NP-Complete Problems

With a short and informal introduction to Computability and Complexity

Álvaro Moreira
alvaro.moreira@inf.ufrgs.br

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil
http://www.inf.ufrgs.br

Contents

Contents

Algorithmics - the science of algorithms

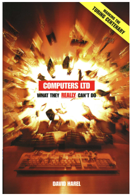
Putting this 3 lectures in the context of a course on Algorithms we have:

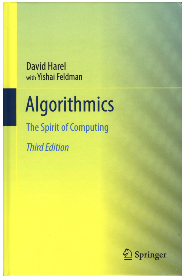
- The good news:
  - Algorithms and data (getting it done)
  - Algorithmic methods (getting it done methodically)
  - The correctness of algorithms (getting it done right)
  - The efficiency of algorithms (getting it done cheaply)
- The bad news:
  - Intractability (you can't always get it done cheaply)
  - Noncomputability (sometimes it can' t be done at all!)

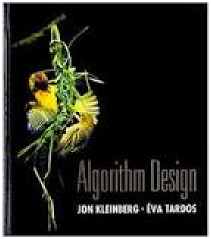
Contents

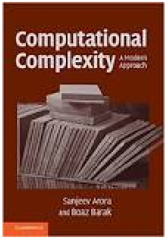
Bibliographic References

These slides are strongly based on the first









More suggestions of reading material will be given along the slides!

Biblografical References II

- **Computers Ltd.: What They Really Can't Do**. David Harel. Oxford University Press.
- **Algorithmics: The Spirit of Computing.** David Harel and Yishai Feldman. Springer
- **Algorithm Design**. By Jon Kleinberg and Eva Tardos. Pearson
- **Computational Complexity: A Modern Approach**. Sanjeev Arora and Boaz Barak. Cambridge University Press.

Topics to be covered

1. Computability (1st lecture)
    - Kinds of problem we are interested in
    - Example of non-computable problems
    - Nothing interesting about computation is computable
    - Church-Turing Thesis
2. Complexity (2nd and 3rd lectures)
    - Lower and upper bounds of problems
    - Tractability x Untractability
    - Problems with tractability status unkown
    - NP-Complete problems
    - P=NP?

Contents

Bad news in computing...

- These 3 lectures are about the **inherent limitations of computing**, such as . . .
    - . . . the **impossibility of solving a problem** with a computer or,
    - . . . the **impossibility of solving a problem efficiently**
- We concentrates on proven, lasting and robust limitations
- And by ”proven" we mean .....

mathematically proven!!

Why to study these limitations?

Why one would care about care about studying or doing research (or even get informed) on the inherent limitations of computing ?

- To satisfy intellectual curiosity
- To discourage futile efforts
- To encourage development of new paradigms
- To make possible the otherwise impossible

Discourage of futile efforts

- If a computational problem has been proved to **admit no solution** then seeking a solution is pointless
- The same goes for computational problems that **do admit solution**, but have been proved to require:
  - Unreasonable amount of space (say, much larger than the entire known universe!!), or that take
  - Unreasonable amount of time (say, a lot more than has elapsed since the Big Bang!!)

Rules of the game I

- We concentrate only on precisely defined computational problems
- We won't focus on problems such as run companies, carry out medical diagnosis, compose music, find a good match for boyfriend or girlfriend, etc...
- No one can say, for instance that he/she has developed an algorithm that **solves** the problem of running a company because "running a company" . . .

. . . is **not** a precisely defined computational problem!!

Rules of the game II

We require that a computational problem

1. be associated with a set of legal inputs
2. its solution should work for any input from the set of legal inputs
3. has an infinite set of legal inputs

(1) and (2) above are clear: we need to know the set of possible inputs to a problem, and the solution should work not only for some of these possible inputs but for all of them.

But what about requirement (3) ?

Rules of the game III

If the set of legal inputs of a problem is finite, them the problem always has a solution!

Example: A problem with a finite set $\{I_1, I_2, \ldots, I_K\}$ of legal inputs, that should answer only **yes** (if the input has some property) or **no** (if the input doesn't have some property).

The problem **has an algorithm** that "contains" a table with the K answers. The algorithm can be the following:

```
(1) if input is $I_1$ then output yes and stop;
(2) if input is $I_2$ then output yes and stop;
    ⋮
(k) if input is $I_K$ then output yes and stop
```

Rules of the game IV

We might not know yet which of the 2^{k} possible algorithms is the correct one. But it certainly exists.

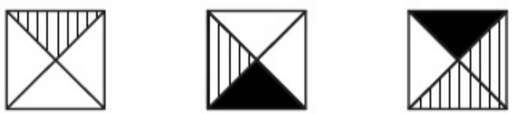
Hence, a problem is interesting for the purposes of investigations on computability only if it has an infinite set of legal inputs

And finally, is very common in the context of computability (and also of complexity) to focus on **decision problem**, i.e, problems that output only **yes** or **no** (or **true** or **false**)

Contents

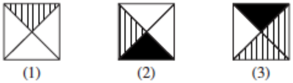
The Tilling Problem I

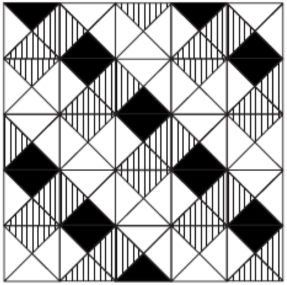
- The problem involves covering large areas using square tiles with colored edges, such that adjacent edges are monochromatic.
- A tile is a 1 by 1 square, divided into four by the two diagonals, each quarter colored with some color.



- We assume that the tiles have fixed orientation and cannot be rotated, and that an unlimited number of tiles of each type is available

The Tilling Problem II



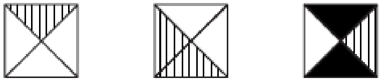


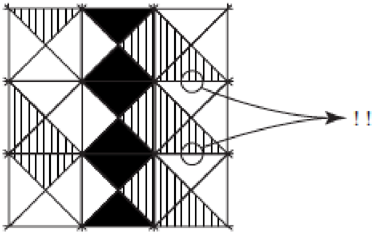
The algorithmic problem

- INPUT: a finite set T of tile descriptions, and
- OUTPUT: “yes" if **any** finite area, of any size, can be covered using only tiles of types in T, such that the colors on any two touching edges are the same. And “no" otherwise.

Given these 3 kinds of tiles we can easily check that it is possible to cover rooms of any size.

The Tilling Problem III





If we exchange the bottom colors of tiles (2) and (3) we can see quite easily that even very small areas cannot be tiled at all.

The Tilling Problem IV

R.Berger: Undecidability of the Domino Problem. Memoirs of the American Mathematical Society 66, 72 pp., 1966

Inputs

Each *tile type* in T is a sequence $t = (n, e, s, w)$ of four symbols, that identify the colors at the top, right, bottom and left edges of the tile.

A problem input $(T, t_{(0,0)})$ is a finite set T of tiles types along with a specification of a distinguished corner tile type $t_{(0,0)} \in T$.

The Tilling Problem V

Requirements

A *tiling* is a function $f : \mathbb{N} \times \mathbb{N} \to T$ that tells which tile type is associated to each square on the infinite quarter-plane.

The requirement of consistency of colors can be written as a pair of conditions

$$f(i, j).1 = f(i, j + 1).3 \quad \text{and} \quad f(i, j).2 = f(i + 1, j).4 \quad \forall i, j \geqslant 0$$

The requirement that the corner tile type $t_{(0,0)}$ be placed in the corner is expressed formally as

$$f(0, 0) = t_{(0,0)}$$

The Tilling Problem is Undecidable

Definition (The Tilling Problem)

INPUT:

- $\langle T, t_{(0,0)} \rangle$

OUTPUT:

- **yes**, if there is a tiling function $f : N \times N \rightarrow T$ with $f(0,0) = t_{(0,0)}$ satisfying the 2 requirements above
- **no**, otherwise

The Word Correspondence Problem I

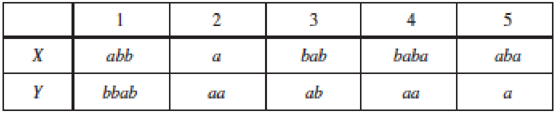
The Word Correspondence Problem, involves forming a word in two different ways.

The inputs for the problem are two groups of words over some finite alphabet. Call them the $\mathrm{X}s$ and the $\mathrm{Y}s$.

The problem asks whether it is possible to concatenate words from the X group, forming a new word, call it Z, so that concatenating the **corresponding** words from among the $\mathrm{Y}s$ forms the very same compound word Z.

The Word Correspondence Problem II

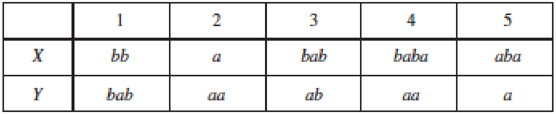
Fig. (a) has an example with 5 words in each group, where the answer is **yes**. Concatenating the words in the sequence 2, 1, 1, 4, 1, 5 from either the Xs or the Ys yields the same word, $aabbabbbabaabbaba$.



	1	2	3	4	5
X	*abb*	*a*	*bab*	*baba*	*aba*
Y	*bbab*	*aa*	*ab*	*aa*	*a*

(a) Admits a correspondence: 2, 1, 1, 4, 1, 5

But the input described in (b), which is obtained from (a) by removing the 1st letter from the 1st word of each group, does not admit any such choice. Its answer is therefore **no**.



	1	2	3	4	5
X	*bb*	*a*	*bab*	*baba*	*aba*
Y	*bab*	*aa*	*ab*	*aa*	*a*

(b) Admits no correspondence

Variations

The *Word Correspondence Problem* is **undecidable** !

The cause of its undecidability seems to be the fact that there is no bound on the size of sequence of words

But variants in which it seems that there are even more cases to check **are decidable**

For instance, a variant that imposes no restriction on the way choices are made from the $\mathrm{X}s$ and $\mathrm{Y}s$ - even the number of words selected need not be the same - is **decidable**

Problems that look very similar might have computability status very different (the same applies to tractability status)

The Halting Problem

The (Program) Halting Problem is undecidable

INPUT: program P and its input X

OUTPUT: Does P terminate when executed with input X? (yes/no?)

Program Verification

The Problem of Program Verification is undecidable

INPUT: specification ϕ about the input, and ψ about the output, both expressed in some logic, and a program P

OUTPUT: if the input is such that is has property given by ϕ, does the output satisfies property ψ after the execution of P? (yes/no?)

Nothing About Computing is Computable!

There is a remarkable result, called Rice's theorem, which shows that not only we cannot verify programs or determine their halting status, but ...

Nothing interesting about programs is computable!!

Interesting is **a property of what the program does** and **not of the particular form that solution takes**.

H. G. Rice, Classes of Recursively Enumerable Sets and Their Decision Problems, Trans. Amer. Math. Soc. 74 (1953), pp. 358–66.

Contents

How to prove that a problem is non-computable? I

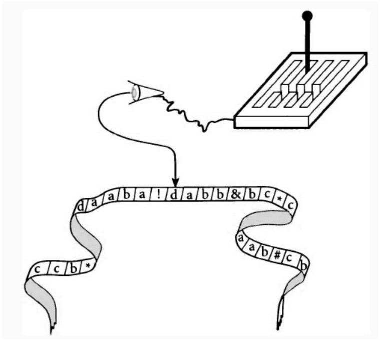
Investigation on (non-)computability started a long time ago in the context of Logics and Mathematics when there were no computers!

It was necessary, of course, to have an agreement of what *effectively computable* was and how to express a computation

Alan Turing, in 1936, proposed a **computer machine** and proved that no machine could be built to solve the Validity Problem for FOL (a decision problem)

Turing, A.M. On Computable Numbers, with an Application to the Entscheidungs problem. Proceedings of the London Mathematical Society. 2 (published 1937). 42: 230–265

How to prove that a problem is non-computable? II



The **Church-Turing Thesis** equates the intuitive notion of effectively computable with the formal notion of computable with a Turing Machine (or any other equivalent computational model)

Church-Turing Thesis

The Church-Turing Thesis **cannot be proved** (Why?)

The Thesis stands firm because:

- Any other computational model invented has been proved to be equivalent to all the others already invented
- So far the thesis has not been disproved (what would be necessary to prove it false?)

The Theory of Computability (and of Complexity) has been built around Turing Machines

Proving the Undecidability of the Halting Problem I

We have to show that it is **impossible to write a program in a given programming language L that solves the halting problem**

After we have proved that one might still think that result depends on the specific language L ...

....and that if we change the programming language we can eventually come up with a program to decide the halting problem

But since programming languages are equivalent to Turing Machines, **by the Church Turing Thesis we can conclude that the Halting problem is undecidable**

Reading

Chapter 2 - *Sometimes We Can't Do It* from the book *Computers Ltd - What they really can't do*, by David Harel

Chapter 8 - *Noncomputability and Undecidability* from the book *Algorithmics: the Spirit of Computing*, by David Harel

Solving the Unsolvable, by Moshe Y. Vardi. Communications of the ACM, Vol. 54 No. 7, Page 5. July 2011

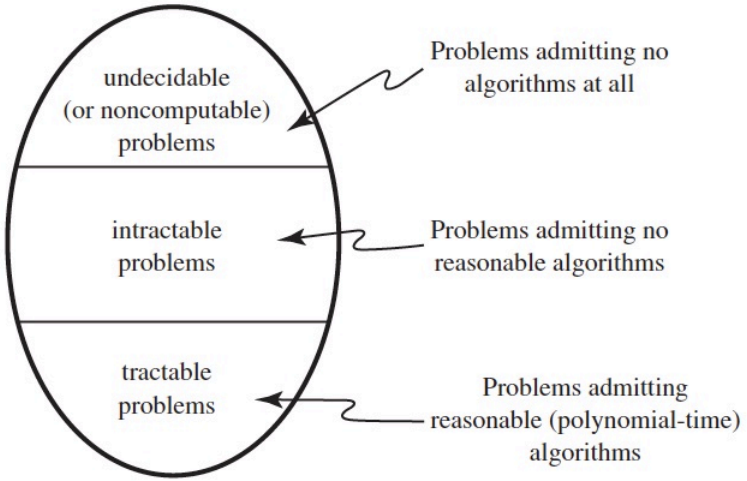
Solving the Unsolvable

From “Solving the Unsolvable", by Moshe Vardi:

> *I believe this **noteworthy progress in proving program termination** ought to force us to reconsider the meaning of unsolvability .... In theory, **unsolvabilty does impose a rigid barrier on computability, but it is less clear how significant this barrier is in practice** .... most real-life programs, if they terminate, do so for rather simple reasons, because programmers almost never conceive of very deep and sophisticated reasons for termination. Therefore, **it should not be shocking that a tool such as Terminator can prove termination for such programs**.*

Contents

Tractability x Intractability

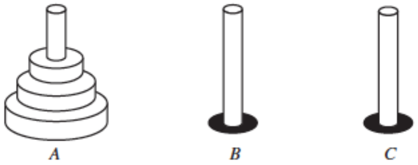

undecidable
(or noncomputable)
problems
intractable
problems
tractable
problems
Problems admitting no algorithms at all
Problems admitting no reasonable algorithms
Problems admitting reasonable (polynomial-time) algorithms

Hanoi Towers I

- Suppose we are given three towers A, B, and C.
- On the first tower, A, there are **three** rings in descending size order, while the others are empty
- We have to move the rings from A to B, using tower C in the process when necessary.
- Rings have to be moved one at a time, and a larger ring can never be placed on top of a smaller one.

Hanoi Towers II

- This puzzle with 3 rings can be solved as follows (with 7 moves):



move A to B;
move A to C;
move B to C;
move A to B;
move C to A;
move C to B;
move A to B.

- With 4 rings on tower A the problem can be solved with 15 move actions

Hanoi Towers III

- We are interested in an algorithm to solve the **general algorithmic problem** associated with the Towers of Hanoi:
    - The **input** is a positive integer N (the number of rings),
    - The **output** is a list of ***"move X to Y"*** actions, that solve the puzzle for N rings.
- There is an algorithm that solves the problem with $2^{\mathrm{N}} - 1$
- It has been **proved** that it is not possible to solve the problem with less than $2^{\mathrm{N}} - 1$ moves. So we **cannot do any better than this**

Hanoi Towers IV

- If we could move **a million rings per second**, with **64 rings** it would take **more than half a million years** to finish!!
- If we could move only **one ring every 10 seconds**, it would takes us more than **five trillion years to finish**.
- One may think the difficulty is because the output is a long sequence of moves and it would take too long to find and exhibit it
- To convince that this is not the case let's take a look at some **decision problems**

Decision Problems

- Optimization version of the Shortest Path problem: *Given data about direct buses between cities (a graph), and given the name of two cities u and v (nodes of the graph),* ***finds the most direct path (shortest in terms of bus changes) between them***

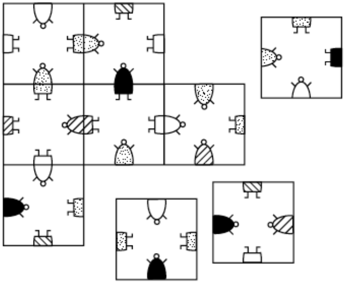
- Decision version of the Shortest Path problem: *Given data about direct buses between cities (a graph), given the name of two cities, u and v (nodes of the graph),* ***and a non-negative integer k, does a path exist between u and v whose length is at most k?***

- **If an optimization problem is easy, its related decision problem is also easy.**

- **If a decision problem is hard, its related optimization problem is hard too**

Monkeys Puzzle I

- Given (descriptions of) N cards, where N is some square number, say, N is M^2, the (original) problem asks for exhibiting, if possible, an M by M square arrangement of the N cards, s.t. touching parts match.



The cards have a fixed direction and they cannot be rotated.

Monkeys Puzzle II

- We concentrate on the decision version of the problem, without asking for one arrangement to be exhibited.
- **A naive algorithm proceeds trying all possible arrangements**
    - it stops with “yes" as soon as it gets a legal arrangement, and
    - it stops with “no" if all arrangements have been tried, and they are all illegal

OBS.: It is possible to be less *brute-force*, by not checking extensions of a partial arrangement that have already been shown to be illegal.

Monkeys Puzzle III

- With a 5x5 grid, there are 25 possibilities for the first card
- Having placed the first card, there are still 24 cards to choose from for the second location, 23 for the third, and so on
- The total number of arrangements to try, can therefore reach...

  $25 \times 24 \times 23 \times \ldots \times 3 \times 2 \times 1 = 25!$ (a 26 digits number!!!)

- How long will the algorithm take in the **worst case**, i.e., when there is no legal arrangement, so that all possible arrangements have to be checked?

Monkeys Puzzle IV

- Suppose a computer can try **a billion arrangements per second**. To try all 25! arrangements it will take **well over 490 million years**

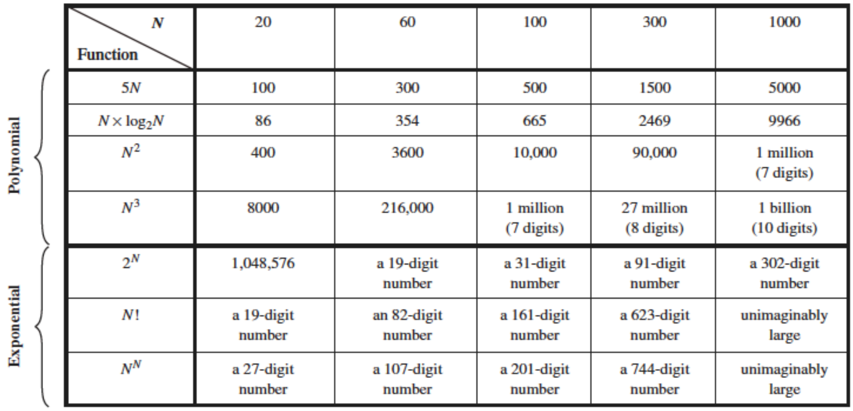
- In a 6x6, the time to try all 36! arrangements it will take **FAR longer than the time that has elapsed since the Big Bang!!**

- And note that, in this context, **the worst-case is the most probable** to happen if the puzzle is well-designed.

- Is there some better solution to the Monkey Puzzle problem practical for a reasonable number of cards?

Probably not, but no one knows for sure.

Function values



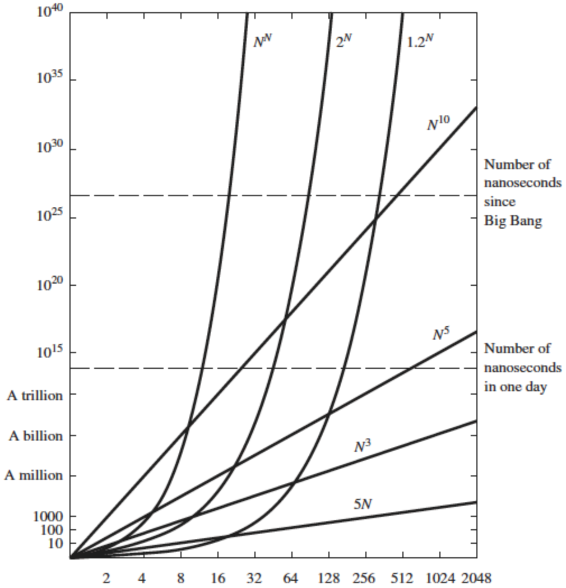
	N / Function	20	60	100	300	1000
Polynomial	$5N$	100	300	500	1500	5000
	$N \times \log_2 N$	86	354	665	2469	9966
	N^2	400	3600	10,000	90,000	1 million (7 digits)
	N^3	8000	216,000	1 million (7 digits)	27 million (8 digits)	1 billion (10 digits)
Exponential	2^N	1,048,576	a 19-digit number	a 31-digit number	a 91-digit number	a 302-digit number
	$N!$	a 19-digit number	an 82-digit number	a 161-digit number	a 623-digit number	unimaginably large
	N^N	a 27-digit number	a 107-digit number	a 201-digit number	a 744-digit number	unimaginably large

This table shows some numbers for some functions. As a reference:
- the number of (known) protons in the universe has 79 digits.
- the number of microseconds since the *Big Bang* has 24 digits.

Function growth I

- If N is 300 the number 2^N is **billions of times larger than the number of protons in the entire known universe!!**.
- N^N grows faster that $N!$ which grows faster than 2^N
- 2^N grows **MUCH** faster than any other functions of the form N^K, for any fixed K.
  - OK that for all N up to 1165, N^{1000} is larger than $N!$, but after that number, $N!$ grows much faster
- 2^N, $N!$, and N^N are all example of "bad" functions because they all grow **MUCH** faster than ("good") N^K functions.

Function growth II



Polynomial (good) x Exponential (bad) I

- These facts lead to a fundamental classification of functions into "good" and "bad" ones.
  - The good ones are the **polynomial** functions
  - The bad ones are the **super-polynomial** functions
- A **polynomial function of** N is any function which is **no greater** in value than N^K for all values of N, from some point on.
- A **super-polynomial function of** N is any function which **is greater** in value than N^K for all values of N, from some point on.

Polynomial (good) x Exponential (bad) II

- Logarithmic ($\log_2 N$), linear (N), and quadratic (N^2) functions, for example, are **polynomial**
- 1.001^N, 5^N, N^N, and $N!$, for instance, are **super-polynomial**

OBS.: To call super-polynomial functions as exponential is an abuse because:

- super-polynomials functions, like $N^{\log 2\ N}$ for example, are not quite exponential,
- and functions like N^N, for instance, are super-exponential

Tractable x Intractable problems

- An **algorithm** that has its time performance bounded from above by N^k, where N is the size of it input, is a **polynomial-time algorithm**
- An **algorithm** that, in the worst case, **requires** super-polynomial, or exponential time, is an **exponential-time algorithm**
- A **problem** is **tractable** if it has a polynomial-time solution
- A **problem** is **intractable** if it **requires** an exponential-time solution

Obs.: off course that an N^{1000} polynomial algorithm is worse than a $N!$ exponential algorithm for $N \leqslant 1165$. But, most of the polynomial-time algorithms have exponent of N that is no more than 5 or 6.

Contents

Lower x Upper Bounds I

- Any algorithmic *problem* has an **inherent *optimal* solution**.
- Suppose someone gives an $O(N^3)$ algorithm for problem X
    - We then know that the optimal solution of X cannot be worse than $O(N^3)$
- Later on, someone discovers a better algorithm, say one that is $O(N^2)$
    - We then know that the problem X cannot be inherently worse than $O(N_2)$ and the previous $O(N_3)$ algorithm becomes obsolete.

Lower x Upper Bounds II

- With better algorithms we get closer the inherent complexity of the problem.
- But is it possible to know **how far can improvements go?**
- Yes, that requires a **proof** of a **lower bound**.
- If, for instance, we **prove** that the problem X cannot be solved in less than O(N^2), then people can stop looking for better algorithms for it since it already has a O(N^2) algorithm.

Lower x Upper Bounds III

- With a **better algorithm** we show that the problem's inherent time performance is **no worse than some upper bound**
- With a **lower bound proof** we show that the problem's inherent time performance is **no better than some lower bound**
- When the **upper and lower bounds meet** (except for the possibly different constant factors) the **algorithmic problem is closed**. Otherwise we say that **there is an algorithmic gap**
- **If a problem is closed as tractable that is good news. If it is closed as intractable, that's bad news, but at least we know something for sure**

Examples I

Towers of Hanoi

- the best algorithm proposed is exponential - **upper bound is O(2^N)**
- we cannot do any better - **lower bound is also O(2^N)**
- the algorithmic problem is **closed**
- and the problem is classified as **intractable**

Monkey Puzzle

- current best algorithm is exponential - **upper bound is O($N!$)**
- current best-known (proved) **lower bound is O(N)**
- there is an **algorithm gap**
- even though the best known algorithm is exponential the **problem** cannot be classified as intractable

Examples II

Linear programming

- Input: a list of m linear inequalities with **rational** coefficients over n variables $x_1, \dots x_n$ (a linear inequality has the form $a_1x1 + a_2x_2 + \dots + a_nx_n \leqslant b$ for some coefficients $a1, \dots a_n, b$),
- Output: is there an assignment of rational numbers to the variables $x1, \dots x_n$ that satisfies all the inequalities?
- This problem is closed with a polynomial time and it is classified as a tractable problem

Examples III

Linear Programming is an important problem with several applications in real-life problems. It also has a very interesting history.

- For many years the best algorithm for it was an exponential-time procedure known as the **simplex method**
- However when the method was used for real problems, even of nontrivial size, it usually performed very well.
- In 1979, a polynomial-time algorithm was found, but the simplex method had a better performance in many of the practical cases
- In 1984 the Indian mathematician **Karmarkar** discovered a very efficient polynomial-time algorithm outperforming the simplex method

Contents

Complexity Theory and Complexity Classes

- Complexity Theory studies the intrinsic complexity of computational problems
- Its goals are:
  - (i) to determine the concrete complexity of problem, and
  - (ii) to identify the connections among the complexity of problems
- This 2nd goal has lead to *discovery* of several **complexity classes** (take a look at the website **The Complexity Zoo**)
- Among these complexity classes, the **NPC** class, for **NP-Complete** (decision) problems, is probably the most well-known

Contents

NP-Complete problems I

- The class NPC of **NP-Complete** (decision) problems contains a great diversity of algorithmic problems, in areas such as operations research, economics, circuits, communication, bioinformatics, game theory, logic, etc, etc
- They all of exhibit the same phenomena:
- The best algorithms that solve them are exponential-time. But **no one has been able to prove that any of these problems really require exponential time**
- And the **best-known lower bounds of most of the problems in this class are O(N)**. Hence it is possible (though unlikely) that they admit very efficient algorithms

NP-Complete problems II

- They share a **remarkable property**

  either they are **all** tractable!! or...

  none of them is!!

- This property explains the nane "complete" in the name of the class

NP-Complete problems III

- This property of problems in the class NPC means that, if someone ever finds **a polynomial-time algorithm for <u>any</u> NP-complete problem....**

  ....there would be **polynomial-time algorithms for <u>all</u> NP-complete problems.**

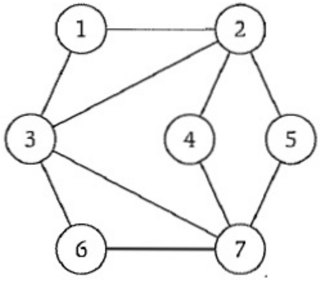
- And if someone ever **proves an exponential-time lower bound for <u>any</u> NP-complete problem....**,

  ...it would follow that **<u>no</u> NP-complete problem can be solved in polynomial time**

- This is not a conjecture, **it has been proved!**

Examples of NP-Complete problems I

INDEPENDENT SET: is a general problem that encodes a situation where you have to choose a subset of some set of objects but there might be pairwise restrictions. For instance: you want to invite a group of friends for dinner but some pairs of them don't get along well, and you want to invite the largest non-conflicting group of friends



- nodes are people, and edges between every two people mean they don't like each other
- for this example the largest group you can invite is

  $\{1, 4, 5, 6\}$

Examples of NP-Complete problems II

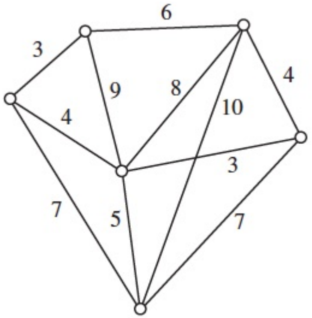
Another use of INDEPENDENT SET: there is one resource to be used by many people one at a time. Those interested, inform beforehand their requests with the period of the day the want to use the resource. Nodes represent the requests, and edge connecting two requests mean that they overlap in time. The resource administrator wants to attend the largest number of requests.

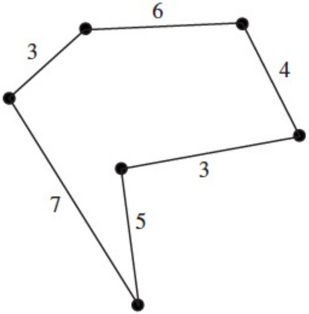
INDEPENDENT SET (decision version):

- Input: a graph G and a number k
- Output: is there a k-size *independent* subset of G's nodes? (y/n?)

Examples of NP-Complete problems III

TRAVELING SALESPERSON: A traveling salesperson has to visit all the cities in a given network before returning to his base. The algorithmic problem asks for the shortest route that passes through all the cities.





In the left is the graph with the networks of cities. In the right is the shortest route passing through all cities in the network (with lenght 28)

Examples of NP-Complete problems IV

Traveling salesperson (decision version):

- Input: a set of n nodes, the distances between each two of these n nodes, and a number k,
- Output: is there a tour that visits every node exactly once and has total length at most k?

Subset sum:

- Input: a multiset of n numbers $A_1, \ldots A_n$ and a number T
- Output: is there a subset of the numbers that sums up to T?

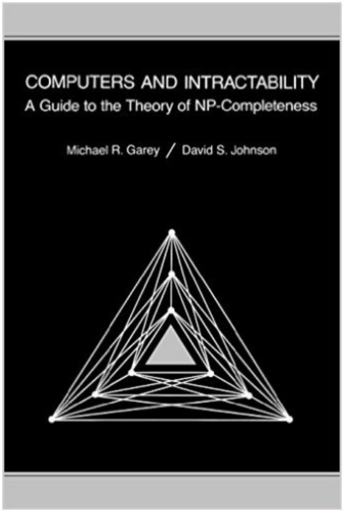
Examples of NP-Complete problems V

KNAPSACK PROBLEM (from Wikipedia) *Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit* W *and the total value is as large as possible.*

Its **decision version**, which asks if a total value of at least V can be achieved without exceeding the weight W, is an **NPC** problem

Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials, selection of investments and portfolios, etc

Examples of NP-Complete problems VI


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

Computers and Intractability - A guide to the theory of NP-Completeness
by M. Garey, and D. Johnson

One of the most cited Computer Science books

Has an appendix with more than 300 NP-Complete problems

Contents

NP Problems - Short Certificates

- Before defining the class of **NPC** problems let's take a step back and start with the class of **NP** problems
- A decision problem is an **NP** problem if, **given a possible solution for it, there is an algorithm that verify/certify, <u>in polynomial time</u>, that it is indeed a correct solution**
- **Monkey Puzzle, for instance, is an NP problem**:
  - when given a supposed solution, i.e. the pieces arranged in a grid, an algorithm can go through all the pieces once, and verify whether the cards match up correctly or not - and this can be done in polynomial time.

More Examples of NP Problems I

Independent set:

- Input: a graph G and a number k
- Output: is there a k-size independent subset of G's nodes? (y/n?)
- Certificate: a list of k nodes forming an independent set

Traveling salesperson:

- Input: a set of n nodes, the distances between each two of these n nodes, and a number k,
- Output: is there a tour that visits every node exactly once and has total length at most k?
- Certificate: the sequence of nodes in the tour.

More Examples of NP Problems II

SUBSET SUM:

- Input: a multiset of n numbers $A_1, \ldots A_n$ and a number T
- Output: is there a subset of the numbers that sums up to T?
- Certificate: the members in this multiset that sums up to T.

LINEAR PROGRAMMING:

- Input: a list of m linear inequalities with **rational** coefficients over n variables $x_1, \ldots x_n$
- Output: is there an assignment of rational numbers to the variables $x1, \ldots x_n$ that satisfies all the inequalities?
- Certificate: the assignment of values to variables

Observe that all decision problems which are tractable, i.e. that have polynomial solutions, belong to the class NP, .i.e P $\subseteq$ NP) !! Why?

NP Problems - Magical Nondetermism

Another way of describing NP problems (that explains the name "NP")

- Whenever it is possible to extend a partial solution, we have a (nondeterministic) machine that can always make a correct choice among the alternatives
- The machine always guess a possibility that leads to a complete solution, if there is a complete solution.
- And if all this "guessing" leading to a complete solution is performed in polynomial time, we say that we the problem has a **N**on-deterministic **P**olynomial-time algorithm for the problem
- Hence, the name **NP**

NPC problems are the hardest NP problems

- **One of the requirements for a problem to be classified as NP complete is that it should be an NP problem**
- That means that, although we don't know how to **solve** an NPC problem in time less than exponential, if we are presented with a solution for it we can verify, in polynomial time, that it is indeed a solution
- **Another requirement for problems to be classified as NPC is that they should be the *hardest* problems among all problems in NP**
- Before we see how to prove that a problem is an NP complete problem, let's take a look to a very important question in Computer Science....

Contents

Is $\mathrm{P} = \mathrm{NP}$? I

- We saw that the class P, of all problems solvable in polynomial time, is contained in the class of NP problems, i.e. $\mathrm{P} \subseteq \mathrm{NP}$

- Recall that for the NPC problems there are two possibilities:
    - **If someone ever finds a polynomial solution for an NPC problem**, since the NPC problems are the *hardest* problems in NP, that would mean that all problems in NP would also have a polynomial solution. In other words that would mean that $\mathrm{NP} \subseteq \mathrm{P}$. And we will be able to say that $\mathrm{P} = \mathrm{NP}$

    - **If someone ever proves that it is impossible to have polynomial solution for NPC problems**, then, of course, it is not the case that $\mathrm{NP} \subseteq \mathrm{P}$. And then we could say that $\mathrm{P} \neq \mathrm{NP}$

Is P = NP? II

- The $\mathrm{P} = \mathrm{NP}?$ problem, as it is called, has been posed in 1971, and is **one of the most difficult unresolved problems in computer science.**

- Most people believe that $\mathrm{P} \neq \mathrm{NP}$, meaning that the NP-complete problems are inherently intractable, but no one knows for sure.

- In any case, showing that an algorithmic problem is NP-complete is an evidence of its probable intractability.

- But how can we prove that a problem is NP-complete? For understanding how such a proof can be done, we still need need to learn about **polynomial-time reduction** of a problem to another.

Contents

Polynomial Time Reduction I

A **polynomial time reduction** is a polynomial time algorithm that reduces one problem to the other in the following sense:

- If someone comes with an input X to the first problem and wants a "yes" or "no" answer, the reduction algorithm **transforms** X into an input Y to the second problem,
- The reduction algorithm does that in such a way that the second problem's answer to Y is precisely the first problem's answer to X.
- We write $Q \leqslant_P R$ to say that problem Q can be reduced in polynomial time to problem R

Polynomial Time Reduction II

- On intuitive way to remember what does it mean for a problem Q to be reducible to a problem R, is to think that, if that is the case, it means that the best algorithm known for problem R, besides solving R, has a logic capable of solving problem Q as well.

- Problem R just needs the inputs of Q to be transformed/translated to the "format" it understands.

- If this transformation/translation of the input of Q as an input of R is done in polynomial time, we say that problem Q can be reduced in polynomial time to problem R (written $Q \leqslant_P R$)

Polynomial Time Reduction III

Suppose $Q \leqslant_P R$:

- If R **can** be solved in polynomial time, then ...

  ... Q **can** also be solved in polynomial time

- If Q **cannot** be solved in polynomial time then ...

  ... R **cannot** be solved in polynomial time

Hamiltonian Path to Traveling Salesman I

HAMILTONIAN PATH

- Input: a graph G with N nodes
- Output: is there a path passing exactly once by every node of the graph G?

Such a path is called an Hamiltonian Path.

- The HAMILTONIAN PATH (HP) problem can be polynomially reduced to the TRAVELING SALESPERSON (TS) problem, i.e:

$$\text{HP} \leqslant_{\text{P}} \text{TS}$$

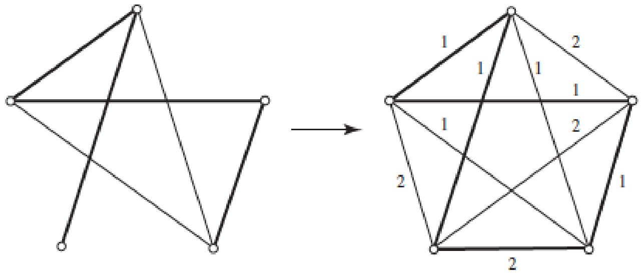
Hamiltonian Path to Traveling Salesman II

Given a graph G with N nodes, the original input for problem HP, transform it to be an input graph T for the problem TS as follows:

- The nodes of T are precisely the nodes of G
- Edges are drawn between every two nodes in T
- Assign cost 1 to an edge of T if it was present in the original graph G, and 2 if it was not.

This transformation of the input of HP to an input to TS can be done in polynomial time!

Hamiltonian Path to Traveling Salesman III

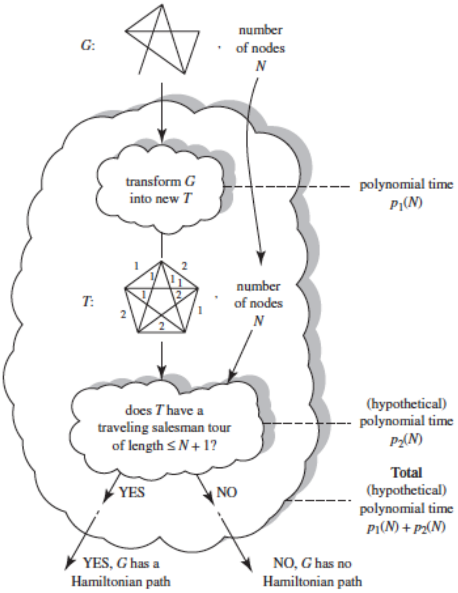


Hamiltonian path emphasized

Traveling salesman tour of length 6 emphasized

- Then ask whether T has a TS tour that is no longer than $N + 1$, where N is the number of nodes in G.
- The answer to the first question on G is “yes” precisely when the answer to the second question on T is “yes”.

Hamiltonian Path to Traveling Salesman IV


G:
number of nodes N
transform G into new T
polynomial time $p_1(N)$
T:
1
2
1
1
1
1
2
2
1
2
number of nodes N
does T have a traveling salesman tour of length $\leq N+1$?
(hypothetical) polynomial time $p_2(N)$
YES
NO
Total (hypothetical) polynomial time $p_1(N) + p_2(N)$
YES, G has a Hamiltonian path
NO, G has no Hamiltonian path

Hamiltonian Path to Traveling Salesman V

- This was a proof that :

$$\mathrm{HP} \leqslant_{\mathrm{P}} \mathrm{TS}$$

- If someone ever finds a polynomial time solution to the TS problem that would imply ....

  .... a polynomial time solution to the HP problem

- If someone ever proves that HP problem cannot be solved in polynomial time, that would imply ...

  ... that TS cannot be solved in polynomial time

NP Complete Problems are Inter-Reducible

Every NP-complete problem is polynomially reducible to each other!

Hence, tractability of one implies tractability of all, and intractability of one implies intractability of all.

How to Prove a Problem is NP-Complete? I

- To establish a problem R is NP-complete (1) reduce in polynomial time R to a problem already known to be NP-complete, call it Q, and (2) reduce in polynomial another problem known to be NP-complete (possibly the same one), call it S, to R:

$$(1) \quad \boxed{\mathsf{R}} \leqslant_P Q \qquad\qquad (2) \quad S \leqslant_P \boxed{\mathsf{R}}$$

- Note that reduction (1) means that R is in NP (why?)

- So, instead of giving two reductions, show that R is in NP (short certificate), and show reduction (2)

How to Prove a Problem is NP-Complete? II

In summary: to show a problem R is NP-Complete:

1 show that $\boxed{R}$ is NP

2 show that $S \leqslant_P \boxed{R}$, where S a problem known to be a NP-complete

1st NP-Complete Problem - Cook's Theorem

- For all this to work there should be a **first problem proved to be NP-complete**.
- In 1971 the satisfiability problem for the propositional calculus, the SAT problem, was shown to be NP-complete by **Stephen Cook**, thus providing the anchor for NP-completeness proofs.
- The result, known as **Cook's theorem**, is considered to be one of the most important results in Computer Science:

  1. SAT is NP
  2. SAT is NP-Hard, i.e., $\mathrm{S} \leqslant_{\mathrm{P}} \mathrm{SAT}$, for any S in NP

Theory x Practice

Do read the following editorial of CACM, by Moshe Vardi

- *On P, NP, and Computational Complexity*
  CACM Vol. 53, No. 11, Page 5, November, 2010

  "While the P vs. NP quandary is a central problem in computer science, we must remember that a resolution of the problem may have limited practical impact. It is conceivable that P = NP, but the polynomial-time algorithms yielded by a proof of the equality are completely impractical, due to a very large degree of the polynomial or a very large multiplicative constant; after all, $(10\mathrm{N})^{1000}$ *is a polynomial! Similarly, it is conceivable that* $\mathrm{P} \neq \mathrm{NP}$*, but NP problems can be solved by algorithms with running time bounded by* $\mathrm{N}^{\log\,\log\,\log \mathrm{N}}$ *— a bound that is not polynomial but incredibly well behaved."*

Cook's Theorem - 1st NP-Complete Problem

- Now, clearly there had to be a first problem proved to be NP-complete.
- Indeed, in 1971 the satisfiability problem for the propositional calculus was shown to be NP-complete thus providing the anchor for NP-completeness proofs.
- The result, known as Cook's theorem, is considered to be one of the most important results in the theory of algorithmic complexity.