LECTURE 9 – ALGORITHM ANALYSIS AND COMPLEXITY CLASSES

Overview

- Time Complexity
- Comparing Algorithms
- The Classes P
- The Class NP
- The Class NP-Complete
- The Class NPI

Time Complexity

We have considered the class of solvable or computable problems and we have looked at some examples of unsolvable problems.

Problems are solvable or unsolvable.

We attempt to separate the practically solvable problems from those that cannot be solved in a reasonable time.

When constructing a computer algorithm it is important to assess how expensive the algorithm is in terms of storage and time.

Algorithm complexity analysis tells how much time does an algorithm takes to solve a problem.

The size of a problem instance is typically measured by the size of the input that describes the given instance.

Algorithm complexity analysis is based on counting primitive operations such as arithmetic, logical, reads, writes, data retrieval, data storage, etc.

$time_A(n)$ expresses the time complexity of algorithm A and is defined as the greatest number of primitive operations that could be performed by the algorithm given a problem instance of size n.

We are usually more interested in the rate of growth of $time_A$ than in the actual number of operations in the algorithm.

Thus, to determine whether it is practical or feasible to implement A we require a not too large upper bound for $time_A(n)$.

Time complexity of algorithms is expressed using the O-notation.

The complexity of a function f(n) is O(g(n)) if $|f(n)| \leq c \cdot |g(n)|$ for all n>0 and some constant c.

Let f and g be functions on n, then:

f(n) = O(g(n)) means that "f is of the order of g", that is, "f grows at the same rate of g or slower".

f(n) = Ω(g(n)) means that "f grows faster than g"

f(n) = Θ(g(n)) means that "f and g grow at the same rate".

An algorithm is said to perform in polynomial time iff its time complexity function is given by $O(n^k)$ for some $k \geq 0$.

That is, an algorithm A is polynomial time if $time_A(n) \leq f(n)$ and f is a polynomial, i.e. f has the form: $a_1n^k + a_2n^{k-1} + \ldots + a_m$

Example. $3x^2 + 2x + 2 = O(x^2)$ because

$$3x^2 + 2x + 2 \leq 5x^2 \text{ for } x > 1$$

$$3x^2 + 2x + 2 \leq 7x^2 \text{ for } x > 0$$

We say that $f \approx g$ (f is equivalent to g) if f = O(g) and g = O(f).

Example. $x^2 = O(3x^2 + 2x + 2)$ because

$$x^2 \leq 3x^2 + 2x + 2 \text{ for } x \geq 0$$

Then we could say that $3x^2 + 2x + 2 \approx x^2$.

An algorithm is said to perform in exponential time if its complexity function is not bounded by a polynomial but instead is given by $O(c^n)$ for some $c > 1$.

Comparing Algorithms

Let two algorithms A and B for solving a problem Π and let $time_A$ and $time_B$ be the time complexities of algorithms A and B respectively.

Algorithm A is said to have the same complexity as algorithm B in solving Π if:

$$time_A \approx time_B$$

Algorithm A is said to be better than algorithm B in solving Π if:

$time_A$ is $O(time_B)$ but $time_B$ is not $O(time_A)$

Example. Let algorithms A, B with time complexities as shown below. How do these two algorithm compare?

$$time_A(n) = \begin{cases} 1{,}000{,}000 & \text{for } n \leq 1{,}000{,}000 \\ n & \text{for } n > 1{,}000{,}000 \end{cases}$$

$$time_B(n) = \begin{cases} n & \text{for } n \leq 1{,}000{,}000 \\ n^2 & \text{for } n > 1{,}000{,}000 \end{cases}$$

$time_A(n) = O(time_B)$ but $time_B(n) \neq O(time_A)$

which means that algorithm A is better than algorithm B (although for practical purposes this may not be the case).

A problem Π is said to be tractable if there is a known polynomial time algorithm to solve it. A problem Π is said to be intractable if it is solvable but there is not known polynomial time algorithm to solve it.

Having a problem Π and an algorithm to solve it that is not polynomial time, then the option is to search for a better algorithm, i.e. a polynomial time one.

Example. Finding the maximum (or minimum) element of a list.

Suppose we have a list of n elements x_1, x_2,…, x_n and suppose we require an algorithm to pick up the maximum element. The following algorithms does this in n–1 steps:

Step 1: Let x = max (x_1,x_2)
Step 2: Let x = max (x,x_3)
and so on…
Step n–1: Let x = max (x,x_n)

Example. Finding the maximum and minimum elements of a list.

The obvious method is:

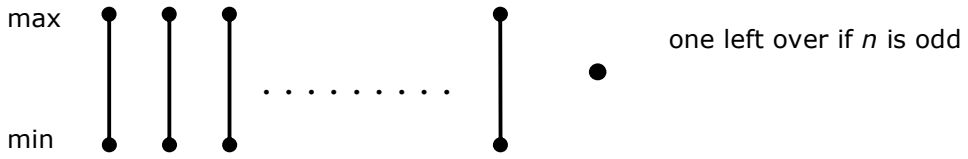
1. Apply the algorithm for finding the maximum element in the list, which takes a total n–1 steps, and then

2. Apply the algorithm for finding the minimum element in the remainder of the list, which takes a total of n–2 steps.

Then the total number of steps is 2n–3 steps.

An alternative method is as follows:

1. Divide the list into disjoint pairs (with one left over if n is odd).
2. Apply max(x,y) to each pair and arrange as shown below, for this we have made $\left\lfloor \frac{n}{2} \right\rfloor$ comparisons.



Now the greatest element of the list is in the top and the least is on the bottom of the arrangement.

3. Apply the algorithm for finding the maximum(minimum) element in the list, to these subsets (with the one left over if n is odd).

Then the total number of comparison is:

If n is even: $\frac{n}{2} + 2(\frac{n}{2} - 1) = \frac{3n}{2} - 2$

If n is odd: $\frac{n}{2} - 1 + 2(\frac{n-1}{2} - 1) + 1 = \frac{n-2}{2} + 2(\frac{n-3}{2}) + 1 = \frac{3n}{2} - 3$

Which is about $\frac{3}{4}$ of the number of steps required with the obvious method.

Example. Determine the time complexity of the binary addition.

Suppose two binary numbers with n digits, $a = a_{n-1}a_{n-2}\ldots a_0$ and $b = b_{n-1}b_{n-2}\ldots b_0$.

Then the sum is given by $a + b = d = d_n\, d_{n-1}\, d_{n-2}\ldots d_0$.

Each d_i is obtained by adding and carrying in the usual way:

$d_i = a_i + b_i + c_i \bmod 2$ where c_{i-1} is the carry from the previous sum.

It is impossible to calculate all the d_i in parallel because of the carry.

Each d_i must await its turn in the order of computation.

Therefore, the complexity of the standard binary addition algorithm is O(n).

The Class P

Given the decision TSP: for a set of n cities $C=\{c_1,c_2,\ldots,c_n\}$ and a distance between each pair of cities c_i and c_j given by $d_{ij} \in \mathbb{Z}^+$. Is there a tour of all cities, starting and ending in city c_1 and visiting each city exactly once, such that the total distance travelled $\leq b$?

The number of possible tours is (n-1)!

A deterministic machine would be able to explore only one tour at a time so it would take exponential time to solve the decision TSP.

There is no known deterministic machine that can solve the decision TSP in polynomial time.

Let be M a deterministic Turing machine and input $x \in \Sigma^*$.

The time complexity $time_M$ of a deterministic Turing machine M is given by

$time_M(n) = \max\{m \mid$ there is an $x \in \Sigma^n$ such that the computation of M on input x is of length m$\}$

If $time_M(n)$ is $O(n^k)$ for some $k \geq 0$ then M is said to take polynomial time.

The Turing machine M is an algorithm is if halts for all inputs and then $time_M(n)$ is defined for all n.

Then, the class **P** is defined as the class of problems which can be solved by polynomial time deterministic algorithms. Problems in **P** can be solved 'quickly'.

P = { L | there is a polynomial time deterministic TM which accepts L}

Every language in **P** is a recursive language.

The Class NP

For the decision TSP stated above, a non-deterministic machine (with unlimited parallelism) would be able to solve the decision TSP in polynomial time.

Let be NDTM a non-deterministic Turing machine and input $x \in \Sigma^*$.

There is at least one computation sequence (possibly more) by which NDM accepts x.

The time taken by NDTM to accept x is given by the length of the shortest of the computation sequences by which NDTM accepts x.

The time complexity $time_{NDM}$ of a non-deterministic Turing machine NDTM is given by:

$$time_{NDM}(n) = \begin{cases} 1 & \text{if no inputs on length n are accepted by NDTM} \\ \min\{m \mid \text{there is an } x \in \Sigma^n \text{ such that the time taken by NDtM to accept x is m }\} & \text{otherwise} \end{cases}$$

We say that NDTM is polynomial time iff $time_{NDM}(n)$ is $O(n^K)$ for some $k > 0$.

Then, the class **NP** is defined as the class of problems which can be solved by polynomial time non-deterministic algorithms. Problems in **NP** can be verified 'quickly'.

NP = { L | there is a polynomial time non-deterministic TM which accepts L}

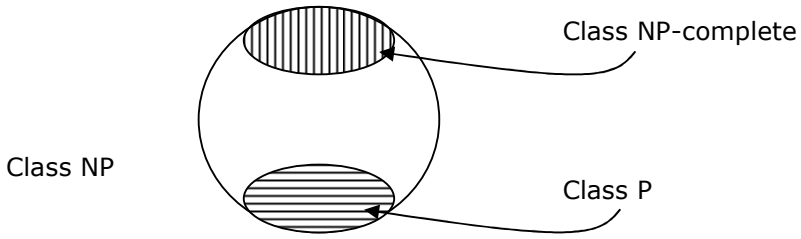
It should be clear that **P** $\subseteq$ **NP**.

It is widely believed that P $\neq$ NP but this has not been formally proven.

NP-Complete problems arise in many domains like: boolean logic; graphs, sets and partitions; sequencing, scheduling, allocation; automata and language theory; network design; compilers, program optimization; hardware design/optimization; number theory, algebra.

The Class NP-complete

Then, within the class **NP** there are easy problems, those in **P**, and very much harder problems called **NP-complete** problems.



Intuitively, the class **NP-complete** are the hardest problems in **NP** as defined by Cook's theorem.

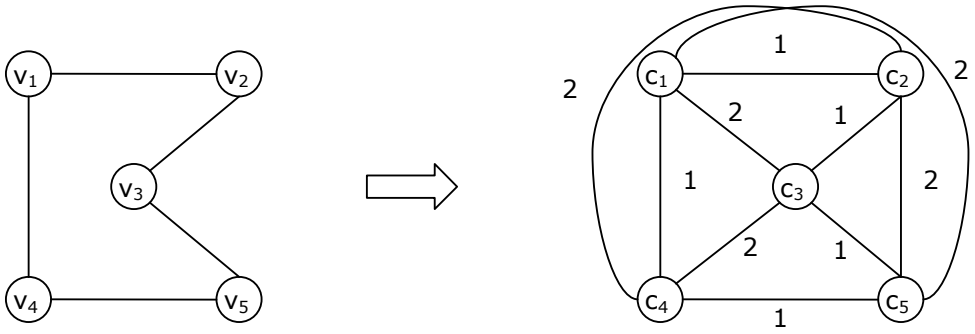
More formally, problem Π is **NP-complete** if $\Pi \in$ **NP** and for any $\Pi' \in$ **NP** $\Pi' \propto \Pi$.

Assume two problems Π_1, Π_2 and an algorithm A to solve Π_2. If each instance of Π_1 can be easily (in polynomial time) transformed into some instance of Π_2 then we could use algorithm A to solve Π_1.

Example. Consider the decision TSP as being Π_2 and let the Hamiltonian circuit decision problem (HC) be Π_1. The HC decision problem: given a graph G, does G contain a Hamiltonian circuit? i.e. a simple cycle connecting all its vertices?

we need: $f: I_{HC} \rightarrow I_{TSP}$

Transformation: an instance of the HC, G = (V,E), can be transformed to an instance of the TSP by representing the TSP instance as G = (V,E) where $C=\{c_1,c_2,\ldots,c_n\} = V$ and the distance between cities c_i and c_j, is $d_{ij} = 1$ if v_i and v_j are connected in G or $d_{ij} = 2$ if v_i and v_j are not connected. Then b = n = |V|.



HC instance TSP instance

We say that problem Π_1 reduces polynomially to problem Π_2, written $\Pi_1 \propto \Pi_2$ if there is a function: $f: \Pi_1 \rightarrow \Pi_2$ and f is computed in polynomial time.

Theorem. If $\Pi_1 \propto \Pi_2$ (respectively $L_1 \propto L_2$) and $\Pi_2 \in$ P then $\Pi_1 \in$ P. Two problems Π_1 and Π_2 (respectively L_1 and L_2) are polynomially equivalent if $\Pi_1 \propto \Pi_2$ and also $\Pi_2 \propto \Pi_1$ (respectively $L_1 \propto L_2$ and $L_2 \propto L_1$).

All evidence so far is that **P ≠ NP** so it seems that no **NP-complete** problem has a known polynomial time algorithm to solve it.

Theorem. If Π_1 is NP-complete and $\Pi_2 \in$ NP then $\Pi_1 \propto \Pi_2$ implies that Π_2 is also NP-complete.

The first problem known to be NP-complete (proof by Cook in 1971) was the satisfiabiability of a set of clauses (SAT). The proof shows first that SAT $\in$ **NP** and then it shows that for all $\Pi' \in$ **NP** then $\Pi' \propto$ SAT.

The SAT problem: given U, a finite set of Boolean variables and C, a finite set of clauses over U, is there an assignment of values to the set of variables U that satisfies the set of clauses C to be true?

Example. An instance of SAT is: given the Boolean variables $U = \{u_1, u_2, u_3\}$ and the set of clauses $C = \{\{u_1 \vee u_2 \vee u_3\} \wedge \{\bar{u}_2 \vee \bar{u}_3\} \wedge \{u_2 \vee \bar{u}_3\}\}$.

The set of clauses is satisfiable by any of the following assignments:

u_1	T	T	F
u_2	T	F	T
u_3	F	F	F

Example. An instance of SAT is: given the Boolean variables $U = \{u_1, u_2, u_3\}$ and the set of clauses $C = \{\{u_1 \vee u_2\} \wedge \{u_1 \vee \bar{u}_2\} \wedge \{\bar{u}_1\}\}$. Is the set C satisfiable?

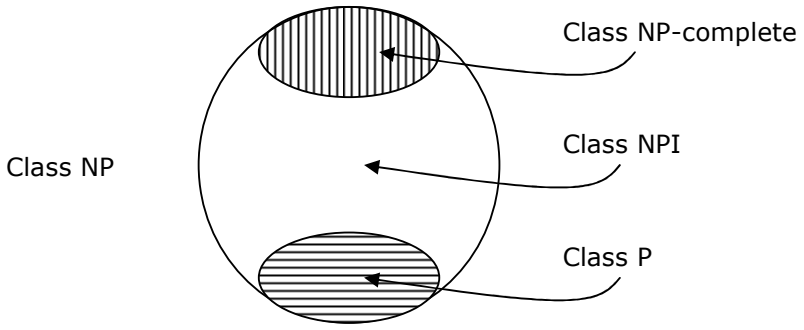
The Class NPI

Accepting that **P ≠ NP** does not mean necessarily that **P** ∪ **NP-complete** = **NP**.

Some problems might not be in **P** neither in **NP-complete**.

It has been shown that if **P** = **NP** then **NPI** = **NP** – (**P** ∪ **NP-complete**).

However, the vast majority of practical problems are in **P** or in **NP-complete**.



Example. It is a conjecture that the decision problems: Graph Isomorphism and Composite Number are both in **NPI**.

Graph isomorphism: given two graphs G_1 and G_2, is G_1 isomorphic to G_2?

For the special case of G_1,G_2 being planar, the problem is solved in polynomial time.

But no polynomial time algorithm has been found to solve the general case. And no one has been able to prove that Graph Isomorphism is **NP-complete**.

Composite Number: given a positive integer k, are there integer m,n > 1 such that k = m·n?

If any polynomial time algorithm can be designed to test primality then the problem would be in **P** but such an algorithm has not been found.

However, there is a polynomial time algorithm for solving the Composite Number problem when k is not a prime number.

Reading: (Rayward-Smith, chapter 6), (Lewis&Papadimitriou, chapter 6)