Chapitre 8

Une introduction à la théorie de la NP-Complétude

1. Introduction: Au chapitre 1, nous avons discuté l'importance d'avoir des solutions de complexité polynomiale. Dans l'étude de la complexité des problèmes, le point important qui concerne aussi bien le théoricien que le praticien est de savoir si, pour un problème donné, il existe un algorithme de complexité polynomiale pour le résoudre.

Dans ce chapitre, nous allons, dans un premier temps, passer en revue les différentes classes de complexité des problèmes. Ensuite, nous allons nous concentrer sur les techniques de réduction pour montrer l'improbabilité d,existence d'algorithme efficace pour résoudre tel ou tel problème. Le lecteur praticien va trouver cette notion de ne prouver rien du tout quelque peu singulière : on investit son temps pour prouver que quelque chose n'existe pas. Pourquoi donc seriez-nous mieux en sachant que quelque chose qu'on ne sait pas faire n'existe pas ?

En réalité, la théorie de la NP-complétude est utile pour le concepteur d'algorithmes, même si elle ne produit que des résultats négatifs : « Quand l'impossible est éliminé, ce qui reste, même improbable, doit être la vérité ». La théorie de la NP-complétude permet de concentrer les efforts d'une manière productive en révélant qu'il y a peu de chance que la recherche, pour une solution efficace pour un problème, soit un succès. Autrement dit, quand on échoue de montrer que tel problème est difficile à résoudre, alors on a plus de chance de lui en trouver un algorithme efficace.

Par ailleurs, la théorie de la NP-complétude nous permet d'identifier les propriétés qui font qu'un problème soit difficile. Cela devrait nous aider à mieux cerner les différentes directions de modélisation et de résolution.

2. Classification de problèmes

Pour des raisons de simplicité et techniques aussi, la théorie de la complexité se limite juste à l'étude des problèmes de décision.

Définition 1. Un problème de décision est un problème dont la solution est formulée en termes oui/non

Exemple 1: Étant donné un graphe G = (X,E), existe t-il un chemin de longueur ≤ L ?

Exemple 2: Étant donné un graphe G=(X,E), les sommets de X peuvent-il être colorés par au plus m couleurs de telle manière que les sommets adjacents soient de couleurs différentes.

Exemple 3. Soit $S = \{a_1, a_2, ..., a_n\}$ un ensemble de *n* entiers. Existe t-il $R \subset S$ tel que

$$\sum_{i \in R} a_i = \sum_{i \in S-R} a_i$$

De cette définition, ce qui importe c'est juste l'existence de la solution. Cette restriction aux problèmes de décision est justifiée par le fait que les autres problèmes qui ne sont pas de décision, comme les problèmes d'optimisation, peuvent être facilement transformés en un problème de décision équivalent.

Le but de la théorie de la complexité est la classification des problèmes de décision suivant leur degré de difficulté de résolution. Dans la littérature, il existe plusieurs classes de complexité, mais les plus connues sont les suivantes.

- **La classe P**. c'est la classe des problème pouvant être résolus en un temps polynomial. C'est la classe des problèmes dits facile.

- **La classe NP :** C'est l'abréviation pour "nondeterministic polynomial time". Cette classe renferme tous les problèmes de décision dont on peut associer à chacun d'eux un ensemble de solutions potentielles (de cardinal au pire exponentiel) tel qu'on puisse vérifier en un temps polynomial si une solution potentielle satisfait la question posée. Le terme nondeterministe désigne un pouvoir qu'on incorpore à un algorithme pour qu'il puisse deviner la bonne solution.

  Exemple de problème dans NP: le problème SAT (de satisfiabilité).

  Soit $F(x_1, x_2, ..., x_n)$ une expression logique de *n* variables. Le problème SAT consiste à trouver des valeur vrai ou faux pour chacune des variables x_i de telle manière à rendre vrai l'expression $F(x_1, x_2, ..., x_n)$.

  Un algorithme nondéterministe résolvant SAT est comme suit :

  For (k=1,k<=n,k++)
  x_k = choix(vrai,faux) ;

  $F(x_1, x_2, ..., x_n)$ = vrai alors F est satisfiable sinon F n'est pas satisfiable.

Une autre manière, plus informelle pour définir la classe **NP** : c'est la classe des problèmes pour lesquels les seuls algorithmes connus de résolution sont ceux ayant une complexité exponentielle.

- **La classe PSPACE**. Problèmes pouvant être résolus en utilisant une quantité d'espace mémoire raisonnable (définie comme une expression polynomiale en fonction de la taille des données) sans considération du temps d'exécution que cela prendra. .

- **Classe EXPTIME**. Problèmes ne pouvant être résolus que par des algorithmes de complexité temporelle exponentielle.

- **Classe Indécidable** Pour certains problèmes, il est possible de montrer qu'il ne peut exister du tout d'algorithmes pour les résoudre, peu importe le temps ou l'espace qui leur est alloué.

Problématiques de théorie de la complexité

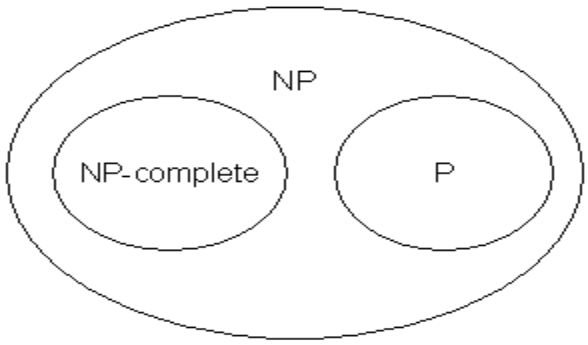
La théorie de la complexité s'intéresse à l'étude de classification de problèmes et les frontières existant entre ces différentes classes. Elle étudie aussi les limites de calcul pour résoudre un problème donné. La problématique centrale, la plus connue, dans cette théorie est la fameuse question : **P = ? NP**. En d'autres termes, s'il est toujours facile de vérifier une solution, est-il aussi facile de trouver une solution ? la réponse à cette question résout plusieurs autre grandes questions de cette discipline, mais bien entendu pas toutes les questions.

Il n'y a pas de raison de croire que l'égalité (**P = NP**) soit vraie. L'attente de la plupart des informaticiens et mathématiciens est que l'égalité soit fausse. Malheureusement, il n'existe pas de preuve pour cette assertion.

Par conséquent, toute une théorie est construite sur les classes **P** et **NP** sans que l'on puisse affirmer s'il existe un problème dans **NP** qui ne soit pas dans **P**. Par conséquent, quelqu'un pourrait dire, à juste titre d'ailleurs, quelle est l'intérêt d'une théorie si elle ne peut répondre à la question de connaître le degré de difficulté d'un problème donné. Une réponse à cette question vient de la notion de **NP-complétude**.

La NP-complétude

La théorie de la **NP-complétude** concerne la reconnaissance des problèmes les plus durs de la classe **NP**. La notion de la difficulté d'un problème qui est introduite dans cette classe est celle d'une classe de problème qui sont équivalents en ce sens que si l'un d'eux est prouvé être facile alors tous les problèmes de **NP** le sont. Inversement, si l'un d'eux est prouvé être difficile, alors la classe **NP** est distincte de la classe **P**.



Définition : Un problème de décision est dit **NP-complet** si tout problème de la classe **NP** lui est polynomialement réductible.

Notons qu'il existe des problèmes dans **NP** qui ne sont pas dans P et qui, vraisemblablement, ne seront pas dans la classe **NPC** (pour dire **NP-complet**)
Le concept central relié à la définition de la **NP-complétude** est celui de la réduction entre problèmes.

La réduction polynomiale

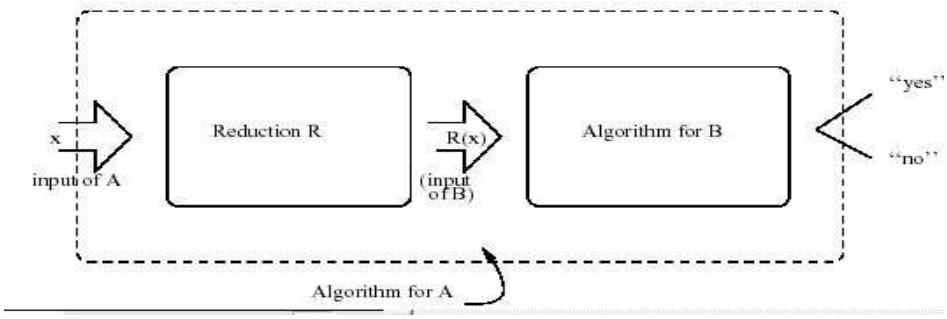
Nous avons vu dans le chapitre précédent que les techniques de réduction nous permettent de relier deux problèmes entre eux. En d'autres termes, la réduction peut être vue comme un moyen pour affirmer qu'un problème est aussi facile ou difficile qu'un autre problème.

L'utilisation de la réduction qu'on a discutée était celle de la résolution d'un problème à l'aide d'un autre problème. Ainsi, pour résoudre un problème donné, nous pouvons le transformer vers un autre problème équivalent dont on connaît la solution.

Dans ce chapitre, nous pouvons aussi utiliser la réduction pour montrer la difficulté de résolution d'un problème en transformant un problème déjà connu comme étant difficile à résoudre vers notre problème de départ. .

Définition 8.1: Un problème A est **réductible** à un problème B s'il existe un algorithme résolvant A qui utilise un algorithme résolvant B.

Définition 8.2: Si l'algorithme résolvant A est polynomial, considérant les appels à l'algorithme résolvant B comme de complexité constante, la réduction est dite polynomiale. On dit que A est polynomialement réductible à B et l'on note A α B**.**



Théorème : la réduction α est transitive, c'est-à-dire

$$P1 \ \alpha \ P2 \ \textbf{et} \ P2 \ \alpha \ P3 \ \Rightarrow \ P1 \ \alpha \ P3.$$

Existence de problèmes NP-complets

Théorème de Cook: Le problème de **satisfiabilité** est **NP-complet** .

Cook (1971) a montré que tous les problèmes de la classe NP sont réductibles au problème de la satisfiabilité d'une expression logique quelconque

Autrement dit, si jamais quelqu'un venait à trouver un algorithme polynomial pour le problème de SAT, alors la question de savoir si P = NP est résolu de fait !

Exemples de problèmes NP-complets

1. **problème du stable:** trouver dans un graphe fini un ensemble de m sommets non reliés (m étant donné)

2. **problème du sac à dos :** soient un ensemble S de nombre entiers positifs et un entier M. Existe t-il une partie A de S telle que $\sum_{i \in A} a_i = M$

3. **problème du cycle hamiltonien :** soit un graphe fini. Existe t-il un cycle de longueur n passant passant une et une seule fois par tous les n sommets.

4. **problème du 3-coloriage :** soit un graphe fini. Peut-on colorier les sommets du graphe à l'aide de 3 couleurs de telle manière que deux sommets adjacents n'aient pas la même couleur.

5. **problème des machine parallèles.** Soient *m* machines identiques et *n* jobs. Chaque job *i* doit être exécuté sans interruption par une des *m* machines pendant un temps p_i. Existe t-

il une partition des *n* jobs sur les *m* machines de telle manière que le temps de fin de tous les jobs ne dépasse pas le temps T

Remarque : Une liste de centaines de problèmes est produite dans le livre de M.R.Garey and D.S. Johnson (1979) : Computer and intractability , Freeman Co.

Comment prouver la NP-complétude d'un problème en pratique

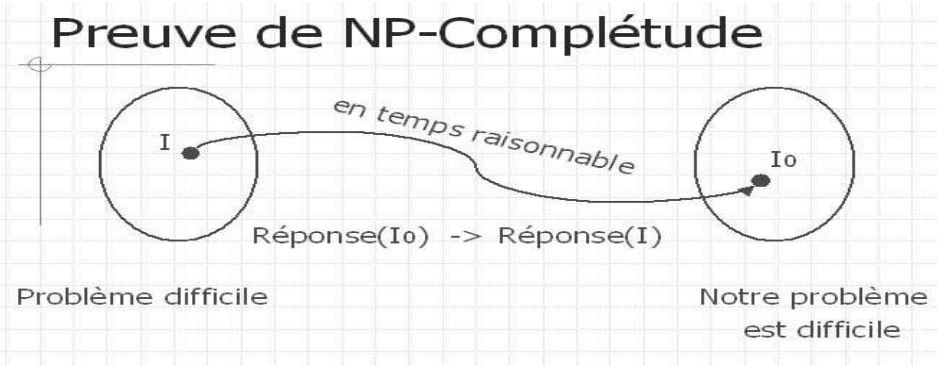
Il est utile de signaler que montrer qu'un problème est **NP-complet** signifie qu'il existe au moins une instance pour laquelle le seul algorithme connu pour résoudre sur cette instance est exponentiel.

Par ailleurs, réduire un problème P1 à un autre problème P2 revient à montrer que P1 est au moins plus facile à résoudre que P2, et P2 est au moins plus difficile à résoudre que P1. Autrement dit, cette réduction signifie que P1 est un cas particulier de P2.

Sachant que la réduction polynomiale est transitive, pour montrer qu'un nouveau problème P est NP-complet, on procède comme suit (voir la figure ci-dessous) :

1. Montrer P est dans NP
2. Choisir un problème, P2, **NP-complet** approprié
3. Construire une réduction f, qui transformant P2 vers P telle que

I une oui-instance de P2 $\Leftrightarrow$ f(I) une oui-instance de P



Note: Si on suspecte la NP-complétude d'un problème, la première étape à suivre est de consulter le livre de Garey et Johnson. La deuxième étape est de lui trouver un problème similaire dans ce livre, et construire une réduction que telle que discutée plus haut. Si cela ne fonctionne pas, cela ne devrait pas nous faire changer de direction et chercher un algorithme efficace pour le problème en question. .

Exemple de preuve de NP-complétude

Soit le problème de *n* jobs à exécuter sur une machine. Chaque job à un temps d'exécution p_i, une date de disponibilité r_i et une date d'échéance d_i. L'objectif est de trouver un ordonnancement qui minimise $L_{\max} = \max(C_i - d_i, 0)$. C_i étant le temps de fin du job *i*. Cette fonction tend à minimiser la longueur des retards des jobs.

Noua allons montrer que le problème de décision correspondant à ce problème est NP-complet.

Le problème de décision correspondant

Soient *n* jobs ayant des temps d'exécution p_i i, des date de disponibilité r_i et des dates d'échéances d_i, et un nombre y. Existe t-il un ordonnancement tel que $L_{\max} \leq y$.

1. Montrons que ce problème est dans NP : Étant donnée une solution, il est clair que le temps pris pour vérifier si elle est valide et si $L_{\max} \leq y$ est $O(n)$.

2. Le problème P2 que nous allons réduire au problème de départ est celui du sac à dos, comme défini plus haut, c'est-à-dire:

un ensemble $S = \{a_1, a_2, ..., a_N\}$ de *N* nombres et un nombre *b* positif tel que $b \leq \sum_{i=1}^{N} a_i$. Existe t-il $A \subset S$ tel que $\sum_{a_i \in A} a_i = b$?

3. L'étape suivante est de construire une réduction polynomiale f du problème de sac vers notre problème telle que telle que, une instance I du sac à dos a une réponse oui si, et seulement si, f(I) une instance de notre problème possède une réponse oui.

Soit I une instance quelconque du problème du sac à dos. Construisons une instance f(I) de notre problème comme suit.

$$n = N + 1;$$
$$r_i = 0; p_i = a_i; d_i = \left(\sum_{i=1}^{N} a_i \right) + 1; i = 1,2,...,N$$
$$r_{N+1} = b; p_{N+1} = 1; d_{N+1} = b + 1;$$
$$y = 0.$$

Notons que la seule date de disponibilité non nulle est celle du job N+ 1. Il est clair que cette construction d'une instance de notre problème, à partir de celle du sac à dos, se fait en un temps polynomial : nous avons juste à affecter (N+5) valeurs faciles à calculer.

1. Nous allons maintenant montrer que $L_{\max} \leq 0$ si et seulement si, il existe A tel que $\sum_{a_i \in A} a_i = b$.

1.supposons que le sous-ensemble A tel que $\sum_{a_i \in A} a_i = b$ existe. Considérons alors l'ordonnancement qui exécute en premier les jobs correspondants à A, dans un ordre quelconque, ensuite le job N+ 1, et ensuite le reste S-A dans un ordre quelconque.

Les jobs de A se termine à l'instant b, car $\sum_{a_i \in A} a_i = b$. Le job b finit à l'instant b+1. Le reste des jobs est exécuté sans temps mort et se finit à l'instant $\sum_{i=1}^{N} a_i + 1$. Par conséquent, aucun job n'est en retard dans cette solution. Autrement dit, $L_{\max} \leq 0$.

On peut donc conclure que si le problème du sac à dos a une solution alors notre problème aussi en possède une.

2. Supposons maintenant que notre problème possède une solution, alors nous devrions montrer que le problème du sac en possède une aussi. Pour montrer cette implication, il est plus simple de montrer sa contraposée. Autrement dit, supposons que le problème du sac à dos ne possède pas de solution. Cela signifie qu'il n'existe pas $A \subset S$ tel que $\sum_{a_i \in A} a_i = b$. Considérons un ordonnancement quelconque. Soit A l'ensemble des jobs exécutés avant le job N+1. Alors cet ensemble A finit à l'instant

$$C_A = \sum_{p_i \in A} p_i = \sum_{p_i \in A} a_i \neq b \text{, par hypothèse.}$$

Si $C_A > b$, alors le job N+1, s'exécutant juste après, finit au temps $C_A + 1$. Autrement dit, ce job sera en retard, c'est-à-dire que $L_{\max} > 0$.

Si $C_A < b$ alors le job N+1 ne peut s'exécuter juste après, car $r_{N+1} = b$. Autrement dit, il y a un temps mort juste avant le job N+1. Le traitement des autres jobs prend un temps total égal à $\sum_{i=1}^{n} p_i - C_A$. Par conséquent, le temps d'accomplissement du dernier job est

$$\begin{aligned} C_{\max} &= b + 1 + \sum_{i=1}^{N} p_i - C_A \\ &> \sum_{i=1}^{N} p_i + 1 \quad \text{car} \quad b > C_A \end{aligned}$$

Comme la date d'échéance de chaque job est $\sum_{i=1}^{N} p_i + 1$, il doit se trouver au moins un job qui est en retard. Autrement dit, nous avons $L_{\max} > 0$. Par conséquent, si le problème du sac à dos à un réponse négative, alors notre aussi a une réponse négative.

Le problème du sac à dos est par conséquent équivalent à notre problème. Nous pouvons donc conclure que notre problème d'ordonnancement est **NP-complet**.

Conclusion

Les machines ont des limites théoriques : elles ne peuvent pas résoudre TOUS les problèmes. Néanmoins, elles en résolvent un grand nombre !