Introduction to Complexity and Computability
NTIN090

Jirka Fink

https://ktiml.mff.cuni.cz/~fink/

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Winter semester 2017/18
Last change on January 11, 2018

General information

E-mail	fink@ktiml.mff.cuni.cz
Homepage	https://ktiml.mff.cuni.cz/˜fink/
Consultations	Thursday, 17:15, office S305

Examination

- Homeworks (theoretical)
- Pass the exam

Content

Literature

- Sipser, M. *Introduction to the Theory of Computation.* Vol. 2. Boston: Thomson Course Technology, 2006.
- Garey, Johnson: *Computers and intractability - a guide to the theory of NP-completeness*, W.H. Freeman 1978
- Soare R.I.: *Recursively enumerable sets and degrees.* Springer-Verlag, 1987
- Odifreddi P.: *Classical recursion theory*, North-Holland, 1989
- Arora S., Barak B.: *Computational Complexity: A Modern Approach*. Cambridge University Press 2009.

Outline

Post correspondence problem (Post, 1946)

Input

- Every domino contains two strings, one on each side; e.g. $\boxed{\frac{a}{ab}}$
- An input to the problem is a collection of dominoes; e.g. $\left\{ \boxed{\frac{b}{ca}}, \boxed{\frac{a}{ab}}, \boxed{\frac{ca}{a}}, \boxed{\frac{abc}{c}} \right\}$

Match

A match a is a list of these dominoes (repetitions permitted) so that the string on the top is the same as the string on the bottom; e.g. $\boxed{\frac{a}{ab}} \boxed{\frac{b}{ca}} \boxed{\frac{ca}{a}} \boxed{\frac{a}{ab}} \boxed{\frac{abc}{c}}$

Problem

Find an algorithm which determines whether for a given collection of dominoes there exists a match.

Hilbert's tenth problem (1900)

Original statement (translated from German)

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

Reformulation

Find an algorithm which for a given multi-variable polynomial with rational coefficients has integral root.

Example of an instance of the problem

Is there an intergral solution of equation $6x^3yz^2 + 3xy^2 - x^3 - 10 = 0$?

Halting problem

Halting problem

Is there an algorithm which for a given program and its input determines whether the program on that input terminates?

Example of a program

```
int main(int argc, char *argv[]) {
  int n, total, x, y, z;
  scanf("%d", &n);
  total=3;
  for(;;) {
    for(x=1; x<=total-2; ++x) {
      for(y=1; y<=total-x-1; ++y) {
        z=total-x-y;
          if(exp(x,n)+exp(y,n)==exp(z,n))
            return 0;
       }
    }
    ++total;
  }
}
```

Random access machine

Components

- Memory divided into cells
  - Cells are indexed by (positive) integers
  - Every cell can store a integer
- Program composed of instructions
  - Arithmetical instructions: additions, subtraction, multiplication, division, modulo
  - Logical instructions: conjunction, disjunction, shift
  - Control instructions: conditional and unconditional jump, halt
- Input and output
  - Special instruction for reading and writing, or
  - storing in memory in pre-defined cells
- Special registers (e.g. program counter)

Arguments of instructions and memory indexing

- Integer, e.g. 3
- Direct addressing, e.g. [4]
- Indirect addressing, e.g. [[5]]
- Example of instruction: [[5]] := 3 + [4]
- Variables can be used to improve readability

Random access machine

Example: Factorial

Calculate factorial of the input stored at [1] and save the results in [2].

```
[2] := 1
loop:
[2] := [1] * [2]
[1] := [1] - 1
jump_if_positive [1], loop
halt
```

Variants of Random access machines

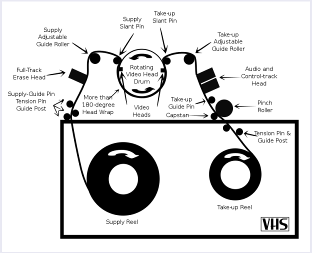
- Memory potentially infinite only in one direction
- Cells with bounded number of bits
- More instructions, e.g. function calls
- More indirections in memory accessing
- Etc.

Memory without random access (Source: Wikipedia)

Video Home System

Compact Cassette

Memory access



Turing machine (Alan Turing, 1936)

Definition

Turing machine is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states (internal memory)
- Σ is a finite tape alphabet
  - i.e. every cell store one value from Σ
  - $\lambda \in \Sigma$ represents an empty cell
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{R, N, L\} \cup \{\perp\}$ is a transition function which for every state of Q and every value in the current cell determines:
  - new state of internal memory,
  - new value to be stored in the current cell,
  - whether the head determining the current cell should be moved to right (R), left (L) or stay (N)
  - $\perp$ represents termination
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

Turing machine

Configuration captures the full state of computation of a Turing machine

A configuration is triple (q, m, h) where

- $q \in Q$ is the current state,
- $m \in \Sigma^{\mathbb{Z}}$ are symbols in all cells of the tape and
- $h \in \mathbb{Z}$ is the index of the current cell (the position of head).

Initial configuration

The initial configuration is $(q_0, m, 0)$ where

- q_0 is the initial state and
- m contains the input word surrounded by λ.

Turing machine

Single step of computation

Assume that the current configuration is (q, m, h) and $\delta(q, m_h) = (q', s, d)$ is result of the transition function. The new configuration after a single step of computation is (q', m', h') where

- $m'_i = \begin{cases} m_i & \text{if } i \neq h \\ s & \text{if } i = h \end{cases}$
- $h' = \begin{cases} h+1 & \text{if } d = R \\ h-1 & \text{if } d = L \\ h & \text{if } d = N \end{cases}$

Termination

The computation terminates if $\delta(q, m_h) = \perp$. The input word is accepted if $q \in F$; otherwise, the word is rejected.

Turing machine

Possible results of computations

For every Turing machine and its every input word, exactly one of the following statements holds:

- Computation terminates and uses finitely many cells of the type.
- Computation loops on finitely many configurations and finitely many cells are used.
- Computation never terminates, never reaches the same configuration twice and uses infinitely many cells.

Variants of Turing machine

- Tape potentially infinite only in one direction
- Multiple tapes
- Multiple heads on tapes
- Non-deterministic Turing machines

All variants are equivalent to the basic model.

Turing machine $\rightarrow$ Random access machine

Theorem

For every Turing machine there exists an equivalent RAM.

Proof

- For simplicity, consider one directional memory and tape
- Represent states Q and tape alphabet Σ by integers
- Memory of RAM is organized as follows:
  - [0] : Position of head
  - [1] : State q
  - [2] and more : Content of the tape
- Rewrite the transition function δ into instructions using “if-then-else” statements
- Simulate the Turing machine on RAM

Random access machine $\to$ Turing machine

Theorem

For every Random access machine (RAM) there exists an equivalent Turing machine (TM).

Proof

We construct 4-tape TM:

- Input tape: Sequence of numbers passed to RAM as the inpun written in binary and separated with $\#$
- Output tape: TM writes here the numbers output by RAM
- Memory of RAM: Content of memory of RAM
  - The tape symbols are $\Sigma = \{0, 1, \#, |, \lambda\}$
  - If the currently used cells are $c_1, c_2, \dots, c_m$ then the tape contains string: $c_1|[c_1]\#c_2|[c_2]\# \cdots \#c_m|[c_m]$
- Auxiliary tape: For computing addition, subtraction, indirect indices, copying part of the memory tape, etc

TM simulates every instruction of RAM as follows: [1]

- Copy arguments from the memory tape to the auxiliary tape
- Evaluate the instruction on the auxiliary tape
- Copy results to the memory tape

1. Program of a RAM consists of finitely many instructions and arguments (indices of memory cells) are bounded numbers. Therefore, the program can be “stored” in a transition functions δ and a program counter can be stored as a part of state space Q.

Words and languages

Terminology

- Alphabet Σ is a finite set of characters (symbols)
- Word is a finite sequece of characters over alphabet Σ
- $\Sigma^\star$ denotes the set of all words
- Language $L \subseteq \Sigma^\star$ is a set of words over alphabet Σ

Decision problem

A decision problem is formalized as a question whether given word belongs to the language.

Note

Every language contains countably many words. The set of all decision problems is uncountable.

Turing decidable languages

Terminology

- TM accept a word $w \in \Sigma^\star$, if computation of TM with input w terminates in an accepting state
- RAM accept a word $w \in \Sigma^\star$, if computation of RAM with input w outputs 1 and terminates
- TM reject a word $w \in \Sigma^\star$, if computation of TM with input w terminates in an non-accepting state
- RAM reject a word $w \in \Sigma^\star$, if computation of RAM with input w outputs 0 terminates
- The set of words accepted by TM M is a language denoted by $L(M)$
- We denote the fact that computation of TM M on w terminates as $M(w)\downarrow$ (computation converges)
- We denote the fact that computation of TM M on w does not terminate as $M(w)\uparrow$ (computation diverges)
- Language L is partially (Turing) decidable (also recursively enumerable), if there is a TM M such that $L = L(M)$
- Language L is (Turing) decidable (also recursive), if there is a TM M which always terminates and $L = L(M)$

Numbering Turing machines

Numbering Random access machines

- Every program in C is a word over alpabet $\{0, 1\}$, i.e. represented by an integer
- Similarly, every RAM can be represented by an integer

Encode a Turing machine $(Q, \Sigma, \delta, q_0, F)$ as a word over a small alphabet Γ

- Consider an alphabet $\Gamma = \{0, 1, L, N, R, |, \#\}$
- Encode every state Q and symbol Σ by a binary string
- The set F is encoded as a concatenation of all states of F separated by $|$
- Instruction $\delta(q, s) = (q', s', d)$ is encoded as word $q|s|q'|s'|d$ over $\{0, 1, |\}$
- Function δ is encoded as a concatenation of all instructions separated with $\#$
- TU $(Q, \Sigma, \delta, q_0, F)$ is encoded as concatenation of q_0, F and δ separated with $\#$

Gödel number

- Encode every symbol of Γ by 3 bits (a word over $\{0, 1\}$)
- Replace the word over Γ encoding TU by a word over $\{0, 1\}$
- Precede the word over $\{0, 1\}$ by the character 1 and read it as a binary integer
- The resulting integer is called Gödel number of a given Turing machine

Existence of an undecidable problems

Notes

1. Some integers may not encode a valid Turing machine
2. If $M \in \mathbb{N}$ is not a valid Turing machine, $M(w)$ rejects every input w
3. Integer representing a Turing machine is not unique
4. For every decidable problem there exist infinitely many Turing machines deciding it
5. There are only countably many Turing machines
6. There are only countably many decidable languages

Corollary

There exists an undecidable language.
Furthermore, there are uncountably many undecidable languages.

Universal Turing machine

Goal

Create a Turing machine which for a given pair (M, w) computes the output of a Turing machine M called on input w. ①

Universal TM has three memory types

- 1st tape contains input, i.e. pair (M, w)
- 2nd tape contains binary encoded tape of M separated by $|$
- 3rd tape contains binary encoded current state of M

Computation of Universal Turing machine

- Copy the input word w on the working (2nd) tape and set the initial state of M on the 3rd tape
- Simulates steps of M
  - Find the transition $\delta(q, s) = (q', s', d)$ on the 1st tape where q is the current state of M stored on 3rd tape and s is the current symbol of M encoded on the current position on 2nd tape
  - Store results q' and s' on the 3rd and 2nd tape and move 2nd head in direction d

1. We already proved that for every for every Random access machine R we can construct a Turing machine M which gives the same output for every input. Now, we create one Turing machine which simulates any TM/RAM.

Diagonalization (Cantor, 1873)

Observation

Let M be a set and 2^M the set of all subsets of M (called a power set). Then, there is no surjection from M to 2^M.

Proof

- For a sake of contradiction, assume that there exists a surjection $f : M \to 2^M$
- Let $Z = \{a \in M;\ a \notin f(a)\}$
- There exists $x \in M$ such that $f(x) = Z$ since f is a surjection
- Does x belong to Z? The following statements are equivalent:
    - $x \in Z$
    - $x \notin f(x)$ by definition of Z
    - $x \notin Z$ since $f(x) = Z$
- Hence $x \in Z \Leftrightarrow x \notin Z$ which is a contradition!

Halting problem

Theorem

Let HALT be the set of all pairs (M, w) such that Turing machine M terminates on its input w. The language HALT is partially decidable but it is not decidable.

Proof (undecidability)

For sake of a contradiction, assume that there exists a Turing machine `halting_solver(`M, w`)` which for a Turing machine M and an input w determines whether M terminates on input w. [1] Consider the following adversary Turing machine.

```
Program adversary(w)
    if halting_solver(w,w) returns "Yes, it terminates" then
        Loop forever
    else
        Halt
```

The following statements are equivalent.

- `adversary(adversary)` terminates
- `halting_solver(adversary, adversary)` returns “Yes, it terminates” [2]
- `adversary(adversary)` loops forever [3]

This is a contradiction!

1. Keep in mind that inputs are represented as integers (or binary strings). Similarly, Turing machines are encoded as integers. Hence, integers represents inputs and also Turing machines and we can use an input as a Turing machine and vice versa. For example, compilers expect programs as their inputs.
2. By definition of Turing machine `halting_solver`.
3. By definition of Turing machine `adversary`.

Undecidability of acceptance

Theorem

Let ACCEPT be the set of all pairs (M, w) such that Turing machine M terminates and accepts its input w. The language ACCEPT is partially decidable but it is not decidable.

Proof (undecidability)

For sake of a contradiction, assume that there exists a Turing machine `accept_solver(M, w)` which for a Turing machine M and an input w determines whether M terminates and accepts input w. Consider the following adversary Turing machine.

```
Program adversary(w)
    if accept_solver(w,w) accepts then
        return reject
    else
        return accept
```

The following statements are equivalent.

- `adversary(adversary)` accepts
- `accept_solver(adversary, adversary)` accepts
- `adversary(adversary)` rejects

This is a contradiction!

Undecidability of empty languages

Theorem

Let EMPTY be the set of all Turing machines which does not accept any input. The language EMPTY undecidable.

Proof (undecidability)

For sake of a contradiction, assume that there exists a Turing machine `empty_solver(`M`)` which for a Turing machine M determines whether accept some input. We construct Turing machine `accept_solver` which is a contradiction.

```
Program accept_solver($M, w$)
    Program helper_$M$($w$)
        if $w = M$ then
            return M(w)
        else
            return reject
    if empty_solver(helper_$M$) accepts then
        return reject
    else
        return accept
```

We prove that our `accept_solver` works properly. First, consider a Turing machine *M* which accept an input *w* which implies the following statements

- `helper` $_M$(M) accepts by definition of `helper`
- L(`helper` $_M$) is non-empty is Turing machine `helper` $_M$ accepts input M
- `empty_solver` (`helper` $_M$) rejects
- `accept_solver(`*M,w*`)` accepts

Next, consider a Turing machine M which does not accept an input *w* which implies the following statements

- `helper` $_M$ does not accept any input
- L(`helper` $_M$) is empty
- `empty_solver` (`helper` $_M$) accept
- `accept_solver(`*M, w*`)` rejects

Complement of languages

Definition

The complement of a language $L \subseteq \Sigma^\star$ is the language $\bar{L} = \Sigma^\star \setminus L$.

Observation

A language L is decidable if and only if the complement $\bar{L}$ is decidable.

Theorem (Post)

Language L is decidable if and only if languages L and $\bar{L}$ are partially decidable.

Corollary

Languages $\overline{\text{ACCEPT}}$, $\overline{\text{HALT}}$ and EMPTY are not partially decidable.

Observation

If L_1 and L_2 are (partially) decidable languages, then the following languages are also (partially) decidable.

- $L_1 \cup L_2$ ①
- $L_1 \cap L_2$
- $L_1 \cdot L_2 = \{ab;\ a \in L_1, b \in L_2\}$, i.e. concatenation of words

1. If we have finitely many (partially) decidable languages $L_1, \dots, L_k$, is the union $L_1 \cup \dots \cup L_k$ (partially) decidable?
   If we have countably many (partially) decidable languages L_i for $i \in \mathbb{N}$, is the union $\bigcup_{i\in\mathbb{N}} L_i$ (partially) decidable?

Turing computable functions

Terminology

- If a Turing machine M terminates on an input w then the output (content of the tape after computation) is denoted by $f_M(w)$.
- Each Turing machine M with tape alphabet Σ computes some partial function $f_M : \Sigma^* \mapsto \Sigma^*$.
- Function $f : \Sigma^* \mapsto \Sigma^*$ is Turing computable, if there is a Turing machine M which computes it.
- To each Turing computable function there is infinitely many Turing machines computing it!

Notes

- A program M (RAM/TM) which for an input x outputs y can be considered as a function $f_M : \Sigma^* \mapsto \Sigma^*$ such that $f_M(x) = y$.
- A function $f : \mathbb{N} \to \mathbb{N}$ composed from some elementary (Turing computable) operations is a program which for an integer (word) x computes $f(x)$.
- Decidability of a language $L \subseteq \Sigma^\star$ is a question whether the function $f : \Sigma^\star \to \{0, 1\}$ is Turing computable where $f(w) = 1$ if and only $w \in L$.

Universal Turing machine, function, language

Universal Turing machine

Universal Turing machine computes for a given pair (M, w) the output of a Turing machine M called on input w.

Universal function

A universal function Ψ satisfies $\Psi(f, x) = f(x)$ for every partially computable function f and every argument of x of the domain of f.

Universal language

A universal language L_u is the set of pairs (M, w) such that Turing machine M accepts w.

Observation

Universal function Ψ is partially computable but it is not computable.
Universal language L_u is partially decidable but it is not decidable.

Proof

Use universal Turing machine . . .

Church-Turing thesis

Intuitively

An algorithm is a finite sequence of simple instructions which leads to a solution of given problem.

Church-Turing thesis

To every algorithm in intuitive sense we can construct a Turing machine which implements it.

Equivalent computation models

According to Church-Turing thesis, intuitive notion of algorithm is also equivalent with

- description of a Turing machine,
- program for RAM,
- derivation of a partial recursive function,
- derivation of a function in λ-calculus,
- program in a programming language C, Pascal, Java, Basic, Lisp, Haskell etc.,

In all these models we can compute the same functions and solve the same problems.

Strings ordering

Defintion

Let Σ be an alphabet and let us assume that $<$ is a strict order over characters in Σ. Let $u, v \in \Sigma^\star$ be two different strings. We say that u is *lexicographically smaller* than v if

- u is shorter than v, or
- u and v have the same length and if i is the first index with $u[i] \neq v[i]$, then $u[i] < v[i]$.

This fact is denoted with $u \prec v$. As usual we extend this notation to $u \preceq v$, $u \succ v$ and $u \succeq v$.

Note

Let u, v be words over $\{0, 1\}$ and $u', v' \in \mathbb{N}$ be the corresponding intergers (preceded by “1”). Then $u \prec v$ if and only $u' < v'$.

Enumerator

Definition

An *enumerator* for a language L is a Turing machine E which

- ignores its input,
- during its computation writes words of L to a special output tape separated with '$\#$', and
- each string $w \in L$ is eventually written by E.
- If L is infinite, then E never stops.

Theorem

1. A language L is partially decidable if and only if there is an enumerator E for L.
2. An infinite language L is decidable if and only if there is an enumerator E for L which outputs elements of L in lexicographic order.

Ideas of proofs:

- Enumerator E for $L \Rightarrow L$ is partially decidable: Run the computation of enumerator E and if it outputs a given word, then accept.
- E enumerates in lexicographic order $\Rightarrow L$ is decidable: Run the computation of enumerator E and if it outputs a given word w, then accept. If E outputs a word lexicographically larger than w, then reject.
- L is decidable $\Rightarrow E$ enumerates in lexicographic order: Decide all words one by one in lexicographic order.
- L is partially decidable $\Rightarrow$ enumerator E for L: Decide all words "in parallel".

Reducibility

Definition

- A language $A \subseteq \Sigma^\star$ is m-reducible to a language $B \subseteq \Sigma^\star$ if there exists a computable function $f : \Sigma^\star \to \Sigma^\star$ such that $w \in A \Leftrightarrow f(w) \in B$ for every $w \in \Sigma^\star$.
- A language $A \subseteq \Sigma^\star$ is 1-reducible to a language $B \subseteq \Sigma^\star$ if there exists a computable injection $f : \Sigma^\star \to \Sigma^\star$ such that $w \in A \Leftrightarrow f(w) \in B$ for every $w \in \Sigma^\star$.
- A language B is m-complete (1-complete) if B is partially decidable and any partially decidable language A is m-reducible (1-reducible) to B.

Example (ACCEPT is *m*-reducible to HALT)

For a Turing machine M and its input w, let $f(M)$ be a Turing machine which calls $M(w)$ and if M accepts w, then $f(M)$ accepts w, otherwise M loops forever. ①

Theorem

If language A is m-reducible to a decidable language B, then A is decidable. ②

Example

Since ACCEPT is reducible to HALT and ACCEPT is undecidable, HALT is also undecidable.

1. Informally, f is a program which obtains a source code of M and returns a source code $f(M)$ which differs from M by adding one if-then-else statement. Therefore, f only trivially modify input source codes and it always terminates. The program $f(M)$ loops if M loops or rejects. Since f is an injection, this is an example of 1-reducibility.
2. For a given $w \in \Sigma^\star$, we compute $f(w)$ and then use a Turing machine recognizing B to determine whether $f(w) \in B$.

S-m-n lemma

s-m-n lemma (computable function)

For any two natural numbers $m, n \geq 1$ there is a 1-1 total computable function $s_n^m : \mathbb{N}^{m+1} \mapsto \mathbb{N}$ such that for every $x, y_1, y_2, \dots, y_m, z_1, \dots, z_n \in \Sigma_b^*$:

$$\varphi^{(n)}_{s_n^m(x, y_1, y_2, \dots, y_m)}(z_1, \dots, z_n) \simeq \varphi_x^{(m+n)}(y_1, \dots, y_m, z_1, \dots, z_n)$$

s-m-n lemma (Informally)

For all $m, n \in N$ and every Turing machine M expecting input of length $m + n$ and every input $y_1, \dots, y_m$ there exists a Turing machine M' expecting input of length n such that for every input $z_1, \dots, z_n$ computations $M(y_1, \dots, y_m, z_1, \dots, z_n)$ and $M'(z_1, \dots, z_n)$ give the same output. Furthermore, there exists a Turing machine SMN which for every M and $y_1, \dots, y_m$ computes M'.

Pseudocode of SMN

```
1 Program SMN ($M, y_1, \dots, y_m$)
2     Program $M'(z_1, \dots, z_n)$
3         return $M(y_1, \dots, y_m, z_1, \dots, z_n)$
4     return $M'$
```

Reducibility

Observations

- The relation of m-reducibility is reflexive and transitive.
- If m-complete language A is m-reducible to a partially decidable language B, then B is m-complete.

Theorem

The following languages are m-complete.

- $\text{ACCEPT} = \{(M, w);\ M \text{ accepts } w\}$
- $\text{HALT} = \{(M, w);\ M \text{ terminates on input } w\}$
- $\text{DIAG} = \{M;\ \text{Turing machine } M \text{ terminates input word } M\}$

Proof (ACCEPT is m-complete)

- ACCEPT is a partially decidable language
- We have to prove that for every Turing machine M there exists a computable function $f_M : \Sigma^\star \to \Sigma^\star$ such that for every input $w \in \Sigma^\star$ it holds that $w \in L(M)$ if and only if $f_M(w) \in \text{ACCEPT}$
- We have to find a computable function f_M such that $f_M(w) = (M, w)$
- We use s-m-m lemma to “hard-code” M, so f_M is $\text{SMN}(\Phi, M)$ where Φ is the universal TM.

Post correspondence problem (Post, 1946)

Input

- Every domino contains two strings, one on each side; e.g. $\boxed{\frac{a}{ab}}$
- An input to the problem is a collection of dominoes; e.g. $\left\{ \boxed{\frac{b}{ca}}, \boxed{\frac{a}{ab}}, \boxed{\frac{ca}{a}}, \boxed{\frac{abc}{c}} \right\}$

Match

A match a is a list of these dominoes (repetitions permitted) so that the string on the top is the same as the string on the bottom; e.g. $\boxed{\frac{a}{ab}} \boxed{\frac{b}{ca}} \boxed{\frac{ca}{a}} \boxed{\frac{a}{ab}} \boxed{\frac{abc}{c}}$

Theorem

A problem whether for a given collection of dominoes there exists a match is undecidable.

Proof

We m-reduce ACCEPT language to the Post correspondence problem (language) as follows.

Simplified Post correspondence problem

First, we consider the following simpler reduction

- Consider a one-directional Turing machine $M = (Q, \Sigma, \delta, q_0, F)$.
- When M terminates, all cells of the tape store the empty symbol λ. ①
- Assume that a match has to start with one given domino.

Terminology ②

- A configuration of a M is a word over $Q \cup \Sigma$ obtained from the word stored on the tape by inserting the current state into the current position of head. ③
- A history of computation is a concatenation all configurations separated with $\#$ and preceded and superseded $\#$.

Our goal is to create a list of dominoes such that the matching word represents a history of computation.

The initial domino

$\boxed{\frac{\#}{\# q_0 w_1 w_2 \cdots w_n \#}}$ where $w_1 w_2 \cdots w_n$ is the input word. ④

1. Turing machine cleans up the tape before it terminates.
2. We slightly modify few terms to simplify the proof.
3. We assume that Q and Σ are disjoin. Formally, a configuration is a concatenation of the following words:
    - the content of the tape before the head
    - the current state
    - the symbol on the current position of head
    - the content of the tape after the head

   Since $Q \cap \Sigma = \emptyset$, the current state also indicates the current position of head on tape.
4. $\# q_0 w_1 w_2 \cdots w_n \#$ is the initial configuration with separating symbols $\#$.

Computation using Post correspondence problem

Dominoes representing the computation

1. For every $s, s' \in \Sigma$ and every $q, q' \in Q$ satisfying $\delta(q, s) = (q', s', N)$ add $\boxed{\frac{qs}{q's'}}$
2. For every $s, s' \in \Sigma$ and every $q, q' \in Q$ satisfying $\delta(q, s) = (q', s', R)$ add $\boxed{\frac{qs}{s'q'}}$
3. For every $s, s', \bar{s} \in \Sigma$ and every $q, q' \in Q$ satisfying $\delta(q, s) = (q', s', L)$ add $\boxed{\frac{\bar{s}qs}{q'\bar{s}s'}}$
4. For every $s \in \Sigma$ add $\boxed{\frac{s}{s}}$
5. Add $\boxed{\frac{\#}{\#}}$, $\boxed{\frac{\#}{\lambda\#}}$ and $\boxed{\frac{\lambda\#}{\#}}$
6. For every $q \in F$ add $\boxed{\frac{q\#\#}{\#}}$

Computation using Post correspondence problem

Reduction to the original Post corresponding problem

For a word $w = w_1 w_2 \cdots w_n$ let

- $\star w$ denotes $\star w_1 \star w_2 \star \cdots \star w_n$
- $w\star$ denotes $w_1 \star w_2 \star \cdots \star w_n \star$
- $\star w \star$ denotes $\star w_1 \star w_2 \star \cdots \star w_n \star$

We reduce an instance of the Simplified Post corresponding problem
$\left\{ \boxed{\frac{t_1}{b_1}}, \boxed{\frac{t_2}{b_2}}, \ldots, \boxed{\frac{t_k}{b_k}} \right\}$ with the initial domino $\boxed{\frac{t_1}{b_1}}$
to an instance of the Post corresponding problem
$\left\{ \boxed{\frac{\star t_1}{\star b_1 \star}}, \boxed{\frac{\star t_1}{b_1 \star}}, \boxed{\frac{\star t_2}{b_2 \star}}, \ldots, \boxed{\frac{\star t_k}{b_k \star}}, \boxed{\frac{\star \heartsuit}{\heartsuit}} \right\}$.

Rice's theorem

Rice's theorem (languages)

Let C be a class of partially decidable languages and let us define $L_C = \{M;\ L(M) \in C\}$. Then language L_C is decidable if and only if C is either empty or it contains all partially decidable languages.

Corollary

The following languages are undecidable.

- Empty language: $L_C = \{M;\ L(M) = \emptyset\}$ where $C = \{\emptyset\}$
- Finite languages: $L_C = \{M;\ L(M) \text{ is finite}\}$ where C is the set of finite languages
- Regular languages: $L_C = \{M;\ L(M) \text{ is regular}\}$ where C is the set of regular languages
- Decidable languages: $L_C = \{M;\ L(M) \text{ is decidable}\}$ where C is the set of decidable languages

Rice's theorem (functions)

Let C be a class of computable functions and let us define $A_C = \{w_e;\ \varphi_e \in \mathrm{C}\}$. Then language A_C is decidable if and only if C is either empty or it contains all computable functions.

Rice's theorem

Rice's theorem (languages)

Let C be a class of partially decidable languages and let us define $L_C = \{M;\ L(M) \in C\}$. Then language L_C is decidable if and only if C is either empty or it contains all partially decidable languages.

Proof (⇒)

For sake of a contradiction, assume that C is decidable by a Turing machine `C_solver`. WLOG: Turing machine M_{reject} rejecting all inputs does not belong to C and C contains a Turing machine M_{in}. We construct Turing machine `accept_solver` which is a contradiction.

```
Program accept_solver($M, w$)
    Program helper $_{M,w}(x)$
        if $M(w)$ accepts then
            Run $M_{in}(x)$ and return its output
        else
            Run $M_{reject}(x)$ and return its output
    Run C_solver(helper $_{M,w}$) and return its output
```

Note that C is decidable if and only if $\bar{C}$ is decidable. So, if $M_{reject} \in C$, we can consider $\bar{C}$ instead of C. We prove that our `accept_solver` works properly. First, consider a Turing machine M which accept an input w which implies the following statements

- `helper` $_{M}$, w(x) and $M_{in}(x)$ give the same output for all inputs x
- $L(\texttt{helper}_{M,w}) = L(M_{in})$
- `C_solver` accepts both M_{in} and $\texttt{helper}_{M,w}$
- `accept_solver(`M,w`)` accepts

Next, consider a Turing machine M which does not accept an input w which implies the following statements

- both `helper` $_{M,w}$ and M_{reject} do not accect any word
- $L(\texttt{helper}_{M,w}) = L(M_{reject})$
- `C_solver` rejects both M_{reject} and $\texttt{helper}_{M,w}$
- `accept_solver(`M,w`)` rejects

Outline

Decision, search and optimization problem

Decision problem

- In a decision problem we want to decide whether a given instance x satisfies a specified condition.
- Formalized as a language $L \subseteq \Sigma^*$ of positive instances and a decision whether $x \in L$.

Search problem

- In a search problem we aim to find for a given instance x an output y which satisfies a specified condition or information that no suitable y exists.
- Formalized as a relation $R \subseteq \Sigma^* \times \Sigma^*$ which contains all pairs (x, y) satisfying a specified condition.

Optimization problem

- In an optimization problem we moreover require the output y to be maximal or minimal with respect to some measure.
- Formalizad as a problem $\arg\max_{y \in \Sigma^*} \{f(x, y);\ (x, y) \in R\}$ where f is a (partial) ordering of all pairs (x, y).

Time and space complexity of a Turing machine

Definition

Let M be (deterministic) Turing machine which halts on every input and let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function.

- We say that M runs in time $f(n)$ if for any string x of length $|x| = n$ the computation of M over x finishes within $f(n)$ steps.
- We say that M works in space $f(n)$ if for any string x of length $|x| = n$ the computation of M over x uses at most $f(n)$ tape cells.

Definition

Let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function, then we define classes:

DTIME($f(n)$) – class of languages which can be accepted by deterministic Turing machines running in time $O(f(n))$.

DSPACE($f(n)$) – class of languages which can be accepted by deterministic Turing machines working in space $O(f(n))$.

Observation

DTIME($f(n)$) $\subseteq$ DSPACE($f(n)$) for any function $f : \mathbb{N} \mapsto \mathbb{N}$.

Important deterministic complexity classes

Definition

- Class of problems solvable in polynomial time:

$$\mathrm{P} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(n^k)$$

- Class of problems solvable in polynomial space:

$$\mathrm{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}(n^k)$$

- Class of problems solvable in exponential time:

$$\mathrm{EXP} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(2^{n^k})$$

Strong Church-Turing thesis

Strong Church-Turing thesis

Real computation models can by simulated on TM with polynomial delay/space increase.

Notes

- Polynomials are closed under composition.
- Polynomial time algorithms are (usually) efficient in practice.
- Definition of P does not depend on particular computational model we use (*as far as it can be polynomially simulated on a TM*).

Verifier

Definition

A verifier for a language A is a Turing machine M, where $A = \{x;\ \exists y : M \text{ accepts } (x, y)\}$.

Notes

- A string y is called a certificate of x if M accepts (x, y).
- Time and space complexities of a verifier is measured only in terms of $|x|$.
- Hence, if time or space complexity of a verifier is $f(n)$, then the length of y is at most $f(n)$.
- A polynomial time verifier runs in time polynomial in $|x|$.

Nondeterministic Turing machine

Definition

Nondeterministic Turing machine is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states (internal memory)
- Σ is a finite tape alphabet
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states
- $\delta : Q \times \Sigma \to 2^{Q\times\Sigma\times\{R,N,L\}\cup\{\perp\}}$ is a transition function which for every state of Q and every value in the current cell determines a set of possible transitions (a new state, a symbol written on type and a movement of the head).

Idea of computation

- In every step of a computation, nondeterministic TM chooses a transition which leads to an accepting state if such a choice exists.
- Nondeterministic TM looks as a parallel computer which in every step “forks” to evaluate all possible transitions and it accepts a given input if at least one “thread” terminates in an accepting state.
- Nondeterministic Turing machine is not a real computation model in the sense of strong Church-Turing thesis.

Nondeterministic Turing machine

Definition

- Computation of Nondeterministic TM M over string x is a sequence of configurations $C_0, C_1, C_2, \dots,$ where
  - C_0 is the initial configuration and
  - C_{i+1} is obtained from C_i by applying one of possible transition defined by δ.
- Computation is accepting if it is finite and M is in an accepting state in the last configuration.
- String x is accepted by NTM M if there is an accepting computation of M over x.
- Language of string accepted by Nondeterministic TM M is denoted as $L(M)$.

Definition

Let M be a nondeterministic Turing machine and let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function.

- We say that M works in time $f(n)$ if every computation of M over any input x of length $|x| = n$ terminates within $f(n)$ steps.
- We say that M works in space $f(n)$ if every computation of M over any input x of length $|x| = n$ uses at most $f(n)$ cells of work tape.

Nondeterministic Turing machine vs. verifier

Definition

NP is the class of languages that have polynomial time verifiers.

Notes

- NP stands for Nondeterministically Polynomial.
- Nondeterminism corresponds to “guessing” the right certificate y of x.
- Corresponds to a class of search problems where we can check if a given string is a solution to our problem.
- Languages in NP are exactly those which are accepted by nondeterministic polynomial time Turing machines.

Basic nondeterministic complexity classes

Definition

Let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function, then we define classes:

- NTIME($f(n)$) – class of languages accepted by nondeterministic TMs working in time $O(f(n))$.
- NSPACE($f(n)$) – class of languages accepted by nondeterministic TMs working in space $O(f(n))$.

Theorem

Class NP consists of languages accepted by nondeterministic Turing machines working in polynomial time, that is

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k).$$

Theorem

For any function $f : \mathbb{N} \mapsto \mathbb{N}$ we have that

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n)).$$

Turing machines using sublinear space

Motivation

Formally, space complexity includes the size of input and output, although “working space” can be sublinear.

Definition

In case of sublinear space complexity, we use three tapes TM:

- Read-only input tape
- Write-only output tape (head moves only to the right)
- Read-write working tapes

Only the working tape is included into space complexity.

Definition

- Class of problems solvable deterministically in logarithmic space $\mathrm{L} = \mathrm{DSPACE}(\log_2 n)$.
- Class of problems solvable nondeterministically in logarithmic space $\mathrm{NL} = \mathrm{NSPACE}(\log_2 n)$.
- Class of problems solvable nondeterministically in polynomial space $\mathrm{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{NSPACE}(n^k)$.

Relations between classes

Notation

Let $f, g : \mathbb{N} \mapsto \mathbb{N}$. We say that $f(n) = o(g(n))$ if for every $\epsilon > 0$ there exists $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$ it holds that $f(n) < \epsilon g(n)$.
Equivalently, $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Lemma

Let M be a deterministic or nondeterministic Turing machine working in space $f(n)$. Then, there exists a constant c_M such that the number all configurations of M is at most

$$\begin{cases} 2^{c_M f(n)} & \text{if } f(n) = \Omega(n) \\ n2^{c_M f(n)} & \text{if } f(n) = o(n) \text{ using TM with sublinear space complexity.} \end{cases}$$ ①

Theorem

Let $f(n)$ be a function satisfying $f(n) \geq \log_2 n$. For any language $L \in \text{NSPACE}(f(n))$ there is a constant c_L so that $L \in \text{TIME}(2^{c_L f(n)})$. ②

Theorem

The following chain of inclusions holds:

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE} \subseteq \text{EXP}.$$

1. - Number of possible contents on whole tape is $|\Sigma|^{f(n)}$.
   - Number of positions of a head is $f(n)$.
   - Number of states is Q.

   The total number of configurations is $|\Sigma|^{f(n)} f(n) Q = 2^{f(n)\log|\Sigma| + \log f(n) + \log Q} \le 2^{f(n)(\log|\Sigma| + 1 + \log Q)} = 2^{c_M f(n)}$ where $c_M = \log|\Sigma| + 1 + \log Q$.

   In the case of sublinear space complexity, we have to include the number of positions of a head on input tape, so the number of configurations is $|\Sigma|^{f(n)} f(n) Q n \le n 2^{c_M f(n)}$.

2. Let M be a NTM deciding L in space $f(n)$. We construct DTM M' deciding L in time $2^{c_L f(n)}$. M' constructs an oriented graph whose vertices are all configurations of M and configuration C_1 and C_2 are connected by an edge if M switch from the configuration C_1 to C_2 in a single step. Then, M' determines whether there exists an oriented path from the initial configuration to an accepting configuration using BFS or DFS.

   The number of vertices is at most $n2^{c_M f(n)}$ and one vertex can be encoded in $f(n)\log|\Sigma| + \log Q + \log f(n) + \log n$ bits. The maximal out-degree of a vertex is $2^{3\Sigma Q}$. Hence, whole graph can be encoded in $2^{c_G f(n)}$ bits for some constant G depending on M only.

   TM can find a path in a graph in time $\mathcal{O}\left(2^{4c_G f(n)}\right)$.

Savitch's theorem

Theorem

For any function $f(n) \geq \log_2 n$ it holds that $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$.

Proof (using yieldability problem)

Given a NTM M working in space $f(n)$, its two configuration C_1 and C_2 and $t \in \mathbb{N}$, can M switch from the configuration C_1 to C_2 in at most t steps? (1)

```
Program yield($C_1$, $C_2$, $t$)
    if $t = 1$ then
        return accept if $M$ can switch from $C_1$ to $C_2$ in a single step, otherwise reject
    else
        foreach configuration $C'$ do
            if both yield($C_1$, $C'$, $\lfloor t/2 \rfloor$) and yield($C'$, $C_2$, $\lceil t/2 \rceil$) accept then
                return accept
        return reject
```

Corollary

$$\text{PSPACE} = \text{NPSPACE}$$

1

- The initial configuration is known and the accepting configuration can be fixed by modifying M so that tape is clear when M accepts.
- The running time of M is $2^{\mathcal{O}(f(n))}$, so DTM accepts if `yield(`C_{init}, C_{accept}, $2^{\mathcal{O}(f(n))}$`)` accepts.
- Depth of the recursion is $\log 2^{\mathcal{O}(f(n))} = \mathcal{O}(f(n))$.
- One configuration can be stored in $\mathcal{O}(f(n))$ cells, so space complexity is $\mathcal{O}\big(f^2(n)\big)$.

Space hierarchy theorem

Definition

A function $f : \mathbb{N} \mapsto \mathbb{N}$, where $f(n) \geq \log n$, is called *space constructible* if the function that maps 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.

Deterministic Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \mapsto \mathbb{N}$, there exists a language A that is decidable in space $\mathcal{O}(f(n))$ but not in space $o(f(n))$.

Corollary

1. For any two functions $f_1, f_2 : \mathbb{N} \mapsto \mathbb{N}$, where $f_1(n) \in o(f_2(n))$ and f_2 is space constructible,
$$\text{DSPACE}(f_1(n)) \subsetneq \text{DSPACE}(f_2(n)).$$
2. For any two real numbers $0 \leq a < b$,
$$\text{DSPACE}(n^a) \subsetneq \text{DSPACE}(n^b).$$
3. $\text{NL} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \text{DSPACE}(2^{n^k})$

Space hierarchy theorem

Deterministic Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \mapsto \mathbb{N}$, there exists a language A that is decidable in space $\mathcal{O}(f(n))$ but not in space $o(f(n))$.

Idea of the proof

- Create a TM D working in space $\mathcal{O}(f(n))$ deciding a language denoted by A
  - Input is a word w of length n encoding a TM M
  - Mark space of length $f(n)$ for simulation $M(w)$
  - If simulation $M(w)$ uses more that $f(n)$ cells or loops, D rejects
  - D accepts w if and only if M rejects w
- No TM M working in space $o(f(n))$ decides A
  - For constradiction, M decides A in space $o(f(n))$ and consider input w encoding M
  - Assume D has enough space to simulate $M(w)$
  - Then, M accepts $w \Leftrightarrow D$ accepts $w \Leftrightarrow M$ rejects w
- Ensure that M has enough space to simulate $M(w)$
  - D has a fixed tape alphabet simulating M with arbitrary tape alphabet
  - If M runs in space $g(n)$, then D uses $dg(n)$ space
  - From $g(n) \in o(f(n))$ it follows that $\exists n_0\ \forall n \geq n_0 : dg(n) < f(n)$
  - Let input word w be $\langle M \rangle 10^{n_0}$ where $\langle M \rangle$ is the code of M
  - Then $|w| = n \geq n_0$ and D can simulates $M(w)$ in space $f(n)$
- D must always terminate but M may loop
  - If M uses $o(f(n))$ space, then M uses $2^{o(f(n))}$ time or loops
  - D has a counter of steps of M and rejects if the counters exceeds $2^{f(n)}$

Space hierarchy theorem

Deterministic Space Hierarchy Theorem

For any space constructible function $f : \mathbb{N} \mapsto \mathbb{N}$, there exists a language A that is decidable in space $\mathcal{O}(f(n))$ but not in space $o(f(n))$.

Algorithm D deciding some language A

```
Program D(w)
    Let $n$ denotes the length of $w$
    Compute $f(n)$ using space constructibility and mark off this much space
    if $w$ is not of the form $\langle M \rangle 10^{n_0}$ for some TM $M$ then
        Reject
    Simulates $M$ on $w$:
        Count the number of steps and reject if the counter excedeeds $2^{f(n)}$
        Reject if the simulation attempts to use more than $f(n)$ space
    if $M$ accepts $w$ then
        Reject
    else
        Accept
```

Time hierarchy theorem

Definition

A function $f : \mathbb{N} \mapsto \mathbb{N}$, where $f(n) \geq n \log n$, is called *time constructible* if the function that maps 1^n to the binary representation of $f(n)$ is computable in time $O(f(n))$.

Deterministic Time Hierarchy Theorem

For any time constructible function $f : \mathbb{N} \mapsto \mathbb{N}$, there exists a language A that is decidable in time $\mathcal{O}(f(n))$ but not in time $o(f(n)/\log f(n))$.

Corollary

1. For any two functions $f_1, f_2 : \mathbb{N} \mapsto \mathbb{N}$, where $f_1(n) \in o(f_2(n)/\log f_2(n))$ and f_2 is time constructible,
$$\mathrm{DTIME}(f_1(n)) \subsetneq \mathrm{DTIME}(f_2(n)).$$
2. For any two real numbers $0 \leq a < b$,
$$\mathrm{DTIME}(n^a) \subsetneq \mathrm{DTIME}(n^b).$$
3. $\mathrm{P} \subsetneq \mathrm{EXP}$

Time hierarchy theorem

Deterministic Time Hierarchy Theorem

For any time constructible function $f : \mathbb{N} \mapsto \mathbb{N}$, there exists a language A that is decidable in time $\mathcal{O}(f(n))$ but not in time $o(f(n)/\log f(n))$.

Algorithm D deciding some language A

```
Program D(w)
    Let n denotes the length of w
    Compute f(n) using time constructibility and store it as a binary counter
    if w is not of the form $\langle M \rangle 10^{n_0}$ for some TM M then
        Reject
    Simulates M on w:
        Decrease the counter of steps and reject if the counter reaches 0
    if M accepts w then
        Reject
    else
        Accept
```

Polynomial reducibility

Definition

Language A is *polynomially reducible* to language B, denoted by $A \le_m^P B$, if there exists a polynomial time computable function $f : \Sigma^* \mapsto \Sigma^*$ satisfying

$$(\forall w \in \Sigma^*)\,[w \in A \iff f(w) \in B]\,.$$

Observation

- $\le_m^P$ is reflexive and transitive relation (quasiorder).
- If $A \le_m^P B$ and $B \in$ P then $A \in$ P.
- If $A \le_m^P B$ and $B \in$ NP then $A \in$ NP.

NP-completeness

Definition

- Language B is NP-*hard*, if any language $A \in$ NP is polynomially reducible to B.
- Language B is NP-*complete* if $B \in$ NP and B is NP-hard.

Observation

- If arbitrary NP-complete problem has polynomial time algorithm then P $=$ NP.
- If for language B it exists NP-complete problem A polynomially reducible to B (i.e. $A \leq_m^P B$), then B is NP-hard.

NP-complete problem

Tiling

Instance: Set of colors B, natural number s, square grid S of size $s \times s$, in which border cells have outer edges colored by colors in B. Set of tile types K, every tile is a square with edges colored by colors in B.

Question: Is there a valid tiling of S with tiles from K? By a valid tiling we mean placing tiles to cells of S without rotation, so that the tiles sharing a border have matching color and the tiles placed in a border cell have the colors matching outer edge colors of S.

Theorem

Tiling is NP-complete problem.

Observation

Tiling is belongs to NP.

NP-hardness of Tiling

Every language $A \in \mathrm{NP}$ is polynomially reducible to Tiling

- There exists NTM M deciding A in polynomial time $p(n)$.
- For simplification we assume:
  1. $F = \{q_1\}$ where $q_1 \neq q_0$, and $\delta(q_1, a) = \emptyset$ for every $a \in \Sigma$.
  2. Computation of M terminates with an empty tape (accepting configuration is unique).
  3. Tape is one-directional.
- Let x be an instance of A and $s = p(|x|)$.
- Rows encode configurations of computation of M:
  - The set of colors is $\Sigma \cup Q \times (\Sigma \cup \{L, R\})$.
  - Upper and bottom edges of the grid are colored by the initial and accepting configuration, resp.
  - Left and right edges of the grid are colored by λ.
  - Types of tiles are defined by the transition function of $M \dots$
- For every valid tiling there exists a computation of M.
  - The sequence of colors between i-th and $(i+1)$-th rows there exists exactly one color from $Q \times \Sigma$ and other colors are from Q.

Satisfiability (SAT)

Terminology

Literal: Variable (e.g. x) or its negation (e.g. $\overline{x}$).

Clause: Disjunction of literals.

Conjunctive normal form (CNF): a formula is in CNF if it is a conjunction of clauses.

Satisfiability (SAT)

Instance: A formula φ in CNF.

Question: Is there an assignment v of truth values to variables so that $\varphi(v)$ is satisfied?

Cook-Levin theorem

SAT is NP-complete problem.

Satisfiability (SAT)

SAT is NP-hard problem

1. Given a set of colors B and types of tiles K and a grid $s \times s$.
2. Variables are $x_{i,j,k}$ for $i, j = 1, \ldots, s$ and $k \in K$.
3. Assignment $x_{i,j,k} = 1$ means that tiles of type K is places on position (i, j).
4. Every position (i, j) has exactly one tile:
   - $\bigvee_{k \in K} x_{i,j,k}$
   - $\overline{x}_{i,j,k} \vee \overline{x}_{i,j,k'}$ for every color $k \neq k'$.
5. Colors are compatible between positions (i, j) a $(i, j+1)$:
   - $\overline{x}_{i,j,k} \vee \overline{x}_{i,j+1,k'}$ whenever the right edge of the tile k on position (i, j) differs from the left edge of the tile k' on position $(i, j+1)$.
6. Color of upper edge of the grid is compatible with the first row:
   - $\bigvee_{k \in U_j} x_{1,j,k}$ where U_j is the set of tiles with upper edge colored by the same color as j-th column of the upper edge of the grid.
7. Similar clauses ensuring compatibility between position (i, j) and $(i+1, j)$ and all edge of the grid.
8. CNF φ is the conjunction of all presented clauses.

3-Satisfiability

Definition

A formula φ is in k-CNF if it is in CNF and every clause consists of exactly k literals where $k \in \mathbb{N}$.

k-SAT

Instance: A formula φ in k-CNF

Question: Is there an assignment v of truth values to variables so that $\varphi(v)$ is satisfied?

Theorem

3-SAT is NP-complete problem. ①

Note

2-SAT is polynomially solvable.

1. Too short clauses can be prolonged using additional variables, e.g. $x_1 \vee \bar{x}_2$ can be replaced by $(x_1 \vee \bar{x}_2 \vee y)\&(x_1 \vee \bar{x}_2 \vee \bar{y})$. Too long clause $\bar{x}_1 \vee x_2 \vee x_3 \vee \cdots \vee x_k$ can be shortened by $(\bar{x}_1 \vee x_2 \vee y)\&(\bar{y} \vee x_3 \vee \cdots \vee x_k)$ using additional variables.

Vertex Cover

Vertex Cover problem

Instance: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.

Question: Is there a set of vertices $S \subseteq V$ of size at most k so that each edge $\{u, v\} \in E$ has one of its endpoints in S (that is $\{u, v\} \cap S \neq \emptyset$)?

Vertex cover problem is NP-complete

We construct a graph G for a given 3-SAT formula φ.

- For every variable x in φ, add two vertices v_x and $v_{\bar{x}}$ joined by an edge.
- For every clause (e.g. $x \vee \bar{y} \vee z$) in φ, add
  - three vertices c_x, $c_{\bar{y}}$, and c_z and
  - edges $c_x c_{\bar{y}}$, $c_x c_z$ and $c_{\bar{y}} c_z$ (forming a triangle on c_x, $c_{\bar{y}}$, and c_z) and
  - edges $c_x v_x$, $c_{\bar{y}} v_{\bar{y}}$ and $c_z v_z$.
- Let $k = v + 2c$ where v and c is the number of variables and clauses, resp.
- Observe that G has no vertex cover smaller than k.
- G has a vertex cover of size k if and only if φ is satisfiable.

NP-complete problems related to Vertex Cover

Clique

Instance: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.

Question: Does G contain k vertices S such that every pair of vertices in S is connected by an edge in G?

Independent set

Instance: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.

Question: Does G contain k vertices S such that every pair of vertices in S is not connected by an edge in G?

Edge cover

Instance: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.

Question: Does G contain k edges covering all vertices of G?

Hamiltonian cycle

Hamiltonian cycle

Instance: An undirected graph $G = (V, E)$.

Question: Does G contain a cycle throught all vertices?

Travelling salesman problem, TSP

Instance: A complete graph $G = (V, E)$ with weights on edges $w : E(G) \to \mathbb{N}$ and a limit d.

Question: Does G contain a Hamiltonian cycle with total weight at most d.

Theorem

Both Hamiltonain cycle and TSP are NP-complete problems.

3-Dimensional Matching

3-Dimensional Matching

Instance: Set $M \subseteq W \times X \times Y$, where W, X, and Y are sets of size q.

Question: Can we find a perfect matching in M? In particular, is there a set $M' \subseteq M$ of size q so that all triples in M' are pairwise disjoint?

Theorem

3-Dimensional Matching is an NP-complete problem.

Theorem

Existence of a perfect matching in an undirected graph is polynomial problem.

Subset-Sum

Subset-Sum

Instance: Positive integers $a_1, \dots, a_k$ and s which are binary coded.

Question: Is there a subset $A \subseteq \{a_1, \dots, a_k\}$ such that $\sum_{a \in A} a = s$?

Subset-Sum is NP-complete problem (reduction from 3-SAT)

- For a given 3-SAT φ on variables $x_1, \dots, x_v$ and clauses $c_1, \dots, c_d$, we construct $2(v+d)+1$ integers $y_1, \dots, y_v, z_1, \dots, z_v, g_1, \dots, g_d, h_1, \dots, h_d, s$, each one is $v+d$ decimal integer of base 10 as follows.
  1. Let $y_i[j]$ denotes j-th digit of integer y_i.
  2. For a positive literal x_i in a clause c_j we set $y_i[j] = 1$.
  3. For a negative literal $\bar{x}_i$ in a clause c_j we set $z_i[j] = 1$.
  4. Let $y_i[d+i] = z_i[d+i] = 1$ for $i = 1, \dots, v$.
  5. Let $y_j[j] = z_j[j] = 1$ for $j = 1, \dots, d$.
  6. All other digits are 0.
  7. Let $s[j] = 3$ for $j = 1, \dots, d$ and $s[d+i] = 1$ for $i = 1, \dots, v$.
- There is no carry in the sum of all integers $y_1, \dots, h_d$ in decimal representation.
- In the subset A, we have to choose either y_i or z_i for every variable $x_i \Rightarrow y_i \in A$ for positive assignment of x_i.
- For every clause c_j, at least one integer of $y_1, \dots, y_v, z_1, \dots, z_v$ has to contribute by 1 in the j-th digit to the sum $\sum_{a \in A} a \Rightarrow$ at least one satisfied linteral in clause c_j. [1]

1. Observe that there a subset A of integers $y_1, \dots, h_d$ with $\sum_{a \in A} a = s$ if and only if φ is satisfiable.

Subset-Sum

Subset-Sum

Instance: Positive integers $a_1, \dots, a_k$ and s which are binary coded.

Question: Is there a subset $A \subseteq \{a_1, \dots, a_k\}$ such that $\sum_{a \in A} a = s$?

Subset-Sum is NP-complete if integers $a_1, \dots, a_k$ and s are binary coded, but it is polynomial if integers are unary coded.

Dynamic programming algorithm for Subset-Sum

1. Consider logical variables $z_{i,j}$ which is true if there exists $A \subseteq \{a_1, \dots, a_i\}$ such that $\sum_{a \in A} = j$ for $j = 0, \dots, s$.
2. Clearly, $z_{0,0}$ is true and $z_{0,j}$ is false for $j > 0$.
3. Observe that $z_{i,j} = z_{i-1,j} \vee z_{i-1,j-a_i}$ for $i = 1, \dots, k$ and $j = 0, \dots, s$.
4. Time complexity is $\mathcal{O}(ks)$ which is polynomial in the length of the input if all integers $a_1, \dots, a_k$ and s are unary coded.

3-partition is NP-complete even if integers are unary coded

Instance: Positive integers $a_1, \dots, a_{3k}$.

Question: Can integers $a_1, \dots, a_{3k}$ be split into k groups, each with 3 elements, so that the sum of integers in each group is the same (i.e. $\frac{1}{k} \sum_{i=1}^{3k} a_i$)?

Scheduling

Scheduling

Instance: A set of tasks U, processing time $d(u) \in \mathbb{N}$ associated with every task $u \in U$, number of processors m, deadline $D \in \mathbb{N}$

Question: Is it possible to assign all tasks to processors so that the (parallel) processing time is at most D?

Theorem

Scheduling is an NP-complete problem.

Number problem

Definition

Let A be a decision problem and let I be an instance of A. Then

$\mathrm{len}(I)$	denotes the length (=number of bits) of encoding of I when using binary encoding of numbers.
$\max(I)$	denotes the value of a maximum number parameter in I.

We say that A is a number problem, if for any polynomial p there is an instance I of A with $\max(I) > p(\mathrm{len}(I))$.

Examples

Number problems	: Subset-Sum, 3-partition, TSP
Non-number problems	: Hamiltonicity, SAT

Pseudopolynomial algorithm

Definition

We say that an algorithm which solves problem A is pseudopolynomial if its running time is bounded by a polynomial in two variables $\mathrm{len}(I)$ and $\max(I)$.

Notes

- We usually measure complexity of an algorithm only with respect to $\mathrm{len}(I)$.
- If for some polynomial p and for every instance I of A we have that $\max(I) \leq p(\mathrm{len}(I))$ then a pseudopolynomial algorithm is actually polynomial.
- Also, if the numbers in I would be encoded in unary, a pseudopolynomial algorithm would run in polynomial time.
- For example, dynamic programming algorithm for Subset-Sum problem is a pseudopolynomial.

Factorization

Trivial algorithm for factorization

```
Input: An integer $n$
$i := 2$
while $i \cdot i \leq n$ do
    if $i$ divides $n$ then
        Output: $i$
        $n := n/i$
    else
        $i := i + 1$
    if $n > 1$ then
        Output: $n$
```

Complexity

- Complexity of the algorithm is $\mathcal{O}(\sqrt{n})$.
- If a given integer n is encoded in unary, then $\sqrt{n}$ is a polynomial function, so the algorithm is pseudopolynomial.
- If a given integer n is encoded in binary using $k = \lceil \log n \rceil$ bits, complexity is $\mathcal{O}\left(2^{k/2}\right)$ which is exponential.

Strong NP-completeness

Definition

- Let A be a decision problem and let p be a polynomial. Then $A(p)$ denotes the restriction of problem A to instances I which satisfy $\max(I) \leq p(\mathrm{len}(I))$.
- We say that problem A is strongly NP-complete, if there is a polynomial p for which $A(p)$ is NP-complete.

Notes

- Any NP-complete problem which is not a number problem is strongly NP-complete.
- If there is a strongly NP-complete problem which can be solved by a pseudopolynomial algorithm then $\mathrm{P} = \mathrm{NP}$.

Unary coding

- Pseudopolynomial = polynomial when considering unary encoding.
- Strongly NP-complete = NP-complete even when considering unary encoding.

Strong NP-completeness of number problems

”Weighted” versions of NP-complete problems

”Weighted” versions of strongly NP-complete problems usually remain strongly NP-complete, e.g.

- Hamiltonicity $\rightarrow$ Travelling Salesman problem
- Vertex cover $\rightarrow$ weighted vertex cover

Number problems

- 3-partition: strongly NP-complete
- Subset-Sum: NP-complete, pseudopolynomial algorithm
- Factorization: pseudopolynomial algorithm
- Prime testing: polynomial algorithm

Optimization problem

Definition

We define optimization problem as a triple $A = (D_A, S_A, \mu_A)$, where

- $D_A \subseteq \Sigma^*$ is a set of instances,
- $S_A(I)$ assigns a set of feasible solutions to each $I \in D_A$
- $\mu_A(I, \sigma)$ assigns a positive rational value to every $I \in D_A$ and every feasible solution $\sigma \in S_A(I)$.

Optimum solution

- If A is a maximization problem, then an optimum solution to instance I is a feasible solution $\sigma \in S_A(I)$, which has the maximum value $\mu_A(I, \sigma)$.
- If A is a minimization problem, then an optimum solution to instance I is a feasible solution $\sigma \in S_A(I)$, which has the minimum value $\mu_A(I, \sigma)$.
- The value of an optimum solution is denoted $\mathrm{opt}(I)$.

Approximation algorithm for Vertex cover

Minimal vertex cover problem

Instance: An undirected graph $G = (V, E)$.

Output: Find a vertex cover (i.e. $S \subseteq V$ so that each edge $\{u, v\} \in E$ has one of its endpoints in S) with minimal size.

Observation

If there exists a polynomial algorithm for minimal vertex cover problem, then P $=$ NP.

Approximation algorithm

```
Input: Graph $G$
1 Let $M$ be a maximal matching of $G$
2 Let $S$ be the set of both endpoints of all edges of $M$
Output: $S$
```

Approximation algorithm for Vertex cover

Approximation algorithm

```
Input: Graph $G$
1 Let $M$ be a maximal matching of $G$
2 Let $S$ be the set of both endpoints of all edges of $M$
Output: $S$
```

Analysis

- The running time is polynomial
- The algorithm outputs a vertex cover
  - If there is an edge uv uncovered by S, then $M \cup \{uv\}$ is a matching, so M is not a maximal matching
- If S' is the minimal vertex cover, then $|S| \leq 2|S'|$
  - For every $uv \in M$, vertex u or v is covered by S'
  - Hence, $|S| = 2|M| \leq 2|S'|$

Approximation algorithm

Definition

Algorithm R is called approximation algorithm for optimization problem A, if for each instance $I \in A$ the output of $R(I)$ is a feasible solution $\sigma \in S_A(I)$ (if there is any).

- If A is a maximization problem, then $c \geq 1$ is an approximation ratio of algorithm R, if for all instances $I \in D_A$ we have that $\mathrm{opt}(I) \leq c \cdot \mu_A(I, R(I))$.
- If A is a minimization problem, then $c \geq 1$ is an approximation ratio of algorithm R, if for all instances $I \in D_A$ we have that $\mu_A(I, R(I)) \leq c \cdot \mathrm{opt}(I)$.

Example: Vertex cover

- The algorithm finds a vertex cover S of size at most $2|S'|$ where S' is the minimal vertex cover.
- Therefore, the approximation ratio is 2.

Inapproximability of maximal independent set

If there exists a polynomial algorithm for maximal independent set with approximation error c for some $c > 1$, then P= NP.

Travelling Salesman problem

TSP with triangular inequality

TSP is NP-complete even if weights of edges satisfies the triangular inequality.

Inapproximability of TSP (without triangular inequality)

If for some $c > 1$ there exists a polynomial algorithm for TSP with approximation error c, then $\mathrm{P} = \mathrm{NP}$.

Approximation of TSP with triangular inequality

For TSP with triangular inequality there exists a polynomial algorithm with approximation error $3/2$.

Bin packing

Bin packing

Instance: Set of k items of rational sizes $a_1, \dots, a_k \in [0, 1]$.

Constrain: Splitting of items to pairwise disjoint bins $B_1, \dots, B_m$, which satisfy that the sum of sizes of items in every bin is at most 1.

Objective: Minimize the number of bins m.

Any-Fit algorithm

Take items as they come and for each item try to find a bin in which it fits. If no such bin exists, add a new bin with the item in it.

Any-Fit algorithm has approximation error 2

- The optimal number of bins m' is at least $\sum_{i=1}^{k} a_i$.
- For every pair of different bins B_i and B_j it holds that $\sum_{l \in B_i} a_l + \sum_{l \in B_j} a_l > 1$.
- By summing last inequality for pairs $(1, 2), (2, 3), \dots, (m-1, m), (m, 1)$ we obtain $2 \sum_{i=1}^{m} \sum_{l \in B_i} l > m$.
- Hence, $m < 2 \sum_{i=1}^{m} \sum_{l \in B_i} l = 2 \sum_{i=1}^{k} a_i \leq 2m'$.

Bin packing

Sorted-Any-Fit algorithm

Sort the items by their value decreasing. Take items from the biggest to smallest and for each item try to find a bin in which it fits. If no such bin exists, add a new bin with the item in it.

Best-fit algorithm

Take items as they come and for each item try to find the most full bin in which it fits. If no such bin exists, add a new bin with the item in it.

Notes

- Best-fit algorithm has approximation error 1.7.
- If m' is the optimal number of bins and m is the number of bins found by Sorted-Any-Fit algorithm, then $m \leq \frac{11}{9} m' + 4$.
- There is no polynomial algorithm with approximation error smaller than $3/2$.

Fully polynomial time approximation scheme

Definitions

- Algorithm ALG is an approximation scheme for an optimization problem A, if on the input instance $I \in D_A$ and a rational number ϵ it returns a solution $\sigma \in S_A(I)$ with approximation ratio $1 + \epsilon$.
- If ALG works in polynomial time with respect to $\mathrm{len}(I)$, then it is a polynomial time approximation scheme.
- If ALG works in polynomial time with respect to both $\mathrm{len}(I)$ and $\frac{1}{\epsilon}$, it is a fully polynomial time approximation schema (FPTAS).

Fully polynomial time approximation scheme

Optimization version of Subset-Sum problem

Instance: Positive integers $a_1, \dots, a_k$ and s which are binary coded.

Feasible solution: A subset $A \subseteq \{a_1, \dots, a_k\}$ such that $\sum_{a \in A} a \leq s$.

Objective: Maximize $\sum_{a \in A} a$.

Algorithm for the optimization version of Subset-sum

```
Input: Numbers $a_1, \dots, a_k, s \in \mathbb{N}$ and $\epsilon > 0$
1 $\delta = \sqrt[k-1]{\frac{1}{1+\epsilon}}$
2 $A_0 = \{0\}$
3 for $i = 1$ to $k$ do
4     $A_i = A_{i-1}$
5     for $t \in A_{i-1}$ do
6         $t' = t + a_i$
7         if $t' \leq s$ and $A_i$ does not contain an integer between $\delta t'$ and $t'$ then
8             Insert $t'$ into $A_i$
9 return max $A_k$
```

Unsatisfiability

Unsatisfiability (UNSAT)

Instance: Formula φ in CNF

Question: Is it true, that for any assignment v of values to variables $\varphi(v) = 0$ (i.e. φ is unsatisfiable)?

Notes

- We do not know a polynomial time verifier for problem UNSAT, this problem most probably does not belong to class NP.
- Language UNSAT is (more or less) the complement of language SAT, because for any formula φ in CNF we have $\varphi \in \text{UNSAT} \iff \varphi \notin \text{SAT}$.

Similar "complementary" problems

- Does a given graph contain no Hamiltonian cycle?
- Given a finite set of integers, does every non-empty subset have a non-zero sum?
- Given a graph G and $k \in \mathbb{N}$, is the size of maximal clique in G at most k?

Class coNP

Definition

We say that language A belongs to the class coNP if and only if its complement $\overline{A}$ belongs to the class NP.

Notes

- For instance UNSAT belongs to coNP. (It is easy to recognize instances which do not encode a formula.)
- Language L belongs to coNP, iff there is a polynomial time verifier V which satisfies that $L = \{x;\ (\forall y)\ V(x, y) \text{ accepts }\}$.
- Clearly, $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$.
- It is not known whether $\mathrm{NP} = \mathrm{coNP}$.
- Clearly, if $\mathrm{P} = \mathrm{NP}$, then $\mathrm{NP} = \mathrm{coNP}$.
- A problem A is polynomially reducible to a problem B if and only if the complementary problem $\bar{A}$ is polynomially reducible to the complementary problem $\bar{B}$.

coNP-completeness

Definition

Problem A is coNP-complete, if

(i) A belongs to class coNP and

(ii) every problem B in coNP is polynomially reducible to A.

Notes

- Language A is coNP-complete, if and only if complement $\overline{A}$ is NP-complete.
- For example UNSAT is an coNP-complete problem.
- If there is an NP-complete language A, which belongs to coNP, then NP $=$ coNP.

Integer factorization

Instance: Positive integers m and n.

Question: Is there an integer k dividing m and satisfying $1 < k < n$.

- Clearly, Integer factorization belongs to NP.
- Agrawal–Kayal–Saxena primality test implies that Integer factorization belongs to coNP.
- It is not known whether there exists a polynomial algorithm for Integer factorization.

Class #P

Motivation

How hard it is to determine the number of

- Hamiltonian cycles in a given graph?
- satisfied assignments of given CNF/DNF formula?
- perfect matchings of a given graph?

These counting problems can be solved in polynomial space.
Are there polynomial time algorithms?

Definition

Function $f : \Sigma^* \mapsto \mathbb{N}$ belongs to class #P, if there is a polynomial time verifier V such that each $x \in \Sigma^*$ satisfies $f(x) = |\{y;\ V(x,y) \text{ accepts}\}|$.

Notes

- We can associate a function #A in #P with every problem $A \in$ NP (given by the “natural” polynomial time verifier for A).
- Natural verifier verifies that y is a solution to the search problem corresponding to A.
- For example the natural verifier for SAT accepts a pair (φ, v), if φ is a CNF and v is a satisfying assignment for φ.
- Then $\#\text{SAT}(\varphi) = |\{v;\ \varphi(v) = 1\}|$.

Class #P (properties)

Nonzero value of $f \in \#\mathrm{P}$

Instance: $x \in \Sigma^*$

Question: $f(x) > 0$?

Notes

- Nonzero Value of $f \in \#\mathrm{P}$ belongs to NP.
- Value of $f \in \#\mathrm{P}$ can be obtained by using polynomial number of queries about an element belonging to the set $\{(x, N);\ f(x) \geq N\}$.
- Value of $f \in \#\mathrm{P}$ can be computed in polynomial space.

Reducing a function to another function

Definition

Function $f : \Sigma^* \mapsto \mathbb{N}$ is polynomial reducible to function $g : \Sigma^* \mapsto \mathbb{N}$ if there are functions $\alpha : \Sigma^* \times \mathbb{N} \mapsto \mathbb{N}$ and $\beta : \Sigma^* \mapsto \Sigma^*$, which can be computed in polynomial time and

$$\forall x \in \Sigma^* : \ f(x) = \alpha(x, g(\beta(x)))$$

Note

This corresponds to the fact that f can be computed in polynomial time with one call of function g (if this call is a constant time operation).

#P-completeness

Definition

We say that function $f : \Sigma^* \mapsto \mathbb{N}$ is #P-complete, if

(i) $f \in \#P$ and

(ii) every function $g \in \#$P is polynomially reducible to f.

Notes

- For example #SAT, #Vertex Cover and other counting versions of NP-complete problems are #P-complete.
- There are problems in P such that their counting versions are #P-complete.

Perfect matching in a bipartite graphs

The following problem is in P but it is #P-complete.

Instance: $G = (A \cup B, E)$ where $E \subseteq A \times B$ and $A \cap B = \emptyset$ and $|A| = |B|$.

Question: Is there a matching in G of size $|A| = |B|$?

Permanent of a matrix

Definition

Let A be a matrix of type $n \times n$. Then we define permanent of A as

$$\mathrm{perm}(A) = \sum_{\pi \in S(n)} \prod_{i=1}^{n} a_{i,\pi(i)},$$

where $S(n)$ is a set of permutations over set $\{1, \dots, n\}$.

Notes

- Like “determinant” without a sign of permutation.
- If A is a adjacency matrix of a bipartite graph G, then $\mathrm{perm}(A)$ computes the number of perfect matchings of G.
- Function perm is #P-complete.

#DNF-SAT

#DNF-SAT

- Term is a conjunction of literals.
- Disjunctive normal form (DNF) is a disjunction of terms.

Instance: Formula φ in DNF.

Question: Is there an assignment v such that $\varphi(v)$ is satisfied?

This problem is in #P while it is decidable in polynomial time.

Parametrized version of Vertex Cover problem

Vertex Cover problem

Instance: An undirected graph $G = (V, E)$ and an integer $k \geq 0$.

Question: Is there a set of vertices $S \subseteq V$ of size at most k so that each edge $\{u, v\} \in E$ has one of its endpoints in S?

This problem is strongly NP-complete. How the complexity depends on k?

Vertex Cover problem parametrized by the size k of a cover

Instance: An undirected graph $G = (V, E)$.

Question: Is there a set of vertices $S \subseteq V$ of size at most k so that each edge $\{u, v\} \in E$ has one of its endpoints in S?

- Solvable by a brutal-force algorithm in time $\mathcal{O}(|V|^k |E|)$.
- For every $k \in \mathbb{N}$, this algorithm is polynomial although the degree of the polynomial depends on k.

Fix-parameter tractability

Goal

- Complexity of brutal-force algorithm for k-vertex cover problem is $\mathcal{O}\left(|V|^k|E|\right)$.
- We would prefer a polynomial time algorithm for k-vertex cover such that the polynomial is independent on k.
- But there is no algorithm polynomial in both k and the size of input (unless P= NP).
- We describe an algorithm running in time $\mathcal{O}(p(|V|)f(k))$ where p is a polynomial function and f is an arbitrary function.

Fix-parameter tractability: Vertex Cover

Algorithm

```
Program Has_Cover(Graph G, size of a vertex cover k)
    if G has no edge then
        return Accept
    if $k = 0$ then
        return Reject
    $uv \leftarrow$ an arbitrary edge of $G$
    if Has_Cover($G \setminus u, k-1$) or Has_Cover($G \setminus v, k-1$) accepts then
        return Accept
    else
        return Reject
```

Observations

- Let G be a graph and uv its edge and $k \geq 1$. Then, G has a vertex cover of size k if and only if $G \setminus u$ or $G \setminus v$ has a vertex cover of size $k-1$.
- Time complexity is $\mathcal{O}(2^k|V|)$.

Fix-parameter tractability: Definitions

Definitions

- A parameter of a language L is a function $k : \Sigma^* \to \mathbb{N}$.
- A parametrized language is a language with a parameter.
- A parameterized language is fix-parameter tractable (FPT) if there exists an algorithm A deciding L, a function $f : \mathbb{N} \to \mathbb{N}$ and a polynomial function $p : \mathbb{N} \to \mathbb{N}$ such that A decides every instance x in time $\mathcal{O}(p(|x|) \cdot f(k(x)))$.

Notes

- Usually, the parameter is a natural property of the problem and it may be a part of the input.
- A language may have many interesting parametrizations.

Observation

A language L with a parameter k is fix-parameter tractable if and only if there exists an algorithm A' deciding L, a function $f' : \mathbb{N} \to \mathbb{N}$ and a polynomial function $p' : \mathbb{N} \to \mathbb{N}$ such that A decides every instance x in time $\mathcal{O}(p'(|x|) + f'(k(x)))$. ①

1. In proves of both implications, we can use the same algorithm and only determine functions giving complexity.
   - $\Rightarrow$ Since $ab \leq a^2b^2$ for $a, b \geq 0$, it follows that $p(|x|) \cdot f(k(x)) \leq p^2(|x|) + f^2(k(x))$.
   - $\Leftarrow$ $p'(|x|) + f'(k(x)) \leq 2p'(|x|) \cdot f'(k(x))$ assuming $p'(|x|), f'(k(x)) \geq 1$.

Vertex Cover: Kernelization

Algorithm

```
Input: Graph $G$, size of a vertex cover $k$
1 for every vertex $v$ in $G$ do
2     if $\deg(v) > k$ then
3         $G \leftarrow G \setminus v$
4         $k \leftarrow k - 1$
5 Remove all isolated vertices from $G$
6 if $k < 0$ or $|E| > k^2$ then
7     return A canonical negative instance
8 return $G$, $k$
```

Observations

- Let v be a vertex of G of degree larger than k. Then G contains a vertex cover of size k if and only if $G \setminus v$ contains a vertex cover of size $k - 1$.
- If a graph of maximal degree k has a vertex cover of size k, then it has at most k^2 edges.
- The resulting graph contains at most k^2 edges and $2k^2$ vertices.
- Complexity is $\mathcal{O}(|V| + |E|)$ and vertex cover can be found in $\mathcal{O}\left(|V| + |E| + k^2 2^k\right)$.

Kernelization

Definition

A kernelization of a language L with a parameter k is a function $g : \Sigma^* \to \Sigma^*$ if

1. g computable in polynomial time and
2. $x \in L \Leftrightarrow g(x) \in L$ for every $x \in \Sigma^*$ and
3. there exists a function $f : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \Sigma^*$ it holds that $|g(x)| \leq f(k(x))$.

Theorem

A decidable parametrized language has a fix-parameter tractable if and only if it has kernelization.

Proof

$\Leftarrow$: Run the kernelization and then the decider.

$\Rightarrow$: Let A be an algorithm of running time $\mathcal{O}(p(|x|) \cdot f(k(x)))$ and x be an instance.

- If $|x| \leq f(k(x))$, then x is already kernelized.
- If $f(k(x)) \leq |x|$, then run A and return a canonical positive or negative instance depending on whether $x \in L$. Running time is $p(|x|) \cdot f(k(x)) \leq |x| \cdot p(|x|)$.