Complexité de Problèmes: les propriétés NP-dures

ACT

Sophie Tison
Université Lille
Master1 Informatique

COMPLEXITÉ DES PROBLÈMES

- Rappels
- Paradigmes d'algorithme
- Complexité des problèmes
  - Généralités
  - La classe P
  - La classe NP
    - Propriétés NP
    - Réductions polynomiales, Propriétés NP-dures
- Que faire face à un problème dur: algorithmique avancée

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.

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.

Si une propriété $NP-$dure se révélait être P, on aurait $P = NP$.

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

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...

Problèmes NP-durs/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: 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 serait polynomial, si on pouvait 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 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 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, le colorier va correspondre à le connecter aux k-1 noeuds des autres couleurs.
Si en le connectant aux $k-1$ autres noeuds "couleurs" un k-coloriage est possible, on le colorie dans cette couleur. On teste toutes les couleurs en utilisant l'algorithme de décision pour la propriété jusqu'à en trouver une correcte.

```
/* 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.

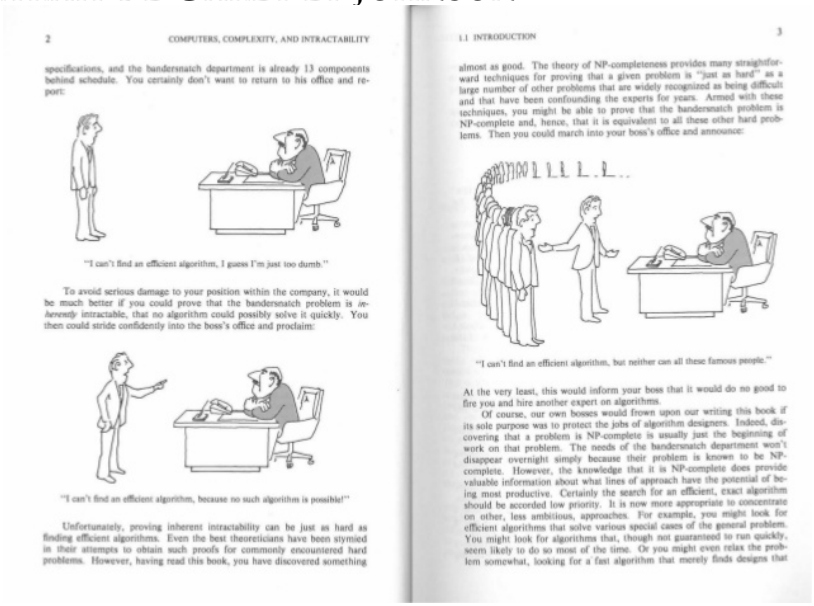
LE CATALOGUE "LES PROPRIÉTÉS *NP*−COMPLÈTES"

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.

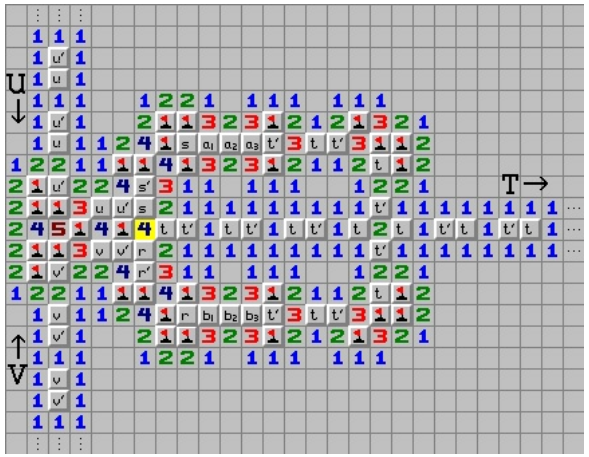
Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight 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 DÉMINEUR

MINESWEEPER IS NP-COMPLETE! BY RICHARD KAYE 2000



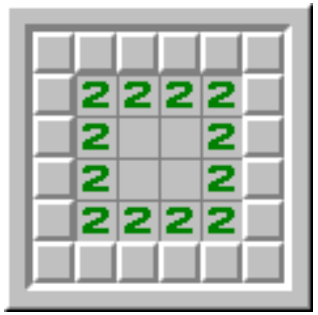
MINESWEEPER IS NP-COMPLETE?

Richard Kaye a montré que le problème suivant est NP-complet:
Donnée: une configuration de taille quelconque partielle: chaque case contient une mine, un nombre de mines voisines (de 0 à 8) ou aucune information
Sortie: Oui, si un placement des mines peut correspondre à cette configuration. Non sinon.

REMARQUES

- la taille est quelconque: généralisation du jeu
- C'est un problème de décision: restriction à une question concernant le jeu
- Si on a un algorithme rapide pour le problème, on peut jouer rapidement sans risque: on part d'une configuration, on se pose la question d'une mine dans un carré: si on peut décider que c'est impossible, on peut dévoiler le carré.
- C'est bien NP: un certificat=un placement des mines

EXEMPLE (DE R. KAYE)



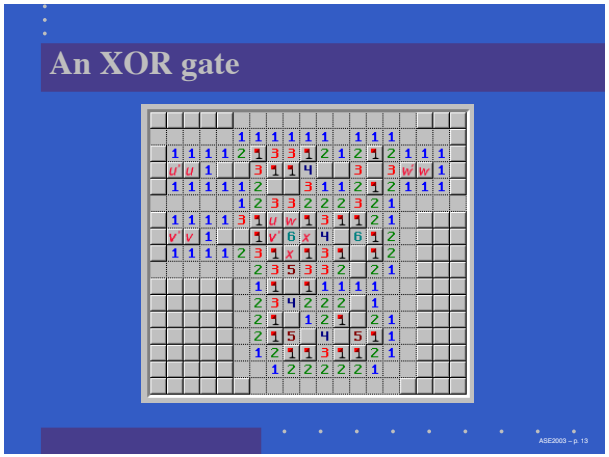
Y a-t-il un placement possible des mines? Oui

Idée de la preuve de R. Kaye

R. Kaye a réduit polynomialement Sat dans le démineur: il code une formule de SAT en une configuration du démineur qui soit "possible" Ssi la formule est satisfiable.

La preuve est technique. Pour en savoir plus: Richard Kaye a un site dédié au démineur:
http://web.mat.bham.ac.uk/R.W.Kaye/minesw/

Un exemple de "widget" (R. Kaye)



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/`

Complexité de problèmes, jeux et puzzles

- Intuitivement un jeu est intéressant si il est (au moins un peu) dur. La complexité de problème (computational complexité) permet d'une certaine façon de mesurer cette difficulté.
- Bien sûr, les jeux qu'on considère ont souvent un nombre de configurations borné (taille de l'échiquier, ...). Il faut les généraliser pour appliquer la théorie de la complexité.
- Pour certains experts, NP-complet est le bon niveau pour les puzzles ou jeux à un joueur: ni trop facile (NP-dur), ni trop dur (NP).
- Voir par exemple, la page de David Eppstein qui classifie quelques jeux à 1 ou 2 joueurs selon leur complexité:

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*.
- à un problème d'optimisation on peut associer en général un problème de décision: si ce dernier est $NP-$ dur, le premier l'est aussi.

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!

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.

Complexité des problèmes

Que faire face à un problème *NP*-dur?

Il paraît raisonnable d'abandonner l'idée d'avoir un algorithme polynomial qui donne à coup sûr la solution (optimale), à moins de prétendre prouver que $P = NP$.
On pourra alors avoir deux attitudes:

- abandonner l'idée d'avoir un algorithme polynomial
- abandonner l'idée d'avoir un algorithme qui donne toujours la solution exacte!

ABANDONNER L'IDÉE D'AVOIR UN ALGORITHME POLYNOMIAL...

On peut essayer d'obtenir un algorithme exact non polynomial "optimisé" et ne l'appliquer que sur des données relativement petites.

ABANDONNER L'IDÉE D'AVOIR UN ALGORITHME POLYNOMIAL...

... ET UN PSEUDO-POLYNOMIAL?

Un algorithme pseudo-polynomialest un algorithme qui est polynomial si les entiers de la donnée sont codés en base 1: par exemple, si la donnée est un entier n, un algorithme en $\Theta(n)$ n'est pas polynomial mais il est pseudo-polynomial.

On dit qu'un problème est faiblement *NP*-dur si il est *NP*−dur et qu'il existe un algo pseudo-polynomial pour le résoudre.

Par exemple, par la programmation dynamique, il y a un algorithme pseudo-polynomial pour le problème du sac à dos: celui-ci est donc faiblement *NP*-dur.

Les pseudo-polynomiaux ne sont pas la panacée: ils restent coûteux et pour beaucoup de problèmes *NP*−durs, on n'en connait pas.

ENUMÉRATION DES SOLUTIONS

- Comme en TP, avec la méthode du British Museum: on énumère des candidats solutions.
- Alternative: une solution peut souvent être construite de façon incrémentale: on peut souvent représenter l'espace des solutions comme un arbre "de choix". Par exemple, pour le TP, une solution partielle est un début de tournée.

BACKTRACKING ET ARBRE DES SOLUTIONS

VERSION RÉCURSIVE POUR LA DÉCISION

```
//explore le sous-arbre de racine sol
Explorer(Solution sol){
 si  sol est faisable{
   si solution est complete
     return Vrai;
   sinon
     pour chaque s  successeur de sol
                              explorer(s);}
 return Faux;}
```

BACKTRACKING ET ARBRE DES SOLUTIONS

VERSION RÉCURSIVE POUR L'OPTIMISATION

```
//explore le sous-arbre de racine sol
Explorer(Solution sol){
 si  sol est faisable{
   si solution est complete
      si elle est meilleure que meilleure trouvée
             mettre à jour la meilleure trouvée;}
   sinon
     pour chaque s  successeur de sol
                          explorer(s);}}
 retourner meilleure trouvée;
```


BACKTRACKING ET ARBRE DES SOLUTIONS

EXEMPLE POUR CIBLE

Une solution partielle est donc par exemple un choix O/N pour les valeurs $x_1, ..., x_i$.

- tester si elle est faisable : si la somme des valeurs prises est $\leq$ cible.
- tester si elle est complète si la somme des valeurs prises est la cible.
- évaluer la solution pour les problèmes d'optimisation: évaluer la somme des valeurs prises.
- énumérer la liste de ses successeurs : simplement on a deux successeurs: prendre ou non x_{i+1}.

BACKTRACKING ET ARBRE DES SOLUTIONS

VERSION ITÉRATIVE POUR OPTIMISATION

```
//explore le sous-arbre de racine sol
Explorer(Solution sol)
 empiler(sol);
 tant que pile non vide  {
   depiler(solution);
   si solution est complete  {
       si elle est meilleure que meilleure trouvée
             mettre à jour la meilleure trouvée;}
   sinon pour chaque s  successeur faisable  de sol
                        empiler(s);
}
```


BACKTRACKING ET ARBRE DES SOLUTIONS

COMMENT AMÉLIORER?

- On peut améliorer: si on peut dire que toute solution qui complète l'actuelle est moins bonne que la meilleure trouvée, on élague.
- Dans l'autre sens, on pourrait explorer en premier les choix qui paraissent le plus prometteurs.

C'est ce qu'on appelle le "Branch and Bound" ou "Séparation et Evaluation".

SÉPARATION ET EVALUATION

La méthode Branch and Bound ou Séparation-Evaluation est basée sur:

- Séparation (Branch): à partir d'un noeud, on explore les différentes branches
- Evaluation (Bound): on évalue l'intérêt des branches.
  - On essaie de trouver une borne inf: toute solution issue de ce noeud aura un coût au moins de MIN et une borne sup- toute solution issue de ce noeud aura un coût au plus de MAX-
  - Si la borne inférieure d 'une région A est supérieure à la meilleur solution déjà trouvée ou à la borne supérieure d'une région B, je peux éviter d'aller explorer A: c'est l'élagage ou le pruning.
  - On explore en premier les branches les plus prometteuses: on peut remplacer la pile de tout à l'heure par une file de priorité par exemple basée sur le MIN associé à une solution.

QUE FAIRE FACE À UN PROBLÈME *NP*-DUR?

- s'acharner à chercher avoir un algorithme polynomial en espérant montrer $P = NP$ ...
- abandonner l'idée d'avoir un algorithme polynomial
  - pour certains problèmes NP-durs, il existe des algorithmes pseudo-polynomiaux.
  - utiliser la spécificité des données; on peut parfois concevoir des algorithmes "efficaces à paramètres fixés" (algorithmes FPT, fixed-parameter tractable)...
  - utiliser des méthodes "améliorées" d'énumération des solutions: exploiter la symétrie des solutions, élaguer l'arbre des solutions (pruning), prioriser les branches les plus prometteuses - cf branch and bound
  - utiliser les solveurs existants, par exemple des solveurs SAT ou de programmation linéaire en entiers.
- abandonner l'idée d'avoir un algorithme qui donne toujours la solution optimale