

Computationally Hard Problems

Algorithms and Running Times

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

Running Time

For the discussion of the running time we use the following example: Consider words (strings) over the alphabet $\Sigma = \{a, b\}$.

Let $L_{\geq 2b} \subseteq \Sigma^*$ be the language of all strings containing at least 2 letters b .

We want to check whether a string $X \in \Sigma^*$ is in $L_{\geq 2b}$, i.e., contains at least two b .

Algorithm **Scan** does this by scanning X from left to right and stopping at the second b saying “bingo”, or “NO” when reaching the end of X without finding two b .

One computational step is checking a letter and moving one position right (can be done in constant time).

Running Time

Definition

Let A be an algorithm.

- ▶ The *running time* $T^A(X)$ of an algorithm A for a fixed input X is the number of atomic computational steps that A performs on input X until it stops
- ▶ If A does not stop on X then $T^A(X) = \infty$.

Example:

$$T^{\text{Scan}}(\text{bbaaaba}) = 2 \quad T^{\text{Scan}}(\text{aabaaabbaab}) = 7$$

$$T^{\text{Scan}}(\text{aaaaaaaaaaaaaaaaaaaaaaaaaab}) = 26$$

Running Time, Worst Case

Definition

- ▶ 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 :

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

Example

One worst-case input is $\mathbf{X} = \underbrace{\text{aaaaa} \dots \text{aaab}}_n$

$$T_w^{\text{Scan}}(n) = n$$

Running Time, Best Case

Definition

- ▶ 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 :

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

Example

One best-case input is $\mathbf{X} = \underbrace{\text{bbaaa} \dots \text{aaaa}}_n$

$$T_b^{\text{Scan}}(n) = 2 \text{ if } n \geq 2$$

Running Time, Average Case (1/3)

Definition

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 :

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

Note that the average running time depends on the choice of the distribution P . Often there is no canonical way to define P .

Running Time, Average Case (2/3)

Example

- ▶ If all strings $X \in \Sigma^n$ are equally likely then “most of them” will have two b among the “first few” letters.
- ▶ Then the **Scan** most of the times stops after a few steps.
- ▶ Here, the average running time of **Scan** is constant.
- ▶

$$T_a^{\text{Scan}}(n) \leq c \text{ for some } c \text{ that does not depend on } n$$

- ▶ Arguments making this rigorous will be given later.

Running Time, Average Case (3/3)

Example

- ▶ Assume that only three strings of length n have a non-zero probability

$$X_1 = \text{abbbbaa} \dots \text{a} \dots \text{aaa} \quad P(X_1) = 0.2$$

$$X_2 = \text{aabbaa} \dots \text{a} \dots \text{aaa} \quad P(X_2) = 0.5$$

$$X_3 = \text{aaabaa} \dots \text{a} \dots \text{aaa} \quad P(X_3) = 0.3$$

- ▶ Then $T^{\text{Scan}}(X_1) = 3$, $T^{\text{Scan}}(X_2) = 4$, $T^{\text{Scan}}(X_3) = n$



$$T_a^A(n) = 0.2 \cdot 3 + 0.5 \cdot 4 + 0.3 \cdot n = 2.6 + 0.3n$$

Asymptotic Notation

We often express the running time in O -notation.

Definition

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 $n \geq 1$.

Examples:

- ▶ $n = O(n)$, $\sqrt{n} = O(n)$, $n/2 = O(n)$, $2n = O(n)$, $n + \sqrt{n} = O(n)$ but NOT $n \log n = O(n)$.
- ▶ $n^2 + n \log n = O(n^2)$, $1000n \ln n = O(n \ln n)$.
- ▶ $2^n = O(2^{n^2})$ but NOT $2^n = O(n^{1000})$ or any other exponent

We do not care much about $o(\dots)$, $\Omega(\dots)$, ... at the moment.

Kahoot

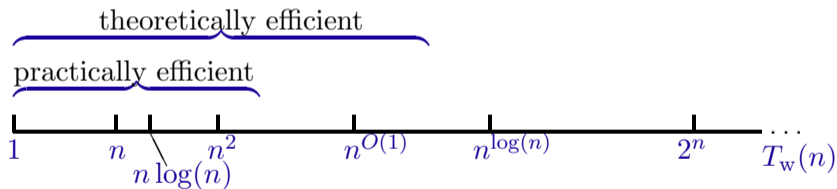
The Zoo of Running times

| Running time | Name |
|--------------------------|------------------------------------|
| $O(1)$ | constant |
| $o(n)$ | sub-linear |
| $O(n)$ | linear |
| $O(n \log(n))$ | — |
| $O(n^2)$ | quadratic |
| $O(n^3)$ | cubic |
| $O(n^{\text{constant}})$ | polynomial |
| $O(n^{\log(n)})$ | super-polynomial / sub-exponential |
| $O(c^n)$ | exponential (e. g. $O(2^n)$) |

The Zoo of Running times

In theory, all polynomial running times are efficient.

In practice, the degree of the polynomial is important.



Kahoot

Properties of Polynomials

The class of polynomials has useful properties.

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

- ▶ Their sum $p(n) + q(n)$ is a polynomial.
- ▶ Their product $p(n) \cdot q(n)$ is a polynomial.
- ▶ Their *composition* $q(p(n))$ is a polynomial.

Example : $p(n) = n^2$, $q(n) = n^3$: Then $q(p(n)) = (n^2)^3 = n^6$.

Especially the second property is important today: Doing a polynomial amount of work polynomially many times is still a polynomial amount of work.

Polynomial versus Exponential

- ▶ An algorithm with polynomial worst-case running time is considered *efficient* (in theory).
- ▶ An algorithm with exponential worst-case running time is NOT considered efficient.
- ▶ Why?
- ▶ Assume that the speed of a computer is doubled.
- ▶ Then an algorithm with polynomial running time n^c can – in the same time as before – solve problems that are a **factor** c_p larger: $N \rightarrow c_p \cdot N$, where N is the problem size.
- ▶ Then an algorithm with exponential running time c^n can – in the same time as before – solve problems that are a **additive term** c_e larger, $N \rightarrow N + c_e$.
- ▶ Further reasons: machine models (out of scope for this course).

Efficient Algorithms

Summing up:

Definition

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

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

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.

Computationally Hard Problems

Decision and Optimization Algorithms

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

Optimization Problems and Decisions Problems

An *optimization problem* (sometimes: “search problem”) is one where one asks for the optimal (e. g., maximal, minimal, largest, smallest, ...) solution.

A *decision problem* is formulated in such a way that the answer is either YES or NO.

Most of the time a decision problem has an associated optimization problem.

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

We shall consider decision problems in the first part of the course because they are formally easier to handle.

Example: The Sorting Problem

Problem [SORTED]

Input: A sequence a_1, \dots, 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 $a_1 \leq a_2 \leq \dots \leq a_n$ and NO otherwise.

Example: Clique

For an undirected graph $G = (V, E)$ a *clique* is a subset $V' \subseteq V$ of the vertices such that all edges are present, i. e., for all $v, w \in V'$, $v \neq w$: $\{v, w\} \in E$. We say that G has a k -clique if $|V'| = k$.

Problem [CLIQUE]

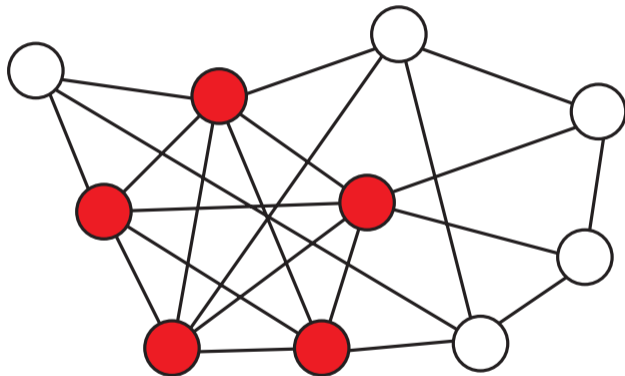
Input for the decision version: An undirected graph G and a number k .

Input for the optimizing version: An undirected graph G .

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

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

Example: Clique



The red vertices form a clique of size 5.

Example: 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.

Example: Satisfiability

Consider the following set C with three clauses:

$$C = \{c_1, c_2, c_3\}$$

$$c_1 = \bar{x}_1 \vee \bar{x}_3 \vee x_5 \vee x_4 \vee x_2$$

$$c_2 = x_2 \vee \bar{x}_4 \vee \bar{x}_5$$

$$c_3 = x_2 \vee \bar{x}_3 \vee x_5 \vee x_4$$

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

$$x_1 = \text{false}$$

$$x_4 = \text{false}$$

$$x_5 = \text{true}$$

The rest of the variables can receive any value.

Decision Problems: A Formal Definition (1/2)

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 Σ_{clauses} :

$$L_{\text{SAT}} := \{w \in L_{\text{clauses}} \mid w \text{ has a satisfying assignment}\}$$

Note that L_{SAT} is a proper subset of L_{clauses} .

One has to (or at least should) distinguish between:

- ▶ The general *problem* which a language.
- ▶ Example: SATISFIABILITY = L_{SAT} .
- ▶ A specific problem instance which is a word in a (not necessarily the same) language.
- ▶ Example: $X \in L_{\text{clauses}}$.

Decision Problem: A Formal Definition (2/2)

Definition

- ▶ 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*.

Kahoot

General Thoughts

- ▶ 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.

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.

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 $A_d(G, k) = \text{YES}$ 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.

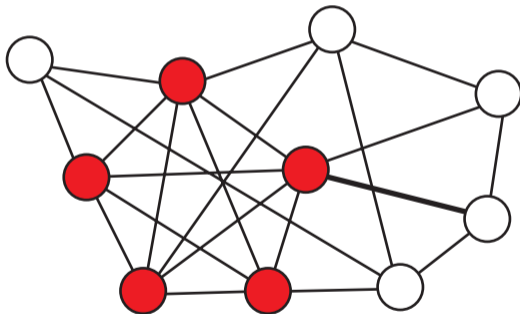
Example: Clique

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

- 1) Remove an edge e from G ($G = (V, E \setminus \{e\})$)
- 2) Call the black box A_d with (G, k) .
- 3a) If $A_d(G, k) = \text{YES}$ then G still contains a k -clique.
Goto 1), i. e., remove another edge.
- 3b) If $A_d(G, k) = \text{NO}$, then the last edge removed, say e' , was a member of all remaining k -cliques.
- 4) Put e' back into G , ($G = (V, E \cup \{e'\})$).
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.

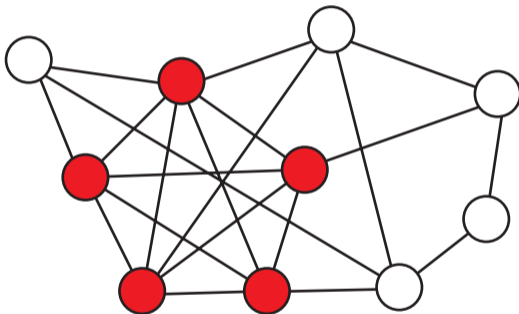
Example: Clique

Select an edge:



Example: Clique

Remove the edge.

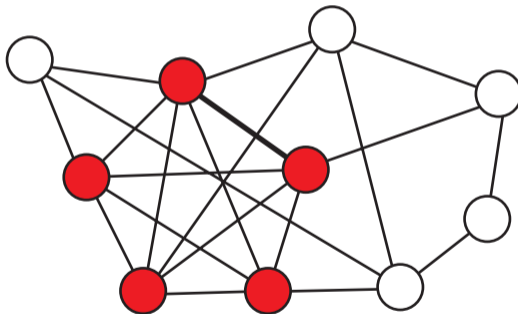


Call the decision algorithm.

The answer is YES, because there still is a 5-clique.

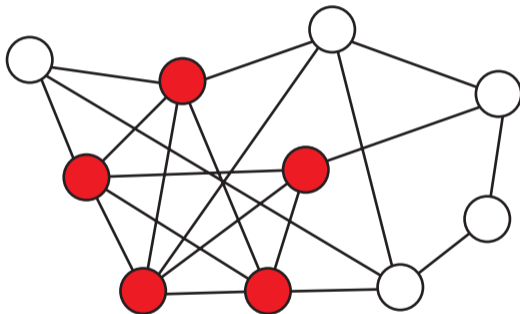
Example: Clique

Remove the edge forever. Select another one.



Example: Clique

Remove the edge.

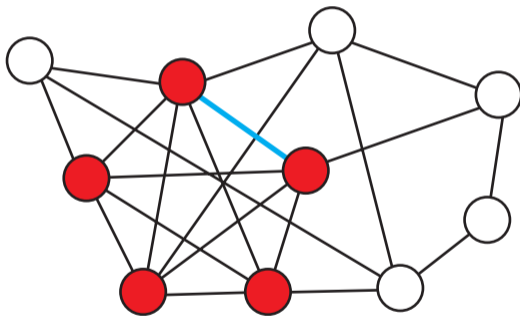


Call the decision algorithm.

The answer is NO, because the only 5-clique is destroyed.

Example: Clique

Put the edge back and mark it “permanent”.



Continue that way.

Example: Clique

Correctness:

- ▶ Phase 1: The binary search correctly determines the size of a maximum clique.
- ▶ Phase 2: As long as $A_d(G, k) = \text{YES}$ the current graph G has a k -clique.
- ▶ If $A_d(G, k) = \text{NO}$, 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.

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.

Alternatives

Can we also remove vertices instead of edges to convert a decision algorithm for Clique to an optimization algorithm?

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.

Computationally Hard Problems

Randomized Algorithms, Complexity Classes and the Class \mathcal{NP}

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

Deterministic Algorithms

- ▶ Most of the algorithms you have seen in your studies are deterministic.
- ▶ A deterministic algorithm A will always produce the same output when the same input is given repeatedly.
- ▶ A deterministic algorithm A will always have the same running time when the same input is given repeatedly.
- ▶ This is a desired and nice behavior.

Randomized Algorithms

- ▶ A randomized algorithm A might produce **different** outputs when the same input is given repeatedly.
- ▶ A randomized algorithm A might have different running times when the same input is given repeatedly.
- ▶ Sometimes it might not stop at all.
- ▶ This behavior is normally undesired and even dangerous.
- ▶ Nevertheless randomized algorithms are very useful if the undesired behavior can be somewhat “statistically” controlled.

Definition

- ▶ A randomized algorithm has access to a *fair coin*.
- ▶ This is a device that on demand outputs a 0 or 1.
- ▶ This happens **independently according to uniform distribution** on $\{0, 1\}$.
- ▶ The probability of seeing a 0 is 0.5 as is that for a 1.
- ▶ A coin flip counts one computational step.

Definition

The coin can be used in form of *randomized flow-control statements*.

```
if (coin = 1) then  
    do something  
else  
    do something else  
end if
```

It cannot be determined beforehand which of the two statements will be executed.

Random Number Generators

As another syntactical tool we introduce a *random number generator*:

This is a subroutine $\text{rand}(a, b)$ which receives two integers a, b , $a < b$ and returns a random number.

The number is drawn **independently according to uniform distribution** from the set $\{a, a + 1, \dots, b - 1, b\}$.

Generating a random number counts as one computational step.

However, it is not necessarily “cheap” and “easy” to obtain real random numbers in practice.