Décidabilité et complexité

Ce chapitre a pour but l'étude des *limites* de ce qu'un ordinateur est capable de faire. Nous verrons que certains problèmes dits *indécidables* sont *mathématiquement* impossibles à résoudre pour un ordinateur. Pour tout une famille d'autres problèmes appelés problèmes *NP-complets*, on ne sait pas à l'heure actuelle s'il existe des algorithmes polynomiaux pour les résoudre. En général on considère qu'ils sont *intractables* c'est-à-dire pas résolubles en pratique de manière exacte et dans un temps raisonnable.

1 Problèmes de décision

Définition

Un **problème de décision** est une *question* portant sur un ensemble de *données* et dont la réponse est soit *oui* soit *non*. Les valeurs particulières que peuvent prendre les données en entrées sont appelées **instances** du problème.

Exemples de problèmes de décision :

- **Entrée :** 3 entiers (a, b, c). **Question :** Existe-t-il dans le plan, un triangle ayant des côtés de longueurs a, b, c dans un certain ordre ?
- $(3, 7, 2)$ est une instance du problème de décision précédent, pour cette instance particulière la réponse est *non*.
- **Entrée :** un mot sur l'alphabet $\Sigma = \{a, b\}$. **Question :** Le mot contient-il autant de a que de b ?
- *aabbbaab* est une instance du problème de décision précédent, pour cette instance particulière la réponse est *oui*.
- **Entrée :** un graphe non-orienté G sous forme de matrice d'adjacence et deux sommets A et B. **Question :** A et B sont-ils dans la même composante connexe ?
- Problème SAT **Entrée :** une formule de la logique propositionnelle. **Question :** la formule est-elle satisfiable ?

Définition

Un problème de décision est **décidable** s'il existe une **machine** qui prend en entrée une instance du problème, dont le calcul **termine toujours** et qui retourne la réponse *oui* ou *non*.

La notion de **machine** dans cette définition est cruciale :

- Vous connaissez déjà un exemple de *machine* : les automates finis prennent bien en entrée des données (sous forme de mots) et répondent oui ou non. Le problème est que cette machine n'est pas assez puissante pour représenter la puissance de calcul d'une machine réelle. Nous savons par exemple qu'un automate ne peut pas tester si un mot contient autant de a que de b alors qu'un ordinateur réel sait parfaitement traiter ce problème.
- Le modèle de machine théorique adapté à ce cours est la *machine de Turing* : elle est hors-programme en MPI.
- En MPI, on considérera que la machine est un ordinateur à mémoire infinie exécutant un programme écrit en `C` ou en `OCaml` prenant en entrée une instance du problème et répondant *oui* ou *non*.

Les problèmes décidables sont donc les problèmes qu'il est possible en théorie de résoudre sur un ordinateur.

2 La classe P

Parmi les problèmes décidables on sait en résoudre certains plus efficacement que d'autres.

La classe P correspond aux problèmes de décision qu'on sait résoudre en temps polynomial. On considère aujourd'hui que ce sont les problèmes qu'on peut aborder *en pratique*, toutefois les algorithmes de résolution proposés peuvent quand même avoir des complexités importantes.

Les problèmes qui ne sont pas dans P sont habituellement considérés comme *insolubles en pratique* : il existe des algorithmes de résolution mais leur complexité en temps est alors supérieure à $O(n^k)$ pour tout k (par exemple la complexité est exponentielle), ce qui rend l'algorithme inutilisable sauf dans le cas d'instances très petites.

De plus la classe P est intéressante car elle est robuste au petits changements : manière dont on compte les opérations élémentaires par exemple.

Définition

Soit un problème de décision et une instance de ce problème. La **taille de l'instance** est le nombre de **bits** utilisés pour coder l'instance en mémoire.

On pourra considérer :

- Si l'instance est un entier k alors la taille de l'instance est $n = \log_2(k)$.
- Si l'instance est un texte, la taille de l'instance est la longueur du texte.
- Si l'intance est un arbre, la taille de l'instance est la taille de l'arbre (nombre de feuilles et de nœuds).
- Si l'intance est une formule logique, la taille de l'instance est le nombre de symboles (connecteurs, quantificateurs, symboles de variables, de constantes, de fonctions) utilisés pour l'écrire.

Définition

Un problème de décision est dans la **classe *P***, s'il existe un réel $k > 0$ et une machine M qui décide ce problème avec un **temps d'exécution** en $O(n^k)$ où n est la **taille de l'instance**.

- Remarque : on obtient une définition équivalente si on dit "s'il existe un polynôme Q tel que le temps d'exécution est en $O(Q(n))$". Ceci explique le nom de la classe, le P signifie polynomial.
- Pour compter le temps, on compte le nombre d'opérations élémentaires effectuées : opérations arithmétiques, lectures/écritures en mémoire, etc...

Attention à ne pas confondre **complexité d'un algorithme** et **classe de complexité d'un problème de décision**. Par exemple, cela n'a aucun sens de dire : "l'algorithme de Dijkstra est dans P" mais on peut dire "l'algorithme de Dijkstra s'exécute en temps polynomial par rapport à n = nombre de sommets".

Exemples : Problème : le tableau t est-il trié, Problème : un graphe G est-il connexe ? Problème : nombre entier n est-il premier ?

Définition

- On dit qu'un problème de décision A **se réduit à** un problème de décision B s'il existe une machine M qui termine toujours, qui prend en entrée une instance I_A du problème A et qui produit en sortie une instance I_B du problème B tel que :

  $$\text{La réponse à } I_A \text{ est } oui \Leftrightarrow \text{La réponse à } I_B \text{ est } oui$$

  On notera alors $A \leq B$.

- Si de plus, il existe un réel $k > 0$ tel que la machine M s'exécute en temps $O(n^k)$ où n est la taille de l'intance I_A, alors on dit que A **se réduit polynomialement à** B et on note $A \leq_P B$.

Point méthode

Pour montrer $A \leq_P B$:

1. On donne un **algorithme de réduction** qui transforme toute instance I_A de A en une instance I_B de B.
2. On vérifie que l'algorithme est de complexité polynomiale par rapport à la taille de l'instance I_A.
3. On justifie l'**équivalence** : I_A est une instance positive de A ssi l'instance I_B construite par l'algorithme de réduction est positive pour le problème B.

Exemples (vu en détail en cours) :

1. réduction de INDEPENDANT-SET vers CLIQUE (et vice-versa)
2. réduction de N-REINES-COMPLETION à SAT
3. réduction de HAMILTON à TSP

Le principe de réduction est un principe général en théorie de la complexité. Il exprime le fait qu'un problème de décision est plus *difficile* qu'un autre. En effet, si je dois résoudre le problème A et que je sais résoudre le problème B, et que $A \leq B$, alors il suffit de transformer mon instance de A en une instance de B pour répondre au problème A. Donc si je sais résoudre B je sais résoudre A ce qui signifie que B est au moins plus difficile que A, d'où la notation $A \leq B$. Selon les propriétés qu'on cherche à établir, on utilisera l'outil de réduction adapté :

Proposition

Soit A, B, C trois problèmes de décision, on a :

1. $A \leq B \Rightarrow A \leq_P B$
2. $A \leq_P A$
3. $A \leq B$ et $B \leq C$ implique $A \leq C$.
4. $A \leq_P B$ et $B \leq_P C$ implique $A \leq_P C$.

Proposition

Soit A et B deux problèmes de décision.

1. Si $A \leq B$ et B est décidable alors A est décidable.
2. Si $A \leq_P B$ et $B \in P$ alors $A \in P$.

Point méthode

Pour démontrer qu'un problème de décision est dans P on peut au choix :

1. Donner un algorithme ou un programme qui répond au problème de décision et ayant une complexité en temps polynomiale par rapport à n = taille de l'instance.
2. Établir rigoureusement que $A \leq_P B$ où B est un problème dont on sait déjà qu'il est dans P.

3 La classe *NP*

Pour beaucoup de problèmes, il n'est pas facile de déterminer un algorithme de résolution polynomial. Cependant, lorsque la réponse à la question est *oui* il est souvent possible d'exhiber une *preuve*, un *témoin* qui justifie que la réponse est *oui*. Il se trouve alors que vérifier que la réponse est bien *oui* avec l'aide du témoin est un problème facile.

La classe **NP** correspond à cette classe de problèmes de décision où l'on sait **vérifier** une solution en temps polynomial. Elle est très importante car les problèmes qu'elle contient ont un fort intérêt pratique en informatique.

Exemples vus en cours : exitence d'un chemin entre deux sommets d'un graphe, satisfiabilité d'une formule logique, SUBSETSUM.

Définition

Un problème de décision A est dans la classe **NP** s'il existe un réel $k \geq 0$ et une machine M à deux entrées (I, C), qui s'exécute en temps $O(n^k)$ par rapport à la taille n de l'instance I et telle que :

$$I \text{ est une instance positive de } A \Leftrightarrow \exists C,\ M(I, C) = \text{oui}$$

L'entrée C est appelée **certificat** et correspond à l'idée de preuve de positivité. Ainsi un problème est dans NP si ses instances positives admettent un certificat qui peut être vérifié en temps polynomial par la machine M appelée **vérificateur**.

On remarque que l'équivalence nous dit aussi que pour toute instance négative, le vérificateur n'accepte aucune preuve :

$$I \text{ est une instance négative de } A \Leftrightarrow \forall C,\ M(I, C) = \text{non}$$

On remarquera qu'on peut se restreindre à des certificats C de taille polynomiale par rapport à n sinon la machine ne pourrait lire le certificat en temps polynomial. **Si vous prouvez qu'un problème est dans NP avec des certificats non polynomiaux par rapport aux instances : il y a problème dans votre preuve**.

Point méthode

Pour montrer qu'un problème est dans NP :

1. On précise ce qu'est un certificat C pour une instance positive du problème.
2. On propose un algorithme ou un programme prenant en entrée une instance et un certificat, qui s'exécute en temps polynomial, et qui retourne *oui* si et seulement si le certificat est valide. (Il ne doit répondre oui que sur les instance positives bien sûr...)

Proposition

P ⊂NP

Démonstration : Soit $A \in P$ décidé par une machine M_A. On considère alors le véricateur $V(I, C) = M_A(I)$ qui ignore l'entrée certicat. Le vérificateur travaille bien en temps polynomial. De plus $\Rightarrow$ est vraie, en considérant un certificat quelconque par exemple $C = \varnothing$ le certificat vide et $\Leftarrow$ est vraie par définition de P.

Aujourd'hui on ne sait toujours pas si $P = NP$. Le problème est ouvert depuis 1971 et la réponse sera récompensée d'1 million de dollars. En langue française la question pourrait s'énoncer ainsi "est-ce qu'un problème facile à vérifier est un problème facile à résoudre ?".

4 La classe NP-complet

Définition

Un problème de décision A est **NP-complet** si :

1. $A \in NP$
2. $\forall B \in NP, \quad B \leq_P A$ (NP-difficile)

Autrement dit les problèmes NP-complets sont les problèmes de NP les plus difficiles.

Proposition

Soit A un problème NP-complet, si on démontre que $A \in P$ alors $P = NP$.

Ainsi, si on suppose que $P \neq NP$, cela signifie qu'il n'existe aucun algorithme polynomial pour résoudre un problème NP-complet. Ils sont donc pratiquement insoluble par ordinateur (sauf sur de très petites instances) **dans le pire cas**.

Point méthode

Pour démontrer qu'un problème de décision A est NP-complet, on montre les **deux** points suivants :

1. On démontre que A est dans NP.
2. On établit rigoureusement que $B \leq_P A$ où B est un problème dont on sait déjà qu'il est NP-complet.

Ainsi plus on connait de problèmes NP-complets plus il est facile de montrer que d'autres problèmes le sont. Il est donc utile de connaître quelques problèmes NP-complets classiques pour répondre à des questions de NP-complétude.

Proposition

Théorème de Cook-Levin (admis)

Problème de décision SAT
Entrée : une formule de la logique propositionnelle
Question : existe-t-il une assignation *vrai*/*faux* des variables qui rend la formule *vraie* ? Autrement dit, la formule est-elle satisfiable ?

Le problème SAT est NP-complet.

Idée de la preuve

- SAT est dans NP : facile
- La logique est très puissante, on peut imaginer qu'on puisse *coder* le fonctionnement d'une machine quelqueconque avec des variables et des règles logiques.
- Si B est un problème de NP, on code avec des formules comme expliqué ci-dessus le fonctionnement de la machine qui vérifie les certificats du problème A.

Réductions vues en cours :

- CLIQUE (existe-t-il une clique de taille k) est NP-complet, réduction depuis $CNF - SAT$, illustration avec $(x \vee x \vee y) \wedge (\neg x \vee \neg y \vee \neg y) \wedge (\neg x \vee y \vee y)$
- 3-SAT est NP-complet : réduction depuis SAT
- CNF-SAT est NP-complet : réduction (triviale) depuis 3-SAT

5 Les problèmes indécidables

Certains problèmes sont théoriquement insolubles. On n'en présente que deux, qui sont fondamentaux. La méthode de la réduction (non nécessairement polynomiale) permet ensuite de démontrer que d'autres problèmes sont indécidables.

Problème de l'arrêt. (démontré en admettant l'existence de la machine universelle)

machine universelle : MU(prog, entree)

On suppose qu'il existe HALT(prog, entree) qui retourne oui si le prog s'arrête sur l'entree. On construit la machine suivante : D(prog): retourne oui si HALT(prog, prog) retourne non boucle sinon

que retourne D(D) ? Le calcul de D(D) termine ssi HALT(D, D) retourne oui ssi D(D) boucle. C'est absurde. Donc la machine HALT n'existe pas.

Problème de correspondance de Post (admis)

En général les problèmes indécidables portent sur des questions plus théoriques que pratiques, voici quelques exemples :

- savoir si une propriété non triviale sur l'ensemble de tous les programmes est vraie ? (Théorème de Rice)
- savoir si une formule est valide dans la logique du premier ordre ?
- savoir si un énoncé arithmétique est vrai ?
- savoir si une équation diophantienne admet une solution ?
- est-il possible de paver un plan avec jeu de tuiles ?