Complexité de Problèmes: les propriétés NP-dures

ACT

Sophie Tison
Université Lille
Master1 Informatique

COMPLEXITÉ DES PROBLÈMES

LE BILAN DE LA DÉFINITION DES NP

NP= non-déterministe polynomial

Une propriété *NP* est une propriété pour laquelle trouver une "solution" (une preuve ..) est peut-être difficile, mais vérifier une solution (une preuve, un certificat...) est facile.

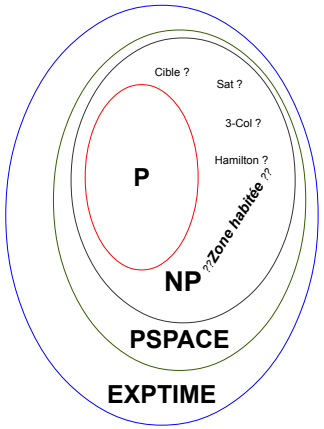
P versus NP

P=easy to find, NP=easy to check

Montrer qu'une propriété est *NP* n'est en général qu'une étape.

MONTRER QU'UNE PROPRIÉTÉ EST *NP* N'EST PAS MONTRER QU'ELLE N'EST PAS *P*!

La hiérarchie



La problématique

On cherche un algorithme pour vérifier une propriété; on a montré qu'elle est NP mais on ne trouve pas d'algorithme P.

Si on arrive à prouver qu'il n' y a pas d'algorithme polynomial, on prouve $P \neq NP$: ça risque d'être difficile...

On peut essayer de montrer qu'elle est $NP-$dure.

La problématique

Une propriété $NP-$dure contient d'une certaine façon toute la difficulté de la classe NP.

Si on trouve un algorithme polynomial pour une telle propriété, on en aurait un pour toute propriété NP.

Montrer qu'une propriété est $NP-$dure justifie en quelque sorte de ne pas avoir trouvé d'algorithme polynomial.

Un outil: Les réductions polynomiales

La notion de réduction permet de traduire qu'un problème n'est pas "plus dur" qu'un autre.

Si un problème A se réduit en un problème B, le problème A est (au moins) aussi facile que B - si la réduction est "facile"-. On peut a priori utiliser la réduction de deux façons:

- la "classique pour un développeur" pour résoudre le problème A en utilisant l'algo qui résout B - si B est facile, A l'est aussi.
- pour montrer qu'un problème est dur: si A est réputé dur, B l'est aussi;

Dans cette partie du cours, c'est ce deuxième raisonnement que l'on va utiliser.

Définition formelle

Réduction polynomiale

Soient L et L' deux langages de Σ^*, correspondant à deux propriétés.

Une **réduction polynomiale** de L dans L' est une application *red* calculable polynomiale de Σ^* dans Σ^* telle que:

$$u \in L \text{ Ssi } red(u) \in L'.$$

Notation: $L \leq_P L'$.

Réduction polynomiale = transformation d'un problème dans un autre.

Par exemple, soient les deux problèmes:

- Pb1
  Donnée: un ensemble d'étudiants. Chacun donne la liste de ceux avec qui il s'entend.
  Sortie: oui si on peut les répartir en trois équipes telles que les membres d'une même équipe s'entendent; non sinon.
- Pb2
  Donnée: un graphe.
  Sortie: oui si on peut le colorier les sommets avec 3 couleurs de telle sorte que deux extrémités d'un arc ne soient pas de la même couleur; non, sinon.

Réduction polynomiale = transformation d'un problème dans un autre.

Une réduction polynomiale du problème pb1 d'équipes dans le problème pb2 de 3-coloriage de graphes sera une transformation:

- qui à toute instance i du pb1 d'équipes, associe une instance $red(i)$ du pb2 de coloriage de graphes
- telle que $red(i)$ a une solution Ssi l'instance i avait une solution.
- cette transformation est polynomiale, i.e. calculable par un algorithme polynomial.

Remarque: on en déduit donc que la taille de $red(i)$ est bornée polynomialement par rapport à celle de i.

Réduction polynomiale = transformation d'un problème dans un autre.

Une réduction polynomiale du problème pb1 d'équipes dans le problème pb2 de 3-coloriage de graphes:

- la réduction: construction du graphe à partir d'une instance d'un problème d'équipes:
  - À chaque étudiant on associe un sommet.
  - Il y a un arc entre deux sommets si les étudiants correspondants ne s'entendent pas.
- la réduction est correcte
  - Si l'instance du problème d'équipes a une solution, en coloriant chaque sommet par le numéro d'équipe, on obtient un 3-coloriage correct.
  - Si le graphe construit a un 3-coloriage correct, la répartition par "équipes-couleurs" est correcte.
- la construction est polynomiale de l'ordre de n^2, alors que la taille de l'instance intiiale est au moins n.

Clique et Indépendant

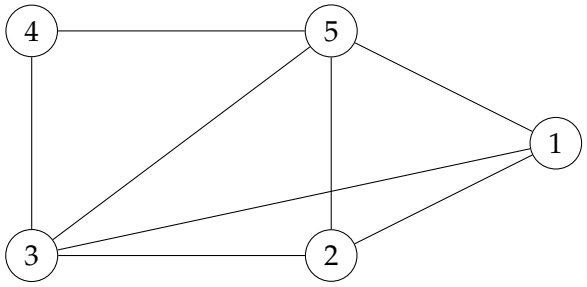
Clique

```
Donnée:
G=(S,A) –un graphe non orienté
k – un entier
Sortie:
Oui, Ssi G contient une clique de cardinal k.
```

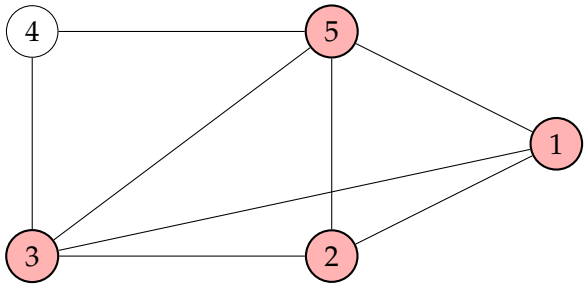
Indépendant

```
Donnée:
G=(S,A) –un graphe non orienté
k – un entier
Sortie:
Oui, Ssi G contient un ensemble indépendant
de cardinal k.
```


Clique

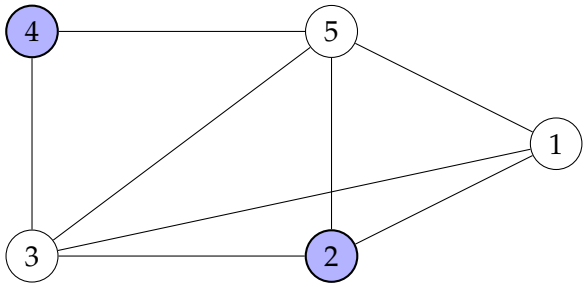


Clique



{1,2,3,5} est une clique: toute paire de sommets est reliée

Indépendant



{2,4} est indépendant: aucune paire de sommets n'est reliée

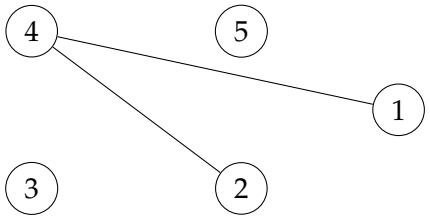
EXEMPLE: SUITE...

On peut choisir comme réduction de `Indépendant` dans `Clique` l'application *red* qui à $(G = (S, A), k)$ associe $(G' = (S, A'), k)$ avec $A' = \{(x, y)/(x, y) \notin A \text{ et } (y, x) \notin A\}$.

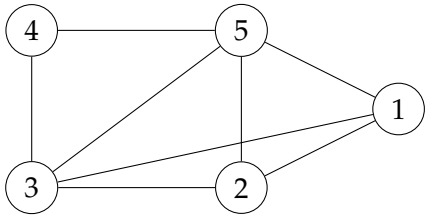
Montrer que c'est bien une réduction polynomiale de `Indépendant` dans `Clique` consiste en:

- Montrer que c'est correct i.e., pour tout (G, k), *Indpendant*(G, k) Ssi *Clique*(G', k)
- Montrer que la transformation est polynomiale.

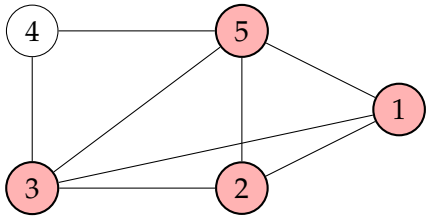
INDÉPENDANT VERS CLIQUE



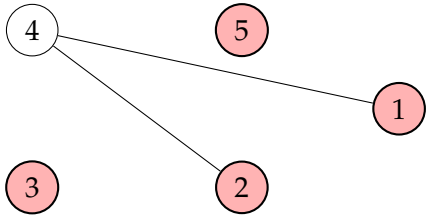
Est transformé en:



INDÉPENDANT VERS CLIQUE



Est transformé en:



EXEMPLE: SUITE...

red transforme l'instance d'`Indépendant` $(G = (S, A), k)$, en l'instance de `Clique` $(G' = (S, A'), k)$ avec $A' = \{(x, y)/(x, y) \notin A \text{ et } (x, y) \notin A\}$.

- Correct? i.e., pour tout (G, k), *Indpendant*(G, k) Ssi *Clique*(G', k)?
  - Si G a un ensemble indépendant C de cardinal k, C est une clique de cardinal k de G'.
  - réciproquement: si G' a une clique C de cardinal k, C est un ensemble indépendant de cardinal k de G.
- La transformation est polynomiale: l'algorithme qui calcule *red*(G) est bien polynomial (en $O(card(S)^2)$).

En résumé

Pour prouver qu'une propriété $P1$ se réduit polynomialement en une autre $P2$, il faut:

- proposer une réduction, i.e. un algo *red* qui à une instance I de $P1$, associe une instance $red(I)$ de $P2$.
- prouver qu'elle est correcte, i.e. qu'une instance I de P1 est positive **Si et Seulement Si** $red(I)$ est une instance positive de $P2$.
- prouver qu'elle est polynomiale.

Pourquoi prouver qu'un problème se réduit en un autre?

Proposition: Si L' est dans P et si $L \leq_P L'$, alors L est dans P.

$u \in L$ Ssi $red(u) \in L'$: donc, pour décider de l'appartenance à L, il suffit d'appliquer la transformation, et de décider de l'appartenance du transformé à L':
Si $AppL'$ est un algo polynomial pour tester l'appartenance à L',

`boolean AppL(u) {return AppL'(red(u));}` est un algo polynomial pour tester l'appartenance à L'.

Si L' est P, L est P.

si L n'est pas P, L' n'est pas P.

Réductions en chaîne

Propriété: La relation $\leq_P$ est transitive: si $L \leq_P L'$ et $L' \leq_P L''$, alors $L \leq_P L"$.

Si $red1$ est une réduction polynomiale de L dans L', et $red2$ est une réduction polynomiale de L' vers $L"$, alors $red2 \circ red1$ est une réduction polynomiale de L vers $L"$.

Les propriétés NP-dures, NP-complètes

Propriété NP-dure

Une propriété R est dite **NP-dure** Si toute propriété NP se réduit polynomialement en R.

Donc, d'après ce qui précède, si une propriété $NP-dure$ était P, on aurait $NP = P$: en effet toute propriété NP se réduirait polynomialement en une propriété P.

Une propriété $NP-dure$ contient donc d'une certaine façon toute la difficulté de NP!

Les propriétés NP-dures, NP-complètes

Propriété NP-complète

Une propriété R est dite **NP-complète** Si elle est NP et $NP-$dure.

Une propriété $NP-$complète est donc une propriété NP qui contient d'une certaine façon toute la difficulté de NP.

La découverte de Cook

Stephen Cook fut le premier à découvrir en l'existence de propriétés NP-dures.

Théorème de Cook - 1971

$3-CNF-SAT$ est $NP-$complète.

$3-CNF-SAT$ (ou $3-SAT$) est le problème de la satisfiabilité d'une formule sous forme conjonctive où chaque clause contient 3 littéraux.

$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$

Comment montrer qu'une propriété est NP-dure?

Pour montrer que R est NP-dure:

- On peut montrer "directement" que toute propriété NP se réduit polynomialement en R.
- Il suffit de montrer qu'une propriété connue $NP-$dure se réduit polynomialement en R.
  En effet, comme $\leq_P$ est transitif, on a:

  Proposition: si $L \leq_P L'$ et L est $NP-$dure, alors L' est $NP-$dure.

Pourquoi montrer qu'une propriété est $NP-$dure?

D'après la proposition précédente, si une propriété $NP-$dure se révélait être P, on aurait $P = NP$, ce qui est peu probable et en tout cas a résisté aux efforts de nombreux chercheurs!

Montrer qu'une propriété est $NP-$dure justifie raisonnablement qu'on n'ait pas trouvé d'algorithme polynomial pour décider de cette propriété!

NP VERSUS *NP*−DURE

Par contre, montrer qu'une propriété est NP, c'est montrer qu'elle n'est pas "si dure" que ça même si elle n'est peut-être pas P...

EXEMPLE: INDÉPENDANT EST *NP*−DUR?

`Indépendant`

```
Entrée:
G=(S,A) --un graphe non orienté
k       -- un entier

Sortie:
Oui, Ssi G contient un ensemble indépendant
de cardinal k.
```


`Indépendant` *NP*−DUR?

$3-CNF-SAT$ est $NP-$dur.

On va réduire polynomialement $3-CNF-SAT$ dans `Indépendant`.

$3-CNF-SAT$
Donnée: $\Phi = C_1 \wedge C_2 ... \wedge C_p$ avec $C_j = l_j^1 \vee l_j^2 \vee l_j^3$

Sortie: Oui, si il existe une valuation v telle que $v(\Phi) = Vrai$.

LA RÉDUCTION DE 3-CNF-SAT EN INDÉPENDANT

A une formule sous forme 3-CNF Φ, on va associer un graphe G_Φ et un entier p tel que la formule Φ est satisfiable Ssi le graphe G_Φ a un ensemble indépendant de cardinal p.

Construction du graphe:

- les sommets = les littéraux des clauses (donc 3p sommets)
- les arcs: il y a un un arc
  - entre deux littéraux de la même clause
  - entre deux littéraux incompatibles de clauses différentes, i.e. l'un est de la forme x et l'autre $\neg x$.
- p: le nombre de clauses de Φ

La réduction est correcte?

G_Φ a un ensemble indépendant de cardinal p Ssi Φ est satisfiable?
Preuve:

- Φ satisfiable: $\exists v$ telle que $v(\Phi) = Vrai$.
  Donc, pour chaque j il existe (au moins un, qu'on choisit) $l_j^{i_j}$ tel que $v(l_j^{i_j}) = Vrai$. Les $l_j^{i_j}$ forment un ensemble indépendant de cardinal p de G_Φ par définition des arcs.
- Dans l'autre sens: supposons que G_Φ ait un ensemble indépendant I de cardinal p.
  Posons $v(x) = vrai$ si il y a un x dans I, $v(x) = faux$ si il y a un $\neg x$, $v(x) = Vrai$ (ou $Faux$) si il n'y a ni x ni $\neg x$.
  v est bien définie: on ne peut avoir à la fois x et $\neg x$ dans I.
  Par construction, il y a un sommet par clause dans I: v valide au moins un littéral par clause; $v(\Phi) = Vrai$, Φ est bien satisfiable.

La réduction est bien polynomiale?

- taille de la formule: 3*p
- taille du graphe: 3*p sommets, au plus de l'ordre de $(3p)^2$ arcs

La réduction est bien polynomiale.

Bilan

- On a bien réduit polynomialement $3-CNF-SAT$ dans `Indépendant` : $3-CNF-SAT \leq_P$ `Indépendant`
- $3-CNF-SAT$ est $NP-$dur.
- donc `Indépendant` est $NP-$ dur
- `Indépendant` est même $NP-$ complet car NP + $NP-$dure.

Remarque: On avait prouvé que `Indépendant` se réduit en *Clique* , donc *Clique* aussi est $NP-$dur!

Question? A-t-on `Indépendant` qui se réduit polynomialement dans $3-CNF-SAT$?

Rép: oui, car `Indépendant` est NP et $3-CNF-SAT$ est $NP-$dure!

Problèmes NP-durs/Propriétés

Ce qui précède ne s'applique qu'aux propriétés.

Parler de problèmes NP, si ce ne sont pas des problèmes de décision, n'a guère de sens;

Par contre on peut parler de problèmes $NP-$durs:

Problème NP-dur

Si l'existence d'un algorithme polynomial pour le problème R implique $P = NP$, on dit que R est $NP-$dur.

Problèmes NP-durs et optimisation

A tout pb d'optimisation de type "trouver une solution de coût minimal (resp. de gain maximal)", on peut associer une propriété:

"existe-t-il une solution de coût inférieur (resp. de gain supérieur) à une valeur donnée en entrée?"

Si le pb de décision est $NP-$dur, le pb d'optimisation le sera aussi.

Si on avait un algorithme P pour le pb d'optimisation, on en aurait un pour celui de décision donc pour toute propriété NP, puisqu'il est $NP-$dur!

Problèmes NP-durs: l'exemple du coloriage de graphes

Le "vrai" problème du coloriage: pour un graphe, trouver un coloriage correct utilisant le moins de couleurs possibles.

Un problème intermédiaire: pour un graphe, trouver le nombre minimal de couleurs nécessaire pour un coloriage correct.

La propriété associée du $k-$coloriage: étant donné un graphe et un entier k, existe-t-il un coloriage correct de G avec k couleurs?

Exemple (suite)

Si il existait un algorithme polynomial pour colorier un graphe avec le moins de couleurs possible, on aurait un algorithme polynomial pour déterminer le nombre de couleurs optimal: on applique l'algorithme qui nous donne un coloriage optimal, et on retourne le nombre de couleurs.

Si il existait un algorithme polynomial pour trouver le nombre minimal de couleurs nécessaire pour un coloriage correct, on aurait un algorithme polynomial pour la propriété du $k-$coloriage: on applique l'algorithme qui nous donne le nombre minimal de couleurs, on vérifie si il est inférieur ou égal à k.

Exemple (suite)

Si il existait un algorithme polynomial pour trouver le nombre minimal de couleurs nécessaire pour un coloriage correct, il existerait un algorithme polynomial pour une propriété $NP-$dure et donc on aurait $P = NP$.

Si il existait un algorithme polynomial pour colorier un graphe avec le moins de couleurs possible, on aurait un algorithme polynomial pour une propriété $NP-$dure et donc $P = NP$.

Le problème de coloriage de graphe est $NP-$dur.

La propriété contient souvent toute la difficulté du problème général

Si il existait un algorithme polynomial pour la propriété du $k-coloriage$, comment en déduire un algorithme polynomial pour trouver le nombre minimal de couleurs pour un coloriage correct?

```
pour nc de 1 à nombre de sommets
  si on peut colorier G avec nc couleurs
  alors return nc
```

L'algorithme est polynomial, si on peut tester en temps polynomial si on peut colorier G avec nc couleurs: sa complexité est au plus *nsommets* fois celle du test de la propriété.
On peut utiliser la dichotomie sur le nombre de couleurs pour avoir une complexité en $(\log nsommets)$ fois celle du test de la propriété.

La classe NP La problématique Les réductions polynomiales Les propriétés NP-dures

La propriété contient souvent toute la difficulté du problème général

Si il existait un algorithme polynomial pour la propriété du $k-coloriage$, comment en déduire un algorithme polynomial pour le coloriage du graphe en un nombre minimal de couleurs?

La classe NP La problématique Les réductions polynomiales Les propriétés NP-dures

La propriété contient souvent toute la difficulté du problème général

"Idée"
On détermine d'abord k le nombre de couleurs optimal comme on vient de faire.

On rajoute au graphe k noeuds correspondant aux k couleurs, tous reliés entre eux.

A chaque étape, on "colorie" un noeud du graphe initial: pour une couleur donnée, si en le connectant aux $k-1$ autres noeuds "couleurs" un k-coloriage est possible, on colorie dans cette couleur. On teste toutes les couleurs en utilisant l'agorithme de décision pour la propriété jusqu'à en trouver une correcte.

La classe NP La problématique Les réductions polynomiales Les propriétés NP-dures

```
/* k est le nombre de couleurs optimal */
pour s de 1 à nombre de sommets {
    /* chercher col  pour s */
    col=1; Trouve=False;
    tant que non Trouve  {
      relier s aux "couleurs"  sauf col
      si on peut colorier le graphe avec k couleurs
      alors Trouve=True
      sinon {enlever les arcs ajoutés; col++;}}}
```

Si l'algorithme de décision était polynomial, on obtiendrait un algorithme polynomial pour colorier le graphe.

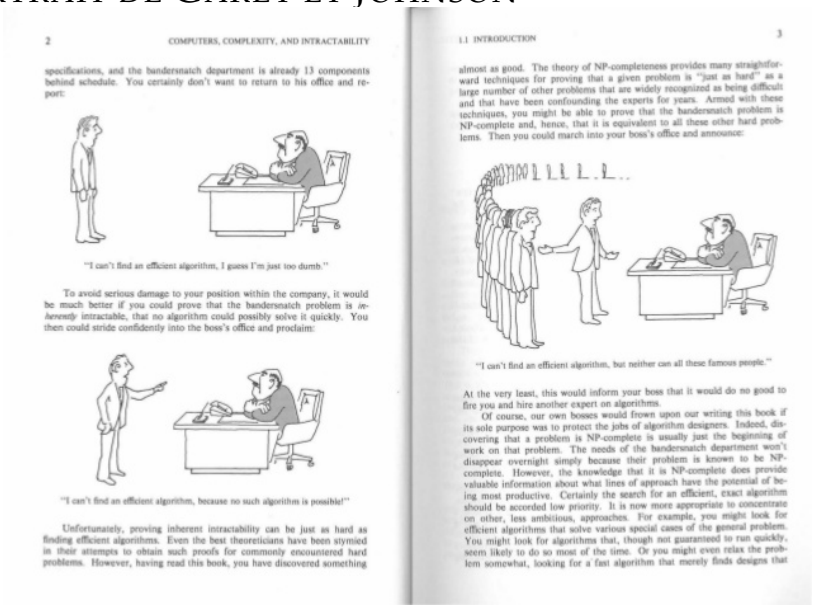
Le catalogue "Les propriétés *NP*−complètes"

Pour prouver qu'une propriété R est $NP-$dure, on peut en choisir une parmi les milliers de propriétés connues NP-dures et essayer de la réduire dans R: le problème est de bien la choisir!

Il existe un catalogue des propriétés $NP-$dures, le livre de Garey et Johnson:" Computers and Intractability: A guide to the theory of NP-completeness " qui contient beaucoup de problèmes mais date de 1979.

Il existe des sites Web qui essaient de tenir à jour le catalogue comme le compendium de problèmes d'optimisations *NP*: http://www.nada.kth.se/ viggo/problemlist/compendium.html.

Extrait de Garey et Johnson



2 COMPUTERS, COMPLEXITY, AND INTRACTABILITY

specifications, and the bandersnatch department is already 13 components behind schedule. You certainly don't want to return to his office and report:

"I can't find an efficient algorithm, I guess I'm just too dumb."

To avoid serious damage to your position within the company, it would be much better if you could prove that the bandersnatch problem is *inherently* intractable, that no algorithm could possibly solve it quickly. You then could stride confidently into the boss's office and proclaim:

"I can't find an efficient algorithm, because no such algorithm is possible!"

Unfortunately, proving inherent intractability can be just as hard as finding efficient algorithms. Even the best theoreticians have been stymied in their attempts to obtain such proofs for commonly encountered hard problems. However, having read this book, you have discovered something

1.1 INTRODUCTION 3

almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years. Armed with these techniques, you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:

"I can't find an efficient algorithm, but neither can all these famous people."

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.

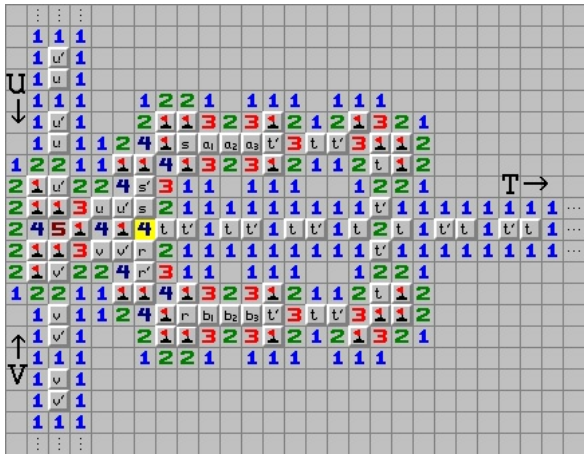
Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight simply because their problem is known to be NP-complete. However, the knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive. Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches. For example, you might look for efficient algorithms that solve various special cases of the general problem. You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time. Or you might even relax the problem somewhat, looking for a fast algorithm that merely finds designs that

Un extrait du catalogue des *NP*−complètes

- 3-CNF-SAT
- 3-coloriage de graphes
- existence d'une clique de taille k dans un graphe.
- le pb du BinPacking
- Le pb du sac à dos (non fractionnable).
- le pb de l'existence d'un circuit hamiltonien.
- Le pb du voyageur de commerce.
- la programmation linéaire en entiers.
- Le pb du recouvrement d'ensembles.
- Le pb du démineur ...

Le démineur

Minesweeper is NP-complete! by Richard Kaye 2000



Classic Nintendo Games are (NP-)Hard

by Greg Aloupis, Erik D. Demaine, Alan Guo, March 9, 2012

We prove NP-hardness results for five of Nintendo's largest video game franchises: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokemon. Our results apply to Super Mario Bros. 1, 3, Lost Levels, and Super Mario World; Donkey Kong Country 13; all Legend of Zelda games except Zelda II: The Adventure of Link; all Metroid games; and all Pokemon role-playing games.

Candy-Crush is (NP-)Hard

Toby Walsh(2014): We prove that playing Candy Crush to achieve a given score in a fixed number of swaps is NP-hard.

Luciano Gualà, Stefano Leucci, Emanuele Natale: Bejeweled, Candy Crush and other Match-Three Games are (NP-)Hard
Pour en savoir plus: `http://candycrush.isnphard.com/`

Perl Regular Expression Matching is NP-Hard

by M-J. Dominus

Perl takes exponential time (in the length of the string to be matched) to check whether or not a string matches certain regex.
One is tempted to wonder whether this difficulty is inherent, or whether there might be a clever way of speeding up the matching algorithm. The answer is that there is probably no clever way to speed up the algorithm, and that the exponential running time of the matching algorithm is probably unavoidable. We show that regex matching is NP-hard when regexes are allowed to have backreferences.

Attention aux faux frères!

L'un est facile (P), l'autre sans doute difficile (NP-dur)!

Le 2-coloriage versus le 3-coloriage??? Le 3-coloriage d'un graphe planaire versus le 4-coloriage d'un graphe planaire??? le pb du plus court chemin versus le pb du plus long chemin sans cycle??? 2-SAT versus 3-SAT??? La programmation linéaire en entiers versus la programmation linéaire en réels???

P	NP-dur
2-coloriage	3-coloriage
4-coloriage planaire	3-coloriage planaire
Chemin le plus court	Chemin le plus long
2-SAT	3-SAT
Programmation linéaire en réels	Programmation linéaire en entiers

COMPLEXITÉ DES PROBLÈMES: BILAN

- *NP* : "easy to check"
- $NP - dur$: contient toute la complexité de *NP*, "aussi dur" que n'importe quel *NP*.
- pour montrer que *R* est $NP-$dur: on cherche à réduire polynomialement un pb connu $NP-$ dur dans *R*.

LA PRÉSENTATION PAR L'INSTITUT CLAY

Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

LA PRÉSENTATION PAR L'INSTITUT CLAY

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair from taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe!