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

Algorithmics - the science of algorithms

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

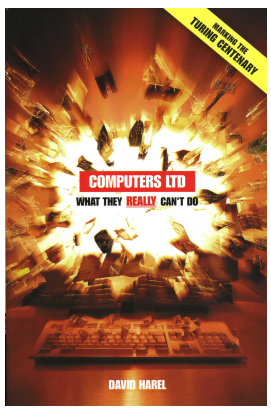
- The good news:
  - Algorithms and data (getting it done)
  - Algorithmic methods (getting it done methodically)
  - The correctness of algorithms (getting it done right)
  - The efficiency of algorithms (getting it done cheaply)

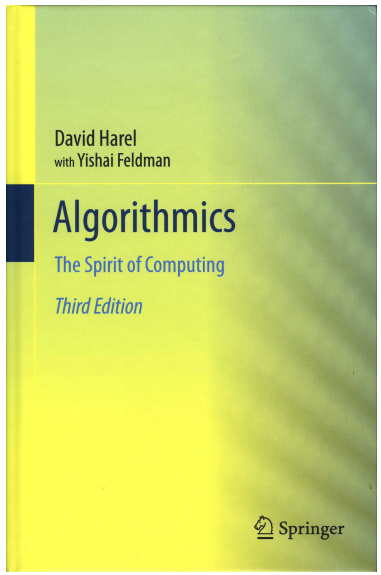
- The bad news:
  - Intractability (you can't always get it done cheaply)
  - Noncomputability (sometimes it can' t be done at all!)

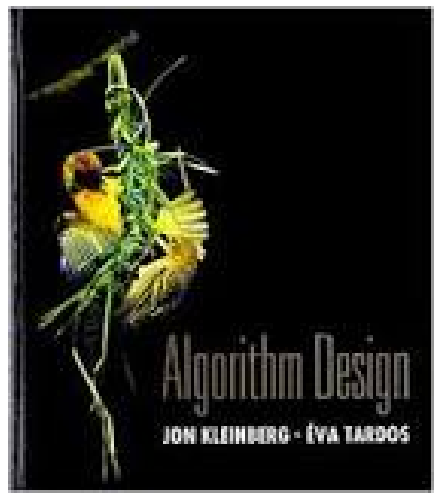
Contents

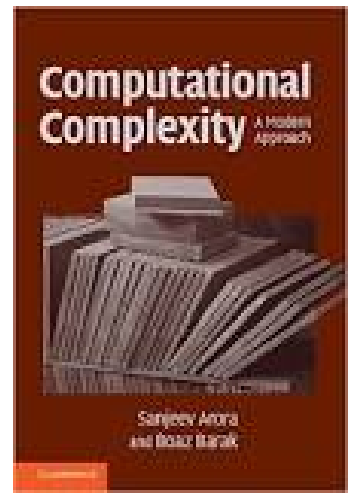
Bibliographic References

These slides are based on the following four books:









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

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
   - Proof that Halting Problem is non-computable
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;`
$\vdots$
(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

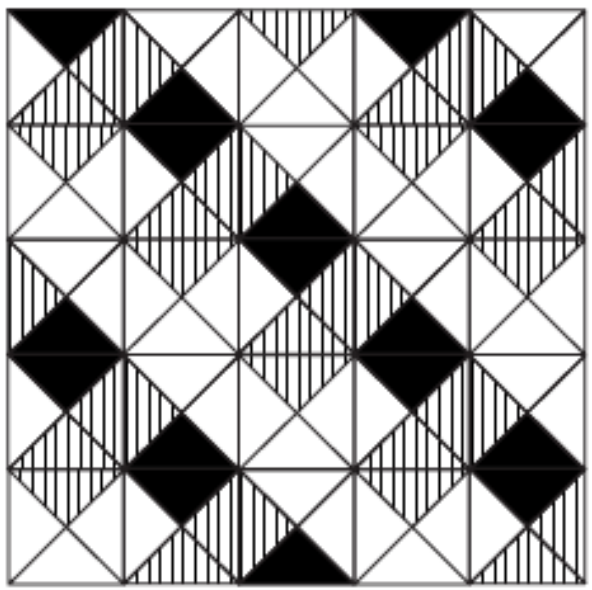
- The problem involves covering large areas using square tiles with colored edges, such that adjacent edges are monochromatic.

- 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

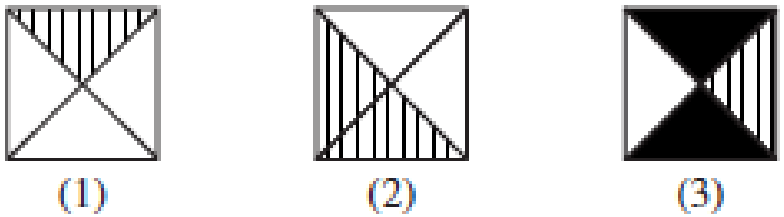


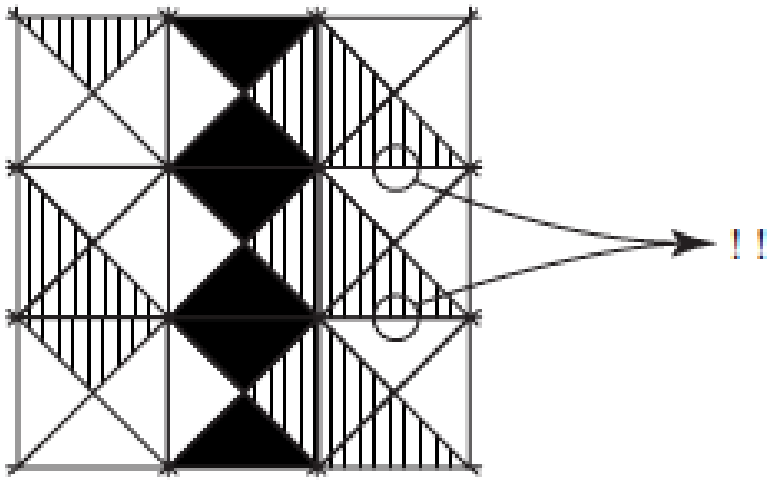
The algorithmic problem

- 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 $\mathrm{t} = (n, e, s, w)$ of four symbols, that identify the colors at the top, right, bottom and left edges of the tile.

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

The Tilling Problem V

Requirements

A *tiling* is a function $f : N \times N \to T$ 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 $f : N \times N \to T$ 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 Xs and the Ys.

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 Ys 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 Xs and Ys - 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

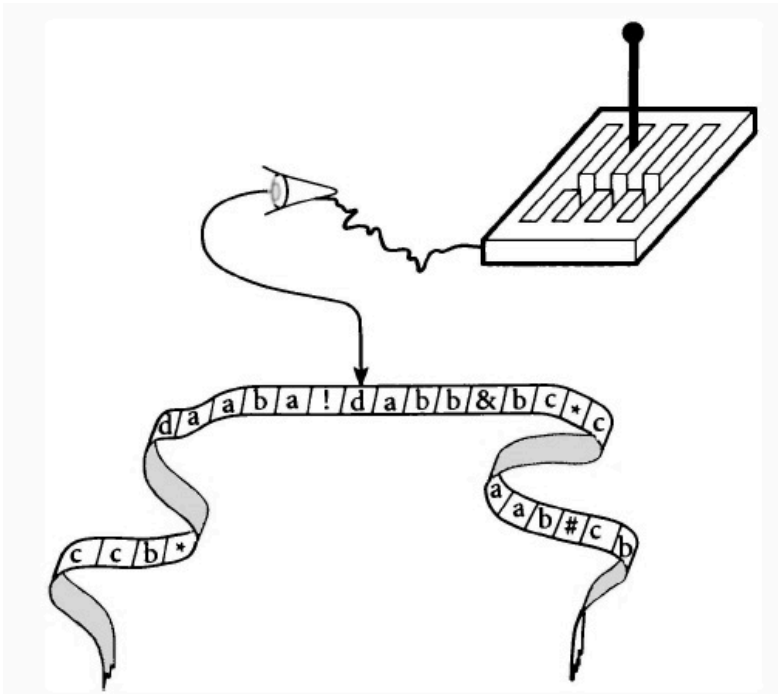
Investigation on (non-)computability started a long time ago in the context of Logics and Mathematics when there were no computers!

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****.*