Računska zahtevnost

Zapiski s predavanj prof. Alexandra Simpsona

Domen Vogrin

Ljubljana, jesen 2023

Kazalo

1 Basics

1.1 Determinant of a matrix

Determinant of matrix

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

is calculated via formula

$$det(A) = \sum_{\sigma \in perm(n)} sgn(\sigma) \cdot \prod_{i=1}^{n} a_{i,\sigma(i)}$$

Naive algorithm requires $n! \cdot (n-1)$ multiplications, which is worse than exponantional. However, there are better algorithms for the permanent. Use Gaussian elimination to factorise A

$$A = L \cdot U \cdot P$$

where L is lower triangle matrix, U is upper triangle matrix, and P is a permutation. This helps, because

$$\begin{aligned} det(A) =& det(L) \cdot det(U) \cdot det(P) \\ =& det(L) \cdot det(U) \end{aligned}$$

We can compute determinants of L and U by just multiplying diagonal elements.

Time complexity od LU decomposition is $\mathcal{O}(n^3)$, which is also time complexity of our algorithm.

Fastest known time for determinant is $\mathcal{O}(n^{2,376})$.

1.2 Primality testing

For given n, is n prime?

Naive algorithm: Count through the odd nimbers from 3 up to $\sqrt{n}$ and check that none divides n. This gives us naive time complexity of $\mathcal{O}(\sqrt{n})$. But the appropriate size measure is $|n|$, which is the length of n. This means, that naive algorithm is exponential in $|n|$.

There are clever more efficient algorithms:

- The algorithm used in practice (Miller-Rabin) uses randomness, and has a positive probability of error. (1970s)
- Breakthrough in 2002 the AKS algorithm: a deterministic polynomial-time algorithm for primality testing.

1.3 Prime factorisation

For given n, compute the list (with multiples) of its prime numbers.

For given n, return two numbers smaller than n whose product is n (if such exist), 0 otherwise. It is an open problem if exist an efficient deterministic/algorithm algorithm to solve this problem.

However, Shor's algorithm (1994) is an efficient quantum algorithm.

1.4 K-colourability

E.g. 3-colourability.

It is an open question if there exists an efficient deterministic (random) algorithm for 3-colourability (or for k-colourabilits if $k \geq 3$).

Equivalent statement: P = NP

These statements are equivalent, because 3-colourability is NP-complete.

Related questions:

- Decision problem: Is a given graph 3-colourable (Does there exists a solution)
- Search problem: Given a graph, find a 3-colouring (if one exists, otherwise return "I can't")
- Counting problem: Count the number of 3-colourings
- Sampling problem: Find a random (w.s.t. uniform distribution) 3-colouring.

1.5 Perfect matchings

Given a bipartite graph (two sided)

A perfect matching is a bijection netween the two sides that follows the edges of the graph.

- Decision problem: efficient algorithm exsists, because
- Search problem: effician algorithm exsists (Ford-Falkerson for network flow ...)
- Counting problem: Open question ($FP \neq \#P$)

We can represent the bipartite graph as an adjacency matrix.

Number of perfect matchings is

$$\sum_{\sigma=perm(n)} \prod_{i=1}^{n} a_{i,\sigma(i)} = perm(A)$$

which is a permanent of A.

2 Asymptotic notation

measures rate of growth.

Typically we want to relate a function

$$T : \mathbb{N} \to \mathbb{R}$$

to a more mathematically natural

$$f : \mathbb{N} \to \mathbb{R}$$

2.1 Big o notation

Main definition:

$$T(n) = \mathcal{O}(t(n)) \iff \exists N \exists c > 0 \forall n \geq N T(n) \leq c \cdot f(n)$$

This is big $\mathcal{O}$ notation.

We are really saying $T \in \mathcal{O}(f)$.

2.2 Little o notation

Main definition:

$$T(n) = o(t(n)) \forall c > 0 \exists N \forall n \geq N T(n) \leq c \cdot f(n)$$

We are really saying $T \in (f)$.

How hard is it to solve computational problems?

We are interested in the resources used, in particular time and space. We need a mathematical model of what an algorithm is that supports an analysis of time and space.

3 Turing machines (1936)

Turing machine is algorithm by definition, sort of computer that runs a fixed program.

- An algorithm should be a finite description of a computational process.
- The algorithm should specify the computational process in its entirety.
- The process of following algorithm should be carried out without any creative input.
- So it is a process that can be carried out by a suitable machine.
- Machine will perform steps in time and need enough working space to carry out calculations.
- We do not bound time and space inadvance.

Definition 3.1. A (deterministic) Turing machine (TM) is specified by:

- A finite tape alphabet Γ with $\square \in \Gamma$ ($\square$ is a blank symbol)
- A finite set Q of states with start $\in Q$.
- A partial function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$.

A machine configuration is a triple (q, t, i) where

- $q \in Q$
- t is a tape configuration
- i is the position of the head.

A tape configuration is a $t : \mathbb{Z} \to \Gamma$ so that $t(m) \neq \square$ for only finitely many m.

3.1 Turing machine execution

We start the machine in a machine configuration (q, t, i), where $q = start$ and i points at the left-most non $\square$ symbol on the tape

$$i = min\{m \in \mathbb{Z} : t(m) \neq \square\}$$

(if the tape is empty, i can be arbitrary.)

One can define mathematically what it means that δ takes us from machine configuration (q, t, i) to (q', t', i') and what it means that δ halts machine configuration (q, t, i).

3.2 Using a TM as a language recogniser

We assume an input alphabet $\Sigma \subseteq \Gamma \setminus \{\square\}$.

(A language is a subset $L \subseteq \Sigma^*$)

We also assume distinguisged halting states accept, reject $\in Q$.

A TM M decides the language L if for any input word $w \in \Sigma^*$

- if $w \in L$ then M halts in the accept state on input w.
- if $w \notin L$ then M halts in the reject state on input w.

A language is decideable, if htere exist a TM M that decides it.

3.3 Using TMs to compute functions

Assume an input alphabet Σ_1 and optput alphabet Σ_2 with $\Sigma_1, \Sigma_2 \subseteq \Gamma \setminus \{\square\}$. A TM M computes a function $f : \Sigma_q^* \to \Sigma_2^*$ if

- for any $q \in \Sigma_1^*$, when given w as input M eventually halts with $f(w)$ on the tape.
- f is computable if there existsa TM that computes f.

3.4 Variant Turing Machines

- Machines in which the tape is infinite in only one direction
- Machines with K tapes (one head per tape)
- Machines with K tapes and many heads per tape, allowing heads to move between tapes

- Machines with multidimensional workspaces instead of one dimentional tapes
- etcetera ...

All such variants are equivalent.

In general a K-tape machine has K tapes each with its own head. So a machine configuration is of the form $(q, t_1, \ldots, t_k, i_1, \ldots, i_k)$ (k tape configurations and k head positions). Transition function has form

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, +1\}^k$$

We execute the machine in the "natural way"as in the example (not written).

Definition 3.2. If M is a (k-tape) language recognising TM we define $time_M : \mathbb{N} \to \mathbb{N}$:

$$time_M(n) = max\{\text{no. of steps taken by input } w : w \in \Sigma^*, |w| = n\}$$

For a language $L \subseteq \Sigma^*$ and a function $T : \mathbb{N} \to \mathbb{N}$ we say that $L \in DTime(T(n))$ if there exist $k - tape$ TM M for some $k \geq 1$ that decides L such that $time_M(n) = \mathcal{O}(T(n))$.

Definition 3.3 (Fundemental def.)**.**

$$P := \cup_{d \geq 1} DTime(n^d)$$

Equivalent between machine models proved by translations between models. Such translations do not preserve $DTime(T(n))$. But all translations are polynomially bounded in space and time. So the class P is invarient under choice of computational model.

4 Nondeterminism

An NTM is:

- Γ as before
- Q as before
- A subset $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{-1, 0, +1\})$

The transition relation: for every $(q, a) \in Q \times \Gamma$ there is a finite set of (q', a', d'), so that $((q, a), (q', a', d')) \in \Delta$. Notation: $(q, a) \to (q', a', d')$

Machine configurations are as before.

A run of the machine is a sequence of configurations starting in the starting configuration and respecting the 'rules' of the transition relation, whose last configuration, if it exists, is a halting/terminating configuration.

NTMs potentially have many different runs from the same starting configuration.

4.1 NTMs as language recognisers

Assume input alphabet $\Sigma \subseteq \Gamma \setminus \{\square\}$ and halting states $accept, reject \in \mathbb{Q}$

An NTM M accepts a word $w \in \Sigma^*$ if there exists a run of M an input w that terminates in accept.

An NTM M rejects a word $w \in \Sigma^*$ if every run of input w terminates in reject.

An NTM M decides the language $L \subseteq \Sigma^*$ if for every $q \in \Sigma^*$:

- if $w \in L$ then M accepts w
- if $w \notin L$ then M rejects w

If M is a NTM, define $\text{time}_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$

$$\begin{aligned}\text{time}_M(n) = max\{&\text{no. of steps in the run } r|w \\ &\in \Sigma^*, |w| = n, r \text{ is a run of M on input } w\}\end{aligned}$$

For a language $L \subseteq \Sigma^*$ and function $T(N) : \mathbb{N} \to \mathbb{N}$, we say that $L \in \text{NTime}(T(n))$ if there exists a NTM M that decides L such that $\text{time}_M(n) = \mathcal{O}(T(n))$.

$$NP = \cup_k NTime(n^k)$$

$$P \subseteq NP$$

because every deterministic TM is nondeterministic.

Big question is:

$$P = NP$$

Example 4.1. Independent set problem - INDSET.

Input: A graph G and a natural number $k \leq |G|$ - number of vertices of G.

Question: Does G have an independent set of size k.

An independant set or anticlique is a subset I of the vertices of G such that no two elements of I have an edge between them.

We code this up as a decision problem for a language over $\{0, 1, ', ', ' \}$

Example encoding (not here) -> Represent a graph as an adjecent matrix. Our input word is 010,101,010;10 (before; is matrix, after; is number of vertices)

If G has n vertices, then our input string has $m^2 + m + \lfloor log_2 k \rfloor + 1$

We write $\ulcorner$(G, k)$\urcorner$ for the encoding in Σ^* representing the input G, k

INDSET:

$\{w \in \{0, 1, , , ; \}$
$*|w$ is of the form $\ulcorner(G, k)\urcorner$ where G is a graph with an independent set of size k$\}$

We now argue that INDSET belongs to NP.

Given an input word

$$a_1 \dots a_M, \dots$$

Scan to the first comma and count number of characters to obtain M as we go along write a nondeterministic sequence of binary bits on second tape. $\mathcal{O}(m)$

Test tp see that there are exactly k 1s on the second tape (if not -> reject). $\mathcal{O}(m^2)$

For each pair of 1s on the second tape, check that there is a 0 in the relevant entry in the adjacency matrix on the first tape. $\mathcal{O}(m^4)$

If so, we accept (otherwise reject)

The time bound is $\mathcal{O}(m^4)$, ie $\mathcal{O}(n^2)$ (because of input is m^2)

Theorem 4.2 (Certificate chatacterisation of NP)**.** T.f.a.e. for $L \subseteq \Sigma^*$

1. Ł $\in NP$
2. There is a polynomial $p(n) : \mathbb{N} \to \mathbb{N}$ and a polynomial time bounded deterministic TM such that for any input $w \in \Sigma^*$ t.f.a.e.
   (a) $w \in L$
   (b) there exists certificate $v \in \Sigma^{p(|w|)}$ such that M accepts w, v

Dokaz. (2) -> (1) Nondeterministically generate $v \in \Sigma^{p(|w|)}$. Then deterministically check if it is a solution. (1) -> (2) Use the sequence of nondeterministic choices as the certificate. ■

P problems solvable in deterministic polynomial time, NP problems solvable in nondetrministic polynomial time.

$$L \in P \equiv L \in DTime(p(n))$$

Problems we can efficiently solve.

$$L \in NP \equiv L \in NTime(p(n))$$

Problems for which we can efficiently verify the correctness of a solution.

This intuition is embodied in the certificate characterisation of LP.

5 Meet some NP problems

All these problems are in fact NP-complete.

Main theorem today: Levin theorem: SAT is NP-complete.

Important today: the notions of NP hardness, polynomial time redneibility??

5.1 SAT

A propositional formula is

- a boolean variable: $u, u', u'', u''', u'''', ...$
- a truth value: 1 (true), 0 (false)
- if ϕ, ψ formulas then so are $\phi \cup \psi, \phi \cap \psi, \neg\phi$

If we assign truth values $\{0, 1\}$ to propositional variables

$$v : \text{variables} \to \{0, 1\}$$

then we compute $v(\phi)$ using truth tables.

ϕ is satisfiable, if there exists an assignment v :variable $\to \{0, 1\}$, such that $v(\phi) = \neg$. We call v a satisfying assignment for ϕ

Example 5.1. $(u \wedge u') \vee (u \wedge u'') \vee (u' \wedge u'')$ This is satisfiable. v is a satisfying assignment $\equiv$ if maps at least two variables to 1.

Example 5.2. $u \wedge \neg u'$ This is not satisfiable.

The SAT problem:

- Input: a propositional formula ϕ
- Question: is ϕ satisfiable?

We represent SAT as a language over the aplhabet $\sigma_{SAT} = \{0, 1, \wedge, \vee, \neg, u, '\}$

$$u \wedge (u'' \vee u''')\phi$$

represent as the

$$\wedge u \vee u''u'''\phi'$$

$$SAT = \{\phi' | \phi \text{ is a satisfiable formula}\}$$

SAT $\in$ NP

A certificate for the satisfiability of a formula ϕ containing variables up to u_k is simply a word $v \in \{0, 1\}*$ representing a stisfying assignment.

We can verify in deterministic polynomial time given input ϕ', v wheter or not v is a satisfying assignment for ϕ.

SAT $\in$ NP by the certificate definition of NP.

Polytime reducibility

We say that a language $L_1 \subseteq \Sigma_1^*$ is polynomial-time (ptime) reducible to a language $L_2 \subseteq \Sigma_2^*$ if there exists a deterministic TM M that compute a function $f : \Sigma_1^* \to \Sigma_2^*$, such that

- for all $w \in \Sigma_1^*$, $f(w) \in L_2 \equiv w \in L_1$ and
- M is polynomial time bounded

$$\exists K s.t. time_M(n) = \mathcal{O}(n^K)$$

Observations

- $L \leq_p L$
- If $L_1 \leq_p L2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$
- If $L' \leq_p L \in P$ then $L' \in P$

- If $L' \leq_p L \in NP$ then $L' \in NP$
- If L is NP hard and $L \leq_p L'$, then L' is also NP hard

NP hardness

L is NP hard, if for any NP language L' we have $L' \leq_p L$.

NP completness

L is NP complete, if $L \in NP$ and L is NP hard.

Theorem 5.3 (Cook/Levin)**.** SAT is NP complete.

5.2 3-SAT

Satisfiability for formulas (glej zapiske)

5.3 3-COL

Given a cojnunction of 3-clauses ϕ. Convert it to a graph that is 3-colourable iff ϕ is satisfiable.

Suppose ϕ has variables up to u_k.

that SAT is NP hard. Suppose $L \in NP$. We show $L \leq_p SAT$.

Let M be a single-tape ND TM that decides L with time bound $p(n) \geq time_M(n)$. (p a polynomial).

If we run M on input w.

The machine can only during its execution look at squares between -p(n) and p(n).

Modify M so that from any halting state it makes a dummy step (does nothing). We will call it M'.

Given an input word $w \in \Sigma$ we construct a propositional formula Φ_w such that

$\Phi_w$satisfiable $\iff$ there exists a run of M' on input w that is in the accept state after $p(|w|)$ st

Φ_w will use $(p(|w|)+1) \times |Q|+(p(|w|)+1) \cdot (2p(|w|)+1) \cdot (|\Gamma|+1)$ variables. This is polynomial in $|w|$ $\mathcal{O}((p(n))^2)$

Mnemonic names for the variables:

- $(\text{state}_t = q)$ for every $t = 0 \dots p(|w|)$, $q \in Q$
- $(\text{symbol}_{t,i} = a)$ for every $t = 0 \dots p(|w|)$, $i = -p(|w|), \dots, p(|w|)$, $a \in \Gamma$
- $(\text{head}_t = i)$ fpr every $t = 0 \dots p(|w|)$, $i = -p(|w|) \dots p(|w|)$

Define Φ_w in such a way that its satiyfying assignments are in correspondence with runs of length (time steps) $p(|w|)$ that end in accept state.

We break its definition into parts.

$Q_{init-w} := \vee_{o \leq i < |w|}(Symbol_{0,i} = w_i) \wedge (head_0 = 0) \wedge (state_0 = start) \wedge (\vee)_{i=-p(|w|)\cdots-1}(symbol_{0,i} = \square$

■