Théorie de la complexité.

Chapitre 1. Introduction et Notations

Dans la partie précédente, nous avons mis en évidence des problèmes sans solution algorithmique générale (problème de l'arrêt notamment). Pour d'autres problèmes des solutions algorithmiques existent. Mais un certain nombre d'algorithmes n'ont aucun intérêt pratique : par exemple générer tous les textes de longueur L sur un alphabet de x lettres nécessiterait un temps très supérieur au temps qui nous sépare du Big Bang même en utilisant tous les ordinateurs aujourd'hui disponibles dès que x est égal à 2 et L de l'ordre de la centaine.

Conventionnellement on dira qu'un algorithme est opérationnel (efficace) si la durée du temps d'exécution est bornée par un polynôme en fonction de la taille de la donnée. Pourquoi la notion de polynomialité est-elle importante ? Cette frontière entre ce qui est réalisable de façon opérationnelle ou pas est-elle purement théorique ? En fait lorsque la durée d'exécution d'un algorithme est polynomiale en fonction de la taille de la donnée, toute multiplication de la puissance de la machine entraîne une multiplication de la taille de la donnée que l'on peut traiter par unité de temps. Ainsi chaque fois que la puissance d'un ordinateur est multipliée par 8, la taille des données que l'on peut traiter par unité de temps est multipliée par 2 avec un algorithme en n^3 mais on ajoute que 3 avec un algorithme en 2^n.

Prenons le problème de l'existence d'un couplage de taille k. Si on prend un sous-ensemble d'arêtes de taille k, il est assez facile de vérifier si cet ensemble est un couplage ou pas (il suffit de vérifier que l'ensemble des extrémités des arêtes n'a pas de doublons et donc un algorithme en 0(k*log(k)) ou en 0(n) peut faire l'affaire). Un algorithme simple consiste donc à donner tous les sous-ensembles de k arêtes et pour chacun d'eux vérifier si le sous ensemble d'arêtes est bien un couplage. Cet algorithme va poser un problème malgré le fait que vérifier si un sous ensemble d'arêtes est un couplage est un problème rapide à résoudre. En effet le nombre de sous ensembles de taille k peut être colossal (petit exercice estimer le temps d'exécution au pire de l'algorithme si m=1000, k=40, le problème précédent se résolvant en 1 nano-seconde). Heureusement nous connaissez un algorithme rapide : on calcule le couplage maximum en 0(nm), si la cardinalité est supérieure à k la réponse est OUI sinon la réponse est NON. Bref nous pouvons répondre à la question de l'existence avec un algorithme de complexité en temps polynomial en fonction de la taille des données (le stockage du graphe nécessite au moins m+n bits et l'algorithme s'exécute en un temps au pire proportionnel à nm). Ce problème est dit de la classe P. On retrouve dans cette classe les problèmes de plus courts chemins, d'arbres couvrants de poids minimum etc. que vous avez déjà vus.

Cependant pour un grand nombre de problèmes pourtant utiles, on ne connaît pas d'algorithme efficace. La question de l'existence est-elle non ou va-t-on en trouver un un jour ? Prenons le problème du stable de taille k. Savoir si un sous-ensemble de taille k est un stable est aussi un problème rapide. Le problème est donc dit NP (comme le problème différent). Plus précisément si la réponse est OUI il existe un stable de taille k, je peux convaincre rapidement quelqu'un (ou prouver que la réponse est OUI rapidement). Remarquons que préparer ma preuve peut être long (pour convaincre que la réponse est OUI je vais par exemple chercher un stable de taille k). Remarquons aussi que la stratégie si je veux prouver que la réponse est NON n'est pas évidente (comment faire autrement que montrer que tous les sous-ensembles de taille k ne sont pas des stables. Pour l'instant on ne connaît pas d'algorithme aussi efficace que pour le couplage. On conjoncture d'ailleurs qu'il n'en existe pas pour ce problème car une réponse positive prouverait que P=NP.

Remarque : ces notions sont indépendantes du modèle de calcul comme pour la calculabilité. Les réponses à l'existence d'un algorithme efficace pour résoudre un problème donné sont donc largement universelles.

Quelques Notations et définitions

Un problème est défini par son nom, les données qu'il nécessite et le résultat attendu. Si ce résultat est de type booléen (OUI ou NON), on parlera de problème de décision (Exemple : Existe-t-il un stable de taille k dans un graphe G ?)

Nom : kStable
Données : Un entier k, un graphe G=(V,E)
Résultat : OUI si et seulement si il existe un stable de taille k dans G.

Sinon le problème est dit d'optimisation. Exemple :

Problème : MaxStable
Données : Un graphe G=(V,E)
Résultat : Donner la cardinalité maximum d'un stable de G.

La notion de réduction polynoniale.

Soit le problème de la couverture des arêtes d'un graphe.

Une couverture C de G=(V,E) est un sous-ensemble de sommets qui intersecte toutes les arêtes de E, c'est-à-dire pour toute arête $\forall e \in E : e \cap C \neq \emptyset$

Problème : kCouverture
Données : Un entier k, un graphe G=(V,E)
Résultat : OUI si et seulement si il existe une couverture de taille k dans G.

Supposons que la procédure kSTABLE résout kStable alors on peut résoudre kCouverture de la façon suivante :

kCOUVERTURE(k,G) :
renvoyer kSTABLE(n-k, G)

En effet, il suffit de remarquer que le complémentaire d'un stable est une couverture et réciproquement.
Autrement dit si j'ai un algorithme polynomial pour résoudre le problème du stable, j'ai un algorithme polynomial pour résoudre le problème de la couverture. Le problème de la couverture est donc plus facile que le problème du stable, au sens de la polynomialité. Elle se réduit polynomialement au problème du stable.
$kStable \in P \Rightarrow kCouverture \in P$
Il est facile de voir que la réciproque est aussi vraie. Donc ces 2 problèmes sont équivalents d'un point de vue de l'existence d'un algorithme polynomial pour les résoudre.

Chapitre 2. Problème NP-Complet.

On ne s'intéresse pour l'instant qu'aux problèmes de décision. Les problèmes d'optimisation correspondants sont qualifiés de NP-difficiles.

1. **La notion de certificat**

Un certificat C d'une réponse positive à un problème Π posé sur une donnée D est une preuve de cette réponse positive. Il est polynomial s'il peut-être vérifié en temps polynomial par rapport à la taille de D.

NP est la classe des problèmes de décision qui admettent un certificat polynomial pour les réponses positives.

Par exemple prenons le problème du cycle hamiltonien.

Données G=(V,E)
Question : Existe-t-il un cycle Hamiltonien dans G ?

Pour prouver que la réponse est OUI, il suffit de donner un ordre sur les sommets x_0, x_1, ... x_i, ...x_{n-1},x_n (avec $x_n=x_0$) et vérifier que $\forall i$, $\{x_i, x_{i+1}\} \in$ E. Donc le problème du cycle hamiltonien appartient à la classe NP.

Remarquons la non symétrie avec la réponse négative : comment prouver qu'un graphe n'a pas de cycle hamiltonien ?

La classe NP est plus grande que la classe P. Strictement ? C'est la grande question. La plupart des chercheurs du domaine pense que la réponse est OUI : la classe NP est strictement plus grande que la classe P.

2. **Définition d'un problème NP-Complet.**

Un problème X se réduit (polynomialement) à un problème Y si étant donné un algorithme efficace pour résoudre Y, on peut trouver un algorithme efficace pour résoudre X (efficace = qui s'exécute en temps polynomial par rapport à la taille de la donnée).

Un problème Π est NP-complet si et seulement si
(i) il est dans NP.
(ii) Tout problème de NP se réduit polynomialement à lui.

Les problèmes NP-complets sont donc les plus difficiles de la classe NP.

Avec la connaissance d'un problème NP-complet, on va facilement pouvoir prouver l'existence d'autres problèmes NP-complets. En effet pour montrer que Π est NP-complet il faut :

- montrer qu'il est dans NP (existence d'un certificat polynomial pour une réponse positive)
- trouver un problème Π' connu NP-complet et trouver une réduction de Π' à Π. C'est-à-dire pour chaque donnée D de Π' trouver une donnée D'=f(D) telle que Π(D')= Π'(D) et f se calcule en temps polynomial par rapport à la taille de D.

Théorème (Cook) : SAT est un problème NP-Complet

Preuve : le résultat est admis.

Présentation de SAT :

Le problème SAT est un problème de logique propositionnelle. Un prédicat est défini sur un ensemble de variables logiques à l'aide de 3 opérations élémentaires : la négation NON ($\neg x$ noté aussi $\bar{x}$), la conjonction ET ($x \wedge y$) et la disjonction OU ($x \vee y$). Un littéral est formé d'une seule variable (x) ou de sa négation ($\neg x$).

Une clause est un prédicat particulier formée uniquement de la disjonction de littéraux (par exemple la clause de taille 3 : $x \vee \neg y \vee z$). Cette clause vaut faux si et seulement si les variables x et z valent faux et la variable y vaut vrai. Une formule est sous forme normale conjonctive si elle s'écrit comme la conjonction de clauses. Le problème SAT consiste à savoir s'il existe une affectation des variables qui permette de rendre vrai une formule logique en forme normale conjonctive (toutes les clauses doivent donc valoir vrai).

Titre : SAT
Données : un ensemble de variables V et un ensemble de clauses C (construites à partir des variables et leurs négations)
Résultat : Existe-t-il une affectation des variables telle que l'ensemble des clauses soient vérifiées ?

Dans la suite on supposera sans perte de généralités qu'il n'y a pas de clause de taille 1 (le littéral correspondant étant obligatoirement vrai si la formule est satisfiable).

Soit V={x, y, z}. Prenons la formule $(\neg x \vee \mathbf{\neg y} \vee z) \wedge (\mathbf{x} \vee y) \wedge (x \vee \mathbf{\neg z})$ (composée donc de trois clauses). Elle est satisfiable en prenons par exemple z=faux, x=vrai et y=faux. En gras, on a un littéral égal à vrai par clause ce qui permet de conclure que toutes les clauses sont vérifiées.

Remarque : Sans perte de généralités on peut supposer que toutes les clauses sont au moins de taille 2.

Soit le problème 3SAT
Données : un ensemble de variables V et un ensemble de clauses C de taille 3 (construites à partir des variables et leurs négations)
Résultat : Existe-t-il une affectation des variables telle que l'ensemble des clauses soient vérifiées ?

Prouvons que 3-SAT est NP-complet.
(i) Le problème est dans NP. En effet en donnant une affectation, on peut en temps polynomial savoir si l'ensemble des clauses sont vérifiées.
(ii) Soit le problème SAT. Montrons que l'on peut réduire SAT à 3-SAT.

L'idée est de réécrire chaque clause comme un ensemble de clauses de taille 3 de telle façon qu'il existe une affectation rendant la clause initiale vraie si et seulement si il existe une affectation rendant l'ensemble des clauses de tailles 3 vraies.

a) Pour les clauses à 2 littéraux : $l_1 \vee l_2$. Il suffit d'ajouter une variable z à V et on remplace la clause par $(l_1 \vee l_2 \vee z)$ et $(l_1 \vee l_2 \vee \neg z)$.

b) Pour les clauses de taille k>3 : $l_1 \vee l_2 \vee \ldots \vee l_{\lfloor k/2 \rfloor} \vee l_{\lfloor k/2 \rfloor+1} \vee \ldots \vee l_k$. On ajoute une variable z à V et on remplace la clause par les deux clauses : $(l_1 \vee l_2 \vee \ldots \vee l_{\lfloor k/2 \rfloor} \vee z)$ et $(l_{\lfloor k/2 \rfloor+1} \vee \ldots \vee l_k \vee \neg z)$.
Soient 2 clauses de taille respectivement $\lfloor k/2 \rfloor+1$ et $\lceil k/2 \rceil +1$ donc strictement inférieure à k. On itère jusqu'à ce que les clauses soient de longueur 3.

Remarquons que la réduction ne marche pas pour 2-SAT. En effet une clause de taille 3 est transformée en une clause de taille 2 et une clause de taille 3. D'ailleurs

Théorème : 2-SAT est polynomial (preuve en TD).

Quelques Réductions vues en cours :

- Le problème du stable de taille k est NP-Complet (à partir de la réduction de 3SAT).

- Corollaire
  - La couverture des arêtes d'un graphe est NP-complet
  - Le problème de la clique max est NP-complet
  - Le problème max2SAT est NP-Complet (réduction à partir de 3-SAT)

Chapitre 3. Résolution de problèmes NP-complets par des algorithmes pseudo-polynomiaux

Comme a priori il n'existe pas d'algorithmes polynomiaux en fonction de la taille des données, on cherche des algorithmes polynomiaux en fonction de la valeur des données.

Remarque : ces notions sont sensiblement différentes, par exemple un unsigned int en C++ se code généralement sur 4 octets. Sa taille est donc de 32 bits mais sa valeur peut atteindre 4 milliards.

Prenons le problème de la somme.

Titre : Somme de sous-ensemble
Données : E une collection de n nombres (e_i est le $i^{ième}$ entier) et un entier T
Question : Existe un sous-ensemble de E dont la somme des éléments vaut T ?

Ce problème est NP-complet.

Algorithme : on va remplir un tableau t de booléen à 2 dimensions avec 1+n lignes et 1+T colonnes.

t[i,j] faudra true si on peut atteindre la somme j avec les seuls entiers 1, 2, …, i.

Ainsi la première contient un True à la première colonne et des False à toutes les autres. La deuxième ligne contient un True à la première colonne et dans la colonne e_1 et des False dans toutes les autres colonnes.

Il est facile de voir que $t[i,j] = t[i-1,j] \vee t[i-1, j-e_i]$. Donc on peut facilement remplir le tableau en 0(nT) et le résultat du problème est la valeur de t[n,T].

Faisons la trace sur un exemple avec E=(2, 2, 3) et T=6.

Remarque : T est la valeur d'un des données (la taille de T est log(T)). Cet algorithme a tout de même de grande chance d'être meilleur que 2^n (prendre toutes les sous-collections possibles).

On peut aussi avoir le programme suivant :

```
bool somme(int i, int j)
{
if (j<0) return false ;
else if  (i==0 ) return j==0 ;
else return somme(i-1,j) || somme (i-1, j-e[i]) ;
}
```

La solution est donnée par l'appel somme(n,T) ;

Quelle est la différence entre les 2 programmes ?
Le premier est nettement meilleur car il permet d'éliminer les calculs redondants (on mémorise les résultats intermédiaires). Cette technique est appelée **programmation dynamique.**

En TD nous verrons d'autres problèmes se résolvant de cette façon : le problème de la partition, le problème du voyageur de commerce.