Eléments de complexité et de calculabilité

Notes de cours - Licence

Christian Attiogbé

Faculté des sciences et des techniques
Université de Nantes
Christian.Attiogbe@irin.univ-nantes.fr

Juillet 1997, mars2001

Table des matières

Avant-propos

La rédaction de ces notes de cours est fortement s'est faite avec le souci de pousser les étudiants à consulter les documents cités en références.
Ces notes doivent donc être prises en tant que telles c'est à dire support des quelques heures donnés en amphi et non comme document de référence.
Merci à Didier Robbes qui a lu et fait améliorer certains aspects du document initial.

Dans le cadre de ce cours, Didier ROBBES encadre les étudiants avec moi depuis 1997.
Christian RETORÉ vient de nous rejoindre (2000/2001) pour continuer cet encadrement.

Cette équipe pédagogique est à votre disposition pour votre initiation aux outils fondamentaux d'analyse des algorithmes.

Introduction

Objectifs

Ce cours s'adresse aux étudiants de la deuxième année de l'IUP-MIAGE. Dans une première partie du cours, nous souhaitons donner aux futurs ingénieurs les connaissances nécessaires pour mesurer la complexité des algorithmes en terme de coût pour les principales ressources que sont le temps d'exécution et l'occupation mémoire. Ces mesures permettront aux étudiants d'évaluer les algorithmes qu'ils écriront et aussi de pouvoir les améliorer et les comparer. La complexité leur servira également de critère pour choisir entre différents algorithmes selon leurs besoins et leurs environnements de travail.
La suite du cours est consacrée à l'étude des problèmes P et NP et une introduction à la calculabilité. Ces aspects du cours apportent aux étudiants les connaissances fondamentales pour cerner la complexité des problèmes et des algorithmes sous-jacents, les notions de décidabilité et de calculabilité.

Prérequis

Première année de l'IUP-MIAGE - Bonne connaissance de l'algorithmique

Chapitre 1

Algorithmes et complexité

1.1 Complexité des algorithmes

Essai de définition d'un algorithme

– Un algorithme est une méthode de résolution d'un problème (ou d'une classe de problèmes), ou bien
– c'est la description logique et chronologique d'une suite d'opérations permettant d'obtenir un résultat bien spécifié sous l'hypothèse de conditions initiales déterminées, ou bien
– c'est le modèle abstrait d'un programme concret.

La notion de *complexité* des algorithmes est utilisée pour préciser l'idée selon laquelle un algorithme est caractérisé par un *coût*.

Le coût d'un algorithme se *mesure* essentiellement par rapport à l'utilisation des deux ressources que sont le *temps et l'espace mémoire*.

On parle alors de *complexité temporelle* d'un algorithme pour indiquer la mesure de la durée d'exécution et de *complexité spatiale* d'un algorithme pour indiquer le volume d'espace-mémoire qu'il exige.

La complexité d'un algorithme est donc la double mesure des temps et espace-mémoire qu'il exige pour sa réalisation.

Pour résoudre un problème (donné par un cahier de charges) plusieurs algorithmes sont souvent candidats (i.e. envisageables). Les exemples classiques sont nombreux :

– problèmes de tri ; algorithmes candidats : bulles, insertion, extraction, fusion, rapide, etc
– problèmes de recherche ; algorithmes candidats : exploration séquentielle, recherche dichotomique, etc

Un choix d'algorithmes est donc nécessaire pour la résolution de problèmes. Ce choix est très souvent basé sur les coûts des algorithmes. On choisira l'algorithme le *moins coûteux* que les autres dans les conditions de réalisations égales. Il peut arriver des cas où le coût n'est pas le critère de choix, d'autres critères tes que la lisibilité, la facilité de maintenance, peuvent être utilisés.

1.2 Choix entre deux algorithmes

On veut trier les éléments d'un tableau *tab* de *n* éléments (de `1..n`). Soit l'algorithme de tri suivant :

```
Algo slowsort(tab : Tableau[1..n] )
Debut
   pour r <-- 1 a n-1
    faire
      pour j <-- r+1 a n
        faire
        si tab[j] < tab[r]
        alors
```

```
            permuter(tab, j, r)
          finsi
        finpour
    finpour
Fin
```

(Idée de l'algorithme : on garde chaque fois le plus petit élément à gauche) Quel est le coût de cet algorithme pour le tri de n nombres ?

On peut étudier plusieurs mesures de coût ; considérons l'unité de coût comme étant une comparaison de deux nombres ; combien l'algorithme fait-il de comparaisons deux à deux ?

Réponse

Nous voyons qu'il fait une comparaison pour chaque valeur de `j` soit `(r+1, r+2, r+3, ... n)` dans la boucle interne. Le nombre total de comparaisons est :

$$\sum_{r=1}^{n-1} \sum_{j=r+1}^{n} 1$$

$$= \sum_{r=1}^{n-1} (n-r)$$

$$= (n-1)n/2$$

$$= \frac{n^2-n}{2}$$

Nous avons un nombre de comparaisons qui est en terme de n^2.

Il existe un algorithme (`trirapide ou quicksort`) qui effectue au maximum cn^2 comparaisons mais en moyenne $O(nlogn)$.

Dans la pratique, remplacer $\theta(n^2)$ en une moyenne de $O(nlogn)$ est appréciable.

Exemple

Une compagnie d'assurance voulant répertorier ses 5000000 de clients ressentira avantageusement la différence entre $n^2 = 25000000000000$ comparaisons et $nlogn = 77.124.740$ comparaisons. (Imaginer le cas d'un ordinateur sur lequel la comparaison prend 10 microsecondes).

Un utilisateur choisit un algorithme parmi plusieurs concurrents en connaissant les complexités temporelle et spatiale de chacun et en fonction des ressources dont il dispose.

1.3 Complexité pratique et complexité théorique

Il est important de distinguer la *complexité pratique* qui est une *mesure précise* des temps de calcul et des tailles de mémoire pour un modèle de machine bien défini, de la *complexité théorique* qui donne des *ordres de grandeur* des coûts, de façon plus indépendante des conditions pratiques de l'exécution de l'algorithme.

Illustration

Soit à écrire un programme trouvant pour tout entier $n > 1$, son plus grand diviseur autre que lui même (le résultat est donc 1 si n est premier).
Le programme suivant (version A) fait le travail :

```
procedure plusgrandiv ( n : Entier) : Entier
variable i : Entier
Debut
        i ← n − 1
        Tanque n mod i ≠ 0 faire
                (* le + grand diviseur de n est inférieur ou égal à i *)
                i ← i − 1
        Finfaire
        plusgrandiv ← i
Fin
```

(* i est initialisé à n-1 et est décrémenté de 1 à chaque tour de boucle, on est sûr que la boucle se terminera puisque $n \bmod 1 = 0$ si auparavant, aucune valeur de i ne satisfait $n \bmod i = 0$*)

Une autre méthode possible est fondée sur la remarque que le résultat cherché est n/i, où i est le plus petit diviseur de n supérieur à 1. Si n est premier, ce "plus petit diviseur" est égal à n ; mais si n n'est pas premier, son "plus petit diviseur" est nécessairement compris entre 2 et $\sqrt{n}$.

On peut donc écrire une autre version (B) du programme :

```
procedure plusgrandiv ( n : Entier) : Entier
variable i : Entier
Debut
        i ← 2
        Tantque (i < √n) et (n mod i ≠ 0) faire
                (* le + petit diviseur de n est supérieur à 1 *)
                i ← i + 1
        Finfaire
        plusgrandiv ← (si n mod i = 0 alors n/i sinon 1)
Fin
```

Pour comparer le ″temps d'exécution″ des versions A et B, on peut remarquer qu'elles ont l'une et l'autre la forme :

```
I1
Tantque C Faire
  I2
Finfaire
I3
```

Soient $t_1,\ t_2,\ t_3$ les temps d'exécution respectifs des instructions $I1,\ I2,\ I3$. Supposons que la boucle `tantque` est représentée en machine de façon habituelle à l'aide d'un branchement conditionnel et d'un branchement inconditionnel, demandant respectivement les temps t_c et t_i. Le temps d'exécution de l'une ou l'autre des deux versions est donc :
$t = t_1 + t_3 + m(t_c + t_2 + t_i) + t_c$ où m est le nombre d'exécution de la boucle.

Le temps d'exécution a donc la forme : $t = P + mQ$ où P et Q sont des constantes (différentes pour les deux versions).

Dans la version A du programme, le nombre maximal d'exécution de la boucle est : $m_A = n - 2$.

Dans la version B, par contre, la boucle est exécutée au plus $m_B = \lceil \sqrt{n} \rceil - 2$ fois.

(**Rappel :** si x est un réel alors
$\lfloor x \rfloor$ désigne la partie entière de x i.e. l'entier n tel que $n \leq x < n+1$
$\lceil x \rceil$ désigne l'entier n' tel que $n' - 1 < x \leq n'$).

La complexité temporelle maximale de la version A est donc : $a + bn$ où a et b sont des constantes ;

La complexité temporelle maximale de la version B est $a' + b'\sqrt{n}$ où a' et b' sont d'autres constantes.

Détailler la complexité pratique des deux versions conduirait à calculer les constantes $a,\ b,\ a',\ b'$ propres à une machine donnée, et dépendant des temps d'exécutions des instructions de base.

La comparaison des complexités théoriques est beaucoup plus intéressante. Lorsque n est grand, c'est le terme proportionnel à n qui l'emporte en A, celui qui est proportionnel à $\sqrt{n}$ en B.

Pour n de l'ordre de 10^{10}, la boucle de la version A prendrait un temps prohibitif sur les machines actuelles (dans le cas où n est premier) ; celui de la version B serait exécutée au plus de l'ordre de 100000 fois, ce qui est réalisable. On dira que les complexités temporelles maximales de A et B sont respectivement : $O(n)$ et $O(\sqrt{n})$ où O signifie "de l'ordre de ".

Chapitre 2

Calcul de coûts théorique et pratique

2.1 Temps d'exécution - Taille des données

2.1.1 Temps d'exécution théorique et pratique

Considérons un algorithme comme une démarche de résolution d'un problème. Etant donné un algorithme *A* conçu pour résoudre un problème *P* donné, on peut construire un programme *Pg*1 comme étant une représentation de *A* dans le modèle de calcul induit par le langage de programmation utilisé (C, Fortran, Pascal, Cobol, ...).

*Pg*1 est exécutable sur un ordinateur en fournissant les données nécessaires à *P*. Ce n'est pas le cas de l'algorithme, qui lui n'est pas exécutable. Cette exécution va prendre du temps qui dépend des facteurs tels que :

- les données de *P* spécialement fournies pour l'exécution,
- la qualité du code objet engendré par le compilateur du langage utilisé pour le programme,
- la nature et la rapidité des instructions disponibles dans le répertoire de l'ordinateur,
- l'efficacité de l'algorithme *A*,
- la qualité du programme écrit par le programmeur.

Remarquons que dans cette suite de facteurs, il y en a beaucoup qui ont un caractère subjectif (données spécifiques, machine particulière, habileté du programmeur).
Une comparaison basée sur ces facteurs serait donc biaisée. On est donc tenté de considérer non plus le programme (temps d'exécution) mais l'algorithme *A* à partir duquel le programme est construit.

De cette manière on est à même de comparer objectivement deux algorithmes ou mieux, trouver un algorithme optimal dans la classe de ceux qui résolvent un même problème. On utilisera alors la complexité théorique qui permet alors de considérer une mesure qui rend compte de la qualité intrinsèque des algorithmes, indépendemment des détails d'implantation qui interviennent eux dans la complexité pratique.

Nous sommes amenés dans la suite à parler du temps d'exécution en termes de coûts théorique (sur les algorithmes) et pratique (sur les programmes).

2.1.2 Temps d'exécution en fonction de la taille des données

Soit un algorithme *A* ; il existe presque toujours un paramètre, soit *n*, caractérisant la taille des données auxquelles *A* s'applique dans chaque cas considéré ; si par exemple *A* est un algorithme de tri, *n* est peut être le nombre de données à trier.

Soit $T(n)$ le temps d'exécution de l'algorithme *A*, exprimé en fonction de *n*. Soit f une certaine fonction connue ; on dira que *A* est de complexité théorique $O(f(n))$ si le rapport $\frac{T(n)}{f(n)}$ est borné lorsque *n* tend vers $+\infty$ (c'est en particulier le cas si $\frac{T(n)}{f(n)}$ tend vers une constante *k*, c'est à dire, en termes mathématiques, si T est équivalent à kf).

Par exemple, un algorithme pourra être de la complexité théorique $O(n)$, $O(n^2)$, $O(2^n)$, $O(nlogn)$ (sans qu'il ne soit nécessaire de préciser la base des logarithmes), etc.

Si une opération s'effectue en un temps fixe ne dépendant pas de la taille du problème, on dira qu'elle est de complexité $O(1)$.

Généralisation

Un algorithme peut être vu comme étant composé de plusieurs sous-algorithmes. L'expression de la complexité peut alors être en fonction de la taille des données des différents sous-algorithmes. On a ainsi des complexités de la forme $O(n \times m)$ ou $(m+n)$, ... si n et m sont les paramètres (tailles des données).

Illustration

On veut savoir si un ensemble de m éléments est inclu dans un autre ensemble de n éléments.

Dans la pratique, le temps d'exécution d'un algorithme dépend non seulement de la taille de l'ensemble des données, mais aussi de leurs valeurs précises ; par exemple, le temps d'exécution de certains algorithmes de tri est considérablement décru si ces données sont partiellement triées à l'origine (alors que d'autres algorithmes de tri sont insensibles à cette propriété).

Pour tenir compte de ce fait, tout en conservant un moyen d'analyser les algorithmes indépendamment de leurs données, on utilise une approche expérimentale où on distingue :

- la complexité maximale, ou complexité du *cas le plus défavorable*, qui est la complexité de la fonction T_{max}, où $T_{max}(n)$ est le temps d'exécution de l'algorithme lorsque l'on choisit les n données entraînant l'exécution la plus longue. C'est cette considération que nous avions faite pour les deux versions du programme "plus grand diviseur".
- la complexité moyenne qui est celle de la fonction T_{moy}, temps moyen d'exécution de l'algorithme appliqué à n données quelconques. Il convient de noter que $T_{moy}(n)$ n'a de sens que si l'on peut faire des hypothèses probabilistes sur la répartition des données, ce qui est loin d'être toujours le cas.
- la complexité minimale, ou complexité du *cas le plus favorable*, c'est le cas ou on choisit les données entraînant l'exécution la plus courte.

Ces notions s'étendent sans difficulté à la mesure du coût d'un algorithme en espace mémoire : on parle de *complexité spatiale* minimale, maximale ou moyenne.

Cette approche théorique de la notion de complexité, ainsi présentée, peut présenter des aspects critiquables parce qu'elle néglige les constantes de proportionnalité pour ne retenir que les comportements asymptotiques. Par exemple les fonctions n^2, $10^{75}n^2$, $1000n^2+10^{75}n+10^{50}$, $\frac{n^2}{10^{75}}+logn$ seront toutes considérées comme étant $O(n^2)$.

De fait, une fonction de complexité $O(2^n)$ peut être inférieure à une fonction de complexité $O(n^2)$ pour toutes les valeurs pratiquement considérées.

Cependant, cette approche présente des aspects qui la justifient [AA89] :

- Soient n_0, n_1, n_2, n_3 les tailles maximales des problèmes que permettent respectivement de traiter quatre algorithmes A, B, C, D de complexité théorique $O(2^n)$, $O(n^2)$, $O(n)$, $O(logn)$.
  Supposons qu'un constructeur X propose une nouvelle machine mille fois plus rapide que la précédente. On peut traiter alors des problèmes de taille plus importante :
  $n'_0 \equiv n_0 + 10$ avec A
  $n'_1 \equiv 32n_1$ avec B
  $n'_2 \equiv 1000n_2$ avec C
  $n'_3 \equiv n_3^{1000}$ avec D
  Contrairement à ce que l'on pourrait penser, les progrès de la technologie rendent plus importante encore la recherche d'algorithmes efficaces du point de vue de la complexité théorique. Quant à la complexité pratique, les coefficients y intervenant deviennent au contraire moins importants.
- Concrètement, il est difficile de donner des résultats précis sur la complexité pratique d'un algorithme sans le lier à une représentation pour une machine particulière.

2.2 Calcul pratique de la complexité : temps d'exécution

Considérons de nouveau (indépendamment du modèle de calcul choisi) une fonction $T_A(n)$ qui exprime en fonction de la taille n le maximum d'un certain coût pour un algorithme A donné. La taille n dépend en principe du codage des données en entrée et le coût dépend des opérations effectuées dans l'algorithme.

Formellement la taille de la donnée est la longueur de la chaîne de caractères (en binaire) qui les code. Cependant, dans la pratique, le codage binaire donne une chaîne dont la longueur est proportionnelle à un paramètre plus simple, qui est souvent facilement reconnaissable et unanimement accepté. Par abus de langage, c'est ce dernier paramètre qui représentera la taille de la donnée.

Quelques exemples de tailles

En considérant des problèmes pour lesquels on écrit des algorithmes on a :

problèmes de polynômes	: le degré (ou le nombre des coefficients)
problèmes de matrices $m \times n$	: le maximum, ou la somme ou le produit de m et n,
problèmes de graphes	: nombre de sommets ou d'arrêtes, ou la somme des deux,
problèmes de tri	: nombre d'éléments à trier
problèmes d'analyse syntaxique	: longueur du mot

2.2.1 Les opérations dont dépend le coût

C'est le concept de classe d'algorithmes (pour résoudre un problème) qui met en évidence les opérations : chaque classe est caractérisée par un ensemble d'opérations dites fondamentales qui y sont utilisées.

Exemples d'opérations fondamentales

comparaison	problèmes de recherche d'un élément dans une liste
comparaison et déplacement	problème de tri d'une liste
addition et multiplication	problèmes de multiplication de matrices
...	

Remarque

Les opérations fondamentales sont différentes des opérations élémentaires qui elles, sont liées à un mode de calcul (par exemple Machine de Turing, Machine à registres).

2.2.2 Hypothèse du coût uniforme

Avec l'hypothèse de *coût uniforme*, toute opération élémentaire prend un temps constant (de l'ordre de $O(1)$) ; le temps d'une opération fondamentale est proportionnelle aux coûts des opérations élémentaires qui la composent et prend lui aussi un temps constant (de l'ordre de $O(1)$).
Dans la pratique, l'hypothèse de coût uniforme n'est acceptable que si tous les nombres rencontrés au cours de l'exécution du programme peuvent être codé sur un même nombre de bits, donc bornés.

En considérant ces modalités pour la taille, les opérations et leur coût, la complexité d'un algorithme A est proportionnelle au nombre maximum d'opérations fondamentales effectuées au cours de son exécution.

2.2.3 Principales règles de comptabilisation des opérations fondamentales

Nous ne considérons ici que le cas des programmes séquentiels. Un programme séquentiel a une structure qui met en évidence une méthode inductive de dénombrement des opérations effectuées. Cette structure dépend des structures de contrôle utilisées.

La séquence

On cumule les temps d'exécution de chacune des actions de la séquence. Soit une séquence S de n actions $a_1; a_2; ...; a_n$. Le coût (ou temps d'exécution) de la séquence S est :

$$coût_S = \sum_{i=1}^{n} T_{a_i}$$

T_{a_i} représente le coût des actions a_i.

La conditionnelle

Soit la conditionnelle :

```
si C
    alors T_1
    sinon T_2
finsi
```

Le temps d'exécution de la conditionnelle avec T_2 éventuellement vide est :

$$Coût_{Si} = coût(C) + Max\{coût(T_1), coût(T_2)\}$$

L'itération

Soit l'itération :

```
tantque  C
   faire
        T
   finfaire
```

Le temps d'exécution de la boucle se calcule de la façon suivante :

$$coût_{tantque} = m * coût(C) + m * coût(T) + coût(non\ C)$$

où m est le nombre de passage dans le corps de la boucle `tantque`.

L'appel de procédure

L'appel de procédure est décomposé, selon le modèle de machine considéré, en plusieurs opérations élémentaires. On a donc un coût uniforme qui prend un temps $O(1)$ c'est à dire majoré par une constante.

2.2.4 *Taille* des données et *forme* des données

Le coût d'un algorithme en fonction des données qu'il a en entrée ne fournit pas une appréciation exacte ; en effet, selon les données le même algorithme peut présenter des coûts proportionnels à la valeur (la forme) des données. C'est le cas par exemple lorsqu'on trie une liste d'éléments partiellement triés.

La complexité maximale (pire des cas), peut être prise en combinant la taille des données et les valeurs des données.

2.3 Temps d'exécution en fonction de la forme des données

Le temps d'exécution d'un algorithme dépend non seulement de la taille des données mais aussi de la forme des données. On va distinguer, en fonction de la forme des données le *pire des cas*, le *meilleur des cas* et *la moyenne*.

Pour certains algorithmes, le temps d'exécution ne dépend que de la taille des données (multiplication de matrices, opérations sur les nombres, ...) ; mais dans la pratique on trouve souvent que la complexité varie aussi, pour une taille fixée des données, en fonction des données elles-mêmes.

Plusieurs quantités caractérisent le comportement d'un algorithme sur l'ensemble D_n des données de taille n. Soit $coût_A(d)$ la complexité en temps de l'algorithme A sur la donnée d (calculée selon les principales règles de comptabilisation) :

– Complexité dans le **meilleur des cas**

$$Min_A(n) = min\{coût_A(d) \mid d \in D_n\}$$

– Complexité dans **le pire des cas**

$$Max_A(n) = max\{coût_A(d) \mid d \in D_n\}$$

– Complexité en **moyenne**

$$Moy_A(n) = \sum_{d \in D_n} p(d).coût_A(d)$$

où $p(d)$ est la probabilité d'avoir d en entrée de l'algorithme.
La *complexité dans le pire des cas* donne une borne supérieure du temps d'exécution ; elle est utile pour donner une estimation de la taille maximale des données qui pourront être traitées par l'algorithme.

La *complexité dans le meilleur des cas* est la borne inférieure du temps d'exécution.

Dans la pratique les cas extrémaux ne sont pas fréquents et on aimerait savoir le comportement en général d'un algorithme, d'où la nécessité de la complexité en moyenne. Si toutes les données son *équiprobables* alors la complexité en moyenne s'exprime par :

$$Moy_A(n) = \frac{1}{\mid D_n \mid}. \sum_{d \in D_n} coût_A(d)$$

Mais en général les données n'ont pas toutes les mêmes probabilités. La définition de la complexité en moyenne nécessite donc l'introduction d'un modèle probabiliste lié au problème.

Souvent on peut partitionner l'ensemble D_n en regroupant les données de taille n de même coût et on évalue la probabilité $p(D_{n,k})$ de chaque classe $D_{n,k}$ de la partition. La complexité en moyenne devient alors :

$$Moy_A(n) = \sum_{D_{n,k} \subseteq D_n} p(D_{n,k}).coût_A(D_{n,k})$$

où $coût_A(D_{n,k})$ représente le coût d'une donnée quelconque de $D_{n,k}$.

Exemple

On veut déterminer la complexité en nombre de comparaisons de l'algorithme de recherche séquentielle d'un élément dans une liste L.

Conclusion

Le coût d'un algorithme en fonction des ressources (temps et mémoire) est un critère important pour comparer des algorithmes en vue d'un choix pour un problème donné. Le coût n'est pas le seul critère utilisé dans la pratique, il ne faut pas perdre de vue les critères de simplicité pour la compréhension et la mise au point.

Le coût théorique d'un algorithme permet de mesurer un algorithme indépendemment de tout critère matériel.

Le coût pratique est lié aux caractéristiques d'une machine donnée. Ce coût ne peut être comparé qu'avec d'autres coûts effectués dans les mêmes conditions.

2.4 Méthode de recherche d'algorithmes efficaces

Un algorithme efficace est un algorithme ayant des complexités spatiale et temporelle faibles.

Dans la pratique il est difficile d'obtenir des algorithmes efficaces, on y arrive souvent au prix d'un compromis entre l'espace et le temps.

Cependant, il existe des principes généraux pour guider la recherche de tels algorithmes.

2.4.1 Diviser pour règner (Divide and conquer)

L'idée essentielle consiste à décomposer un problème de taille n soit en k problèmes de tailles m_1, m_2, m_3, ... m_k telles que $m_1 + m_2 + m_3 + \ldots + m_k \leq n$ soit en une opération d'une certaine complexité $O(n)$ ou $O(1)$ par exemple et en un problème de taille $m < k$.

C'est l'idée qui est utilisée dans la récursion : paramétrage, résolution du cas trivial, traitement du cas général (en le ramenant à un cas simple). Toutes les méthodes de tri découlent de cette idée.

2.4.2 Equilibrage

La division en sous-problèmes ne donne pas toujours de bons résultats sur le plan de la complexité théorique.

Exemple

Soit une méthode ramenant un problème de taille n à un problème de taille $n-1$, moyennant des opérations de complexité $O(n)$.

Soit $T(n)$ la complexité temporelle de l'algorithme.

On a par définition :

$T(n) \leq cn + T(n-1)$ où c est une constante.

En développant :

$$T(n) \leq cn + c(n-1) + c(n-2) + \ldots + c + T(0)$$

$$T(n) \leq c.\frac{n(n+1)}{2} + T(0)$$

donc

$T(n) \in O(n^2)$.

Soit maintenant une méthode permettant de ramener un problème de taille n à deux opérations de complexité $O(n)$.

La complexité temporelle $T'(n)$ sera :

$$T'(n) \leq cn + 2T'(\lfloor \frac{n}{2} \rfloor)$$

donc

$$T'(n) \leq cn + 2c\,|\,\frac{n}{2}\,| + 4c\,|\,\frac{n}{4}\,| + \ldots + 2^{|log_2^0 n|}T'(1)$$

c'est à dire

$$T'(n) \leq \underbrace{cn + cn + cn + \ldots +}_{|log_2^n| termes} nT'(1)$$

donc

$$T'(n) \in O(nlogn)$$

Cette deuxième méthode est donc bien meilleure que la première.

En conclusion, il ne suffit pas de décomposer (diviser) pour obtenir des algorithmes, il faut aussi équilibrer les morceaux résultant de la division.

Exemple

La recherche dichotomique est un exemple direct de ce principe.

Il peut arriver des cas où l'optimum précis est atteint pour un découpage non équilibré.

2.4.3 Les compromis espace-temps

En programmation, on est souvent obligé de choisir entre une amélioration de la complexité spatiale et une amélioration de la complexité temporelle, l'une se faisant au détriment de l'autre.

Exemple

Supposons trois grandeurs X, Y, Z reliées par une équation de la forme $X = f(X, Z)$

Si l'on a un très grand nombre (de l'ordre de millions) de triplets (X, Y, Z), on peut se poser le problème de savoir s'il faut conserver tous les triplets, ou seulement les doublets (Y, Z), les X correspondants étant recalculés par $X_i = f(Y_i, Z_i)$ au besoin.

Chapitre 3

Outils mathématiques

3.1 Estimation asymptotique - Ordre de grandeur

Dans les problèmes d'évaluation, on utilise souvent des fonctions de dénombrement. Une fonction de dénombrement peut parfois être très compliquée. Or une estimation de l'ordre de grandeur asymptotique de la fonction fournit suffisamment d'informations. On va donc se servir de l'estimation de l'ordre de grandeur plutôt que de la fonction de dénombrement précise.

Considérons uniquement les fonctions de $\mathbb{N} \rightarrow \mathbb{R}$, elles sont interprétées comme des fonctions de dénombrement.

Définition

On dit qu'une fonction $f : \mathbb{N} \rightarrow \mathbb{R}$ tend vers a lorsque n tend vers l'infini lorsque :
$\forall\, \upsilon,\ \exists\, n_0$ tel que $n \geq n_0 \Rightarrow \mid f(n) - a \mid \leq \upsilon$

Définition

On dit que f tend vers l'infini quand n tend vers l'infini lorsque :
$\forall K,\ \exists\, n_0$ tel que $n \geq n_0 \Rightarrow \mid f(n) \mid \geq K$

Les expressions suivantes sont synonymes :
$n \rightarrow \infty$,
pour n assez grand, ou
n au voisinage de l'infini
Le fait qu'une fonction f tende vers une limite finie ou infinie quand n tend vers l'infini ne constitue pas un renseignement précis (intéressant). Pour avoir plus d'informations utiles, on compare f avec des fonctions dont connait le comportement au voisinage de l'infini.

Soit g une fonction positive qui ne s'annule pas au voisinage de l'infini :

Définition

On dit que f est asymptotiquement négligeable devant g et on écrit $f = o(g)$ ou $f(n) = o(g(n))$ (on prononce "petit o de g"), lorsque le rapport $\frac{f}{g}$ tend vers 0 quand $n \rightarrow \infty$

Définition

On dit que f est asymptotiquement équivalente à g et n écrit $f \equiv g$, lorsque $\frac{f}{g}$ tend vers 1 quand $n \rightarrow \infty$.

Définition

On dit que f est asymptotiquement dominée par g et on écrit $f = O(g)$ ou $f(n) = O(g(n))$ (on prononce f égale "grand O de g") lorsqu'il existe une constante $c > 0$ telle que, pour n assez grand, on a $| f(n) | \geq c.g(n)$

Définition

On dit que f est asymptotiquement du même ordre de grandeur que g et on écrit $f = \Theta(g)$ ou $f(n) = \Theta(g(n))$ (on prononce f égale theta de g) lorsque $f = O(g)$ et $g = O(f)$.

Remarque

Il faut garder à l'esprit que, la fonction g étant donnée, $o(g)$, $O(g)$, $\Theta(g)$ représentent des classes de fonctions suivantes :
$o(g) = \{f \mid f \text{ est asymtotiquement negligeable devant } g\}$
$O(g) = \{f \mid f \text{ est asymptotiquement dominee par } G\}$
$\Theta(g) = \{f \mid f = O(g) \text{ et } g = O(f)\}$

Dans ce cadre l'écriture $f = O(g)$ signifie $f \in O(g)$.
(sinon on aurait des absurdités telles que :

$$\left.\begin{array}{rl} n^2+n = & O(n^2) \\ 2n^2 = & O(n^2) \end{array}\right\} \Rightarrow n^2+n = 2n^2$$

)

3.1.1 Echelle de comparaison

On dit qu'une fonction f est :
– *bornée ou constante* si $f = O(1)$
– *logarithmique* si $f = O(logn)$
– *linéaire* si $f = O(n)$
– *quadratique* si $f = O(n^2)$
– *exponentielle* s'il existe une constante $c > 0$ telle que $f = O(c^n)$.
Remarque : Lorsque $f = O(g)$ ou mieux encore $f = \theta(g)$ on dit que g est un ordre de grandeur de f.

3.1.2 Séries

$$\begin{array}{rl} \frac{1-x^n}{1-x} & = 1+x+x^2+\ldots+x^{n-1} \\ & = \sum_{i=0}^{n-1} x^i \end{array}$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$\sum_{m=0}^{\infty} \frac{x^m}{m!} = e^x$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

3.1.3 Divers

– Matrices
Exemple

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

Le *ijème mineur* d'une matrice A de taille $n \times n$, pour $n > 1$ est la matrice $A_{[ij]}$ de taille $(n-1) \times (n-1)$ obtenue en supprimant la *ième* ligne et la *jème* colonne de A.
Le déterminant : il est défini récursivement en fonction des mineurs de la matrice.

$$det(A) = \begin{cases} a_{11} & si\ n\ =\ 1 \\ a_{11}det(A_{[11]}) - a_{12}det(A_{[12]}) + \ldots + (-1)^n a_{[1n]}det(A_{[1n]}) & si\ n\ >\ 1 \end{cases}$$

Le terme $(-1)^{i+j}det(A_{[ij]})$ s'appelle le $cofacteur$ de l'élément a_{ij}.
– Nombre d'inversions d'une permutation
Le nombre d'inversions $Inv(\sigma)$ d'une permutation σ représentée par un tableau t est le nombre de couples d'éléments (j,i) tels que $j < i$ et $t[j] > t[i]$
– Approximation de *Stirling*

$$n! = \sqrt{2\pi n}(\frac{n^n}{e}(1 + \theta(\frac{1}{n}))$$

e est la base du logarithme naturel ($ln\ n\ =\ log_e n$).

Chapitre 4

Problèmes - Calculabilité

4.1 Motivations

Nous donnons ici quelques rudiments de la *NP-complétude* pour introduire et sensibiliser aux problèmes particuliers qu'un informaticien peut être amener à résoudre.
Ces rudiments de la *NP-complétude* sont indispensables pour la bonne conception des algorithmes.

En effet il y a des problèmes pour lesquels il n'existe pas d'algorithmes efficaces (pour les résoudre). Lorsqu'on arrive à identifier ou à dire, face à un cahier de charges, qu'on se trouve confronté à de tels cas de problèmes, il est inutile de chercher un algorithme efficace. La seule issue est de recourir à un algorithme d'approximation.

De plus, certains problèmes réels semblent n'être pas plus difficiles que la recherche dans un graphe ou le tri mais en réalité, ils sont *NP-complets* ou de la classe des *NP-complets*, le concepteur qui n'est pas familier à cette classe de problèmes court alors le risque de perte de temps considérable.

4.2 Algorithmes efficaces

Il existe une hiérarchie des fonctions liées à leur rapidité de croissance.

Une fonction exponentielle croît beaucoup plus vite que n'importe quelle fonction polynomiale. Un algorithme dont la complexité croît exponentiellement avec la taille des données d'entrée n'est pas utilisable dans la pratique.

4.2.1 Définition

Un algorithme est dit polynomial si sa complexité est $O(n^p)$ pour un certain entier p. Les algorithmes polynomiaux sont dits *efficaces* ou *rapides*.

L'*efficacité* dans ce cadre est un concept purement mathématique.

4.2.2 Exemples

L'algorithme de multiplication de matrices $(n x n)$ a un coût de n^3 pour des données de taille n. C'est une complexité polynomiale. L'algorithme est efficace.

L'algorithme d'énumération des parties d'un ensemble de n éléments a un coût de (2^n) pour des données de taille n. C'est d'une complexité exponentielle. L'algorithme n'est pas efficace.

4.3 Décidables et indécidables

Les travaux en Logique Mathématique ont mis en évidence une partition de l'ensemble des problèmes en deux classes. La classe des problèmes pour lesquels on ne trouvera jamais d'algorithmes de résolution, ils sont dits *indécidables* et la classe des problèmes pour lesquels il existe une solution algorithmique, ces problèmes sont dits *décidables*.

Lorsqu'on décide de résoudre un problème par l'algorithmique, on prend donc le risque :

– de ne trouver qu'une solution algorithmique inefficace,
– de ne pas trouver de solution algorithmique.

Dans les deux cas, la question fondamentale est de savoir si cela est dû à la nature du problème ou aux connaissances (relativement limitées) de la personne qui cherche à résoudre le problème.

Il est alors important de vouloir répondre à la question : "Existe-t-il des problèmes qui n'ont pas de solution algorithmique ?"

Pour répondre à une telle question, une formalisation ou à défaut une formulation précise de la notion de problème est nécessaire.

4.4 Problèmes et Langages

4.4.1 Problèmes

On définit un problème (abstrait) Q comme une relation binaire sur un ensemble I d'instances d'un problème et un ensemble $\mathcal{S}$ de solutions d'un problème.

$$Q : I \leftrightarrow \mathcal{S}$$

Certains aspects de la théorie des langages permettent d'illustrer simplement des propriétés des problèmes. Nous rappelons brièvement dans la suite quelques définitions (simplifiées) sur les langages.

4.4.2 Formalisation des problèmes

Si un problème doit être résolu par un programme (ou une procédure effective), il faut que les instances du problème soient représentées par une fonction accessible à ce programme.

Par exemple si c'est un programme PASCAL , alors l'instance du problème (les données sur lesquelles le programme opérera) devra être représentée par une collection d'entiers, de chaînes de caractères, ...

Donc il faut un alphabet qui permette de construire les mots qui représentent l'instance du problème. L'ensemble des mots ainsi définis forme un langage. On dit qu'un problème est caractérisé par le langage des encodages de ses *instances positives* (ou instances oui). La résolution d'un problème est alors comparable à la *reconnaissance du langage* des instances positives du problème.

Les problèmes pour lesquels il existe une solution algorithmique, donc un traitement par ordinateur, se répartissent en deux classes :

– Les *problèmes de décision*. Ils sont modélisés par la notion de langage : déterminer si un mot appartient ou non à un langage.
– Les problèmes de calcul conduisant à un résultat qui est une valeur (numérique ou non). Ils sont modélisés par une fonction de transformation des mots d'un alphabet d'entrée en des mots d'un autre alphabet de sortie.

4.4.3 Problèmes de décision

Définitions

Un problème de décision est une classe d'énoncés auxquels on doit répondre par oui ou par non.

Chaque énoncé est appelé une *instance* du problème dont la réponse appartient à l'ensemble $\{oui, non\}$.

Donner une réponse positive à un problème de décision consiste à exhiber un algorithme pour le résoudre.

Donner une réponse négative à un problème de décision consiste à prouver qu'il n'existe pas d'algorithme pour le résoudre. (Une tâche souvent difficile).

Un énoncé d'un problème de décision se formule souvent sous la forme d'une question.

Exemple

- Etant donné un entier n, existe-t-il deux entiers p et q inférieurs à n tels que $n = p * q$?
  (si oui n est dit composé).
- Soient un graphe G et un entier k, G admet-il un cycle de longueur $\geq k$?
- Soient P et P' deux programmes, P et P' sont-ils équivalents ?
- Soit un graphe G, G peut-il être colorié avec 5 couleurs (sans nœud voisins de même couleur) ?

Généralement on décrit un problème de décision sous la forme d'une description générique d'objets (de valeurs fixées ou non) et d'une question.

Exemple

COMP	Instance : un entier n
	Question : n est-il composé ?

COMP est le nom (généralement connu) du problème.

Puisque tout problème de décision peut recevoir une réponse *oui* ou *non*, on comprend aisément la modélisation qui consiste à le penser comme la question de savoir si un mot donnée (codage en entrée) appartient ou non à un certain langage, celui représentant la totalité des mots pour lesquels la réponse est *oui*.

Tous les problèmes ne sont pas des problèmes de décision, mais beaucoup de problèmes peuvent se ramener à un problème de décision ou à un problème d'optimisation.

Quel est le nombre minimum de couleurs avec lesquelles on peut colorier un graphe G ?
(appelé nombre chromatique)

De façon générale, lorsqu'on trouve une solution *rapide* pour un problème de décision, on arrive avec un peu d'effort à trouver une solution au problème d'optimisation correspondant.

Dans les problèmes de décision, on s'intéresse surtout à la *vérification*. Dire qu'une instance I d'un problème admet la réponse *oui* équivaut à affirmer que la proposition "I possède une certaine propriété L" est vraie (donc il en existe une preuve convaincante).

L'existence et la nature de la preuve ne relèvent pas forcément d'une procédure efficace. L'important c'est que la vérification de la correction de cette preuve vis à vis de la réponse *oui* puisse être entreprise efficacement sur le plan algorithmique.

Mathématiquement, cette perception s'apparente à la différenciation des deux activités suivantes :

- Prouver un théorème.

– Vérifier une preuve d'un théorème.

4.4.4 Problèmes de calcul

Nous ne développons pas cette partie dans ces notes de cours.
Le lecteur intéressé peut consulter les fonctions calculables. Les ouvrages suivants traitent du sujet : [Wol93],[RLC90].

4.4.5 Réductibilité

Lorsqu'on se trouve face à un problème ou à un cahier de charges, une façon naturelle de réagir consiste à se ramener à un problème connu et déjà résolu.
Plus généralement, soient P et P' deux problèmes de décision. On dit que P' est *rapidement réductible* en P si l'on peut convertir, en un temps polynomial, toute instance I' de P' en une instance I de P, de telle sorte que I et I' aient la même réponse *oui* ou *non*.

Supposons que nous voulions résoudre un système linéaire de 100 équations à 100 inconnues de la forme $Ax = b$ (où A est une matrice). Admettons que nous nous procurons d'un logiciel et que nous nous apercevons désespérement que le logiciel ne fonctionne que sur des systèmes pour lesquels la matrice A est symétrique alors que notre matrice ne l'est pas.

Une issue possible est de nous ramener à une matrice symétrique A^TA en cherchant la solution du système $A^TAx = A^Tb$.

Nous aurons ainsi réduit le problème pour utiliser le logiciel. Nous aurons réduit notre problème en un problème connu et résolu.

Remarque

La démarche de réduction ne change pas la nature du problème. Il faut donc avoir à l'esprit le fait que les problèmes indécidables ne peuvent pas se ramener à des problèmes décidables.

Théorème

Soient deux problèmes P et P' tels que P soit réductible en P' :

– si P est indécidable alors P' l'est aussi,
– si P' est décidable alors P l'est aussi.

4.5 Les classes P et NP

La notion d'efficacité des algorithmes permet de diviser les problèmes décidables en deux classes de complexité. La classe P des problèmes pour lesquels il existe une solution algorithmique de coût polynomial et la classe des autres.
Dans la pratique, lorsqu'on recherche pour un problème un algorithme effectivement utilisable, il faut d'abord identifier la classe de complexité du problème.

Les problèmes décidables qui appartiennent à la classe P sont dits *faciles*.

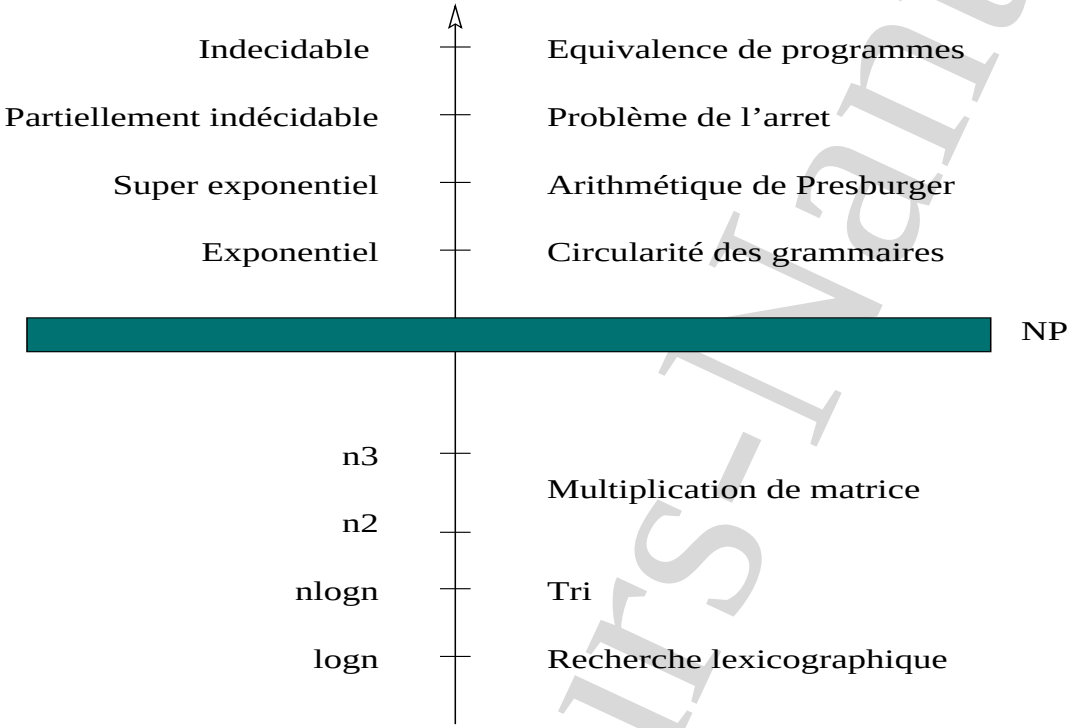
Les problèmes décidables qui n'appartiennent pas à la classe P sont dits *difficiles* ou *intraitables*.

Bien que les deux goupes de problèmes faciles et de problèmes difficiles soient distincts, il n'est pas toujours simple de rattacher un problème donné à l'un des deux groupes.

En effet, il faut exhiber un algorithme de coût polynomial ou démontrer l'inexistence d'un tel algorithme ; cette dernière démonstration n'est pas facile.

Il y a entre les problèmes faciles et les problèmes difficiles, une série de problèmes intéressants de par leurs propriétés et pour lesquels on ne connaît pas d'algorithmes efficaces. Les meilleurs algorithmes connus actuellement pour ces problèmes ont un coût (en temps d'exécution) exponentiel et personne n'a encore réussi à démontrer que ces problèmes n'appartiennent pas à la classe *P*. En attendant, on les met dans une classe dénommée *NP* (*Nondeterministic Polynomial*).

Le schéma qui suit, extrait de [Xuo92] donne un aperçu des principaux problèmes.



4.5.1 La classe *P*

Un problème de décision est dit de la classe *P* s'il existe un algorithme polynomial qui reconnait le langage des *instances oui*. Un tel problème est dit *facile*.

En d'autres termes, un problème de décision appartient à la classe *P* s'il existe un algorithme *A* et un nombre *c* tel que **pour toute instance I** du problème, l'algorithme *A* donne une solution en un temps polynomial $(O(n^c))$ où *n* est la taille de l'instance (par exemple le nombre de bits de la chaîne d'entrée qui représente l'instance *I*).

4.5.2 La classe *NP*

Un problème de décision *Q* appartient à la classe *NP* s'il existe un algorithme *A* tel que :

1. à chaque mot du langage *Q* (c'est à dire à chaque instance *I* pour laquelle la réponse est *oui*), un certificat $C(I)$ est associé, tel que lorsque la paire $(I, C(I))$ est une entrée de l'algorithme *A*, celui ci reconnaît que *I* appartient au langage *Q*,
2. si *I* est un mot qui n'appartient pas au langage *Q*, il n'existe aucun certificat tel que *A* reconnaisse *I*,
3. l'algorithme *A* s'arrête en un temps polynomial.

NP est la classe des problèmes de décision pour lesquels *il est aisé de* **vérifier** que les réponses sont correctes. On ne cherche pas un moyen de trouver une solution mais de vérifier qu'une solution donnée est correcte.

On dit aussi que les problèmes de la classe *P* ont des solutions **faciles à trouver** alors que ceux de la classe *NP* ont des solutions **faciles à vérifier** même si elles sont difficiles à trouver.

Considérons le problème du coloriage de graphes comme problème de décision Q. Il n'est pas dans P mais dans NP.

Prenons un graphe G coloriable par k couleurs, le certificat de G peut être une liste de couleurs affectées à chaque sommet de G.

Comment avons nous obtenu cette liste de couleurs ? (ceci est un problème ardu !)

En fait, nous ne nous intéressons qu'à la vérification. Pour vérifier qu'un certain graphe G est k coloriable, il suffit d'avoir la couleur de chaque sommet telle que le graphe soit correct.

Si on a un certificat, la vérification de l'algorithme est simple. Il suffit de vérifier d'abord que chaque sommet a bien une et une seule couleur, puis qu'on n'a pas utilisé plus de k couleurs et enfin que pour toute arête e de G les deux extrémités sont bien de deux couleurs différentes.

Le problème du coloriage appartient donc bien à la clase NP.

Quelques problèmes classiques

- Le voyageur de commerce.
  Soient n points du plan d'une ville et une distance D. Existe-t-il un chemin qui passe par toutes les villes, retourne au point de départ et dont la longueur totale n'excède pas D ?
- Le coloriage de graphe.
  Soit un graphe G et un entier k. Peut-on colorier *correctement* les sommets de G avec k couleurs ?
- Le bin-packing.
  Soient un ensemble fini S d'entiers positifs, un entier k (la "bin-capacity") et un entier N (le nombre de "bin"). Existe-t-il une partition S en N sous-ensembles au plus, telle que la somme des entiers de chaque sous-ensemble soit $\leq k$?
  En d'autres termes, peut-on "empaqueter" les entiers de S dans au plus N cases sachant que la capacité de chaque case est k ?
- ...

Ces problèmes sont difficiles à résoudre. Considérons celui du coloriage de graphe. On peut essayer tous les moyens possibles de colorier les sommets de G en k couleurs pour voir si ça marche. Si G a n sommets, il y a k^n possibiltés. Si $n = 50$ et $k = 10$, ... les capacités des machines d'aujoudhui restent encore limitées.

4.6 Les problèmes NP-complets

Un problème de décision est *NP-complet* si :
- il est NP et
- si chaque problème NP est rapidement réductible en lui.

Les problèmes *NP-complets* sont nombreux. Ainsi, si l'on devrait trouver un algorithme de temps polynomial pour un certain problème *NP-complet* Q, on aurait trouvé par la même occasion un algorithme rapide pour tous les problèmes NP.

Soit une certaine instance I' d'un problème NP Q'. Puisque Q' est rapidement réductible en Q, on peut transformer I' en une instance I de Q. On applique alors l'algorithme trouvé à I. On aura ainsi utilisé du début à la fin un temps polynomial.

Supposons que l'on arrive un jour à démontrer que le problème de coloriage de graphes est *NP-complet* et que l'on trouve le jour suivant un algorithme rapide pour le résoudre.

Considérons alors une instance du "bin-packing". Elle peut être convertie en une instance de coloriage ayant la même réponse *oui* ou *non*. Utilisons alors l'algorithme rapide trouvé pour le coloriage, la réponse obtenue est aussi correcte pour le problème initial.

Réciproquement, supposons que l'on montre qu'il est impossible de trouver un algorithme rapide pour un certain problème Q *NP-complet*. On ne peut alors trouver d'algorithme rapide pour aucun des problèmes *NP-complets* Q', car sinon, on pourrait résoudre les instances de Q en les réduisant en instances de Q'.

Si on peut démontrer qu'il n'existe pas de méthode rapide pour tester si un entier est premier, on aurait alors montré qu'il n'existe pas de méthode rapide pour décider si des graphes sont $k-$coloriables, car le premier problème est *NP* est le deuxième est *NP-complet*.

En résumé, la rapidité pour un problème *NP-complet* entraîne la rapidité pour tous les *NP* ; la lenteur prouvée pour un problème *NP* entraîne la lenteur de tous les problèmes *NP-complets*.

Quelques propriétés des problèmes *NP-complets*

- Les problèmes sont tous très difficiles à résoudre et aucun algorithme se terminant en un temps polynomial n'a été trouvé pour aucun d'entre eux.
- L'impossibilité de trouver des algorithmes polynomiaux n'a pas été prouvée.
- Ce n'est pas simplement une liste de problèmes difficiles. Si un algorithme rapide pouvait être trouvé pour un d'entre eux, il y aurait des algorithmes rapide pour chacun d'entre eux.
- Réciproquement, s'il pouvait être prouvé qu'aucun algorithme n'existe pour un des problèmes *NP-complets*, il ne pourrait y en avoir pour aucun autre.
- L'existence ou non d'algorithmes polynomiaux pour les problèmes *NP-complets* est probablement considéré comme un des principaux problèmes encore irésolus en Informatique.

4.7 Les algorithmes non déterministes

(Voir études en TD/TP)

Bibliographie

[AA89] J. Ullman A. Aho, J. Hopcroft. *Structures de données et algorithmes*. Collection IIA. InterEditions, 1989.

[CF94] M. Soria C. Froidevaux, M-C. Gaudel. *Types de données et algorithmes*. Ediscience. McGrawHill, 1994.

[eCB84] B. Meyer et C. Baudoin. *Méthodes de programmation*. Eyrolles, 1984.

[eIG97] A. Arnold et I. Guessarian. *Mathématiques pour l'informatique*. Enseignement de l'informatique. Masson, 1997. 3e édition.

[RLC90] Rivest, Leserson, and Cormen. *Introduction à l'algorithmique*. Dunod, 1990.

[SF96] Robert Sedgewick and Philippe Flajolet. *Introduction à l'analyse des algorithmes*. International Thomson Publishing, 1996.

[Ste90] Stern. *Fondements mathématiques de l'informatique*. MacGrawHill, 1990.

[Wil89] H.S. Wilf. *Algorithmes et complexité*. Collection logique mathématiques et Informatique. Masson - Prentice Hall, 1989.

[Wol93] P. Wolper. *Introduction à la calculabilité*. Masson, 1993.

[Xuo92] N.X. Xuong. *Mathématiques discrètes et Informatiques*. Collection logique mathématiques et Informatique. Masson, 1992.