02249- Computationally hard problems- 09-09-2020

Many problems in real life are hard.
To describe problems in unambiguous way we need FORMAL LANGUAGES.

Slide 7
We looked at a scan algorithm to solve exactly the work problem, we talked about running time as atomic computational steps.
Checking a letter, counting up position and moving one step further.
Everything that can be done in constant time :
Computing a logarithm cannot be done in constant time (approximates a serie and makes a computation).

Slide 8-13
The algorithm is a abstract description of a problem: algorithm has a certain running time defined by T= time Â= algorithm (X = input). If the loop is infinite then T is infinite. This is a very specific definition. Since its the running time of the algorithm over a certain specific Input X.

The worst-case running time $T_w^A(n)$ of algorithm A for input size n is the longest running time of the algorithm on an input of size n:

$$T_w^A(n) := \max\{T^A(X) \mid \|X\| = n\}$$

One often considers the worst case prospective
W = worst case
(n) = all input of size n

The best-case running time $T_b^A(n)$ of algorithm A for input size n is the shortest running time of the algorithm on an input of size n:

$$T_b^A(n) := \min\{T^A(X) \mid \|X\| = n\}$$

One rarely considers the best case running time

The definition of average-case running time $T_a^A(n)$ of algorithm A for input size n assumes a distribution P_n on the inputs of size n. It then is the average with respect to P of all running times on inputs of size n:

$$T_a^A(n) :=\ P_n(X) \cdot T^A(X) \ X{:}\ \|X\|{=}n$$

The running time is weighted on the probability of having that running time. The average running time depends on the distribution P in practice. And it is difficult to define.

// MISSED CONTACT//

Asymptotic Notation -slide 14

O-notation. Gives us the order of growth of the running time.

How does the run time scale with the size of the input? We are going to consider the order of growth.

Definition

1. Let $f(n)$ and $g(n)$ be non-negative functions of n. We say $f(n) = O(g(n))$ if $f(n) \leq c \cdot g(n)$ for some constant $c > 0$ and all $n \geq$

We are going to consider non negative function (running time is never negative).

$n=O(n)$,

$n=O(n)$,

$n/2=O(n)$,

$2n=O(n)$,$n+ n=O(n)$

NOT $n \log n = O(n)$.

$n^2+nlogn=O(n^2)$,

$1000nlnn=O(nlnn)$.

$2^n = O(2^{n^2})$

NOT $2^n = O(n^{1000})$ or any other exponent

We can’t bond exponential run time with a polynomial, it is not enough put a constant in front of a polynomial to bound exponential growth.

Kahoot
n(2.5) + n^3 /(loglogn) +(long n is O(n^3) is O(n^3) since it is the dominant term. log log n is smaller, so the growth of n^3 /loglog n can be bounded by O(n^3)

Running time	Name
$O(1)$	constant
$o(n)$	sub-linear
$O(n)$	linear
$O(n \log(n))$	—
On^2	quadratic
On^3	cubic

$On^{constant}$	polynomial
$On^{log(n)}$	super-polynomial / sub-exponential
$O(c^n)$	exponential (e. g. $O(2^n)$)

If the exponent of the running time depends on n, we are in shit.
POLYNOMIAL == GOOD
SUPER POLYNOMIAL/ SUB EXPONENTIAL == NOT GOOD
EXPONENTIAL == BAD

In theory n^constant (polynomial) is efficient, in practice the c counts, meaning that not all c in practice are really doing the work.

Which function is bounded by a polynomial? 1 ad (n^3 /log n) —> this is bounded with n^3 an bigger than 1
n^(log n)is exponential non polynomial
2(sqrt(n)) is exponential non polynomial

We really like polynomial, because of their properties.

Let $p(n)$ and $q(n)$ be two polynomials. Then:

1.Their sum $p(n) + q(n)$ is a polynomial.

2.Their product $p(n) \cdot q(n)$ is a polynomial.

3.Their composition $q(p(n))$ is a polynomial. 23236

Example:$p(n)=n$,$q(n)=n$:Then$q(p(n))=(n\ ) =n$.

Property 2 is important:we can run an algorithm with polynomial worth case running time as many times! MULTIPLICATIVE SCALING

An algorithm with polynomial worst-case running time is considered efficient (in theory)

algorithm with polynomial running time n^c can – in the same time as before – solve problems that are a factor c_p larger: $N \to c_p \cdot N$, where N is the problem size.

Running time is linear, double speed of computer: the run time scales with multiplicative factor (41%)

Alg with exponential worst-case running time is NOT considered efficient: ADDITIVE SCALING ONLY
If the exponential run time, with a double speed we only have an addictive term

algorithm with exponential running time c^n can – in the same time as before – solve problems that are a additive term c_e larger, $N \to N + c_e$.

WHEN IS AN ALGORITHM EFFICIENT?

Let A be an algorithm. We call A efficient if its worst-case running time is upper bounded by a polynomial, i. e., if there is a polynomial p such that

$\forall n \in N$: $T_w{}^A(n) \leq p(n)$.

Note that an algorithm with worst case running time $p(n)$ might be much faster on some problem instances of size n.
This is due to the specific structure of these problem instances.

Of course on some inputs of size n the algorithm can stop earlier (remember the scan algorithm with linear running time), that is why we talk about the worst case scenario.

OPTIMISATION PROBLEMS AND DECISION PROBLEMS

Optimization problem: gimme the optimal solution to for example maximise the work span, or gimme the minimal path. This is also called "search problem".
Decision problem: formulated in a way you have only a binary answer: yes or no.

It is often possible to solve an optimisation problem by solving a number of related decision problems

Problem [Sorted]
Input: A sequence $a_1, \ldots, a_n$ of n integers.

Output for the optimizing version: A increasingly sorted sequence consisting of the input integers.

Output for the decision version: YES if the sequence is increasing, i. e., if $a_1 \leq a_2 \leq \cdots \leq a_n$ and NO otherwise.

THE CLIQUE PROBLEM

For an undirected graph $G = (V, E)$ a clique is a subset $V' \subseteq V$ of the vertices such that all ′

edgesarepresent,i.e.,forall$v,w \in V$,$v /= w$:$\{v,w\} \in E$.WesaythatGhasak-cliqueif $|V'| = k$.

V' (vi prime)
Lets see the two versions of the same problem:

Problem [Clique]

Input for the optimizing version: An undirected graph G.

Output for the optimizing version: A set of vertices which forms a clique of maximum size.

Input for the decision version: An undirected graph G and a number k.

Output for the decision version: YES if G contains a clique of (at least) k vertices and NO otherwise.

In the decision version, the input will not only have the object itself but also k.

Take a bunch of vertices.
Take each pair in the bunch.
For every pair of vertices the interconnected edge is present in the subgraph.

SATISFIABILITY

Problem [Satisfiability]
Input: A set of clauses over n boolean variables.

Output for the decision version: YES if there is an assignment of truth values to the variables which satisfies all clauses together and NO otherwise.

Output for the optimizing version: An assignment of truth values to the variables which satisfies all clauses together if such an assignment exists and NO otherwise.

Ex1
C1 = X1 OR X2
C2 = X1 OR NOT(X2)
C3 = NOT(X1) OR X2
C4 = NOT(X1) NOT (X2)

Is not possible to satisfy all 4 clauses at once.
If the answer would be YES than we want to have the satisfying assignment.

Ex2

Consider the following set C with three clauses:

$C = \{c_1, c_2, c_3\}$
$c_1 = x_1 \vee x_3 \vee x_5 \vee x_4 \vee x_2$

$c_2 = x_2 \vee x_4 \vee x_5$
$c_3 = x_2 \vee x_3 \vee x_5 \vee x_4$

The following is a possible satisfying assignment (there are others):

$x_1 = \text{false}$ $x_4 = \text{false}$ $x_5 = \text{true}$

The following is a possible satisfying assignment (there are others):

x2 = true

One can define a decision problem with A FORMAL LANGUAGE.

To formally define what a decision problem is we use formal languages.
The Satisfiability problem can thus be written as the language L_{SAT} over the alphabet

$\Sigma_{clauses}$:
Note that L_{SAT} is a proper subset of $L_{clauses}$.

General problem is the Language, the problem instance which is a word.
We are going to solve LSAT!
(See diagram)

A decision problem P is a language L_P over an input alphabet Σ.

We say that an algorithm A solves the problem P if it solves the word problem for L_P . That is, if it decides whether a word $w \in \Sigma^*$ is in L_P or not. We call A a decision algorithm.

Decision problems question:
Determine whether a graph is a triangle is a decision problem, because is the only one that you can answer with yes or no.

Schedule jobs in parallel
Sort n natural numbers
Compute a satisfying assignment
Are NOT DECISION problems, but OPTIMISATION problems

IN REALITY DECISION PROBLEMS DO NOT GIVE ARE A REAL SOLUTION, WHEREAS THE OPTIMISATION PROBLEMS CAN.
Can one make the other?
Sometimes YES.

Lets assume we have a fast algorithm for decision problems, than we can build an efficient algorithm for the optimisation problem. NOTE: we often NOT have the decision problems efficient black box/subrutine.

The decision version is not so useful; it does not give a “real” solution.

Can a solution to the decision version be used to solve the optimization version?

Often: YES!

How: By solving a number of decision versions.

The solution algorithm for the decision version is used as a black box (sub-routine).

The conversion is called efficient if it is polynomial in the input size, where every use of the black box counts one step.

Hence, if the black box can be realized with polynomial running time, the total running time is polynomial.

HOW TO TRANSFORM A DECISION ALG TO A OPTIMISATION PROBLEM:

Example: Clique

Suppose we have a decision algorithm A_d for Clique.

Given input (G, k), it will answer YES iff G has a k-clique.

We use A_d as a black box, i. e., we know its input-output semantics but not how it internally works.

We are given a graph $G = (V, E)$ and want to find a clique of maximum size (there might be many).

We solve this problem by feeding the black box with different inputs.

Algo is in two phases:

Example: Clique

Phase 1: Determine the size of a maximum clique by binary search.

Set $k \leftarrow n$, where $n = |V|/2$.

Call the black box A_d with input (G, k).

If $A_d(G, k) = YES$ then set k to the average of the previous k and an upper bound on the largest clique size (initially n)

else set k to the average of the previous k and a lower bound on the largest clique size (initially 0).

Proceed accordingly, updating upper and lower bounds until the interval of possible values has been narrowed down to 1 element.

Then k is the size of a maximum clique.

Alternatively, one could use a linear search, counting k down from n until A_d answers YES for the first time.

1- solve another optimisation problem: max size of a clique. In principle we want the vertices, for now we focus on the number of the vertices.

We use binary search (also linear would work but slower), we start in the middle. If n = 1000 we ask is the clique of 500? YES or NO.
YES: is the clique of size 750? (3/4 *n)YES/NO
NO: is the clique of size 250? YES/NO

Updating upper and lower bound, we half the size of the range with binary search, so it runs in log

(n)
Now we know the size of the largest cliques, eg 5

K IS THE LARGEST CLIQUE SIZE

2- Now we need to find the edges of the clique.

Example: Clique

Phase 2: Find the edges forming a maximum clique (their endpoints are the vertices of a maximal clique).

1) RemoveanedgeefromG(G=(V,E\{e}))

2) Call the black box A_d with (G, k).

3a) If $A_d(G, k) = YES$ then G still contains a k-clique.

Goto 1), i. e., remove another edge.

3b) If $A_d(G, k) = NO$, then the last edge removed, say e , was a member of all remaining

k-cliques.

4) Pute$'$ backintoG,(G=(V,E∪{e$'$}).

Mark e' as permanent, i. e., it will never be removed again.

5) Goto 1) and iterate 1) through 5) until $k(k-1)/2$ edges are permanent. The endpoints of these edges form a k-clique.

Sort the edge in a whatever way.
Pick any edge, remove it.
Use the black box routine and ask if the clique of size k is still there after removing the edge.

We are going to find only one clique, cause there could be too many to write them down.

Show that the algorithm is efficient == running on polynomial running time

---

Example: Clique

Correctness:

Phase 1: The binary search correctly determines the size of a maximum clique.

Phase 2: As long as $A_d(G, k) = YES$ the current graph G has a k-clique.
If $A_d(G, k) = NO$, then the previous call of the black box resulted in YES.

Thus every permanent edge is in some k-clique.

The permanent edges form one k-clique:

Suppose the permanent edges e_1, e_2 are in different k-cliques.

Then the answer after removing e_1 (or e_2) would have been YES and that edge would have been removed forever.

The algorithm will not make permanent two edges part of two different cliques.
It is not possible that permanent edges are in different k-cliques.

Running time?

Example: Clique

Running time:

"Phase 1", the binary search, issues at most $\lceil \log|V|\rceil$ calls to A_d. All operations of Phase 1 run in constant time.

Analysis of "Phase 2": Graph G originally contains at most $n(n-1)/2$ edges, $n = |V|$.

Every edge is considered exactly once in 1) – 4).

The time for 1) – 4) is constant, assuming that the call to the black box A_d is one computational step.

The total time is $O(n^2)$.

Summing up: The optimization version of Clique can be solved in polynomial time with polynomially many calls of the decision algorithm.

T = O(log(V)) + O(n(n-1)/2) = O(n^2)
The time is polynomial.

Can we remove vertices instead of edges? Yes, we remove nodes. Is still a clique of size k? YES, gone. NO, restore forever. With the node we also remove all its edges (so the rung time analysis becomes slightly more complicated).

Note of caution!

Decision vs. Optimization: One Note of Caution

When using a decision algorithm as subroutine, make sure you call it with an input that fits the problem. Examples:

If the input is an undirected graph, do not give a directed graph to the subroutine.

If the input is a set of clauses consisting of negated or unnegated variables, you cannot give additional restrictions such as “set x_1 to true”. Make sure to use a set of clauses, nothing else.

Please keep this in mind for the exercises.

Next week: randomised algorithms for defining hardness of problems.

DETERMINISTIC ALGORITHMS: same output when running it over and over again, with same running time. Nice and desired.

RANDOMISED ALGORITHMS: really useful and not so unreliable. Different output with same input and different running time. The algorithm might even not stop at all! This is not nice. But one can prove that is very unlikely that the algorithm gives a wrong answer.

Eg: fair coin = device that gives a random bit (0 or 1) INDIPENDENTLY ACCORDING TO UNIFORM DISTRIBUTION.
P = 0.5
Calling the Radom number generator Is one computational step.

Random Number Generator:
Rand (a,b) random number drawn between a and b.

It is a difficult task, not trivial. Real random numbers are by looking at natural processes. We instead use pseudorandom algorithms.