NP-Complete Problems

With a short and informal introduction to Computability Complexity

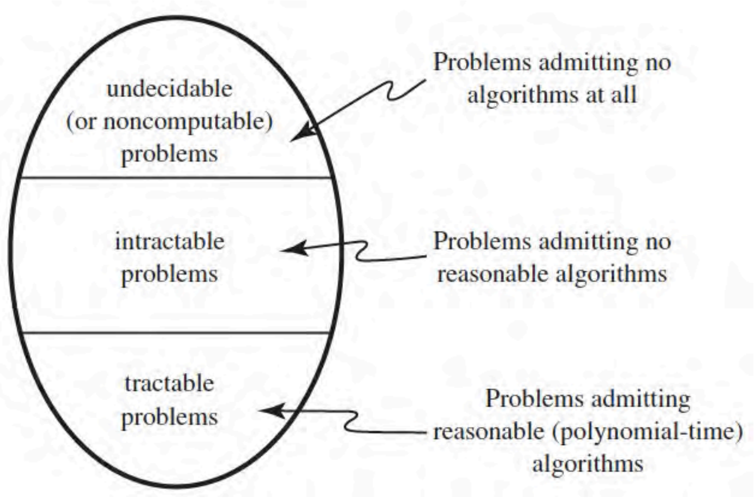
Álvaro Moreira
alvaro.moreira@inf.ufrgs.br

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

Contents

Contents

Tractability x Untractability



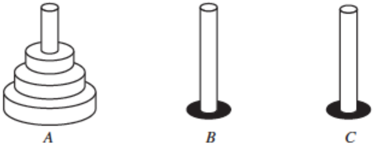
Hanoi Towers I

- Suppose we are given three towers, or three pegs, A, B, and C.
- On the first peg, 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 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.



Hanoi Towers III

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

- We are interested in an algorithm to solve the **general algorithmic problem** associated with the Towers of Hanoi

- The input for the algorithm is a positive integer N (the number of rings), and the desired output is a list of ***"move X to Y"*** actions, that solve the puzzle for N rings.

- There is an algorithm where the **number of move actions produced**, for an N ring case, is precisely $2^{\mathrm{N}} - 1$

Hanoi Towers IV

- Also, it has been **proved** that $2^N - 1$ is a **lower bound** on the required number of moves for solving the problem, so we **cannot do any better than this**

- If we were able to move **a million rings every second**, with **64 rings** it would take **more than half a million years** to complete the process!!

- If we were able to move only **one ring every 10 seconds**, it would takes us more than **five trillion years to finish**.

Decision Problems I

- One may think the difficulty is because the output is a sequence of moves. Since many moves are required, it takes too long to find and exhibit all of them.
- To convince that this is not the case lets examine **decision problems** (problems requiring only a ”yes“/”no“ solution)
- Most of the problems of interest in practice however, are not decision problems...
- Problems which are not decision problems will be **recast** as decision problems.

Decision Problems II

Example: Consider the problem that, given data about direct buses between cities (a graph), and given the name of two cities, finds a the most direct path (shortest in terms of bus changes) between them.

A decision problem related to problem above is :

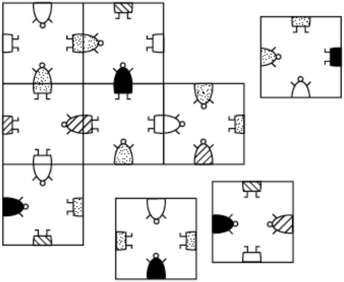
Given info about direct bus connections between cities (a graph), two cities u and v (vertices), and a non-negative integer k, does a path exist between u and v whose length is at most k?

The number k is a bound on the value to be optimized.

If an optimization problem is easy then its related decision problem is easy as well. **If a decision problem is hard, its related optimization problem is also hard.**

Monkeys Puzzle I

- Given (descriptions of) N cards, where N is some square number, say, N is M^2, the (original) problem calls for exhibiting, if possible, an M by M square arrangement of the N cards, so that colors and halves 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 has already been shown to be illegal.

Monkeys Puzzle III

- If we are dealing, for instance, with a 5x5 grid, there are 25 possibilities for choosing a card to be placed in the first location.
- Having placed some card in that location, there are 24 cards to choose from for the second location, 23 for the third, and so on.
- The total number of arrangements can, therefore, reach: $25 \times 24 \times 23 \times \ldots \times 3 \times 2 \times 1 = 25!$ which is 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

- A computer that can try **a billion arrangements every second** will take **well over 490 million years** to try all 25! arrangements!!!.
- In a 6x6, the time to try all 36! arrangements would be **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 game is well-designed.
- These impressive numbers consider the **brute force** solution. 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.

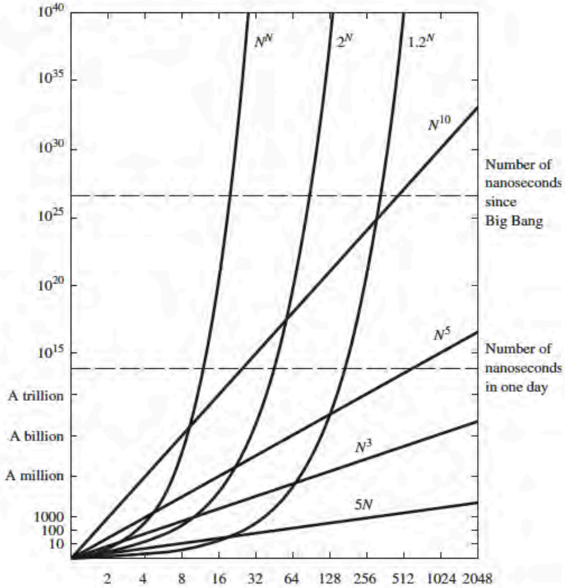
Function values

	Function \ N	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



Function growth II

- 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.

Polynomial x Exponential (good x bad) I

- These facts lead to a fundamental classification of functions into "good" and "bad" ones.
  - The good ones are **polynomial** functions
  - The bad ones are **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 x Exponential (good x 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**

- It is common to abuse terminology and use **exponential** as a synonym for **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 I

- An **algorithm** whose (order-of-magnitude) time performance is bounded from above by a polynomial function of N, where N is the size of its inputs, is called a **polynomial-time algorithm**
- Similarly, an **algorithm** that, in the worst case, **requires** super-polynomial, or exponential time, will be called a **exponential algorithm**
- An **algorithmic problem** is **tractable** if it admits a polynomial-time solution. It is **intractable** if it **only** admits an exponential-time solution

Tractable x Intractable problems II

- Of course that an N^{1000} polynomial algorithm is worse than a $N!$ exponential algorithm for inputs under size 1165
- But the majority of exponential algorithms are really useless, and most polynomial algorithms are really useful in practice
- These facts give credibility to this distinction between *good* (polynomial) and *bad* (exponential)
- In fact, the vast majority of polynomial-time algorithms for practical problems feature an exponent of N that is no more than 5 or 6.

Contents

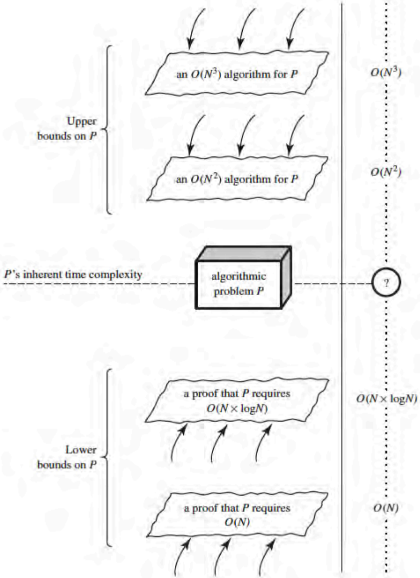
Lower x Upper Bounds I

- Any algorithmic *problem* has an **inherent *optimal* solution**.
- Suppose someone gives an $O(N^3)$ algorithm for problem P
    - We then know that the optimal solution of P 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 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, beforehand, **how far can improvements go?**
- Yes, that requires a **proof** of a **lower bound**.
- If, for instance, we **prove** that the problem P cannot be solved in less than $O(N^2)$, then people can stop looking for better algorithms for it

Lower x Upper Bounds III



Closed x Open Problems

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

Examples III

OBS.: Linear Programming is a very important problem with many 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

Organizing the World of Computational Problems I

- The Monkey Puzzle is just one of close to 1000 algorithmic problems, all of which 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**, i.e. no one has been able to prove that their lower-bounds are also exponential
- The **best-known lower bounds of most of the problems in this class are O(N)**. Hence it is conceivable (though unlikely) that they admit very efficient linear-time algorithms.

Organizing the World of Computational Problems II

- This class of (decision) problems is called NPC. Its elements are called **NP-Complete problems** (for **N**ondeterministic **P**olynomial Time Complete problems)
- The class NPC contains an ever-growing diversity of algorithmic problems, arising in such areas as operations research, economics, graph theory, game theory, and logic
- They share a remarkable property: either they are all tractable or none of them is!!
- Before defining NP-Complete Problems let's take a step back and start first with the class **NP** of **NP** problems.

Contents

NP Problems - Short Certificates I

- The set of NP problems is the set of all decision problems that are **verifiable in polynomial time**
- If we have a possible solution for it, we can verify, in polynomial time, that it is indeed a correct solution
- We can, for instance verify that a monkey puzzle solution is correct in polynomial time (linear time, even!), when presented with a solution.
- When given the pieces arranged in a square, a supposed solution, we simply go through all the pieces once, and verify that the monkeys match up correctly or not - this can be done in polynomial time.

NP Problems - Short Certificates II

- So, our verification algorithm always has a yes/no (true/false) answer in polynomial time
- We call this answer, along with it's explanation, a **certificate**.

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 vertices?
- Certificate: a list of k vertices 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 closed circuit, i.e. 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 list 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 list of members in this subset that sums up to T.

Linear programming:

- Input: a list of m linear inequalities with rational coefficients over n variables $x_{1}, \ldots x_{n}$ (a linear inequality has the form $a_1x1 + a_2x_2 + \ldots + a_nx_n \leqslant b$ for some coefficients $a1, \ldots a_n, b$),
- Output: is there an assignment of rational numbers to the variables $x1, \ldots x_n$ that satisfies all the inequalities?
- Certificate: is the assignment.

NP Problems - Magic Coins I

- There is another way of describing NP problems
- Assume we have a very special “magic coin”....
- Whenever it is possible to extend a partial solution in two ways (for example, two monkey cards can be legally placed at a currently empty location, the coin is flipped and the choice is made according to the outcome.
- However, the coin does not fall at random; it possesses magical insight, always indicating the best possibility.

NP Problems - Magic Coins II

- The coin will always select a possibility that leads to a complete solution, if there is a complete solution.
- Technically, we say that algorithms that use such magic coins are **nondeterministic**,
- They always "guess" which of the available options is better, rather than having to employ some deterministic procedure to go through them all.
- Thus, NP problems are apparently intractable, but become "tractable" by using magical nondeterminism.

Contents

NP Completeness I

- NP-complete problems are NP problems which are open w.r.t tractability status (such as the Monkey's Puzzle) that have a additional and remarkable property:
- **Either they are all are tractable, or none of them is!**
- The term "complete" is used to signify this additional property

If someone finds **a polynomial-time algorithm for any single NP-complete problem**,

- there would immediately be **polynomial time algorithms for all NP-complete problems.**

NP Completeness II

Also, if someone were to **prove an exponential-time lower bound for any NP-complete problem**,

- it would follow immediately that **no NP-complete problem can be solved in polynomial time**

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