P vs NP: A very short brief introduction

Created by: Dang Quang Vinh

Eindhoven University of Technology, the Netherlands

Version: 1.0

December 27, 2013

Summary

The article summaries some important points about P vs NP problem. P vs NP is quite famous, but it does not mean that everyone understands it well. I have known the problem for more than eight years, but still recent I have a concrete idea about it. The article may be useful for undergraduate (mostly) or postgraduate computer science students, or anyone related.

Terminology

This section describes some fundamental terminologies for P vs NP understanding.

Algorithm

You can find an official definition of algorithm easily in the Internet, but I will give my definition, for the easier understanding purpose

“An algorithm is a sequence of instructions to guide a computer do something”

Characteristics of an algorithm

The most important characteristics of an algorithm is: an algorithm is independent from running machine.

It means, even you run the algorithm on a Pentium II or newest computer, or even you perform an algorithm with paper and pencil, you should expect the same output with the same input.

Complexity

There are several kinds of algorithm complexity:

- Time complexity
- Space complexity
- Implementation complexity
- Etc

In this article, we only focus on time complexity.

Intuitively, time complexity of an algorithm is the running time of the algorithm: how much time the algorithm need to be done?

The time complexity of an algorithm in general case is not a constant, but depends on many factors. The common factor be used to calculate is the size of input and structure of input.

Again, intuitively, usually we can expect that, bigger and more “chaos” input, bigger time complexity.

To determine the time complexity of an algorithm, we focus to the worst case, i.e, the most “chaos” kind of input.

In this case, the time complexity is a function of input size.

Input Size

It is not easy problem as it seems to be, but to keep it simple, in this article (an many same – level documents), we can consider all elements of the input have the size 1, and the input size is the number of input elements.

For example, if an input is a list or an array, the input size is the length of the list or array, and we do not care what the individual size of individual elements is.

Big O Notation

We can represent the time complexity as a function of input size, as:

time = f (n)

We focus to the case n is very large (i.e, n $\rightarrow \infty$). Why?

Actually, if n is quite small, modern computers (even smartphone or tablets) can run almost algorithm quite fast enough.

We use Big O Notation to represent the time complexity of algorithm.

For example,

- If f(n) = 3 * n + 2 $\rightarrow$ the algorithm has the time complexity O(n)

Deterministic RAM model

Of course, we know that the processing time on different machines, or even on same machine at different times are different. So, to calculate the time complexity of an algorithm, we do not count the wall - clock running time, but count **how many instructions** the processer need to perform to finish an algorithm.

From practical view, we can implement an algorithm, deploy it on a computer, run and try to count the number of CPU instructions.

This practical approach has some disadvantages:

- Takes time and effort
    - Sometimes, it is not easy to implement an algorithm and run it.
- Depends on programming language we use
- Depends on operating system of the machine
- Even we can eliminate the effects of running environment, we cannot receive the final result, because we cannot run with every possible input (in general case).

To analyze the time complexity of an algorithm without implementation, we need a common machine model, and we can analyze how the algorithm runs on this machine.

Model

In deterministic RAM model, we have:

- Access RAM is free
    - For e.g, x = 3 or x = y is free operation
- Basic operation take 1 instruction:
    - For e.g, 1 + 1 or 2 * 3
    - In many cases, this assumption is not correct.
        - For e.g, divide two integer numbers with size ~ 1024 bits

Use these above assumptions, we can analyze the time complexity of an algorithm.

P vs NP

Polynomial and Exponential

In deterministic RAM model, there are many problems so far we cannot find a good algorithm.

A good algorithm means, this algorithm has the time complexity as a polynomial function, such as $O(n^k)$.

The best algorithms we know for these problems have the time complexity as $O(2^n)$.

We know that:

$$\lim_{x \to \infty} \frac{2^n}{n^k} = \infty$$

So, with n even is not really big, the running time may be come to millions of year, or more[1].

[1] We can make a simple example:
1 year = 365 days = 31 536 000 seconds ~ 2^{25} seconds
Consider an algorithm with time complexity 2^n, with n = 64 (size of chess board).
Assume that the computer can perform 2^{30} instructions per second (it is really optimistic assumption now, at least for personal computer, but we can expect with Moore's law) – you can read more about Titan super computer at: http://en.wikipedia.org/wiki/Titan_(supercomputer)

To solve these problems, we need a different computer model

Non Deterministic RAM model

The main different of this model, compared with Deterministic RAM model, is a new “if” statement.

In traditional computer (i.e, the computer we are using now in home or office), we have “if” statement like below:

```
if condition:
    do something
else:
    do other tasks
```

We always need to perform the condition to determine what branch we will go.

But in Non deterministic RAM model, the computer can know what branch is correct without condition computation. (i.e, the computer knows condition is true or false, without compute it)[2].

With this kind of computer (if we can build), we can run all exponential problems in traditional computers in polynomial time. Great!

Definition of P and NP

We define:

P is the set of all problems can be solved in deterministic RAM model in polynomial time (i.e, we can find an algorithm which run in our home computer in polynomial time).

NP is the set of all problems can be solved in non-deterministic RAM model in polynomial time.

NP – Complete

In 1971, Cook and Levin proved that, all problems in NP can be deducted into a single problem[3].

So, with this super computer, we need $2^{64}/2^{25}/2^{30} = 2^{9}$ years = 512 years
Remember that, for e.g, if n = 70 instead of 64, the running time is ~32 000 years

[2] It seems like a magic, but you can read more about quantum computer here: http://en.wikipedia.org/wiki/Quantum_computer

[3] http://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem

P vs NP

Easy to see that NP ≥ P[4].

The problem (and famous problem) is: > or =

It means, P = NP or P != NP (P != NP means NP > P)

To prove P = NP, you can pick any problem in NP (for example, vertex cover problem[5]) and find a new algorithm which can run on deterministic RAM model (again, it means your computer) in polynomial time.

To prove P != NP, you need to prove that all these problems (i.e, vertex cover and so on) cannot be solved in polynomial time with traditional computer (again, deterministic RAM model).

And, hope that you know about Millennium problem of ClayMath Institute:

http://www.claymath.org/millenium-problems/p-vs-np-problem

References

Wernicke, S., Bennett, S., Norell, S. (2013). *An Introduction to Theoretical Computer Science.* Udacity course.

Woeginger, G.J (2013). *P vs NP page.*

http://www.win.tue.nl/~gwoegi/P-versus-NP.htm

Accessed December 27, 2013

[4] If you do not know why, you can think about it.
[5] http://en.wikipedia.org/wiki/Vertex_cover