A JOURNEY OVER COMPUTATION

P = NP dilemma

Abstract

This paper sheds light over computational theory in Computer Science. It illustrates some of the important concepts in computation and it gives clear sight over how to solve the famous P = NP problem.

Keywords

Theory of Computation, NP-Complete, Programming languages, Computer Science, Computation

DECEMBER 28, 2019

AMERICAN UNIVERSITY OF BEIRUT
Dr. Jad Matta

Table of Contents

Table of Figure

Computation Definition

Computation is a general term for any type of information processing that can be represented as an algorithm precisely (mathematically). For instance,

- Adding tow numbers in our brains, on a piece of paper or using a calculator
- Converting a decimal number to its binary presentation or vice versa
- Finding the greatest common divisors of two numbers

A very fundamental and traditional branch of **Theory of Computation** seeks:

1. A more tangible definition for the intuitive notion of algorithm which results in a more concrete definition for **computation**
2. Finding the boundaries (limitations) of computation

Algorithm

- A finite sequence of simple instructions that is guaranteed to halt in a finite amount of time
- This is a very naïve definition since we didn't specify the nature of this simple instructions and we didn't specify the entity which can execute these instructions

An Abstract Machine

- To make a more solid definition of algorithm we need to define an abstract (general) machine which can perform any algorithm that can be executed by any computer
- Then, we need to show that indeed this machine can run any algorithm that can be executed by any other computer. Then,
    - We can associate the notion of algorithm with this abstract machine
    - We can study this machine to find the limitations of computations, (Problems with no computation available to solve)

Regular Languages

Regular languages = languages denoted by regular expressions

= languages accepted by DFAs

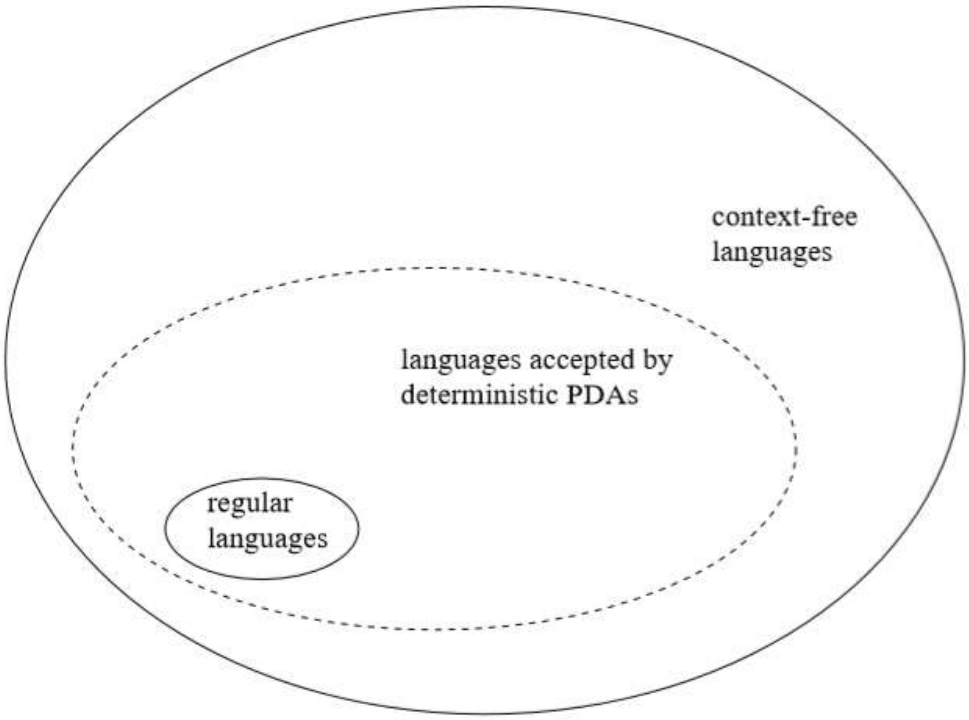
= languages accepted by NFAs

The class of regular languages is strictly contained in the deterministic context-free languages (DCFL) which in turn are strictly contained in the (general) context-free languages.

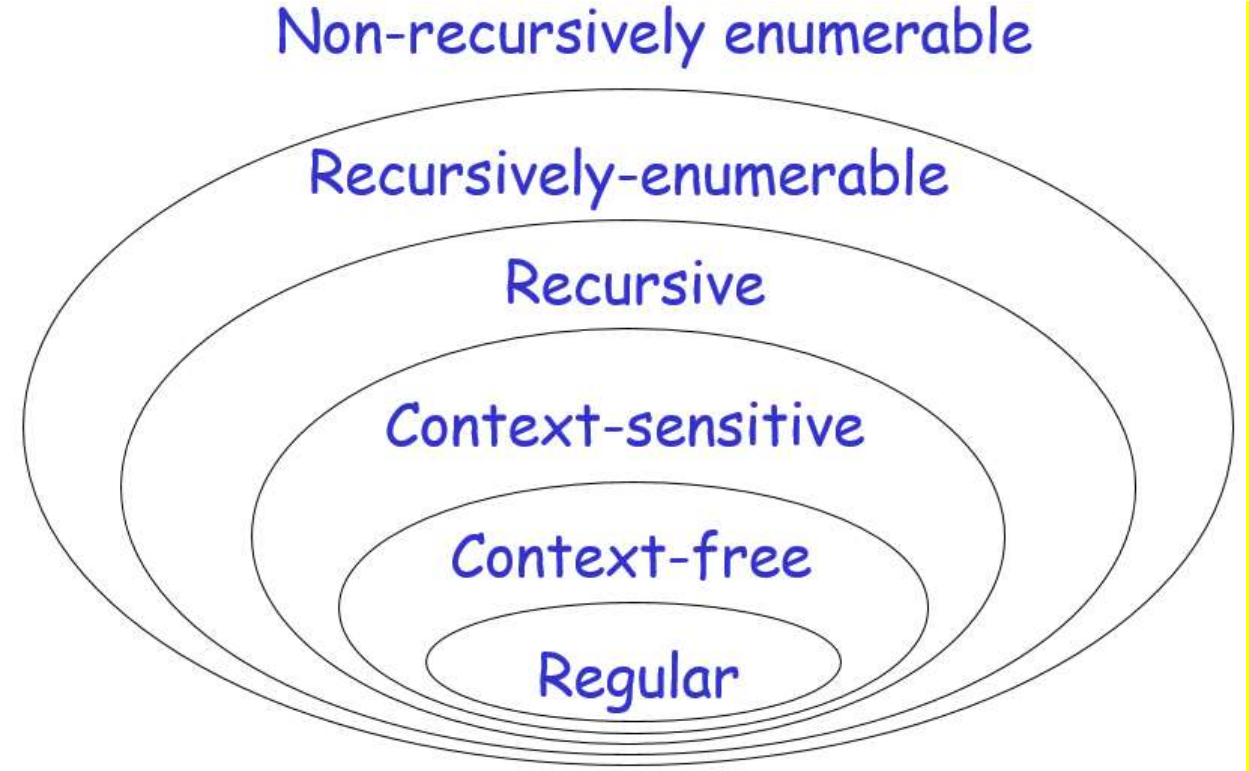
The class DCFL consists of languages recognized by deterministic pushdown automata.

A decision problem is a restricted type of an algorithmic problem where for each input there are only two possible outputs.

- A decision problem is a function that associates with each input instance of the problem a truth value true or false
- A decision algorithm is an algorithm that computes the correct truth value for each input instance of a decision problem. The algorithm has to terminate on all inputs
- A decision problem is decidable if there exists a decision algorithm for it. Otherwise, it is undecidable



The Chomsky Hierarchy



Turing Machine

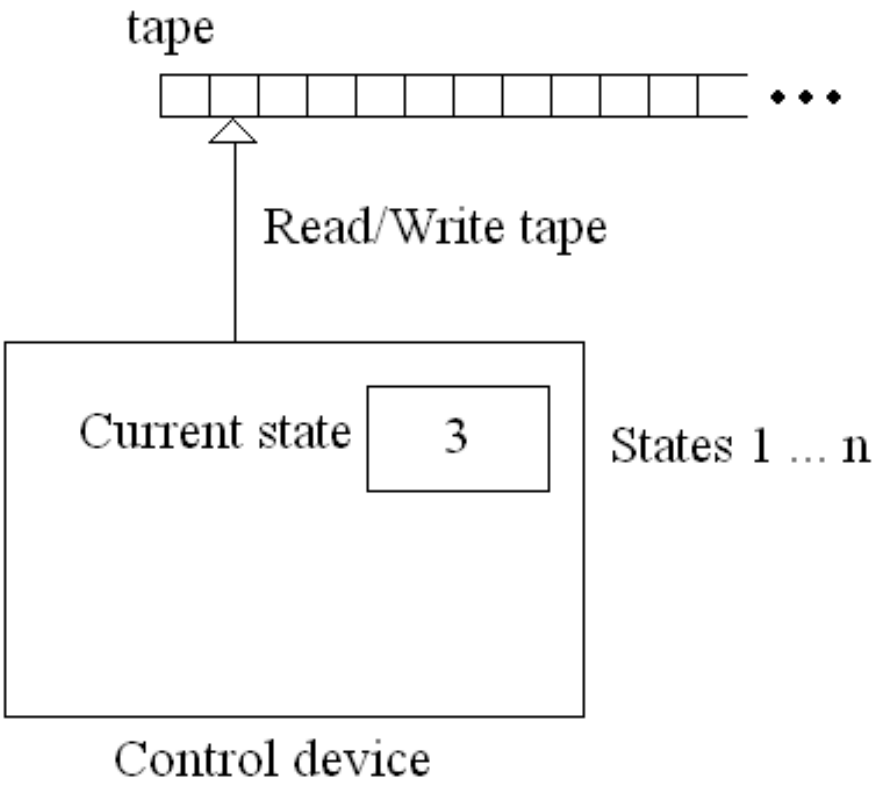
- A conceptual model for general purpose computers proposed by Alan Turing in 1936
- A Turing machine has an unlimited and unrestricted amount of memory
- A Turing machine can do everything a real computer can do
- Nevertheless there are problems that a Turing machine cannot solve
- In real sense, these problems are beyond the theoretical limits of computations

Turing Machine Specification

Components of Turing Machine:

1. An unlimited length tape of discrete cells
2. A head which reads and writes on tape

3. A control device with a finite number of states which can
    a. Instruct the head to read the symbol on the tape currently under head
    b. Instruct the head to write a symbol on the cell of the tape currently under tape
    c. Move the head one cell to left or right
    d. Change its current state



Turing Machine Instructions

- Instructions of Turing Machine have the following format:

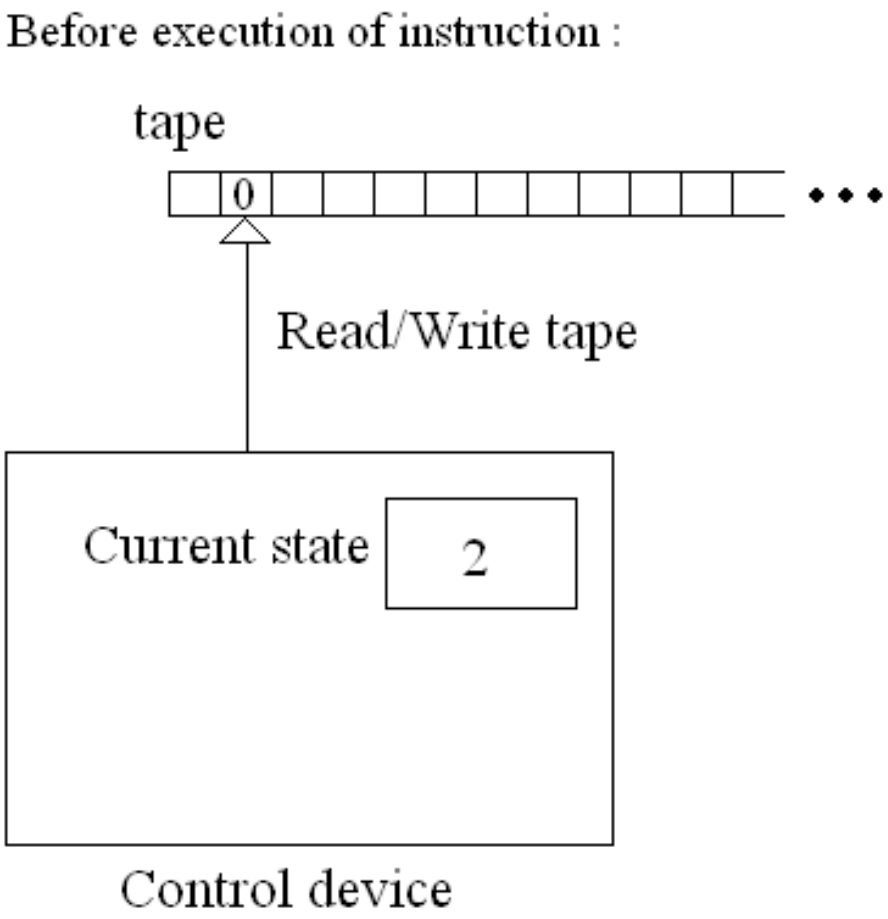
(Current state, current symbol, Write, Move L/R or No move, New State)

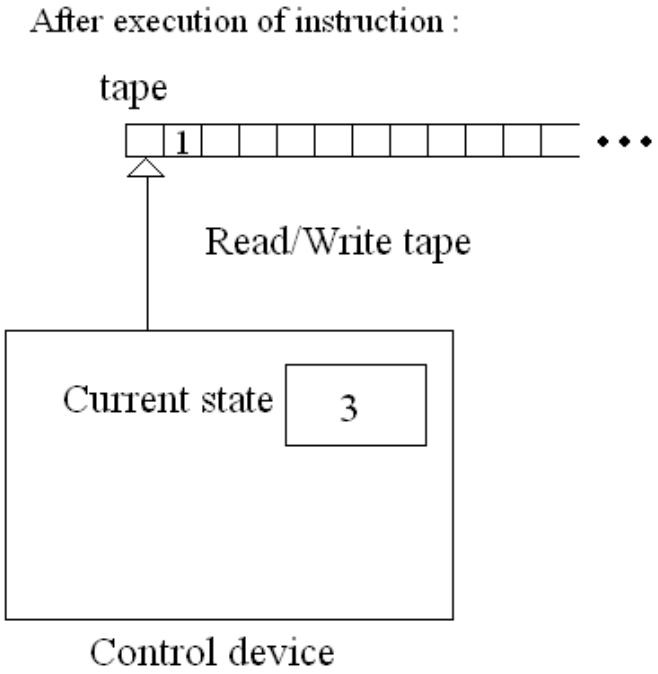
- The interpretation of the TM (Turing Machine) instructions:
    - (2, 0, 1, L, 3)

- When Turing machine (the control unit of TM) is at state 2 and the current tape symbol is 0, write symbol 1 at current tape cell and go to state 3

Visualization of TM instruction

(2, 0, 1, L, 3)





TM Conventions

- We always use state 1 as the initial state
- The tape is used for recording input and output, one symbol per cell. Initially, the string to serve as input to our computation is recorded beginning from the leftmost tape cell
- Initially, the position of head is at left most cell

The output of TM

- The output of a TM program or algorithm is the sequence of symbols on the tape when the TM halts on that program

TM Programs

- A Turing machine program is a set of TM instructions
- Turing machine halts on a program if there is no instruction in the program which its current state is the current state of the machine and its current symbol is the current symbol of the tape of the machine (symbol under head of the machine)

Example

{(1, 1, 1, R, 2), (2, 1, 1, R, 2), (2, blank, blank, R, 3), (3, 1, blank, L, 4), (4, blank, 1, R, 2)}

- This program outputs the sum of two integers m and n given as input
- The numbers are in base 1 (unary notation)
- Examples of integers in unary notation
  1 = 1 2 = 11 3 = 111 4 = 1 1 1 1 ….
  Number n = n number of 1s.

The definition of Algorithm

We have reasons to believe that for any algorithm (finite sequence of steps which stops in a finite amount on time) that can be executed on any machine, there is a TM algorithm (program) which can be executed on TM and performs the same action.

Decidable Problems

- Problems, for which we can't find an algorithm that answer all possible instances of the problem
- That is there is not TM program which answer all possible instances of the problem in a finite amount of time
- For a decidable problem there is a program such that if an instance of the problem has solution, the program eventually halts with answer. But there is no solution for that instance, the program will not ever halt.

Can we consider such programs as algorithms?

Answer: No, because they might not halt

A problem is decidable if some Turing machine decides (solves) the problem

Decidable languages are often called also recursive languages.

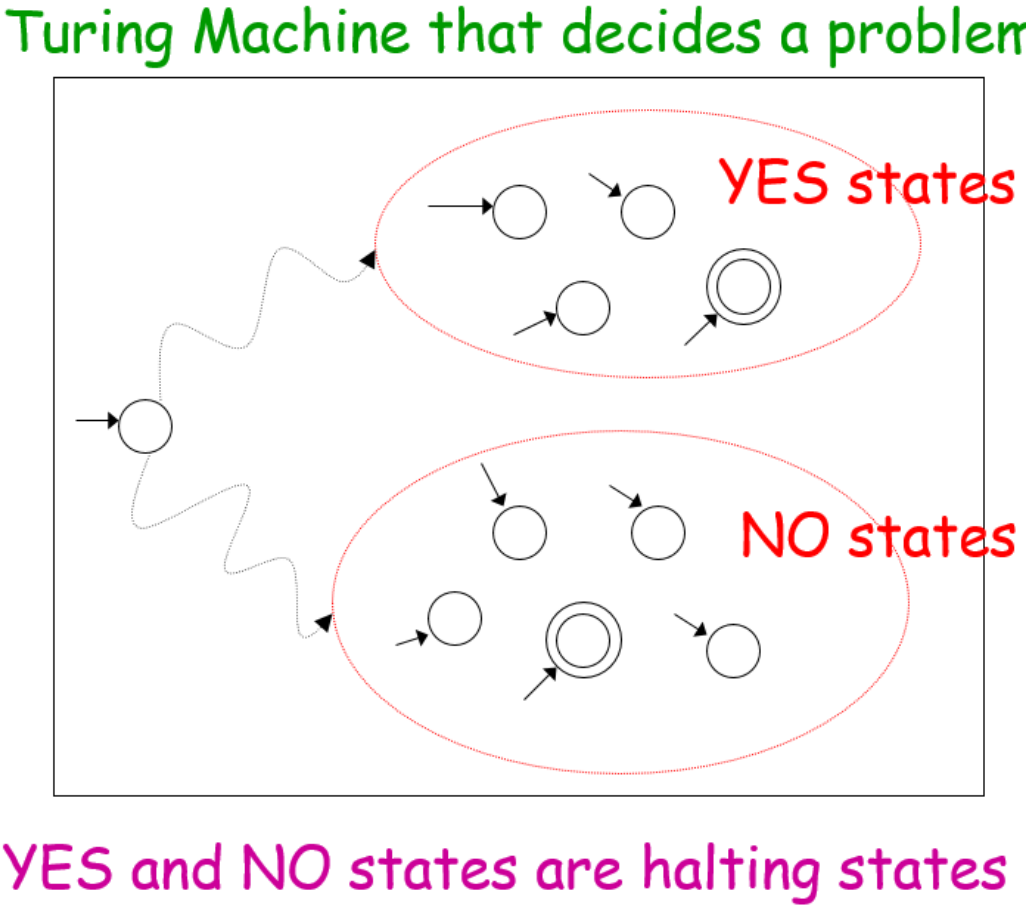
Decidable problems:

- Does Machine M have three states?

- Is string w a binary number?
- Does DFA M accept any input?

The machine that decides (solves) a problem:

- If the answer is yes then halts in a yes state
- If the answer is no then halts in a no state

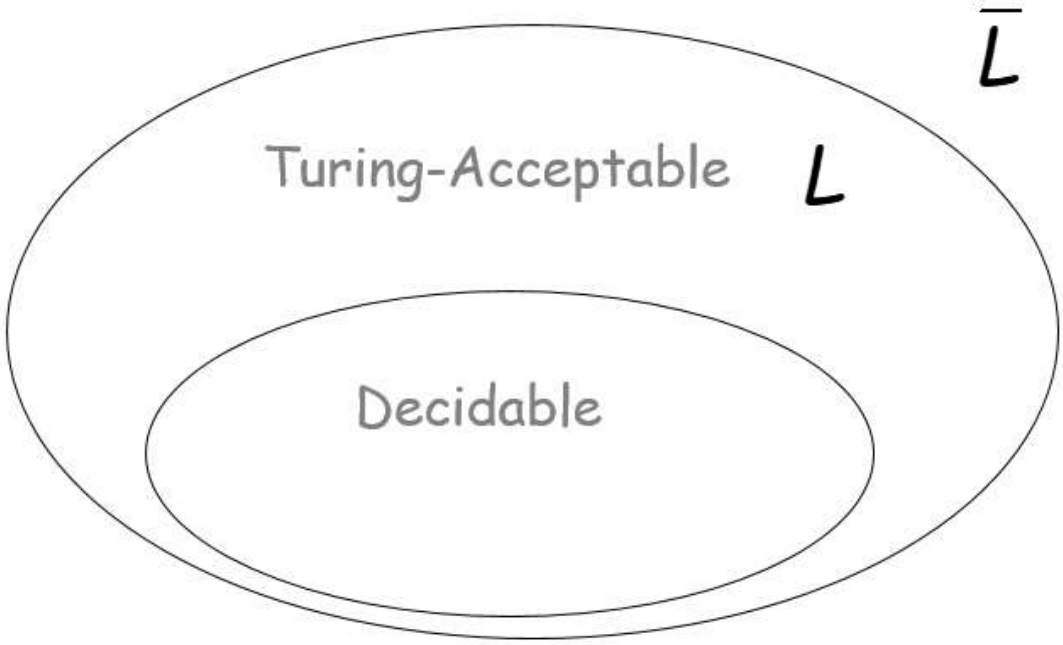


Undecidable Language

There is no Turing Machine which accepts the language and makes a decision (halts) for every input string.

(machine may make decision for some input strings)

A language is Turing-recognizable (or recursively enumerable) if it is recognized by a TM. That is, all words in the language are accepted by the TM. On words not belonging to the language, the computation of the TM either rejects or goes on forever.



L is Turing-Acceptable and undecidable

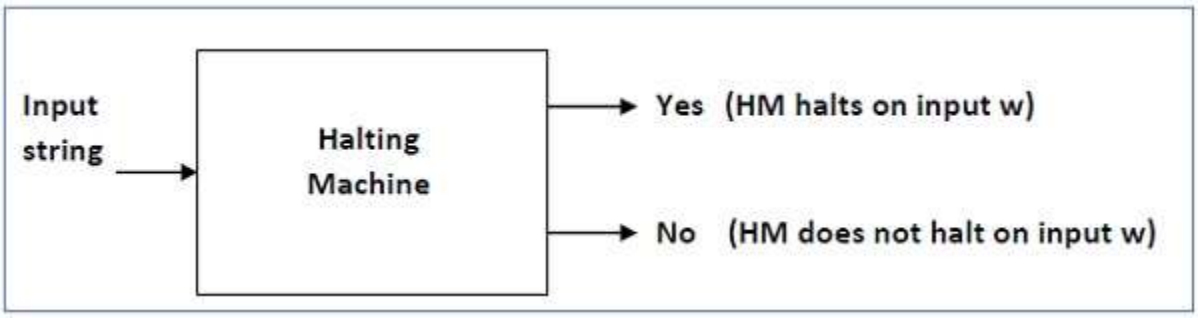
The Halting Problem

Given a program/algorithm will ever halt or not?

Taking a procedure and an input evaluate to **#t** if the procedure would terminate on that input, and to **#f** if would not terminate.

Halting means that the program on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. Basically halting means terminating. So can we have an

algorithm that will tell that the given program will halt or not. In terms of Turing machine, will it terminate when run on some machine with some particular given input string.



This is an undecidable problem because we cannot have an algorithm which will tell us whether a given program will halt or not in a generalized way by having specific program/algorithm. The best possible way is to run the program and see whether it halts or not. In this way for many programs we can see that it will sometimes loop and always halt.

Limits of Computation: Tractable and Intractable Problems

A problem is in P if it admits an algorithm with worst-case time-demand in $O(n^k)$ for some integer k.

N.B: a problem may also have algorithmic solutions whose time-demand grows unreasonably. For instance, a naïve solution for the determinant would take O(n!) while it can take $O(n^3)$ using the Gaussian elimination method.

However there are some problems for which it is known that there are no algorithms which can solve them in polynomial time, these are referred to as **provably intractable** and as being in the class **EXPTIME (EXP**onential **Time) –** or wose.

A problem is in the class EXPTIME if all algorithms to solve it have a worst-case time demand which is in $O(2^{p(n)})$ for some polynomial p(n).

Example: the Towers of Hanoi

Higher time-complexity classes

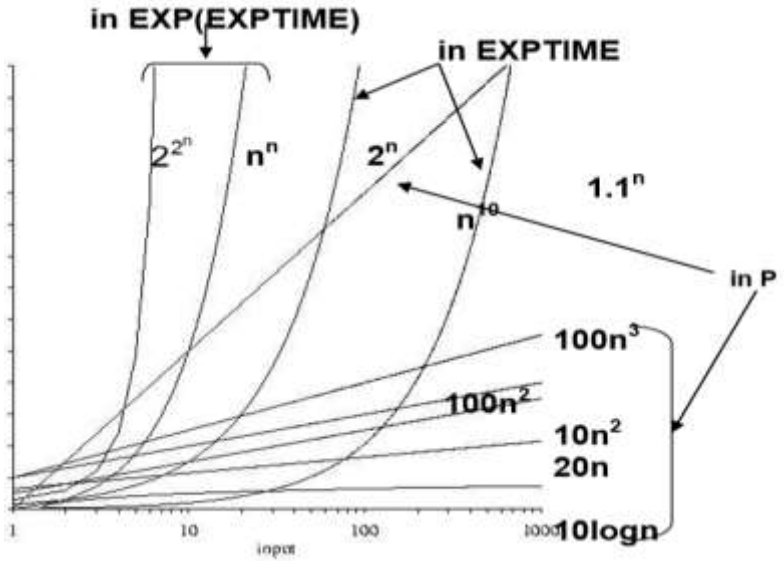
There are other classes of problems for which the time demand cannot be bounded above even by a function of the form $2^{p(n)}$. In fact there are a hierarchy of these higher time-complexity classes such that a problem within a given class is considered 'more intractable' than all those within lower-ranked classes.

So beyond EXPTIME we can have EXP(EXPTIME), for which the time-demands of all known solutions are bounded above by a multiple of $2^{2^{p(n)}}$, EXP(EXP(EXPTIME)) problems which are in $O(2^{2^{2^{p(n)}}})\cdots$ and there are problems whose time-complexity is even worse, and cannot be bounded by any

$$2^{2^{\cdot^{\cdot^{\cdot^{2^{p(n)}}}}}}$$

(referred to as 'non-elementary' problems.

All these classes of provably intractable problems, from EXPTIME upward, can be referred to as having a **super-polynomial** time demand.



However it turns out that the most interesting class of problems is a class which lies on some sense between the class of tractable problem P and those of the provably intractable, super-polynomial time problems.

These are problems which are probably intractable – but we are not sure.

The classes NP and NP Complete

The Hamiltonian Circuit Problem

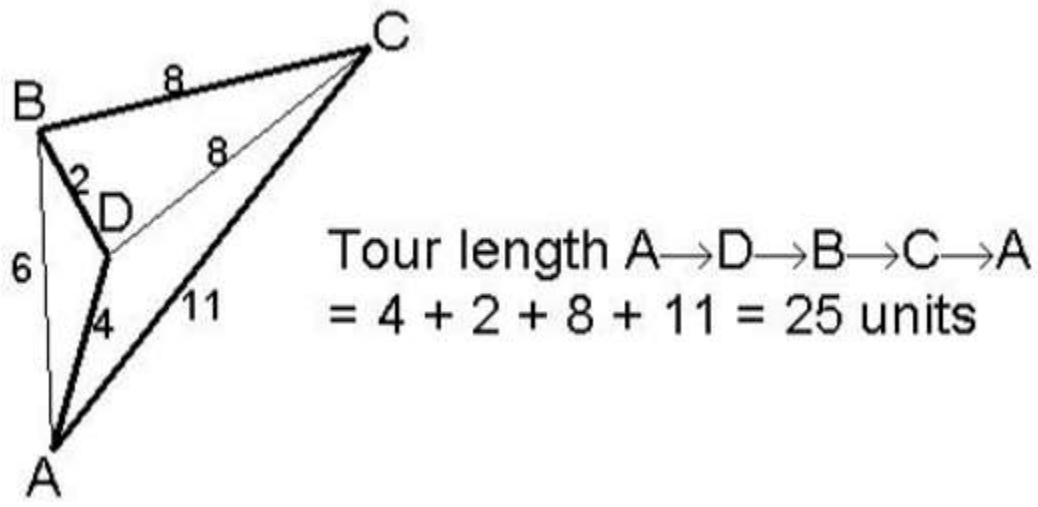
A connected, undirected, unweighted graph G has a Hamiltonian circuit if there is a way to link all of the nodes via a closed route that visits each node once, and only once.

It is strongly believed that there are no polynomial time algorithms for this problem.

The Traveling Salesman Problem (TSP)

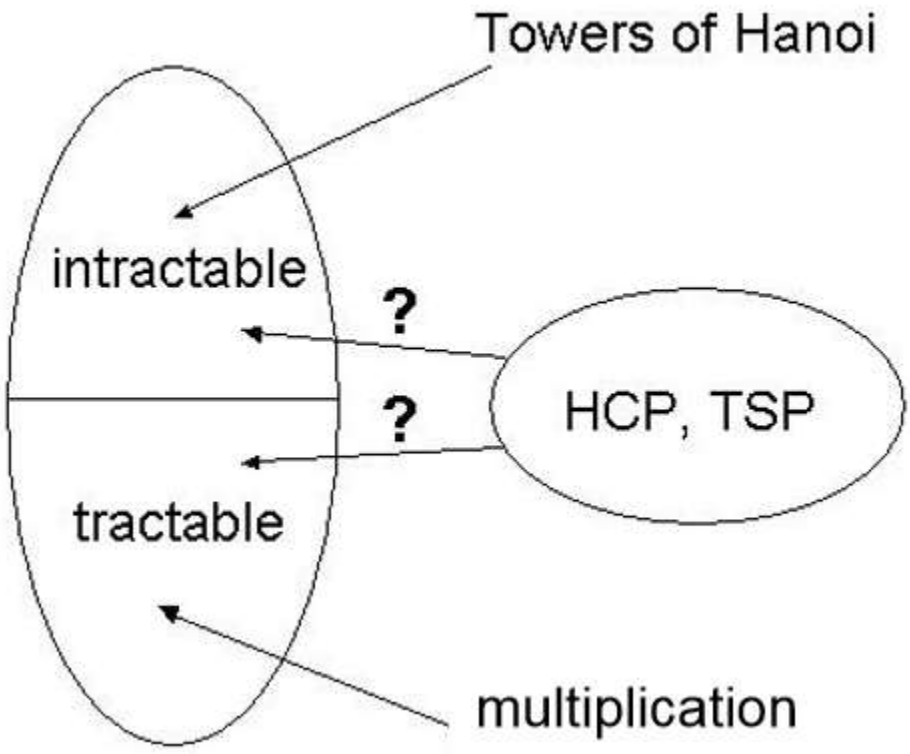
The TSP shares the extremely bad scaling behavior of the Hamiltonian circuit problem, and is one of the best-known examples of a problem in this 'probably intractable' class. This graph problem is similar to the Hamiltonian circuit problem in that it looks for a route with the same properties as required by the Hamiltonian circuit problem, but now of minimal length as well.

Given a connected, undirected, weighted graph (G, W), where W is the set of edge weights ('city distances'), The TSP seeks to find the shortest valid tour (a circuit visiting each node once and only once).



The Hamiltonian Circuit Problem and Traveling Salesman Problem like the Towers of Hanoi because although no-one has yet found a polynomial time algorithm for them, no-one has proved that no such algorithm exists.

The Hamiltonian Circuit Problem and Traveling Salesman Problem belong to the class **NP-Complete**, which is a subset of the larger problem class **NP**. NP-Complete is a class of problems whose time-complexity is presently unknown, though strongly believed to be super-polynomial, and can thus be thought of as being 'probably intractable'.



Thousands of problems are now known to have this probably-intractable character, including **optimization problems** such as the TSP, **scheduling problems** (such as the timetabling of lectures and exams), **decision problems** such as whether a map or graph can be colored in a certain way, whether an area of a given size can be covered by a specified set of patterned tiles, or if a logical assertion can be satisfied.

Polynomial time (p-time) reduction

In general that a problem A reduces in p-time to another problem B, written as

$$A \leq_p B$$

Means that there is some procedure, taking no more than polynomial time as a function of the size of the input to A, which

- Converts an input instance of A into an input instance of B
- Allows a suitable algorithm for problem B to be executed
- Provides a mechanism whereby the output obtained by this algorithm for problem B can be translated back into an output for problem A

The algorithm for problem B thus also provides a solution to problem A. Moreover A's solution will be obtained in a time which is in the same complexity class as the algorithm which solves B, since the extra work needed to translate is just in p-time. Most importantly, if we know – or in the case of NP and NP-Complete, suspect – that we have a lower bound on the time demand of all possible algorithms for B, we can say that in terms of its fundamental difficulty problem A is no worse than problem B.

There are three basic defining properties of problems in NP and NP-Complete

(i) **Problems in NP and NP-Complete are very hard to solve but easy to check**

The problems are hard because they appear to only admit algorithms whose time-demand behavior is described by super-polynomial functions.

However if a solution to a yes-instance of the problem is asserted then it can be checked in polynomial time; this ability to check a solution for correctness in polynomial time is referred to as a short certificate for the problem.

(ii) **Problems in NP-Complete are the hardest problems in NP**

An **NP-hard problem** (which may not itself be in NP) is one to which any problem in NP can be reduced in polynomial time:

If A is NP-hard, for all B in NP it is true that $B \leq_p A$ (be Reduced in p-time to A)

The class NP-Complete is the class of problems within NP itself which have this property: **NP-Complete = NP $\cap$ NP-hard**

(iii) **Problems in NP-Complete stand or fall together**

Any problem in the class NP-Complete can be shown to be **reducible in polynomial time** to any other problem in the class, meaning that there is a way in which any problem A can be mapped onto any other problem B using a number of steps taking no more than polynomial time such that a solution for B also provides a solution for A, and that the converse can also be done.

If A, B $\in$ NPC then $A \leq_p B$ *(A reduces in p-time to B)* and $B \leq_p A$ *(B reduces in p-time to A)*

The completeness property of the Non-deterministic Polynomial Complete problems – a solution to any one of them in this sense provides a solution to any other. It is the best-known property of problems in NP-Complete because it means that should a –time algorithm be found for just one problem in NP-Complete, then all NP-Complete problems would be solvable in p-time. Moreover if this were to happen all the problem in NP would be pulled in too; thousands of previously-intractable problems would then in principle become solvable in reasonable amounts of time.

P = NP?

This problem is the most famous problem in theoretical computer science.

To show P = NP

We have to find a polynomial time algorithm for any of the problems in NP-Complete. This also would provide, in principle, a polynomial algorithm for all problems in the class of NP.

To show P ≠ NP

We need to provide just one counterexample to the assertion P=NP would be sufficient to show that the sets P and NP are not in fact equal.

It is important to emphasize that the stand or fall together property applies only to problems in NP-Complete, and not to those in NP which are not also NP-Hard, those in the class NP\ NP-Complete.