme 2 *Un graphe G sans cycle, ayant au moins une arête, possède nécessairement un sommet pendant.*

Preuve: Il existe au moins une chaîne dans G. Considérons une chaîne $[x = x_1, \ldots, x_k = y]$ de longueur maximale.

Si x n'est pas un sommet pendant alors il admet un autre voisin z différent de x_2. Si $z \in \{x_3, \ldots, x_k\}$, alors l'arbre possède un cycle ce qui est exclu.

Donc $z \notin \{x_2, \ldots, x_k\}$, mais alors la chaîne choisie n'était pas maximale. □

Revenons à la preuve du théorème. Le premier lemme permet de montrer par induction 4 implique 3.

En effet d'après ce lemme un graphe G, vérifiant la condition 4, admet au moins un sommet pendant x. Si l'on considère le graphe $G' = G - x$. G' est sans cycle ssi G est sans cycle. En outre G' a un sommet et une arête de moins que G et vérifie donc aussi la condition 4. On peut donc réitérer le procédé jusqu'à arriver sur le graphe vide réduit à un sommet qui est évidemment sans cycle. Donc 4 implique 3.

Un raisonnement analogue basé sur le lemme 2 permet de montrer 5 implique 3.

Montrons maintenant 3 implique 4 et 5. Le seul graphe à un sommet est connexe et sans cycle et il a bien $n - 1 = 0$ arête.

Suposons maintenant vraie jusqu'à l'ordre $n - 1$ que la condition 3 implique 4 et 5. On considère un graphe $G = (X, E)$ ayant n sommets. Soit $a \in X$. $G - a$ possède $k = d(a) \geq 1$ composantes connexes $G_i = (X_i, E_i)$ qui sont nécessairement connexes et sans cycle, vérifiant ainsi la condition 3. L'hypothèse d'induction sur les graphes G_i donne :

$|E_i| = |X_i| - 1$. D'où l'on déduit :

$|E| = \Sigma_{1 \leq i \leq k} |E_i| + k = \Sigma_{1 \leq i \leq k} (|X_i| - 1) + k = \Sigma_{1 \leq i \leq k} |X_i| = |X| - 1$. $\square$,

Corollaire 2 *Tout arbre non réduit à un sommet admet au moins deux sommets pendants (sommet de degré 1)*

Remarquons que les deux preuves des lemmes précédents permettent d'affirmer que si l'arbre n'est pas réduit à un sommet, il admet au moins **deux** sommets pendants. En outre lorsqu'on supprime un sommet pendant à un arbre, le graphe obtenu est encore un arbre, ce qui nous donne un schéma récursif simple pour manipuler les arbres.

4.1 Exercices

1. Montrer qu'un graphe non orienté sans boucle ni arête multiple possède au moins deux sommets de même degré.

4.2 Dualité cycle-cocycle

Nous allons maintenant exhiber une propriété d'échange ainsi qu'une dualité.

Définition 6 *Un cocyle $\theta(A)$ (ou une coupe (A, X-A)) d'un graphe $G = (X, E)$ est un ensemble d'arêtes incidentes à un sous ensemble $A \subset X$, i.e. l'ensemble des arêtes n'ayant qu'une extrémité dans A l'autre extrémité étant dans X-A.*

Par définition $\theta(A) = \theta(X - A)$.

Lemme 3 *Si un cycle μ et un cocycle θ ont une arête en commun, ils en ont au moins une deuxième.*

Il est possible de préciser un peu le résultat : un cycle et un cocyle ont en commun un nombre pair d'arêtes.

Lemme 4 *Soit $G = (X, E)$ un graphe connexe et $T = (X, F)$ un arbre recouvrant de G, pour toute arête $e \in E - F$ il existe une arête $f \in F$ telle que $T' = T - f + e$ soit un arbre recouvrant de G.*

Une arête d'un graphe est appelée un **isthme** lorsque sa suppression disconnecte le graphe.

Lemme 5 *Soit $G = (X, E)$ un graphe connexe et $T = (X, F)$ un arbre recouvrant de G, pour toute arête $f \in F$, si $G - f$ est toujours connexe (i.e. f n'est pas un isthme de G), il existe une arête $e \in E - F$ telle que $T' = T - f + e$ soit un arbre recouvrant de G.*

Dans les deux cas, on dit alors avoir procédé à un échange. Il est facile d'en déduire que si T_1 et T_2 sont deux arbres recouvrants de G, il existe une suite finie d'échanges qui permet de passer de T_1 à T_2.

4.3 Espaces vectoriels des cycles-coycles

À chaque cycle (resp. cocycle) on associe le vecteur caractéristique de ses arêtes, un vecteur de $\{0, 1\}^m$.

On considère les coefficients dans $Z/2Z$.

Il est facile de vérifier que : $\theta(A) = \Sigma_{x \in A} \theta(x)$

Ce qui permet de définir la somme de cocycles :

$\theta(A) + \theta(B) = \theta(A \Delta B)$

Nous avons ainsi un sous espace vectoriel des cocycles.

Lorsqu'on considère la somme de cycles, les arêtes communes disparaissent, cependant la définition de la somme de cycles quelconques, nous oblige à considérer comme cycle une union disjointe de cycles. À ce prix on peut définir le sous espace vectoriel des cycles.

On vérifie bien que :

$\forall$ cycle μ et $\forall$ cocycle $\theta(A)$, on a bien

$\mu.\theta(A) = 0$. Les deux sous-espaces sont donc orthogonaux.

Théorème 9 *On considère un graphe $G = (X, E)$ connexe, la dimension de l'espace vectoriel des cycles (resp. cocycles) est $m - n + 1$ (resp. $n - 1$).*

Preuve: Soit $T = (X, F)$ un arbre recouvrant de G. Pour $\forall e \in E - F$, $T+e$ contient un cycle unique μ_e. Ceci nous permet de construire un ensemble de $m - n + 1$ cycles indépendants, car ils ont tous une arête qui les distingue de tous les autres. Donc $dim(Cycles) \geq m - n + 1$.

De même pour $\forall f \in F$, $T - f$ admet deux composantes connexes, et il existe donc un cocyle unique de G, noté θ_f. On peut ainsi construire $n - 1$ cocyles indépendants. Donc $dim(Cocycles) \geq n - 1$.

Comme les espaces vectoriels sont orthogonaux,

$dim(Cycles) + dim(Cocyles) \leq m$, d'où le résultat annoncé. $\square$

On appelle base fondamentale de cycles d'un graphe, une base obtenue à l'aide d'un arbre recouvrant. Il est facile de voir en considérant le graphe 3-soleil, qu'il existe des bases de cycles qui ne sont pas fondamentales.

Ces bases de cycles sont utiles en chimie organique pour l'analyse des fonctions associées à une molécule.

Chapitre 5

Arbre recouvrant de poids minimum et algorithmes gloutons

On considère, un graphe non orienté connexe $G = (X, E)$ et une fonction de poids $\omega : E \to R_+$. Et l'on cherche donc un un sous-graphe $T = (X, F)$ connexe dont la somme des poids des arêtes est minimale.

Comme application on peut imaginer que le graphe G représente le réseau des rues d'une ville que l'on cherche à cabler. Il faut donc trouver un graphe **connexe** (chaque sommet doit avoir accès au réseau) et de coût minimum. En effet le coût d'installation d'un cable peut dépendre de la rue choisie (faire passer un cable sous un tramway ou un monument historique peut coûter cher) c'est ce que modélise la valuation ω.

Il est facile de voir, comme la valuation ω est positive qu'une solution optimale d'un tel problème est nécessairement un arbre (graphe connexe minimal, cf. le chapitre précédent). Ce problème est donc appelé **le problème de l'arbre recouvrant de poids minimum**.

Ce problème admet une solution, car l'ensemble des arbres recouvrants est un ensemble fini, donc il admet un élément de coût minimal. Tout le problème c'est de le trouver sans énumérer tous les arbres recouvrants car il peut y en avoir beaucoup (ex : n^{n-2} pour un graphe complet, d'après le célèbre théorème de Cayley publié en 1889).

Pour ce faire on va construire un schéma d'algorithme, basé sur une bi-coloration en rouge ou bleu des arêtes de G. Cette présentation très élégante est due à Tarjan [32].

Au départ les arêtes ne sont pas colorées et pour les colorier (de manière définitive) nous allons utiliser les deux règles suivantes :

Règle Bleue : Choisir un cocycle θ qui ne contient pas d'arête bleue, colorier en bleu une arête non colorée de coût minimal de θ.

Règle Rouge : Choisir un cycle μ qui ne contient pas d'arête rouge, colorier en rouge une arête non colorée de coût maximal de μ

On remarquera que ces règles sont duales l'une de l'autre, par l'échange cycle-cocyle et bleu-rouge.

Arbre recouvrant de poids minimum
tant que *une règle est applicable* **faire**
⌊ l'appliquer

Théorème 10 *Si G est connexe, l'algorithme précédent calcule un arbre recouvrant de poids minimum.*

Preuve:

Le graphe étant fini, il existe un nombre fini d'arbres recouvrants, parmi ceux-ci il en existe un de poids minimum, notons le $T = (X, F)$.

Nous allons prouver que l'invariant suivant est maintenu dans l'algorithme :

Invariant principal : Il existe un arbre de poids minimum qui contient toutes les arêtes bleues et aucune rouge.

La preuve se fait par récurrence sur le nombre d'application des règles. Initialement l'invariant est trivialement vrai.

Montrons tout d'abord que les arêtes bleues forment une forêt de G (i.e. un graphe sans cycle). La preuve se fait par induction sur le nombre d'applications de la règle bleue. Au départ il n'y a pas d'arêtes bleues donc pas de cycle et supposons que l'application de la ième règle bleue sur l'arête $ab \in \theta$ introduise un cycle entièrement bleu μ. Mais d'après le lemme 3 θ et μ ont une autre arête en commun, d'où la contradiction.

Supposons le vrai lorsque k arêtes ont déjà été coloriées par application des ces deux règles, il existe donc un arbre $T = (X, F)$ recouvrant de poids minimal qui contient toutes les arêtes bleues et aucune rouge.

Considérons une nouvelle application, par exemple de la règle bleue. On colore donc en bleu une arête f d'un cocyle θ. Dans le cas où $f \in F$ alors l'invariant est encore trivialement vérifié. Supposons donc le contraire, f n'appartient pas à F. Le graphe $T = (X, F \cup \{f\})$ admet donc un cycle μ. f appartient à μ et θ, or un cycle et un cocycle ont au moins deux arêtes en

commun, soit g une telle arête. Nécessairement g $\in \theta$ et d'après l'application de la règle bleue $\omega(f) \leq \omega(g)$.

Cependant $T' = (X, F \cup \{f\} - g)$ est aussi un arbre recouvrant (échange des arêtes f et g, on note $T' = T + f - g$. Nous avons $\omega(T') = \omega(T) + \omega(f) - \omega(g)$, la minimalité de T impose donc que $\omega(f) = \omega(g)$ et donc $\omega(T') = \omega(T)$, et l'arbre T' permet de montrer que l'invariant est encore vrai.

Le raisonnement est similaire dans le cas d'une application de la règle rouge.

Examinons ce que se passe lorsqu'on ne peut plus appliquer de règle. Supposons que les arêtes bleues forment un graphe partiel ayant plusieurs composantes connexes sur les ensembles de sommets $X_1, \ldots, X_k$, avec $k \geq 2$. Considérons le cocyle engendré par X_1, $\theta(X_1)$ qui ne contient par définition aucune arête bleue. Mais est-il possible que toutes ses arêtes soient rouges ? Montrons le contraire. L'ivariant étant toujours vrai, il existe un arbre de poids minimal $T_{final} = (X, F_{final})$ qui contient toutes les arêtes bleues et aucune rouge. Cet arbre utilise au moins une arête de $\theta(X_1)$, et cette arête ne peut donc être rouge, donc il existe au moins une arête de $\theta(X_1)$ incolore. On peut appliquer sur $\theta(X_1)$ la règle bleue et colorier en bleu l'arête incolore de poids minimal, d'où la contradiction.

Ceci implique qu'à la fin les arêtes bleues forment un graphe partiel connexe de G lorsque celui-ci est connexe. Ce graphe est nécessairement un arbre d'après l'invariant principal car les arêtes bleues sont incluses dans un arbre de poids minimum.

Montrons maintenant que toutes les arêtes sont bien coloriées. Pour ce faire supposons qu'il existe une arête $g \in E$ non coloriée. Nécessairement $g \notin F$, mais alors il existe un cycle unique dans $T + g$ sur lequel nous allons pouvoir appliquer la règle rouge, contradiction.

Conclusions : l'algorithme s'arrête donc et lorsqu'aucune règle n'est applicable, toutes les arêtes sont coloriées. Les arêtes bleues constituent un arbre recouvrant de poids minimum et les arêtes rouges un coarbre (un coarbre est par définition le complémentaire d'un arbre recouvrant de G).

□

L'algorithme générique proposé est dit **glouton** car il ne remet jamais en question la coloration d'une arête. Il existe plusieurs mises en application différentes de cet algorithme générique. Les plus célèbres sont dues à Kruskal [23] et à Prim [27].

Dans tous les algorithmes présentés ci-après pour la recherche d'un arbre recouvrant de poids minimum, le graphe initial G est supposé connexe.

Algorithme de Kruskal
Trier les arêtes de G par poids croissant en une liste L
$F \leftarrow \emptyset$
tant que $|F| < n-1$ *et* $L \neq \emptyset$ **faire**
  $e \leftarrow Premier(L)$
  Retrait(L, e) ; **si** $T = (X, F \cup \{e\})$ *n'a pas de cycle* **alors**
    $F \leftarrow F \cup \{e\}$

Cet algorithme revient à utiliser n-1 fois la règle bleue lorsqu'on ajoute une arête dans F, et un autant de fois la règle rouge que l'on détecte la présence d'un cycle en ajoutant l'arête e. En outre il est facile de montrer qu'à chaque itération la propriété : F *est sans cycle* est maintenue, et donc à la fin F est un arbre recouvrant de G.

Il n'est donc pas nécessaire de colorier explicitement les arêtes non considérées (les arêtes de L à la fin de l'algorithme) car elles sont nécessairement rouges et ne peuvent intervenir dans un arbre de poids minimal (cf. l'invariant).

L'algorithme de Prim revient aussi à toujours appliquer la règle bleue, en travaillant toujours sur un seul cocyle que l'on maintient au cours de l'algorithme. En outre la propriété F *engendre un graphe connexe est maintenue*, et à la fin F est nécessairement un arbre recouvrant de G.

Algorithme de Prim
$F \leftarrow \emptyset$
$A \leftarrow \{x_0\}$
tant que $|F| < n-1$ **faire**
  Calculer l'arête $e = (a, x)$ (avec $a \in A$) de coût minimal de $\theta(A)$
  $F \leftarrow F \cup \{e\}$
  $A \leftarrow A \cup \{x\}$

Une variante due à Boruvka (1926) d'après [19], est d'autant plus intéressante qu'elle précédait historiquement celles de Kruskal et de Prim.

Algorithme de Boruvka
$F \leftarrow \emptyset$
tant que G *n'est pas trivial* **faire**
    Détruire les boucles de G
    Remplacer les arêtes multiples entre deux sommets, par une seule dont le poids est le minimum des poids de ces arêtes multiples
    pour $\forall x \in G$ **faire**
        trouver l'arête e_x de poids minimum adjacente à x
        $F \leftarrow F \cup \{e_x\}$
        Contracter e_x

Montrer que cet algorithme se prête aisément à la parallélisation.

5.1 Quelques variantes

Kruskal à l'envers
Trier les arêtes de G par poids décroissant en une liste L
$F \leftarrow E$
tant que $|F| > n-1 et L \neq \emptyset$ **faire**
  $e \leftarrow Premier(L)$
  Retrait(L, e) ; **si** $T = (X, F - \{e\})$ *est connexe* **alors**
    $F \leftarrow F - \{e\}$

L'algorithme "Kruskal à l'envers" est intéressant à utiliser lorsque le nombre d'arêtes du graphe G est très proche de n-1, car le nombre de passages dans la boucle **tant que** est borné par $|E| - n + 1$.

Kruskal-dans-le-désordre
Construire une liste L avec les arêtes de G
$F \leftarrow \emptyset$
tant que $L \neq \emptyset$ **faire**
  $e \leftarrow Premier(L)$
  Retrait(L, e)
  si $T = (X, F \cup \{e\})$ *n'a pas de cycle* **alors**
    $F \leftarrow F \cup \{e\}$
  sinon
    si $\omega(e) < \omega(f)$, *où f est une arête de poids maximum du cycle de* $F \cup \{e\}$ **alors**
      $F \leftarrow (F \cup \{e\}) - \{f\}$

L'algorithme **Kruskal-dans-le-désordre** permet d'éviter la phase initiale de tri des arêtes du graphe suivant leur poids.

5.1.1 Ordre lexicographique sur les arbres

L'ensemble des arbres recouvrants d'un graphe peuvent être muni d'un ordre partiel $\preccurlyeq$, défini comme suit :

À chaque arbre T on associe le mot $m(T)$ constitué des poids des arêtes de l'arbre classées par ordre croissant.

Nous pouvons alors définir $T \preccurlyeq T'$ ssi $m(T) \leq_{lex} m(T')$.

L'ordre défini ci-dessus n'est qu'un ordre partiel, car un même mot peut-

être associé à deux arbres, qui ont alors le même poids.

Il est facile de constater que l'algorithme de Kruskal construit un arbre minimal pour l'ordre $\preccurlyeq$. Cependant nous avons un résultat beaucoup plus fort.

Propriété 10 *Un arbre T est de poids minimum ssi il est minimal pour l'ordre $\preccurlyeq$.*

Chapitre 6

Graphes orientés

6.1 Définitions de base sur les graphes orientés

Nous noterons un graphe orienté $G = (X, U)$, où X est un ensemble **fini** de sommets et $U \subseteq X^2$ un ensemble d'arcs. Un arc (x, y) est donc constituée de deux sommets : une origine x et une extrémité terminale y. Les graphes orientés considérés dans ce cours seront, sauf mention contraire, **simples** : sans boucle (arête de type xx) et sans arc multiple (plusieurs arcs entre deux sommets).

$H = (Y, V)$ est un **sous-graphe induit** de $G = (X, U)$, si $Y \subseteq X$ et si $V = \{(x, y) \in U \mid x, y \in Y\}$.

$H = (Y, V)$ est un **sous-graphe partiel** de $G = (X, U)$, si $Y \subseteq X$ et si $V \subseteq U \cap Y^2$.

En général $|X| = n$ et $|E| = m$. Le **degré** d'un sommet noté $d+(x)$ (resp. $d^-(x)$) est le nombre d'arcs sortant (resp. rentrant) de x. Un sommet **pendant** dans un graphe, est un sommet x tel que : $d(x) = 1$.

Une **chemin** de longueur k est un graphe :

$P = (\{x_0, x_1, \ldots, x_k\}, \{(x_0, x_1), (x_1, x_2), \ldots (x_{k-1}, x_k)\})$ ayant $k + 1$ sommets et k arcs, noté P_k. Ce graphe s'appelle un chemin de longueur k joignant x_0 à x_k.

On confond souvent le chemin avec la séquence : $[x_0, x_1, \ldots, x_k]$. Un graphe réduit à un sommet et sans arc est une chemin de longueur 0.

Un **circuit** de longueur k est un graphe :

$C = (\{x_1, \ldots, x_k\}, \{(x_1, x_2), (x_2, x_3), \ldots (x_{k-1}, x_k), (x_k, x_1)\})$ ayant k sommets et k arcs, noté C_k. On confond souvent le circuit avec l'une des sé-

quences : $[x_1, x_2, \ldots, x_k, x_1]$ (à une permutation circulaire près).

Avec de telles définitions, un chemin (resp. un circuit) ne passe qu'une fois par un sommet, cela correspond à une chaîne (resp. circuit) élémentaire chez certains auteurs.

On appelle **marche orientée** dans $G = (X, U)$ une séquence non nulle $M = x_0 u_1 x_1 u_2 \ldots u_k x_k$ alternant sommets et arcs telle que :

$\forall 1 \leq i \leq k$, l'origine de u_i est x_{i_1} et son extrémité terminale x_i. On dit que la marche orientée est de longueur k et qu'elle joint x_0 à x_k.

Lorsque $x_0 = x_k$ on dit que la marche est **fermée** ou un **pseudocycle**. Si le graphe est simple, nous pouvons omettre les arcs et noter la marche par $[x_0, x_1, \ldots, x_k]$.

Lorsque tous les arcs de la marche orientée sont différents on dit que la marche est une **piste**[1], lorsque tous les sommets sont distincts la marche est un chemin.

Pour les graphes orientés, les choses se compliquent un peu, car nous retrouvons les objets du chapitre précédents dès que l'on considère le graphe non orienté sous-jacent, sur lequel on peut identifier des chaînes, cycles et autres marches.

Proposition 5 *S'il existe une marche orientée finie joignant x à y dans G, alors il existe un chemin joignant x à y dans G, en outre les sommets et les arcs de ce chemin appartiennent à la marche.*

Preuve: Soit M la marche orientée de G joignant x à y. Il existe donc une marche orientée de longueur minimum joignant x à y dans G. Notons M', cette marche orientée. Si elle n'est pas élémentaire, on peut extraire une marche orientée de longueur strictement plus petite, d'où la contradiction. □

N.B. Cette preuve repose sur la finitude du graphe, car si M est de longueur infinie, il n'est pas sûr que M' soit de longueur strictement inférieure.

6.2 Arborescences

Une **racine** d'un graphe orienté $G = (X, U)$ est un sommet $r \in X$ tel que $\forall x \in X$, $x \neq r$, il existe un chemin de r à x.

[1] Ces définitions de chemins, marches, pistes et autres circuits ne sont pas standard dans la littérature.

Théorème 11 *Pour un graphe orienté $G = (X, U)$ avec $|X| \geq 2$, les 6 conditions suivantes sont équivalentes et caractérisent les arborescences :*

1. *G est connexe sans cycle et admet une racine*
2. *G possède une racine et a $n - 1$ arcs*
3. *G possède une racine r et $\forall x \in X$, il existe un chemin unique de r à x*
4. *G possède une racine et est minimal avec cette propriété (tout retrait d'un arc de G supprime la propriété)*
5. *G est connexe et il existe un sommet $r \in X$, avec $d^-(r) = 0$ et $\forall x \in X$, $x \neq r$, $d^-(x) = 1$*
6. *G est sans cycle et il existe un sommet $r \in X$, avec $d^-(r) = 0$ et $\forall x \in X$, $x \neq r$, $d^-(x) = 1$*

Preuve: (2) $\rightarrow$ (3) :

L'existence d'une racine implique la connexité de G, le nombre d'arcs permet de conclure que le graphe non orienté sous-jacent est un arbre. D'après le théorème sur les arbres, $\forall x \in X$, $x \neq r$ le chemin de r à x est nécessairement unique.

(3) $\rightarrow$ (4) : trivial.

(4) $\rightarrow$ (5) :

□

Ainsi dans une arborescence, tout sommet sauf la racine admet un prédécesseur unique ($d^-(x) = 1$) et donc on peut représenter une arborescence à l'aide de la fonction Parent : $X \rightarrow X$.

Chapitre 7

Parcours de graphes

7.1 Parcours générique

Un algorithme de parcours dans un graphe peut se formaliser à l'aide de deux ensembles de sommets OUVERTS et FERMES, le premier contenant les sommets à explorer et le deuxième les sommets déjà explorés. L'exploration d'un sommet consistant à examiner tous les voisins d'un sommet : visite de tous les arcs sortants du sommet.

ParcoursGénérique(G, x_0)
Données: un graphe orienté $G = (X, U)$, un sommet $x_0 \in X$
Résultat: une arborescence de chemins issus de x_0
$OUVERTS \leftarrow \{x_0\}$
$FERMES \leftarrow \emptyset$
$Parent(x_0) \leftarrow NIL$
$\forall y \neq x_0$, $Parent(y) \leftarrow y$
tant que $OUVERTS \neq \emptyset$ **faire**
  $z \leftarrow Choix(OUVERTS)$
  $Ajout(z, FERMES)$
  Explorer(z)
  $Retrait(z, OUVERTS)$

Explorer(z)
pour *Tous les voisins y de z* **faire**
  si $y \in FERMES$ **alors**
    Ne rien faire
  si $y \in OUVERTS$ **alors**
    Ne rien faire
  $Ajout(y, OUVERTS)$
  $Parent(y) \leftarrow z$

Théorème 12 *L'algorithme précédent calcule en $O(n+m)$ une arborescence de racine x_0 recouvrant l'ensemble des sommets atteignables à partir de x_0.*

Preuve: Il est facile de vérifier que l'algorithme précédent vérifie bien les invariants suivants :

Invariants :

– La fonction Parent définit une arborescence de racine x_0,

– $\forall x \in OUVERTS$, il existe un chemin de x_0 à x.

Ainsi à la fin du parcours, l'ensembles des sommets FERMES est égal à l'ensemble des sommets atteignables dans le graphe G à partir de x_0. Cet ensemble est décrit à l'aide de la fonction Parent. Un sommet est exploré au plus une fois et donc l'algorithme est en $O(n+m)$ si le graphe est représenté par ses listes d'adjacence. □

Ce parcours s'applique aux graphes non orientés et orientés. Le déroulement d'un parcours permet d'ordonner (numéroter) les sommets d'un graphe. Ces ordres sont liés à la gestion de l'ensemble des sommets OUVERTS et à la fonction de choix du prochain sommet à explorer. Cet ensemble peut être géré comme une pile (Parcours en profondeur) ou une File (Parcours en largeur).

Lorsque tous les sommets du graphe ne sont atteignables à partir de x_0 si l'on veut parcourir tout le graphe, il suffit d'ajouter les instructions suivantes et de représenter l'ensemble des sommets FERMES à l'aide d'un tableau de booléens :

pour tous les $v \in X$ **faire**
  $Ferme(x) \leftarrow Faux$
pour tous les $v \in X$ **faire**
  si $Ferme(x) = Faux$ **alors**
    ParcoursGénérique(G, x)

Dans ce cas l'algorithme calcule une forêt d'arborescences recouvrante de G.

7.1.1 Parcours en largeur

On obtient un parcours en largeur dès que l'on gère l'ensembles des sommets OUVERTS comme une file (premier entré, premier sorti).

7.1.2 Parcours en profondeur

Défini en premier par Tremeaux 1882 puis par Tarry 1895 pour des problèmes de parcours dans les labyrinthes. Redécouvert en 1972 par Tarjan et Hopcroft afin de concevoir des algorithmes de résolution des problèmes de recherche dans un graphe : le calcul des composantes fortement connexes, celui des composantes 2-arêtes connexes, le test de planarité ... On obtient un parcours en profondeur dès que l'on gère l'ensembles des sommets OUVERTS comme une pile (dernier entré, premier sorti). Cette gestion d'une pile se prête à une écriture récursive du programme, comme suit :

$DFS(G)$:
forall $v \in X$ **do**
  $Ferme(x) \leftarrow Faux$
forall $v \in X$ **do**
  si $Ferme(x) = Faux$ **alors**
    $Explorer(G, x)$

$Explorer(G, x)$:
$Ferme(x) \leftarrow Vrai$;
pre(x) ;
forall $xy \in U$ **do**
  si $Ferme(y) = Faux$ **alors**
    $Explorer(G, y)$
  post(x) ;

Et les deux fonctions suivantes utilisant deux variables : comptpre et comptpost étant initialisées à 1.

$pre(x)$:
$pre(x) \leftarrow comptpre$;
$comptpre \leftarrow comptpre + 1$;

$post(x)$:
$post(x) \leftarrow comptpost$;
$comptpost \leftarrow comptpost + 1$;

En utilisant les propriétés des numérotations $pre(x)$ et $post(x)$, il est possible de reconstituer le fonctionnement de la pile qui a permis de gérer l'ensemble de sommets OUVERTS. Ce parcours a permis d'écrire des algorithmes linéaires (en $O(n+m)$) pour la recherche des composantes fortement connexes d'un graphe orienté, de recherche des composantes 2-arête connexes d'un graphe non orienté, ainsi qu'un algorithme qu'un test de planarité.

7.1.2.1 Une partition des arcs de G

Comme tous les parcours, un parcours en profondeur permet de construire une forêt d'arborescences. On distingue trois types d'arcs : les arcs de retour, les arcs traversiers et enfin les arcs de transitivité de la forêt d'arborescences Les ordres pre et post permettent de les distinguer, comme l'indiquent les trois équivalences suivantes :

- xy arc de retour ssi $pre(y) \leq pre(x)$ and $post(x) \leq post(y)$
- xy arc traversier ssi $pre(y) \leq pre(x)$ and $post(y) \leq post(x)$
- xy arc de transitivité de la forêt d'arborescences ssi $pre(x) \leq pre(y)$ and $post(y) \leq post(x)$
- Le cas où $pre(x) \leq pre(y)$ and $post(x) \leq post(y)$ étant impossible par le principe du parcours en profondeur.

  Ces trois types d'arcs partitionnent les arcs du graphe.

Définition 7 *Une **extension linéaire** d'un graphe sans circuit $G = (X, U)$ est un ordre total sur les sommet de G, noté τ, vérifiant :*
$\forall x, y \in X$, $xy \in U$ *implique* $x \leq_\tau y$.

Certains auteurs appellent une extension linéaire un **tri topologique** ou encore **un ordre total compatible**.

Théorème 13 *Si G est un graphe sans circuit, alors le parcours en profondeur appliqué sur G vérifie la propriété :*
L'ordre inverse de dépilement (noté $post^d$) est une extension linéaire de G.

Preuve: Considérons deux sommets $x, y \in X$ et supposons $xy \in U$. Deux cas sont possibles :

1. $pre(x) < pre(y)$. Mais alors l'exploration de y se terminera donc avant celle de x et nous avons bien : $post^d(x) < post^d(y)$.
2. $pre(y) < pre(x)$. Comme il n'existe pas de chemin de y à x, car sinon G aurait un circuit, nous avons que l'exploration de y se terminera avant celle de x. Et donc $post^d(x) < post^d(y)$.

□

Ce théorème peut aussi s'écrire comme suit :

Théorème 14 *Après un parcours en profondeur sur un graphe G les 3 propriétés suivantes sont équivalentes :*

i *G est sans circuit*

ii *Il n'y a pas d'arc de retour*

iii *$post^d$ est une extension linéaire de G*

Corollaire 3 *$post^d$ est une extension linéaire de G/P le graphe G quotienté par la partition en composantes fortement connexes de G.*

Le parcours en profondeur permet donc de calculer une extension linéaire d'un graphe sans circuit. Il existe d'autres algorithmes, le plus classique étant celui basé sur la remarque que tout graphe sans circuit admet une source (i.e. un sommet tel que $d^-(x) = 0$). Si on modifie un parcours générique comme suit :

ExtensionLineaire(G, S)
Données: Un graphe orienté $G = (X, U)$ sans circuit, S l'ensemble des sources de G
Résultat: Une numérotation des sommets
$OUVERTS \leftarrow \{S\}$;
$FERMES \leftarrow \emptyset$;
$compteur \leftarrow 1$;
tant que $OUVERTS \neq \emptyset$ **faire**
  $z \leftarrow Choix(OUVERTS)$
  $numero(z) \leftarrow compteur$;
  $compteur \leftarrow compteur + 1$;
  $Ajout(z, FERMES)$
  Explorer(z)
  $Retrait(z, OUVERTS)$

Explorer(z)
pour *Tous les successeurs y de z* **faire**
  $d^-(y) \leftarrow d^-(y) - 1$;
  si $d^-(y) = 0$ **alors**
    $Ajout(y, OUVERTS)$

Cette procédure fournit une numérotation des sommets qui est une extension linéaire de G. Si l'on gère l'ensemble des sommets OUVERTS comme une File, l'extension linéaire obtenue est appelée numérotation par niveau, ou par rang.

7.1.2.2 Algorithme de Tarjan 1972

Il suffit de transformer la fonction d'exploration du parcours en profondeur comme suit :

Explorer(G, x) :
$Ferme(x) \leftarrow Vrai$;
$pre(x)$;
$racine(x) \leftarrow pre(x)$;
forall $xy \in U$ **do**
  si $Ferme(y) = Faux$ **alors**
    $Explorer(G, y)$;
    $racine(x) \leftarrow min\{racine(x), racine(y)\}$ **;**
$post(x)$;

Les composantes fortement connexes se repèrent récursivement avec le test $racine(x) = pre(x)$ à la fin de l'exploration de x.

7.1.2.3 Algorithme de Kosaraju 1978, Sharir 1981

1. Exécuter un parcours en profondeur sur G.
2. Faire un autre parcours en profondeur sur G^- avec pour ordre initial l'ordre $post^d$
3. Les arbres de la forêt recouvrante de G^-, sont les composantes fortement connexes de G

7.1.2.4 Implémentations

Dans un parcours en profondeur nous avons le choix de parcourir les successeurs d'un sommet dans un ordre qui peut soit provenir :

- d'un parcours précédent comme dans l'algorithme de Kosaraju et Sharir.
- d'un ordre explicite de préférence sur les liens (utilisé pour les algorithmes d'héritage en programmation objet)
- Un parcours en profondeur traverse chaque arête une fois„
  alors qu'un parcours en largeur examine tout le voisinage d'un sommet à la fois voisinage noté $N(x)$

Chapitre 8

Algorithmes de plus courts chemins

8.1 Introduction

On considère un graphe $G = (X, U)$ orienté et l'on suppose les arcs munis d'étiquettes, i.e. une valuation $\omega : U \to R$.

Le problème de la recherche d'une plus court *chemin* dans un graphe orienté a de très nombreuses applications pratiques, telles :

- Routages de paquets dans les réseaux, routages de véhicules dans les réseaux de transports
- Diamètre d'un réseau de télécommunications (qualité de service)
- Problèmes de Transport
- Jeux (graphe dont les sommets sont les états du jeu et les arcs les transitions légales)
- Investissements, ordonnancements
- Navigation (routeurs de course au large : Vendée Globe Challenge, Route du Rhum, ...)

En général la valuation d'un chemin est la somme des valuations des arcs qui constituent le chemin.

8.2 Les différents problèmes

Pour un graphe $G = (X, U)$ orienté dont les arcs sont valués

- 1. Etant donné deux sommets a et b, trouver une plus courte marche allant de a jusqu'à b dans G.
- 2. Etant donné un sommet a trouver pour chaque $x \in X$ une plus courte marche allant de a jusqu'à x dans G.
- 3. Pour tout $x, y \in G$, trouver une plus courte marche allant de x jusqu'à y.

Dans l'énumération ci-dessus, la complexité du problème est croissante. En effet un algorithme qui résout le problème 3, permet de résoudre le problème 2 et un algorithme qui résout le problème 2 permet de résoudre le problème 1.

Cependant lorsque les valuations choisies sont de signe quelconque, la solution au problème 1 peut-être une marche infinie comprenant un circuit de valeur négative (circuit dit **absorbant**).

Exemple : $valuation(ij) = cout(ij) + taxe - subvention$

Il est tentant de transformer le problème en :

Etant donné deux sommets a et b, trouver le plus court chemin allant de a jusqu'à b dans G.

Mais alors le problème devient NP-difficile.

8.2.1 Algorithme de Dijkstra [16]

Algorithme de Dijkstra 1959
Données: un graphe orienté $G = (X, U)$, une fonction de coût
$\omega : U \to \mathcal{R}^+$
Résultat: une arborescence de chemins issus de x_0
$OUVERTS \leftarrow \{x_0\}$
$FERMES \leftarrow \emptyset$
$Parent(x_0) \leftarrow NIL$
$\forall y \neq x_0\ ,\ Parent(y) \leftarrow y$
$d(x_0) \leftarrow 0$
$\forall y \neq x_0\ ,\ d(y) \leftarrow \top$ (l'élément maximal de T)
tant que $OUVERTS \neq \emptyset$ **faire**
  Choisir un sommet $z \in OUVERTS$ tel que
  $d(z) = min_{y \in OUVERTS}\{d(y)\}$
  $Ajout(z, FERMES)$
  Explorer(z)
  $Virer(z, OUVERTS)$

Explorer(z)
pour *Tous les voisins y de z* **faire**
  si $y \in FERMES$ **alors**
    Ne rien faire
  si $y \in OUVERTS$ **alors**
  sinon
    si $d(z) + \omega(z, y) < d(y)$ **alors**
      $Parent(y) \leftarrow z$
      $d(y) \leftarrow d(z) + \omega(z, y)$
  $Ajout(y, OUVERTS)$
  $Parent(y) \leftarrow z$
  $d(y) \leftarrow d(z) + \omega(z, y)$

Théorème 15 *Lorsque la valuation ω est à valeurs positives, l'algorithme calcule bien une arborescence des plus courts chemins issus de x_0.*

Preuve:

Considérons les invariants principaux de l'algorithme.

Invariants

1. $\forall x \in OUVERTS$, il existe un chemin μ de x_0 à x tel que, $d(x) = \omega(\mu)$
2. À la fin de la ième exploration d'un sommet, $\forall x \in OUVERTS \cup FERMES$, $d(x)$ est égale à la valuation minimale d'un chemin de G de x_0 à x n'utilisant que des sommets $FERMES$ sauf éventuellement x.
3. $\forall u \in FERMES$ and $\forall v \in OUVERTS$, $d(u) \leq d(v)$.

L'invariant 1 se montre facilement par récurrence, montrons l'invariant 2 aussi par récurrence. Pour se faire indexons par i tous les objets de l'algorithme à la fin de l'exploration du ième sommet.

À l'initialisation lorsque $i = 0$, l'invariant est trivialement vrai. Supposons le maintenant vrai jusqu'à l'exploration du (i-1)ème sommet. Soit x le sommet exploré à la ième itération.

Considérons $z \in OUVERTS_i \cup FERMES_i$, il y a plusieurs cas possibles.

– Soit $xz \notin U$, mais alors il n'existe pas de chemin joignant x_0 à z n'utilisant que des sommets de $FERMES_i$ sauf éventuellement z. Comme $d_i(z) = d_{i-1}(z)$, l'hypothèse de récurrence permet de conclure.

- Soit $xz \in U$ et $z \notin OUVERTS_{i-1} \cup FERMES_{i-1}$. Dans ce cas le seul chemin joignant x_0 à z n'utilisant que des sommets de $FERMES_i$ sauf éventuellement z passe par x et l'algorithme affecte bien à $d_i(z)$ la valeur de ce chemin.
- Soit $xz \in U$ et $z \in OUVERTS_{i-1}$. Quand le sommet x est fermé à la ième étape, il existe un chemin joignant x_0 à z n'utilisant que des sommets de $FERMES_i$ sauf éventuellement z passant par x. L'algorithme prend en compte la valeur de ce chemin pour calculer $d_i(z)$, on peut conclure grâce à l'hypothèse de récurrence.
- Soit $xz \in U$ et $z \in FERMES_{i-1}$. Dans ce cas l'algorithme ne fait rien (i.e. $d_i(z) = d_{i-1}(z)$) montrons que c'est justifié. En effet supposons qu'il existe un chemin $\mu = [x_0, x_1, \dots x_k, x_{k+1} = x, z]$ allant de x_0 à z tel que $\omega(\mu) < d_i(z)$.
  Soit $j \in [0, k]$ l'indice du dernier sommet de μ qui ait été exploré avant que z ne soit exploré à l'étape h. Par hypothèse de récurrence $d_h(x_{j+1}) = \omega(\mu[x_0, x_{j+1}])$. Comme les valuations des arcs sont positives, on en déduit : $d_h(x_{j+1}) < d_h(z)$ ce qui contredit l'hypothèse sur j, car l'algorithme aurait du explorer x_{j+1} avant z.

□

8.2.2 Algorithme de Bellman - Ford

Données: un graphe orienté $G = (X, U)$, une fonction de coût $\omega : U \to \mathcal{R}$
Résultat: une arborescence de chemins issus de x_0 ou un circuit négatif

$Parent(x_0) \leftarrow NIL\ \forall y \neq x_0\ ,\ Parent(y) \leftarrow y$
$d(x_0) \leftarrow 0\,;\ \forall y \neq x_0\ ,\ d(y) \leftarrow infini)$
répéter
  $\forall xy \in U$
  si $d(y) < d(x) + \omega(xy)$ **alors**
    $d(y) \leftarrow d(x) + \omega(xy)$ et $pere(y) \leftarrow x$
jusqu'à $|X| - 1$ *fois*

pour chaque $xy \in U$ **faire**
  si $d(y) < d(x) + \omega(xy)$ **alors**
    Il existe un circuit négatif

- Complexité en $O(n.m)$
- Invariant :
  $\forall i, x$,
  $d_i(x)$ = la longueur minimum d'une marche orientée de s à x ayant au plus i arcs.
- A la fin de l'algo la fonction père représente une arborescence des plus courts chemins.

8.2.3 L'algorithme A*

Si la fonction choix fournit un sommet z vérifiant :
$d(z) + h(z) = min_{y \in OUVERTS}\{d(y) + h(y)\}$ où h(y) est une information "heuristique" sur la distance qui reste à parcourir.

et si l'instruction : "Ne rien faire" de l'algorithme de Dijkstra est remplacée par :

si $d(z) + \omega(z, y) < d(y)$ **alors**
  $Parent(y) \leftarrow z$
  $d(y) \leftarrow d(z) + \omega(z, y)$
  $Ajout(y, OUVERTS)$

On suppose que cette fonction $h(y)$ est une information disponible en chaque sommet du graphe. L'usage de cet algorithme est adapté au cas où l'on cherche un plus court chemin d'un sommet x_0 à un sommet t. La valeur de $h(x)$ pour un sommet x donné, étant une estimation de la distance qu'il reste à parcourir pour aller de x à t.

Dans les deux instanciations précédentes, le goulot d'étranglement de complexité provient de la gestion de l'ensembles des OUVERTS. Il faut utiliser une structure de données qui permette le calcul du minimum en $O(logn)$ ou mieux.

8.2.3.1 Algorithme de Prim

L'algorithme de Prim que nous avons vu au chapitre sur les arbres recouvrants de poids minimum peut aussi se voir comme un algorithme de plus court chemin. Pour cela il suffit de remarquer que le graphe est non orienté, les arêtes sont valuées et si le sommet à explorer est celui qui vérifie :
$d(z) = min_{y \in OUVERTS}\{d(y)\}$

et si l'exploration devient :

Explorer(z)
pour *Tous les voisins y de z* **faire**
  si $y \in FERMES$ **alors**
    Ne rien faire
  si $y \in OUVERTS$ **alors**
  sinon
    si $\omega(z,y) < d(y)$ **alors**
      $Parent(y) \leftarrow z$
      $d(y) \leftarrow \omega(z,y)$
  $Ajout(y, OUVERTS)$
  $Parent(y) \leftarrow z$
  $d(y) \leftarrow \omega(z,y)$

L'algorithme ci-dessus calcule un arbre de poids minimum de G et c'est une implémentation de l'algorithme de Prim. En outre, il est facile de montrer que l'arborescence obtenue est celle des distances minimum (suivant la distance du max) issue de x_0.

Chapitre 9

Fermeture transitive

- Dans cette section, nous considérons les graphes orientés comme des relations binaires, et l'on peut donc se poser la question de leur transitivité.
- Etant donné $G = (X, U)$ un graphe orienté, calculer $G^t = (X, U^t)$ le graphe de la plus petite relation transitive contenant U.

Propriété 11 *$xy \in U^t$ ss'il existe un chemin de x à y dans G*

Un premier algorithme en $O(n^3)$.

pour $i = 1$ *à* n **faire**
  pour $j = 1$ *à* n **faire**
    pour $k = 1$ *à* n **faire**
      si $A[j, i] = 1$ *et* $A[i, k] = 1$ **alors**
        $A[j, k] \leftarrow 1$

Cet algorithme est connu sous le nom d'algorithme de Roy-Warshall.
L'invariant :
A la fin de la boucle en i,
Pour tout $i' \leq i$, le graphe G_i n'admet pas de triplet du type $ji'eti'k \in G_i$ mais $jk \notin G_i$.

9.1 Calcul via des produits de matrices

Soit A la matrice d'incidence de G.

- $A^2[i,j] = 1$ ss'il existe un chemin de longueur exactement 2 entre i et j dans le graphe.
- $(A + A^2)[i,j] = 1$ ss'il existe un chemin de longueur ≤ 2 entre i et j dans le graphe.
- Il suffit donc de calculer si l'on considère la fermeture réflexo-transitive (on ajoute toutes les boucles)
  $I + A + \ldots + A^{n-1} = (I + A)^{n-1}$
- Calcul de X^n
- Le meilleur algorithme de produit de Matrices
- $O(n^\alpha log n)$

Produit de matrices

- Considérons deux matrices carées (n,n), A, B. Rappelons que le produit $C = A.B$ vérifie :
- $C[i,j] = \bigoplus_{k=1}^{k=n} A[i,k] \bigotimes B[k,j]$
- Les opérateurs $\bigoplus$ et $\bigotimes$ doivent être associatifs et distributifs.

9.2 Algèbres max, plus

Dans les applications ces opérateurs $\bigoplus$ et $\bigotimes$, vont être surchargés comme suit :

- Existence d'un chemin, fermeture transitive
  Opérateurs logiques : $\bigoplus = \vee, \bigotimes = \wedge$
- Calcul du nombre de chemins :
  Opérateurs : $\bigoplus = +$ **et** $\bigotimes = x$
- Calcul des plus courts chemins
  Opérateurs : $\bigoplus = min$ **et** $\bigotimes = +$
  (avec la convention $A[i,j] = a_{ij}$ la valuation de l'arc ij s'il existe et ∞ si ij n'est pas un arc de G)

Chapitre 10

Algorithme d'Euclide 3 siècles avant J.C.

10.1

Commençons par la version récursive classique.

PGCD par soustractions successives
$PGCD(P,Q)$ Données : P et Q deux entiers
si $P = Q$ **alors**
  STOP répondre P
si $P < Q$ **alors**
  Echanger P et Q
$PGCD = PGCD(P - Q, Q)$

PGCD par divisions successives
Données : deux entiers P et Q
X, Y, Z, T des variables entières
$X \leftarrow P; Y \leftarrow Q; Z \leftarrow 1$
si $X < Y$ **alors**
  Echanger X et Y
tant que $Z \neq 0$ **faire**
  $T \leftarrow XdivY$; //Division entière de X par Y
  $Z \leftarrow X - T \times Y$
  $X \leftarrow Y$
  $Y \leftarrow Z$

Euclide étendu
Données : deux entiers P et Q
X, Y, Z, T, C, C', D, D' des variables entières
$X \leftarrow P; Y \leftarrow Q; Z \leftarrow 1$
$C \leftarrow 0; C' \leftarrow 1$
$D \leftarrow 1; D' \leftarrow 0$
si $X < Y$ **alors**
  Echanger X et Y
tant que $Z \neq 0$ **faire**
  $T \leftarrow X div Y$; // division entière de X par Y
  $Z \leftarrow X - T \times Y$
  $U \leftarrow C - C' \times T$
  $V \leftarrow D - D' \times T$
  $C' \leftarrow C$; $D' \leftarrow D$
  $C \leftarrow U$; $D \leftarrow V$
  $X \leftarrow Y$
  $Y \leftarrow Z$

Cet algorithme permet le calcul de l'inverse modulo.

Chapitre 11

Cryptographie à clef publique

Le but de la science du chiffre, c'est d'assurer la confidentialité des communications en présence d'adversaires. La cryptologie se divise en cryptographie (à clé secrète depuis la nuit des temps et depuis 1976 à clef publique) et cryptanalyse.

La notion de clé publique a été inventée par Diffie et Hellman en 76 [15]. Le mécanisme du chiffrement ou du codage se fait à l'aide d'une clef disponible dans le domaine public et associée au receveur du message. Ce dernier décodera le message à l'aide d'une clé privée.

Le principe du codage à clef publique est parfaitement adapté au monde moderne actuel dans lequel à chaque instant tout individu est rélié à au moins un réseau. Dans une base de données accessible par tous (le domaine public) chaque individu (ou chaque machine) peut déposer les informations nécessaires à tous ceux qui veulent communiquer avec lui (ou elle) de manière cryptée. On peut imaginer que cette base de données soit indexée par les numéros IP. Ainsi le receveur de message n'a pas à savoir l'identité de ceux qui vont communiquer de manière cryptée avec lui, c'est la grosse différence avec le cryptographie à clef secrète dans laquelle l'émetteur et le recepteur doivent s'échanger des clés (et donc se connaître) en préalable à toute communication.

11.1 RSA

Etudions en détail le codage à clef publique RSA, inventé par Rivest, Shamir et Adelmann en 1978.

Supposons qu'un émetteur A veuille envoyer un message crypté à un re-

cepteur B. Il y a cependant un travail de B, préalable à toute communication cryptée.

11.1.1 Le travail préliminaire de B

1. B choisit deux nombres premiers distincts assez grands p et q (de l'ordre de 10^{100}).
2. B fixe $n = pq$ et il publie n (B dépose n dans le domaine public).
3. B calcule $\phi(n) = (p-1)(q-1)$
4. B choisit e tel que $pgcd(e, \phi(n)) = 1$. Autrement dit e et $\phi(n)$ sont premiers entre eux. B publie e. e comme "encipher" ou encodeur.
5. B calcule d tel que $ed = 1$ *modulo* $\phi(n)$. Autrement dit d est l'inverse modulo $\phi(n)$ de e. d est la clé privée, d comme "decipher" ou décoder.

11.1.2 L'émission d'un message crypté par A

A commence par lire les données associées à B dans le domaine public (une base de données accessible sur le réseau), il trouve ainsi n et e.

Le message que A veut envoyer à B, est un nombre M écrit en binaire. On supposera $M < n$, dans le cas contraire on segmente M en paquets de taille inférieure à n.

A calcule $C = M^e$ modulo n qu'il envoie par le réseau à B.

B lorsqu'il reçoit le message C, il calcule : $D = C^d$ modulo n.

Ainsi les opérations de codages et décodages sont très simples, il suffit de faire de élévations à une puissance pour lesquelles il existe des algorithmes très performants (vus en TD). La seule chose qui manque pour mettre en oeuvre RSA, est donc le calcul d'un inverse modulo, mais ceci à été vu en TD et peut se faire en même temps que le calcul du pgcd.

11.1.3 Justifications

$D = C^d = M^{de}$ modulo n. Or d et e étant deux éléments inverses modulo $\phi(n)$, nous avons $de = 1 + k\phi(n)$. Donc $D = M.M^{k\phi(n)}$.

D'après le théorème d'Euler : $M^{\phi(n)} = 1$ modulo n lorsque $pgcd(M, n) = 1$.

Ainsi lorsque M et n sont premiers entre eux :

$D = M$ modulo n, B a bien récupéré en clair le message envoyé par A.

Les cas où M et n ne sont pas premiers entre eux, ce qui veut dire que M est un multiple de p ou q, sont assez rares et dans ce cas le codage RSA ne marche pas nécessairement.

11.2 Quelques notions utiles d'arithmétique

La fonction d'Euler notée $\phi(n)$ représente le nombre d'entiers m, $0 \leq M \leq n$ tels que $pgcd(m, n) = 1$.

$\phi(1) = 1$, $\phi(5) = 4$, $\phi(6) = 2$. Lorsque $n > 1$ est premier $\phi(n) = n - 1$.

Si n est le produit de nombre premiers p et q, alors $\phi(n) = (p-1)(q-1)$.

Le "petit" théorème de Fermat affirme :

si n est premier alors pour tout entier $0 \leq b < n$, $b^{n-1} = 1$ modulo n.

Avec les notations précédentes on peut l'écrire :

Pour tout entier $0 \leq b < n$, $b^{\phi(n)} = 1$ modulo n. Et c'est donc un cas particulier du théorème d'Euler : $b^{\phi(n)} = 1$ modulo n, lorsque $pgcd(b, n) = 1$.

On peut comprendre ce théorème d'Euler, en remarquant que dans l'anneau Z/nZ les entiers premiers avec n forment un sous-groupe multiplicatif (tout entier premier avec n admet un inverse modulo n). La suite du raisonnement se fait en remarquant que l'ordre de tout élément d'un groupe divise l'ordre du groupe.

11.2.1 Cryptanalyse de RSA

Si un intrus intercepte le message C sur le réseau, pour le décoder il lui faudrait connaître d, l'inverse modulo $\phi(n)$ de e. Donc il faudrait connaître $\phi(n)$, c'est à dire p et q, autrement dit les facteurs premiers de n (remarquons que seuls n et e qui sont dans le domaine public sont connus par l'intrus).

Cependant il n'existe pas d'algorithme efficace (**publié à ce jour**) qui calcule les facteurs premiers d'un entier. Dans la littérature ils sont référencés sous le nom d'algorithmes de factorisation d'entiers. Les algorithmes publiés sont tous de complexité exponentielle en n et l'on recommande donc de choisir n grand afin de d'être sûr de ne pas être décrypté même si l'intrus possède un très gros ordinateur.

Chapitre 12

Une fonction qui croit énormément

12.1 Fonction d'Ackermann

La fonction d'Ackermann doublement récursive, se définit classiquement :

```
Ackermann(m, n)=
if m=0 then
    n+1
else
    if n=0 then
        A(m-1, 1)
    else
        A(m-1, A(m, n-1))
```

Cette fonction croit extraordinairement vite et on peut lui associer une espèce d'inverse :

$\alpha(m,n) = min_{i\geq 1}\{A(i, \lfloor m/n \rfloor) > logn\}$.

Fonction qui va croitre très lentement.

Variante : certains auteurs utilisent une fonction d'une variable

$\alpha(n) = min_{i\geq 1}\{A(i,i) > logn\}$.

Considérons les fonctions logarithmique itérées :

$log^{(i)}n = log\ log\ log\ \ldots\ logn$ où le logarithme est itéré i fois.

Ou encore :

$log^*n = min\{y \in N | 2^{2^{2^{\cdots}}} \geq n$,"une tour d'exponentielle de hauteur y"$\}$.

Autrement dit log^*n donne le nombre d'itérations successives de la fonction logarithme nécessaires pour obtenir à partir de n une valeur inférieure à 1.

$log^*n = i \Longleftrightarrow log^{i-1}n > 1$ et $log^i n \leq 1$.

Et cette fonction vérifie : $\alpha(n) \leq log^*n$, pour $\forall n \in N$.

Chapitre 13

Recueil d'examens

Les algorithmes proposés doivent être justifiés et analysés et bien sûr ce qui nous intéresse à chaque fois, c'est un algorithme correct ayant la meilleure complexité possible.

13.1 Janvier 2000

13.1.1 Première partie : complexité

1. $f(n) = A^n$ si n est pair et A^{2n+1} sinon.
   Évaluer à l'aide des notations O, Ω et Θ la complexité du meilleur algorithme calculant f, sachant que A est une matrice carrée booléenne (matrice dont les éléments sont des 0 ou 1) ayant m lignes et m colonnes.
2. On considère le programme récursif suivant :

   f(n) :
   si $n = 1$ **alors**
   | Afficher 1
   sinon
   | $C \leftarrow f(2)$
   | $f(n-1) + C$

   Evaluer sa complexité.
3. Considérons les équations de récurrence :

$$\begin{cases} T(1) = 1 \\ T(n) = 3T(n/4) + 6n \end{cases}$$

Evaluer T(n) de deux manières différentes.

13.1.2 Deuxième partie : algorithmes

1. On considère maintenant un graphe orienté connexe $G = (X, U)$ et une valuation des arcs $\omega : U \to R$.

   On veut calculer une arborescence des plus courts chemins issue d'un point $x_1 \in X$, proposer un algorithme (justifier votre choix) dans les cas suivants :

   a) Des graphes clairsemés ou peu denses, ayant peu d'arêtes.

   b) Des graphes pour lesquels la valuation est à valeur dans $[1, |X|]$.

   c) Des graphes pour lesquels la valuation est à valeur dans $[1, |U|^2]$

   À chaque fois on précisera et justifiera l'algorithme choisi et la structure de données préconisée pour atteindre la meilleure complexité.

2. Donner une preuve ou un contre-exemple aux énoncés suivants

   a) Si toutes les valuations des arêtes sont différentes l'arbre de poids minimum est unique.

   b) Si toutes les valuations des arcs sont différentes l'arborescence des plus courts chemins issus de x_1 est unique.

   c) S'il existe un unique arc de valuation minimale dans le graphe, il appartient nécessairement à toute arborescence des plus courts chemins issus de x_1.

   d) S'il existe une unique arête de valuation minimale dans le graphe, elle appartient nécessairement à toute arbre recouvrant de poids minimum.

3. On considère un graphe non orienté connexe $G = (X, E)$ et une fonction de poids $\omega : E \to R_+$.

   On notera $d_G(x, y)$ la longueur en nombre d'arêtes d'un plus court chemin de G (lorsqu'il existe) entre x et y. Par convention lorsqu'il n'existe pas de plus court chemin on posera $d_G(x, y) = \infty$. On appelle *largeur* du graphe la valeur :

   $Larg(G) = Max_{x,y \in X}(d_G(x, y))$

   Proposer un algorithme de calcul de Larg(G), évaluer sa complexité.

4. * Écrire un algorithme linéaire de calcul de Larg(G) lorsque G est un arbre.
5. Montrer sur un exemple que des arbres recouvrants de poids minimum peuvent avoir des largeurs différentes.
6. ** Proposer un algorithme qui calcule un arbre de poids minimum et de largeur minimum.

13.1.3 Question de cours

Comment adapter un des algorithmes de recherche de motif dans un texte vus en cours, afin de tenir compte d'un caractère " ?" dans le motif. Ce caractère signifie n'importe quelle lettre de l'alphabet.

13.2 Test

Ceci est un test que vous devez faire sans l'aide de documents autres que notes de cours et TD, que vous pouvez nous rendre et que nous corrigerons

Complexité

1. Donner la définition des mots suivants : algorithme, calcul, programme, complexité.
2. Ecrire et **analyser la complexité** d'un algorithme qui calcule le produit de deux nombres décimaux exprimés comme suit :
   $A = 0.a_1 \dots a_n$ et $B = 0.b_1 \dots b_n$
   Où $\forall i$ a_i et b_j sont des nombres compris entre 0 et 9.
   Même question pour un algorithme qui calcule la racine carrée. (Précisez dans les deux cas la précision du calcul et la valeur du résultat).
3. On considère le sous-programme suivant :
   I entier ;
   tant que I $\geq$ 0
   faire I = I*I
   fintantque
   Que se passe-t-il si l'on démarre cette séquence d'instructions avec :
   I=0, I=1, I=2, I=25 ?
   On supposera que les entiers sont codés sur 64 bits.

4. On considère les instructions suivantes :
   a entier
   a=3 ; j=0
   tant que a $<$ n
   faire $a = a * 3$ et $j = j + 1$
   fintantque
   Que valent a et j en fonction de n à la fin d'un tel calcul ? Complexité du calcul.
5. Si dans votre stage à venir, il vous fallait programmer un algorithme de tri, quels critères de sélection allez vous utiliser ?
6. Proposer un algorithme de tri répondant aux critères de sélection de la question précédente et analyser sa complexité.
7. Que veut dire la phrase : il existe des algorithmes linéaires pour le tri ?
8. Lorsqu'on doit trier des mots de longueur k, nous avons vu en cours un algorithme très efficace, rappelez son principe et sa complexité. Est-il intéressant d'utiliser cette idée lorsque l'alphabet est $\{0, 1\}$. Quelle est la complexité de l'algorithme ainsi obtenu ?

13.3 Janvier 2001

13.3.1 Complexité

1. $$\begin{cases} T(1) = c \\ T(n) = 3T(\lceil \sqrt{n}/4 \rceil) + c \end{cases}$$
   Evaluer la complexité de $T(n)$ solution de cette équation de récurrence.
2. Proposer deux implémentations différentes de l'algorithme suivant et préciser leur complexité :
   Soit X un ensemble d'entiers tous différents
   $Y \leftarrow \emptyset$, $i \leftarrow 0$,
   tant que $X \neq \emptyset$ **faire**
   - $e \leftarrow Pluspetitelement(X)$
   - Retrait(X, e)
   - $i \leftarrow i + 1$,
   - $e \leftarrow e + i$,
   - Ajout(Y, e)

13.3.2 Arbres recouvrants optimaux

Ce problème est centré sur l'arbre recouvrant optimal et les algorithmes associés, on considère donc dans toute cette partie, un graphe non orienté connexe $G = (X, E)$ et une fonction de poids $\omega : E \to R_+$. Et l'on cherche donc un arbre $T = (X, F)$ dont la somme des poids des arêtes est optimale.

1. Comment modifier les algorithmes de Kruskal et Prim de recherche d'un arbre recouvrant de poids minimum, si le graphe possède des arêtes multiples (i.e. entre deux sommets il peut exister plusieurs arêtes de poids différents).

2. Même question si les poids des arêtes sont des entiers relatifs.

3. On considère maintenant cet algorithme, extrait du polycopié (hélas sans preuve, on ne peut donc avoir confiance en personne !)

   Kruskal-dans-le-désordre
   Construire une liste L avec les arêtes de G
   $F \leftarrow \emptyset$
   tant que $L \neq \emptyset$ **faire**
   - $e \leftarrow Premier(L)$
   - Retrait(L, e)
   - **si** $T = (X, F \cup \{e\})$ *n'a pas de cycle* **alors**
     - $F \leftarrow F \cup \{e\}$
   - **else**
     - **si** $\omega(e) < \omega(f)$, *où* f *est une arête de poids maximum du cycle de* $F \cup \{e\}$ **alors**
       - $F \leftarrow (F \cup \{e\}) - \{f\}$

   Ecrire une preuve de cet algorithme **Kruskal-dans-le-désordre**.

4. Proposer une implémentation en précisant les structures de données utilisées et la complexité temporelle ainsi obtenue.

5. Pour quel type de données votre algorithme est-il intéressant ?

6. On s'intéresse maintenant au problème de l'arbre recouvrant de poids maximal. Peut-on adapter les algorithmes de Kruskal et Prim ?

13.3.3 Applications

13.3.3.1 Réseaux

On considère un réseau $R = (X, E)$, par exemple l'internet, sorte de graphe non orienté dans lequel chaque arête e est munie d'une mesure de fiabilité $f(e) > 0$. On supposera R connexe. On définit la fiabilité d'un chemin comme le minimum des fiabilités des arêtes du chemin.

1. Montrer qu'il existe un arbre recouvrant T de R qui vérifie la propriété suivante : Pour $\forall x, y \in X$ l'unique chemin de x à y dans T est un chemin de fiabilité maximale dans R. On peut commencer par étudier sur un exemple.
2. En déduire un algorithme du calcul de cet arbre.

13.3.4 Spécifications

Spécifier une commande de recherche d'un motif dans un texte, pour les deux cas suivants :

1. Un éditeur de texte simple,
2. Un moteur de recherche sur le WEB,

Quels algorithmes proposeriez vous dans ces deux cas ?

13.4 Janvier 2002

13.4.1 Première partie : complexité

1. Evaluer l'ordre de grandeur de la fonction suiviante :

$$\begin{cases} f(n) = \frac{2^n}{3}, \text{ si n est un carré parfait ;} \\ \quad = \frac{n^2}{2}, \text{ si n est pair mais pas un carré parfait;} \\ \quad = 4n, \text{ si n est impair mais pas un carré parfait.} \end{cases}$$

2. Considérons les équations de récurrence :

$$\begin{cases} T(1) = 1 \\ n \geq 2, T(n) = T(n/2) + H(n) \\ \\ H(1) = 1 \\ n \geq 2, H(n) = H(n-1) + n \end{cases}$$

Evaluer l'ordre de grandeur de T(n) de deux manières différentes.

3.
Un petit programme bizarre :
$a \leftarrow n\,;\ j \leftarrow 0\,;$
tant que $a \neq 0$ **faire**
  si a *est impair* **alors**
    $a \leftarrow a - 1\,;$
  $a \leftarrow \frac{a}{2}\,;\ j \leftarrow j + 2\,;$

Exprimer j puis a en fonction de n et prouver que l'algorithme calcule bien votre expression. Déterminer la complexité de l'algorithme.

4. * Que peut-on dire de l'évaluation de $T(n)$ qui vérifie les équations de récurrence suivantes ?

$$\begin{cases} T(1) = 1 \\ n \geq 2, T(n) = 3T(n/4) + 2T(n/5) \end{cases}$$

13.4.2 Deuxième partie : algorithmes

Une petite question sur les tableaux pour commencer. On considère une matrice carrée $A[n, n]$, dont les éléments sont entiers et rangés en ordre croissant en ligne (de gauche à droite) et en colonne (de bas en haut) .

Ecrire un algorithme qui recherche si un élément x est dans la matrice. Complexité de l'algorithme ?

Arbres de poids minimum

On considère mainteant un graphe non orienté connexe $G = (X, E)$ et une fonction de poids $\omega : E \to R_+$.

1. Ecrire un algorithme qui calcule s'il existe un arbre recouvrant de poids minimum de G dans lequel deux sommets fixés a et b sont pendants dans l'arbre. (i.e. $d_T(a) = d_T(b) = 1$).
2. Dans le cas ou il n'existe pas un tel arbre de poids minimum, écrire un algorithme qui calcule le meilleur arbre recouvrant (s'il en existe un) vérifiant cette contrainte sur a et b.
3. Mêmes questions lorsqu'on remplace les sommets a et b, par un ensemble de sommets $S \subseteq X$.
4. Rappelez la condition pour qu'il existe plusieurs arbres recouvrants de poids minimum. Dans un tel cas écrire un algorithme qui calcule parmi

ces arbres recouvrants de poids minimum, celui dont l'arête de plus fort poids est minimale.

5. On suppose maintenant que les poids des arêtes de G sont des entiers compris entre 1 et $|X| = n$. Evaluer la complexité d'une bonne implémentation de l'algorithme de Kruskal dans ce cas particulier.

13.4.3 Question de cours

Ecrire et démontrer un algorithme de recherche d'un arbre recouvrant de poids minimum basé sur le principe suivant. Choisir un cycle, enlever l'arête de plus fort poids de ce cycle, ... jusqu'à plus soif.

Evaluer la complexité de l'implémentation de votre algorithme.

13.5 Corrigé janvier 2002

13.5.1 Complexité

1. Il faut distinguer la complexité du calcul de la fonction f, de celle de la fonction elle-même.

   La complexité de f est en $O(2^n)$ et en $\Omega(n)$.

   Quand à la complexité du calcul de f, il faudrait d'abord écrire un programme qui calcule f et ensuite l'analyser, mais ce n'était pas la question.

2. Les équations de récurrence :

   D'abord résoudre directement les équations de récurrence de H, et l'on obtient

   $H(n) = 1/2.n(n+1)$

   Puis pour T, on peut soit utiliser le théorème général du polycopié, soit un changement de variables, soit une preuve par induction.

3. Analyse du petit programme :

   On remarque qu'à chaque passage dans la boucle a est divisé par deux et qu'on ajoute 2 à j. La valeur finale de j est donc $2logn$.

   La complexité de ce programme est donc en $O(logn)$.

4. En remarquant que $3/4 + 2/5 > 1$, la solution de ces équations ne peut être linéaire.

13.5.2 Algorithmes

La recherche d'un élément x dans une matrice A ordonnée en lignes et en colonnes, peut se faire de deux façons différentes en utilisant l'ordre partiel des éléments de A.

La première consiste à faire une espèce de dichotomie, on prend un élément au milieu et l'on otient l'équation de récurrence :

$T(n) = T(n/4) + T(n/2) + a$

ce qui donne $T(n) \in 0(n)$.

La deuxième méthode fait une espèce de parcours diagonal de la matrice A et l'algorithme obtenu est aussi linéaire.

Arbre de poids minimum

1. S'il existe un tel arbre il utilise en a et b des arêtes ax et by de poids minimum sur les cocycles de a et de b.

   On calcule un arbre recouvrant T de poids minimum de G, et l'on compare au poids d'un arbre recouvrant T' sur $G - a - b$. Il suffit de comparer $\omega(T)$ et $\omega(T') + \omega(ax) + \omega(by)$.

2. L'arbre T' calculé à la question précédente auquel on adjoint les arêtes ax et by répond à la question.

3. Généralisation immédiate des questions précédentes.

4. Pour qu'il existe plusieurs arbres de poids minimum, il est nécessaire que plusieurs arêtes aient le même poids. (Condition non suffisante).

   La fin de la question était un piège, car pour tous les arbres recouvrants de poids minimum d'un graphe, le poids de l'arête de poids maximal est identique. Donc l'algorithme demandé était trivial, il suffit d'en calculer un.

5. L'hypothèse sur les poids des arêtes permet d'utiliser un algorithme de tri linéaire et donc d'avoir une implémentation très intéressante de l'algorithme de Kruskal.

13.6 Janvier 2003

13.6.1 Compréhension et Analyse d'algorithmes

(a) Résoudre à l'aide de trois méthodes différentes, les équations de récurrence suivantes :

$T(1) = 1$ et pour $n \geq 2$, $T(n) = 3T(n/4) + 12n^{100}$

(b) Un archéologue a trouvé dans les restes d'un camp romain une pierre datée de 100 avant J.C. qui contenait une inscription décrivant une méthode de calcul. Traduit dans notre langage algorithmique usuel, on obtient le calcul ci-dessus.

Calcul romain
Donnée deux entiers P et Q
X, Y, Z, T des variables entières
$X \leftarrow P; Y \leftarrow Q; Z \leftarrow 1$
si $X < Y$ **alors**
  Echanger X et Y
tant que $Z \neq 0$ **faire**
  $T \leftarrow X div Y$; //Division entière de X par Y
  $Z \leftarrow X - T \times Y$
  $X \leftarrow Y$
  $Y \leftarrow Z$

(c) Que vaut X à la fin du calcul précédent, en fonction de P et Q ? Démontrer votre affirmation, on démontrera que le calcul s'arrête, puis l'invariant.

(d) Complexité de l'algorithme.

(e) La pierre est-elle un faux, ou est-il possible que les romains aient eu une telle connaissance ?

(f) *Question plus difficile, à traiter en dernier*
En fait la pierre était un peu usée, il nous a semblé deviner le calcul suivant :

Calcul romain
Donnée deux entiers P et Q
X, Y, Z, T, C, C', D, D' des variables entières
$X \leftarrow P; Y \leftarrow Q; Z \leftarrow 1$
$C \leftarrow 0; C' \leftarrow 1$
$D \leftarrow 1; D' \leftarrow 0$
si $X < Y$ **alors**
  Echanger X et Y
tant que $Z \neq 0$ **faire**
  $T \leftarrow X div Y$; // division entière de X par Y
  $Z \leftarrow X - T \times Y$
  $U \leftarrow C - C' \times T$
  $V \leftarrow D - D' \times T$
  $C' \leftarrow C$; $D' \leftarrow D$
  $C \leftarrow U$; $D \leftarrow V$
  $X \leftarrow Y$
  $Y \leftarrow Z$

Que valent C et D à la fin du calcul ? Si votre interprétation est bonne, faut-il reprendre la réponse à la question 4 ?

(g) Résoudre à l'aide de trois méthodes différentes, les équations de récurrence suivantes :

$T(1) = 1$ et pour $n \geq 2$ $T(n) = 3T(n/4) + 12n$

13.6.2 Algorithmes

(a) On considère un tableau $TAB[1..n]$ d'entiers.

Un tel tableau est en fait constitué de k sous-séquences monotones (croissantes ou décroissantes) maximales. Exemple : le tableau suivant [1, 3, 5, 2, 4, 9, 7, 6, 20, 22] est constitué des sous-séquences monotones [1, 3, 5] (croissante) puis [2, 4, 9] (croissante) puis [7, 6] (décroissante) puis [20, 22] (croissante).

Ecrire un algorithme qui calcule ces séquences à partir du tableau.

(b) En déduire un algorithme de tri. Evaluer sa complexité en fonction du nombre k de sous-séquences monotones initiales.

(c) On considère un graphe simple (sans arêtes multiples) sans boucles, non orienté et connexe $G = (X, E)$ et une fonction de poids (ou valuation) sur les arêtes $\omega : E \rightarrow R_+$.

En fait les valuations arêtes sont des probabilités de panne. On suppose les pannes des arêtes mutuellement indépendantes . On cherche l'arbre recouvrant ayant la probabilité de panne minimale. La probabilité de panne d'un arbre étant le produit des probabilités des arêtes qui le constituent. Peut-on utliser l'algorithme de Kruskal ? Donnez soit un contre-exemple, soit une preuve.

13.6.3 Question de cours

Est-il possible de concevoir un algorithme de construction d'un arbre de poids minimum, qui alterne une application de la règle bleue, avec une application de la règle rouge, jusqu'à obtenir un arbre de poids minimal ? Dans le cas positif, évaluer la complexité de votre algorithme ?

Question de cours

Il s'agit de l'algorithme Kruskal à l'envers (vu en cours).

13.7 Janvier 2004

13.7.1 Algorithmes et Complexité

1. On considère les équations de récurrence suivantes :
   $T(1) = 1$
   et pour $n \geq 2$, $T(n) = T(\lfloor n/6 \rfloor) + T(\lfloor 7n/12 \rfloor) + 3n$
   Montrez que l'on peut les résoudre à l'aide d'un résultat énoncé dans le polycopié. Précisez l'application de ce résultat (le numéro du résultat et l'instanciation précise des paramètres).
   Vérifier le résultat proposé par le polycopié, soit en fournissant une preuve par récurrence, soit en analysant directement l'arbre des appels récursifs.
   Mêmes questions pour les équations :
   $T(1) = 1$
   et pour $n \geq 2$, $T(n) = 7T(\lfloor n/3 \rfloor) + n^2$
2. On considère la fonction $f : N \longrightarrow N$
   si n est pair $f(n) = 2n^{16} - 3n^5 + 2$
   et si n est impair $f(n) = 2n^{16} + 3n^5 + 2000$
   Evaluez cette fonction en utilisant les notations O, Ω, Θ.
3. On considère la fonction g s'appliquant sur des matrices carrées entières (k, k), définie comme suit :
   $g(A) = 2A^{16} + 3A^5 + 2000J$
   Où J est la matrice carrée ayant tous ses éléments égaux à 1. Proposer un algorithme qui calcule $g(A)$, évaluer sa complexité en fonction de k.
   Piste : il est possible d'utiliser des sous-programmes Produit(A,B) (resp. Somme(A, B)) qui calculent le produit (resp. la somme) de deux matrices carrées de taille (k, k) et dont la complexité algorithmique est $p(k)$ (resp. $s(k)$).

13.7.2 Structures de données

Dans cette partie, une donnée sera identifiée à sa clé que l'on peut considérer comme un entier tenant sur un mot mémoire de la machine.

1. On doit gérer un ensemble de clés F soumis aux contraintes suivantes. $n \leq |F| \leq 2n$. Et l'on doit pour voir réaliser les opérations suivantes :

Rechercher un élément dans F, ajouter un élément à F, enlever un élément à F.

Proposer une structure de données pour l'implémentation de F et analyser la complexité des algorithmes qui réalisent les trois opérations précédentes sur F.

2. Rappelons la structure de données des B-arbres utilisée dans les bases de données.

   Il s'agit de la généralisation des arbres binaires de recherche. Un B-arbre d'ordre $n \geq 2$, est un ensemble arborescent de pages et vérifie :
   - Chaque page contient au plus 2n clés ordonnées.
   - Chaque page à l'exception de la page racine, contient au moins n clés.
   - Toute page non feuille contenant m éléments $K_1 \leq K_2 \ldots \leq K_m$ admet $m+1$ pages descendantes $P_0, P_1, \ldots P_m$. Et pour toute clé $h_i \in P_i$ pour $i \in [0, m]$, nous avons $h_0 \leq K_1 \leq h_1 \ldots \leq K_m \leq h_m$.
   - Toutes les pages non feuilles sont à la même distance de la racine.

   Que se passe-t-il si l'on applique à la gestion des pages d'un B-arbre, la structure de données que vous avez proposé à la question précédente pour gérer un ensemble F.

   Evaluer précisemment la complexité des algorithmes de recherche et d'insertion d'un élément dans votre nouvelle structure de données.

3. Considérons maintenant un arbre binaire de recherche rouge et noir (ARN) (nous avons étudiés de tels arbres lors du TD numéro 7). Appliquons lui la transformation suivante : Chaque sommet noir absorbe ses fils rouges dans une même structure. Que devient l'ARN ?

13.7.3 Applications du cours

Un(e) adolescent(e) veut communiquer via une messagerie électronique avec son groupe d'amis en utilisant une échelle de valeur : un peu, beaucoup, passionnément, à la folie, pas du tout.

Pour ce faire, il(elle) ne peut utiliser que l'ordinateur familial (assez mal configuré un seul compte, pas de mots de passe ...).

1. Que lui conseillez-vous ?

   N.B. Pour les besoins de cette question, évitez les réponses du type : il suffit d'acheter un ordinateur portable personnel, d'ouvrir une messagerie chez une fournisseur différent de celui utilisé par la famille ....

Quel codage lui conseillez-vous ? Décrivez précisemment le protocole de codage de votre solution.

2. Même question si l'un des parents est spécialiste de crypto. Proposeriez-vous une autre solution ?

13.8 Janvier 2005

13.8.1 Algorithmes et Complexité

1. On considère les équations de récurrence suivantes :
   $T(1) = 1$
   et pour $n \geq 2$, $T(n) = T(\lfloor n/7 \rfloor) + T(\lfloor 11n/14 \rfloor) + 100n$

   (a) Montrez que l'on peut les résoudre à l'aide d'un résultat énoncé dans le polycopié. Précisez l'application de ce résultat (le numéro du résultat et l'instanciation précise des paramètres).

   (b) Vérifier le résultat proposé par le polycopié, soit en fournissant une preuve par récurrence, soit en analysant directement l'arbre des appels récursifs.

   (c) Pour l'analyse de quel algorithme du cours avons nous utilisé, une variante de ce résultat ?

   Mêmes questions pour les équations :
   $T(1) = 1$
   et pour $n \geq 2$, $T(n) = 2T(\lfloor n/2 \rfloor) + 5nlogn$

13.8.2 Conception d'algorithmes

1. Proposez un algorithme qui étant donné un tableau Tab contenant n entiers positifs, calcule le plus grand entier apparaissant le plus grand nombre de fois dans le tableau, et NIL lorsque les entiers contenus dans Tab sont tous différents.

   Analyser la complexité de votre algorithme. Peut-on faire mieux ?

2. On considère deux graphes orientés $G_1 = (X, U_1)$ et $G_2 = (X, U_2)$ sur le même ensemble de sommets, proposer un algorithme qui calcule les composantes fortement connexes communes à G_1 et G_2. Donner un

exemple dans lequel les composantes fortement connexes communes ne sont pas triviales.

Analyser la complexité de votre algorithme. Peut-on faire mieux ?

NB Toute ressemblance avec un projet java récent ne peut-être que fortuite !

13.8.3 Algorithmes gloutons et algorithmes de graphes

1. On considère un graphe orienté $G = (X, U)$ et deux sommets s, p de G. Les arcs sont munis d'une probabilité de panne, et les pannes sont supposées indépendantes.

   Proposer un algorithme qui calcule le chemin le plus sûr de s à p. (Piste : la probabilités de panne d'un chemin est le produit des probabilit'es des arcs du chemin).

2. On considère un graphe non orienté $G = (X, E)$, les arêtes étant munies d'une valuation $\omega E :\rightarrow R$ qui mesure la fiabilité d'une arête.

   Proposer un algorithme qui calcule un arbre recouvrant de G de fiabilité maximale. Proposer une définition de la fiabilité maximale d'un arbre recouvrant.

3. Dans le village d'une tribu reculée, on pratique la polyandrie, c'est à dire qu'une femme peut avoir plusieurs époux. Par contre, chaque homme ne peut avoir qu'une femme. Chaque couple potentiel peut rapporter, si le mariage a lieu, une certaine somme de biens au village, variant selon les couples.

   Le problème est de trouver comment organiser les mariages pour qu'ils rapportent le plus possible au village.

   Proposer un algorithme glouton pour résoudre ce problème d'optimisation. A-t-on la solution optimale ?

   Que se passe-t-il si par décret on fixe à k le nombre maximal de maris que peut posséder une femme.

[]

Bibliographie

[1] J. R. Abrial. *The B-book, assigning programs to meanings.* Cambridge University Press, 1996.

[2] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. 2002.

[3] A. V. Aho, I. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Welsey, 1983.

[4] M. Aigner and G.M. Ziegler. *Proofs for the BOOK.* Springer-Verlag, 1999.

[5] J.D. Boissonnat and M. Yvinec. *Géométrie Algorithmique.* Ediscience International, 1995.

[6] P.B. Borwein. On the complexity of calculating factorials. *Journals of Algorithms*, 6 :376–380, 1985.

[7] Catalan. Notes sur une équation aux différences finies. *J. Math. Pures et Appliquées*, 1838.

[8] L. Comtet. *Analyse Combinatoire, volumes I et II.* Presses Universitaires de France, 1970.

[9] S.A. Cook and R.A. Reckhow. Time bounded random machines. *Journal of Computer Science and Systems*, pages 354–375, 1973.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press. En français chez InterEditions.

[11] J.P. D'Angelo and D.B. West. *Mathematical Thinking : problem solving and proofs.* Prentice-Hall, 1997.

[12] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms.* Mc Graw Hill, 2006.

[13] M.P. Delest and G. Viennot. Algebraic languages and polyominoes enumeration. *Theoretical Computer Science*, 1984.

[14] R. Diestel. *Graph Theory*. Springer-verlag, 1997.

[15] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. on Information Theory*, 22(6) :644–654, 1976.

[16] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, (1) :269–271, 1959.

[17] I. Dutour. *Grammaires d'objets : énumération, bijections et génération aléatoire*. PhD thesis, université Bordeaux, 1996.

[18] P.C. Gilmore and R.E. Gomory. Sequencing a one-state-variable machine : a solvable case of the traveling salesman problem. *Operation Research*, pages 655–679, 1964.

[19] R.L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1) :43–57, 1985.

[20] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete mathematics, a foundation for computer science*. Addison -Welsey.

[21] D.E. Knuth. Big omicron and big omega and big theta. *Sigact news*, April–June :18–24, 1976.

[22] C. Kozen. *The design and analysis of algorithms*. Texts and monographs in computer science. Springer-Verlag, 1991.

[23] J.B. Kruskal. On the shortest spanning tree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, pages 48–50, 1956.

[24] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP, volume I*. Benjamin Cummings Publishing Company, 1991.

[25] R. Motvani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[26] G. Pick. Geometrisches zur zahlenlehre. *Sitzungsber lotos, Prague*, 1900.

[27] R.C. Prim. Shortest connections networks and some generalizations. *Bell Syst. Tech. J.*, pages 1389–1401, 1957.

[28] R. Rivest, A. Shamir, and Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

[29] J.E. Savage. *Models of Computation : exploring the power of computing*. Addison-Welsey, 1998.

[30] D. Shasha and C. Lazere. *Out of their minds*. Copernicus, 1995.

[31] I. Stewart. *Visions géométriques*. Belin, 1994.

[32] R.E. Tarjan. *Data structures and network algorithms*, volume 44. SIAM Monography, 1983.

[33] A. Troesch. Interpretation géométrique de l'algorithme d'euclide et reconnaissance de segments. *Theoretical Computer Science*, 1993.

[34] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques ; deuxième mémoire : Recherche sur les parallèloèdres primitifs. *J. Reine Angew. Math.*, pages 198–287, 1908.

[35] D. West. *Introduction to Graph Theory*. Prentice Hall, 1996.