Pina Kolling
piko0011

Summary: Lecture 2

Summary for the chapters *7.1 Complexity classes* and *7.2 Hierarchy problem.* [3]

Complexity classes

Background knowledge:

A complexity class is a set which contains problems with similar complexities. The complexities are examined in regards of a specific ressource, for example time or space. For the problems the most efficient solution/algorithm is analysed.

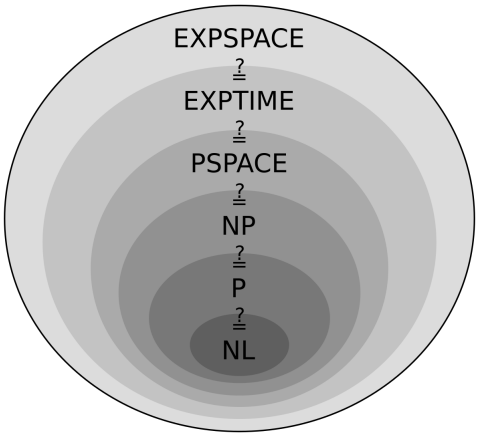


Figure 1: Complexity classes [2]

Usually the complexity depends on the input size. With the asymptotic complexity, classes are build, which are the complexity classes. [4]

Summary:

Parameters of complexity classes: [1, 3]

- **Model of computation:**
  here: multistring Turing Machine
- **Mode of computation:**
  for example: deterministic or non-deterministic (deterministic: the computer will always produce the same output for a given input while going through the same states, non-deterministic: can show different behaviors for the same input)
- **Resources:**
  something *expensive* that the machine uses up, for example: time or space
- **Restrictions/Bound:**
  for example: upper bound, lower bound as a function $f : \mathbb{N} \to \mathbb{N}$

Definition of complexity classes: [3]
The complexity class is the set of all languages which are decided by a Turing Mashine M that is operating in the defined mode and for any input x, M uses at most $f(|x|)$ units of the defined resource.

Pina Kolling
piko0011

Definition proper complexity function: [3, 1, 4]

- $f : \mathbb{N} \to \mathbb{N}$
- $\forall n \in \mathbb{N}\ f(n+1) >= f(n)$ (f is non-decreasing)
- It exists a multistring Turing Machine M that fullfills the following conditions with an input of size n:
  - M halts after $O(n + f(n))$ steps (runs in time $O(n + f(n))$)
  - M uses $O(f(n))$ space
  - M maps 1^n to $1^{f(n)}$

Examples of proper functions: [3]

$$f(n) = \log n^2$$
$$f(n) = n \log n$$
$$f(n) = n^2$$
$$f(n) = n^3 + 3n$$
$$f(n) = 2^n$$
$$f(n) = \sqrt{n}$$
$$f(n) = n!$$

If the function f and g are proper, $f + g$, $f \cdot g$ and 2^g are proper, too.

Definition precise Turing Machine: [3, 1]
A multistring Turing Machine M is called a precise Turing Machine, if there are functions f and g such that, for every input x of length n, M stops after exactly $f(n)$ steps with exactly $g(n)$ blanks on strings $2, ..., k$.

If M is a precise Turing Machine and f is a proper complexity functon such that, M decides a language in $f(n)$, then there exists a precise Turing Machine M' of the same type as M which decides the same language in $O(f(n))$.

Complexity classes: [3, 1]

Class name	Description
TIME(f)	deterministic time
SPACE(f)	deterministic space
NTIME(f)	non-deterministic time
NSPACE(f)	non-deterministic space

(f is a proper complexity function)

Sometimes f is not a particular function but a family of function which are parametrized by an integer $k >= 0$.

Class	Function	Description
P	$\bigcup_{k>=0}$TIME(n^k)	Polynomial time
NP	$\bigcup_{k>=0}$NTIME(n^k)	Non-deterministic polynomial time
EXP	$\bigcup_{k>=0}$TIME(2^{n^k})	Exponential time
L	SPACE$\log n$	Logarithmic space
NL	NSPACE($\log n$)	Non-deterministic logarithmic space
PSPACE	$\bigcup_{k>=0}$SPACE(n^k)	Polynomial space
NPSPACE	$\bigcup_{k>=0}$NSPACE(n^k)	Non-deterministic polynomial space

Pina Kolling
piko0011

Complement classes: [3, 1]

For a string that is part of a language, one *yes* input needs to be found. For a string to be not part of a lanugage, all the paths must be a *no*. The complement of a language $L \subseteq \Sigma^*$ is the set of all valid inputs that do not belong to L. It is denoted as $\bar{L}$ with $\bar{L} = \Sigma^* - L$. This can be extended to decision problems. The complement of a decision problem A is called A COMPLEMENT. The *yes* and *no* answers on an Turing Machine can be switched to solve the complement problems.

The complement of a complexity class C, the class of all the complements is denoded as coC. The deterministic classes are closed under complement, for examlpe $coP = P$. That does not cound for non-deterministic classes.

Hierarchy problem

Definitions: [3, 1]
Let $f(n) >= n$ be a proper complexity function and H_f a time-bounded version of the HALTING language H with

$$H_f = \{M; x : M \text{ accepts input } x \text{ after at most } f(|x|) \text{ steps}\}.$$

M is a deteministic multistring Turing Machine. The following can be concluded:

$$H_f \in \text{TIME}(f(n)^3)) \qquad \text{and} \qquad H_f \notin \text{TIME}(f(\lfloor \frac{n}{2} \rfloor)$$

If $f(n) >= n$ is a proper complexity function, then the class TIME($f(n)$) is strictly contained within TIME($(f(2n+1))^3$). This has the consequence, that there is an infitely growing proper hierarchy of clompexity classes within P: P $\neq$TIME(n^k) for every k and P$\subset$EXP because P$\subseteq$TIME(2^n) $\subset$TIME($2^{(2n+1)^3}$) $\subseteq$EXP.

If $f(n)$ is a proper complexity function, then the class SPACE($f(n)$) is a proper subset of SPACE($f(n) \log f(n)$)

There is a recursive function $f : \mathbb{N} \to \mathbb{N}$ such that TIME($f(n)$) =TIME($2^{f(n)}$).

Questions and problems

- I did not understand the proofs of the hierarchy problem yet but there will be more time and afford put into it.
- What is a parametrized function family? Are they numbered?

References

[1] Martin Berglund. *Lecture notes in Computational Complexity.*

[2] *Complexity classes diagram image source.* `https://en.wikipedia.org/wiki/Complexity_class`.

[3] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[4] Prof. Dr. Thomas Schwentick. *Lecture notes in Grundbegriffe der theoretischen Informatik.* `https://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Archiv/Bachelor_LA_GyGe_Inf_Modellv/_Module/INF-BfP-GTI/index.html`.

Pina Kolling
piko0011

Summary: Lecture 3

Summary for the chapter *7.5* until page 150. [3, 4]

Background knowledge

The P vs. NP problem is an unsolved problem in complexity theory. Mathematical problems which are solved by a computer are classified as P or NP problems: All problems that can be solved efficiently by a computer belong to the class P. In the case of NP problems it is unknown whether they can be solved efficiently or not. In this context, efficient means that the required computing time of a solution algorithm grows at most polynomially (e.g. quadratically) with increasing complexity. The only thing that is currently clear is that a correct solution to an NP problem can be checked for correctness in polynomial time.
Developing an algorithm for an NP problem is usually very difficult. Computer scientists and mathematicians do not only try to work out effective algorithms, they also try to determine whether P = NP. In other words, they are trying to find out whether P and NP are really different problem classes or whether it is possible to solve NP problems in polynomial time as well. Scientists all over the world are trying to prove that P != NP.
The reason why experts wish NP problems to remain almost unsolvable is called cryptography. Unlike many other fields, complexity in cryptography is not only desirable, but necessary. It is important to know that most encryption methods used today are based solely on the fact that the effort to *guess* the key is too high. The problem of *guessing* is therefore an NP problem. Not only theoretically, but also practically, the proof of the solvability of NP problems means the end of all currently used encryption methods.
But there is still hope for cryptography. The NP problem has often been supposedly solved. Both P = NP and P != NP have been attempted to be proven many times. It turned out that every single solution and every single proof attempt so far turned out to be wrong or incomprehensible. [1, 4, 5]

Basic relations between complexity classes

The hierarchy theorem shows how deterministic classes of the same kind (time or space) relate to each other. Here are the relationships between classes of a different kind exmined: P and NP. If $f(n)$ is a proper complexity function:

$$\text{SPACE}(f(n)) \subseteq \text{NSPACE}(f(n)) \text{ and } \text{TIME}(f(n)) \subseteq \text{NTIME}(f(n)) \tag{1}$$

$$\text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n)) \tag{2}$$

$$\text{NSPACE}(f(n)) \subseteq \text{TIME}(k^{\log n + f(n)}) \tag{3}$$

Proof for (1)**:**
Any deterministic Turing Machine is also a non-deterministic Turing Machine, which leads to $\text{SPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$ and $\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n))$.

Proof for (2)**:**
To prove (2), a language $L \in \text{NTIME}(f(n))$ is considered. It exists a precis non-deterministic Turing Machine M that decides L in time $f(n)$ and for the prove a deterministic Turing Machine M' has to be designed that decides L in space $f(n)$. Memory space can be reused, time cannot. There are d choices in every step: $1, ..., d$. For each choice the non-determinstic Turing Machine is simulated and the previous simulations can be deleted to reuse space. Only the currently simulated and the next simulation need to be saved and tracked.

Configuration of a Turing Machine

Snapshot of the computation of M with an given input x. For a k-string Turing Machine M the configuration is a $2k+1$-tuple with the state, the k strigs and the k head positions.

Proof for (3)**:**
This proof involves a method for space-bounded Turing Machines, which is called **reachability method**.
For this, a multistring non-deterministic Turing Machine M with input and output is given, which decides a lanugage L within space $f(n)$.
A determistic method to simulate the nondeterministic computation of M has to be found and it has to be in time $c^{\log n+f(n)}$, where n is the length of the input and c is a constand depending on M. The amount of configurations for M is at most $c^{\log n+f(n)}$ with c being a constand depending on M.
Then a configuration graph $G(M,x)$ is defined, the nodes are the possible configurations and the edges is the input. If an accepting configuration can be reached with an input, the word is contained. In this way, the problem was reduces to the REACHBILITY problem, which was already prooved to be in polynomial time.
If the graph is too big to construct, ist must be implicit. (An implicit graph representation is a graph edges are not represented as explicit objects in a computer's memory, but rather are determined algorithmically from some other input, for example a computable function. [2])

Combining the previous knowledge, it can be concluded:

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

and

$$\text{L} \subset \text{PSPACE}$$

This means, that at least on of the inclusions has to be proper.

Questions: Why does the meaning of n change?

Problem: GRAPH REACHABILITY

Given a graph G and two nodes $n_1, n_2 \in V$, is there path from n_1 to n_2?
(A graph $G = (V, E)$ is a finite set V of nodes and a set E of edges (node pairs).)

Savitch's theorem

$$\text{REACHABILITY} \in \text{SPACE}(\log^2 n)$$

Proof:
PATH$(startnode, endnode, m)$ checks if there is a path from startnode to endnode with the length of at most 2^m.
The parameters of the function can be split up (like divide and conquer):

$$\text{PATH}(i,j,m) \Leftrightarrow \exists k : \text{PATH}(i,k,m-1) \text{ and } \text{PATH}(k,j,m-1)$$

Some internal node k is choosen and it is checked recursively if there is a path from *startnode* to k and from *endnode* to n. This is then recursively done with the new midpoint.

Pina Kolling
piko0011

There can be $\log n$ many segments to work on – the recursion depth and the record size both are at most $\log n$. This leads to a recursion stack of $O(\log^2 n)$.
The complexity function is at least $\log n$.

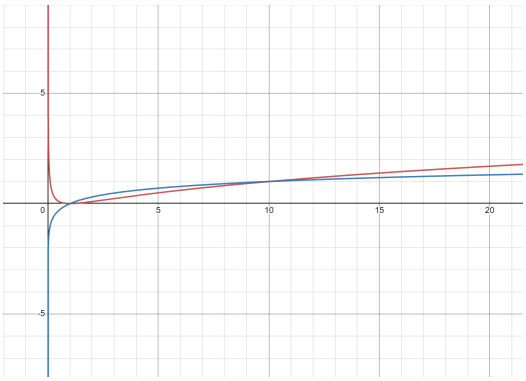


Figure 1: Red graph $f(n) = \log^2 n$ and blue graph $g(n) = \log n$

References

[1] Dr Datenschutz (Website). *P vs. NP: Ein Geschenk der Informatik an die Mathematik*. Last opened 11.11.2022. URL: https://www.dr-datenschutz.de/p-vs-np-ein-geschenk-der-informatik-an-die-mathematik/#:~:text=Hierbei%20werden%20von%20einem%20Computer,effizient%20l%C3%B6sen%20lassen%20oder%20nicht..

[2] Wikipedia (Website). *Implicit graph*. Last opened 12.11.2022. URL: https://en.wikipedia.org/wiki/Implicit_graph.

[3] Martin Berglund. *Lecture notes in Computational Complexity*.

[4] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[5] Prof. Dr. Thomas Schwentick. *Lecture notes in Grundbegriffe der theoretischen Informatik*. https://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Archiv/Bachelor_LA_GyGe_Inf_Modellv/_Module/INF-BfP-GTI/index.html.

Pina Kolling
piko0011

Summary: Lecture 4

Summary for the chapter *7.3* from page 150 on. [3]

Nondeterministic Turing Machine

A nondeterministic Turing machine (NTM) has states, which have more than one possible next state for an action. The states are not completely determined by its action and the current symbol it sees, (unlike a deterministic Turing Machine).
NTMs are for example used in thought experiments. One of the most important problems in is the P versus NP problem: How difficult it is to simulate nondeterministic computation with a deterministic computer? [1, 3]

Asymmetry of non-deterministism

Asymmetry of nondeterministic acceptance:

- Example: find out if a formula φ is satisfiable ($\varphi \in SAT$):
  - choose truth values for the variables nondeterministically
  - check if they make φ become true
- this approach seems to be unpractical so check whether φ is not satisfiable ($\varphi \in \overline{SAT}$)
- Question whether NP = coNP is a statement about *all* options

Asymmetry of nondeterministic space:

- Example: REACHABILITY $\in$ NL
  - starting at start node 1
  - algorithm walks through nondeterminstically chosen edges $\leq n$ times
  - only current position is remembered ($\log n$ space)
  - accepts if current node is node n
- this approach seems to be unpractical so check if node n is *not* reachable from node 1

$\log n$ space

A graph algorithm using $O(\log n)$ space stores a fixed number of pointers, independent of n, and manipulates them in some way. [2]

Nondeterministically computing functions

- A nondeterminstic Turing Machine M computes a function f if the the following hold for every input x:
  - one of the computations of M stops in the halting state $h$with the correct result $f(x)$ on the output tape
  - all computations that do not correctly output $f(x)$ stop instead in a *no*-state (this path failed then)

Problem: GRAPH REACHABILITY

Given a graph G and two nodes $n_1, n_2 \in V$, is there path from n_1 to n_2?
A graph $G = (V, E)$ is a finite set V of nodes and a set E of edges as node pairs.

REACHIBILITY can be nondeterministically solved in space $\log n$.

Pina Kolling
piko0011

Immerman-Szelepscènyi

Theorem (Counting problem):
Given a graph G and a start node x, the number of nodes that are reachable from x in G can nondeterministically be computed in space $\log n$ (where n is the number of nodes of G).

- can be non-deterministically solved
- solving as an extension to REACHABILITY
- counting of nodes that can *not* be reached similar: substract result from n
  $\rightarrow$ counting problem and its complement are identical

Algorithm:

- nodes $1, ..., n$ with start node 1
- $S(i)$ is the set of nodes which are reachable from the startnode with a pathlength of i
  - $S(0)$ will contain node 1
  - $s(1)$ will contain all neighbours of 1
- Algorithm consitsts out of 4 nested for-loops:
  - outer for loop:
    - computes number of nodes reachable from initial node (for loop with k steps) as $|S(1)|, |S(2)|, ..., |S(n-1)|$
    - $|S(n-1)|$ is the desired answer (n is the number of nodes)
    - $|S(0)| = 1$ (contains only start node)
    - $|S(k)|$ is computed after producting $|S(k-1)|$
    - in each step the previous set is overwritten with the next one because the space is limited
  - second for loop:
    - it is computed how far the previous steps got and summed up how far it can get
    - a counter l is initialized to 0
    - l gets incremented for each node u which is in $S(k)$ (in the end: $l = |S(k)|$)
  - third loop:
    - deciding if node u belongs to $S(k)$
    - iterating over all nodes $v \in V$ one by one to reuse space
    - if node v is in $S(k-1)$, a counter m is incremented
      m counts the members of $S(k-1)$ that were found so far
    - if $u = v$ or there is an edge from u to v: $u \in S(k)$
      $\rightarrow$ variable *reply* gets set to true
    - if end is reached:
    - $u \notin S(k)$ if end is reached and reply is false:
      if $m < |S(k-1)|$ not all members of $S(k-1)$ have been ecountered: return *no*
    - else return *reply*
  - fourth loop:
    - checking whether $v \in S(k-1)$ with non-determinism (similar to REACHBILITY)

- nodes can't be marked (this would use linear space)
- runs in space $\log n$ with a Truring Machine M
- M has seperate strings holding each of the variables: $k, |S(k-1)|, l, u, m, v, p, w_p, w_{p-1}, input, output$
  all of those need only to be compared to each other and incremented by 1
  all bounded by n

$\overline{\text{REACHABILITY}} \in$ NL

- NL = nondeterministic logarithmic space
- $\overline{\text{REACHABILITY}} \in$ NL

Modify the non-deterministic Turing Machine from above so that it returns *yes* if the innermost subroutine ever reaches the target node n, otherwise, return *no*.

- run previous algorithm
- if target node is found, yes is returned
- else algorithm continues

Questions:
Did I understand this right?

NSPACE is closed under complement

$$\text{NSPACE}(f(n)) = \text{coNSPACE}(f(n))$$

for all proper complexity functions $f(n) \geq \log n$.

Proof idea:

- Language $L \in$NSPACE$(f(n))$ is decided by an non-deterministic Turing Machine M
- M is space bounded in $f(n)$
- to show: there is a $f(n)$ space bounded non-deterministic Turing Machine $\overline{M}$ which decides $\overline{L}$:
  - On input x $\overline{M}$ runs the recursive algorithm of the *Savitch-Theorem-proof* (choses internal node on the middle) on the configuration graph of M
  - the algorithm decides if two nodea are connected on the basis of x and the transition function of M
  - if $\overline{M}$ comes to an accepting configuration U, it halts and rejects
  - otherwise (if it is computed and no accepting configuration has been found) $\overline{M}$ accepts

References

[1] Jeff Erickson. *Nondeterministic Turing Machine*. `http://jeffe.cs.illinois.edu/teaching/algorithms/models/09-nondeterminism.pdf`. 2016.

[2] *Logarithmic Space and NL-Completeness*. `http://www.cs.toronto.edu/~ashe/logspace_handout.pdf`. 2020.

[3] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

Pina Kolling
piko0011

Summary: Lecture 5

Summary for the chapter *8.1.* [6]

Reduction

Examples of NP-problems:

- Travelling Salesman Problem
- SATISFIABLE
- REACHBILITY (in P)
- CIRCUIT VALUE (in P)

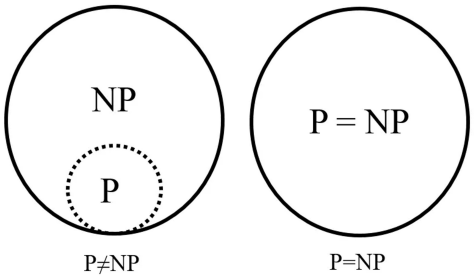


Figure 1: P and NP sets [3]

- reduction: a problem is at least as hard as another
- problem A is at least as hard as problem B if B reduces to A
- B reduces to A if there is a transformation R
  - R produces for every input x of B an equivalent input $R(x)$ of A
  - the answer of input x on B and input $R(x)$ on A have to be the same
- to solve B on input x, A can be solved instead with input $R(x)$

Reduction

Problem A is at least as hard as problem B if B reduces to A.

Transformation function:

- tranformation function R should not be too hard to compute
  $\rightarrow$ R should be limited

- efficient reduction R: $\log n$ space bounded

  Transformation function

  A language L_1 is reducible to L_2 if there is a function R computable by a deterministic Turing Machine in space $O(\log n)$ and $x \in L_1 \Leftrightarrow R(x) \in L_2$.

  R is called a reduction from L_1 to L_2.

- A Turing Machine M that computes a reduction R halts for all inputs x after a polynomial number of steps.
  - there are $O(n \cdot c^{\log n})$ possible configurtions for M on an input of length n
  - deterministic: no configuration can be repeated
  - computation of length at most $O(n^k)$

Pina Kolling
piko0011

Reduction HAMILTONIAN PATH to SATISFIABLE

Problem: HAMILTON PATH

The Hamiltonian Path problem asks whether there is a route in a directed graph G from a start node to an ending node, visiting each node exactly once. (Is there a path in G that visits each node one?) [1]

Problem: SAT

The SAT (satisfiability) problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. [7]

- HAMILTON PATH can be reduced to SAT
  $\rightarrow$ demonstrates HAMILTON PATH is not significantly harder that SAT
- construct a boolean expression $R(G)$ that is satisfiable only if G has a Hamilton path
  $\rightarrow$ write a logical formular that only becomes true when HP is true
- instance: Graph $G = (V, E)$ with n nodes $(1, 2, ..., n)$
- $R(G)$ has n^2 boolean variables $x_{i,j}$ then
- node j is the ith node in the HAMILTON PATH
- $R(G)$ is in conjuctive normal form (CNF: $(a \vee b) \wedge (\neg a \vee c)$)
- conjuncted clauses of $R(x)$:
  - each node j must appear in the path $x_{1,j} \vee x_{2,j} \vee ... \vee x_{n,j}$ – for every node j
  - no node j appears twice in the path: $\neg x_{i,j} \vee \neg x_{k,j}$ for all i, j, k with $i \neq k$
  - every position i on the path must be occupied – $x_{i,1} \vee x_{i,2} \vee ... \vee x_{i,n}$ for each i
  - no two nodes j and k occupy the same position in the path – $\neg x_{i,j} \vee \neg x_{i,k}$ for all i, j, k with $j \neq k$
  - nonadjacent nodes i and j cannot be adjacent in the path – $\neg x_{k,i} \vee \neg x_{k+1,j}$ for all $(i, j) \notin E$ and $k = 1, 2, ..., n - 1$

  [4]

Proof idea:

- to show:
  - for any graph G, $R(G)$ has a satisfying truth assignment only if and only if G has a Hamilton path
  - R can be computed in space $\log n$
- $R(G)$ contains $O(n^3)clauses$:
  - for each node j exists a unique position i $(x_{i,j})$
  - for each position i exists a unique node j $(x_{i,j})$
  - permutaion π of the nodes with $\pi(i) = j$ if $x_{i,j}$
  - $(\pi(1), \pi(2), ..., \pi(n))$ is a hamilton path
  - the truth assignment $T(x_{i,j}) = true$ if and only if $\pi(i) = j$

Boolean Circuit

A Boolean circuit is a mathematical tree model for logic formulas.
Boolean circuits are defined in terms of the logic gates they contain. For example, a circuit might contain binary AND and OR gates and unary NOT gates, or be entirely described by binary NAND gates.

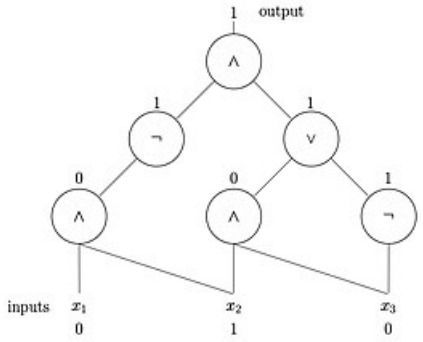


Figure 2: Boolean circuit example [2]

Reduction REACHABILITY to CIRCUIT VALUE

Problem: GRAPH REACHABILITY

Given a graph G and two nodes $n_1, n_2 \in V$, is there path from n_1 to n_2?
A graph $G = (V, E)$ is a finite set V of nodes and a set E of edges as node pairs.

REACHIBILITY can be nondeterministically solved in space $\log n$.

Problem: CIRCUIT VALUE

The CIRCUIT VALUE Problem is the problem of computing the output of a given Boolean circuit on a given input.
In terms of time complexity, it can be solved in linear time (topological sort).
The problem is closely related to the SAT (Boolean Satisfiability) problem which is complete for NP and its complement, which is complete for co-NP.

- REACHABILITY can be reduced to CIRCUIT VALUE
  $\rightarrow$ demonstrates REACHABILITY is not significantly harder that CIRCUIT VALUE
- construct a variable-free circuit $R(G)$ that has *true* as output only if G has a path from the start node to node n
- $R(x)$ uses no $\neg$ gates (monotone circuit)
- (both problema are in P)
- idea: use the Floyd-Warshall algorithm (dynamic programming)
- instance: Graph $G = (V, E)$ with n nodes $(1, 2, ..., n)$
- the gates:

  - $g_{i,j,k}$ with $1 \leq i, j \leq n$ and $0 \leq k \leq n$
    - there is a path from node i to node j without passing through a node bigger than k
    - $g_{i,j,0}$ is true if and only if $i = j$ or i and j are neighbours
  - $h_{i,j,k}$ with $1 \leq i, j, k \leq n$
    - there is a path from node i to node j passing through k but not any node bigger than k

- $h_{i,j,k}$ is an *and* gate with predecessors $g_{i,k,k-1}$ and $g_{k,j,k-1}$ where $k = 1, 2, ..., n$
- $g_{i,j,k}$ is an *or* gate with predecessors $g_{i,j,k-1}$ and $h_{i,j,k}$ where $k = 1, 2, ..., n$
- $g_{1,n,n}$ is the output gate

[5]

Proof idea:

- prove by induction on k
  $\rightarrow$ prove that the gates work as described
- claim is true for $k = 0$
- if claim is true for $k - 1$: true for k too

- $g_{1,n,n}$ is true if and only if there is a path from 1 to n and there is a path from 1 to n in G
- R can be computed in $\log n$ space
- circuit $R(G)$ is derived from the Floyd-Warshall algorithm

Reduction CIRCUIT SAT to SAT

Problem: CIRCUIT SAT

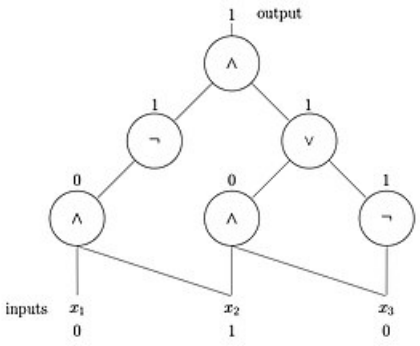
The circuit satisfiability problem (CIRCUIT SAT) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true.

- CIRCUIT SAT can be reduced to SAT
  $\rightarrow$ demonstrates CIRCUIT SAT is not significantly harder that SAT
- given a circuit C, construct a boolean expression $R(C)$ such that $R(C)$ is satisfiable if and only if C is satisfiable
- the variables of $R(C)$ are those of C plus g for each gate g of C
- clauses of $R(C)$:
  - g is a variable gate x:
    Add clauses $(\neg g \vee x)$ and $(g \vee \neg x) \equiv g \Leftrightarrow x$
  - g is a *true* gate:
    Add clause $(g) \rightarrow g$ must be true to make $R(C)$ true
  - g is a *false* gate:
    Add clause $(\neg g) \rightarrow g$ must be false to make $R(C)$ true
  - g is a $\neg$ gate with predecessor gate h:
    Add clauses $(\neg g \vee \neg h)$ and $(g \vee h) \equiv g \Leftrightarrow \neg h$

- g is a $\vee$ gate with predecessor gates h and h':
  Add clauses $(\neg h \vee g)$, $(\neg h' \vee g)$ and $(h \vee h' \vee \neg g)$, meaning: $g \Leftrightarrow (h \vee h')$
- g is a $\wedge$ gate with predecessor gates h and h':
  Add clauses $(\neg g \vee h)$, $(\neg g \vee h')$, and $(\neg h \vee \neg h' \vee g)$, meaning: $g \Leftrightarrow (h \wedge h')$
- g is the output gate:
  Add clause (g), meaning: g must be true to make $R(C)$ true

[5]

Example:



- g as variable gate: x_1 x_2 x_3
- g as $\wedge$ gate: $(x_1 \wedge x_2)$ $(x_2 \wedge x_3)$
- g as $\neg$ gate: $(\neg x_3)$ $\neg(x_1 \wedge x_2)$
- g as $\vee$ gate: $(x_2 \wedge x_3) \vee \neg x_3$
- g as $\wedge$ gate: $\neg(x_1 \wedge x_2) \wedge ((x_2 \wedge x_3) \vee \neg x_3)$

Figure 3: Boolean circuit example [2]

Generalization

- generalizations are a special form of reductions
- problem A is a generalization of problem B if every instance of B is also an instance of A $\rightarrow B$ can be reduced to A
- inputs of A are a subset of inputs of B and on those inputs A and B have the same answers

Generalization

Problem A is a generalization of problem B if every instance of B is also an instance of A. This implies B can be reduced to A.
The inputs of A are a subset of inputs of B and on those inputs A and B have the same answers.

Closedness under composition

Closedness under composition

If R is a reduction from a language L_1 to a lanugage L_2 and R' is a reduction from L_2 to a language L_3, then the composition $R \cdot R'$ is a reduction from L_1 to L_3.
$x \in L_1 \equiv R'(R(c)) \in L_3$

Pina Kolling
piko0011

Proof idea:

- $R \cdot R'$ can be computed in space $\log n$
- compose two Turing Machines M and M'
- simulate the machine for R by keeping on a counter which tape position i is
- when M' is about to do the ith write, jump back to the simulation of R and use this value

References

[1] J. Baumgardner, K. Acker, O. Adefuye, and et al. "Solving a Hamiltonian Path Problem with a bacterial computer". In: *J Biol Eng* 3.11 (2009). DOI: https://doi.org/10.1186/1754-1611-3-11.

[2] *Image source: Boolean Circuit.* https://upload.wikimedia.org/wikipedia/en/thumb/d/df/Three_input_Boolean_circuit.jpg/300px-Three_input_Boolean_circuit.jpg.

[3] *Image source: P-NP sets.* https://www.techno-science.net/actualite/np-conjecture-000-000-partie-denouee-N21607.html.

[4] Prof. Yuh-Dauh Lyuu. *Lecture slides on Reduction of hamiltonian path to sat.* https://www.csie.ntu.edu.tw/~lyuu/complexity/2011/20111018.pdf. 2011.

[5] Prof. Yuh-Dauh Lyuu. *Lecture slides on Reductions.* https://www.csie.ntu.edu.tw/~lyuu/complexity/2004/c_20041020.pdf. 2004.

[6] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[7] Prof. Dr. Thomas Schwentick. *Lecture slides in Grundbegriffe der theoretischen Informatik.* https://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Archiv/Bachelor_LA_GyGe_Inf_Modellv/_Module/INF-BfP-GTI/index.html.

Pina Kolling
piko0011

Summary: Lecture 6

Summary for the chapter *8.2.* [1, 7]

Completeness

Let C be a complexity class and let L be a language in C. L is called *C-complete* if any language $L' \in C$ can be reduced to L.

(Every language of a complexity class can be reduced to L.)

- reducitbility is transitive → problems are ordered by difficulty
- complete problems can capture the difficulty of a class
- problem is seen as cpmpletely understood if the problem is complete

Question:
Which problems can be reduced to a formal language?

SAT can be expressed as formal language. [6]
⇒ SAT can be reduced to a formal language. (?)
SAT is in NP. [7]

Because CIRCUIT SAT can be reduced to SAT: CIRCUIT SAT can be reduced to a formal language. (?) CIRCUIT SAT is NP-complete. [3]
Any formal language $L \in$ NP can be reduced to CIRCUIT SAT? OR the other way around?

Formal language

Formal languages are abstract languages, which define the syntax of the words that get accepted by that language. It is a set of words that get accepted by the language and has a set of symbols that is called alphabet, which contains all the possible characters of the words. Those characters are called nonterminal symbols. [2, 8]

Kleene star
The Kleene star Σ^* of an alphabet Σ is the set of all words that can be created through concatenation of the symbols of the alphabet Σ. The empty word ϵ is included.

Formal language
A formal language L over an alphabet Σ is a subset of the Kleene star of the alphabet: $L \subseteq \Sigma^*$

Where to set the line between lanuguage decisions and other problems? Can every problem be contrcuted as a formal language?

Is everything that is reducable to SAT reducable to a formal language because of the transitivity?

I assume it does not have an influence on the complexity of a problem if it can be expressed as a formal language? Are formal languages part of specific complexity classes?

Closed under reduction

The following complexity classes are all closed under reductions:

P NP coNP L NL PSPACE EXP

A class C is closed under reductions if whenever L is reducible to L' and $L' \in C'$, then $L\ in C'$.

If a complete problem in C belongs in a class $C' \subseteq C$, $C = C'$.

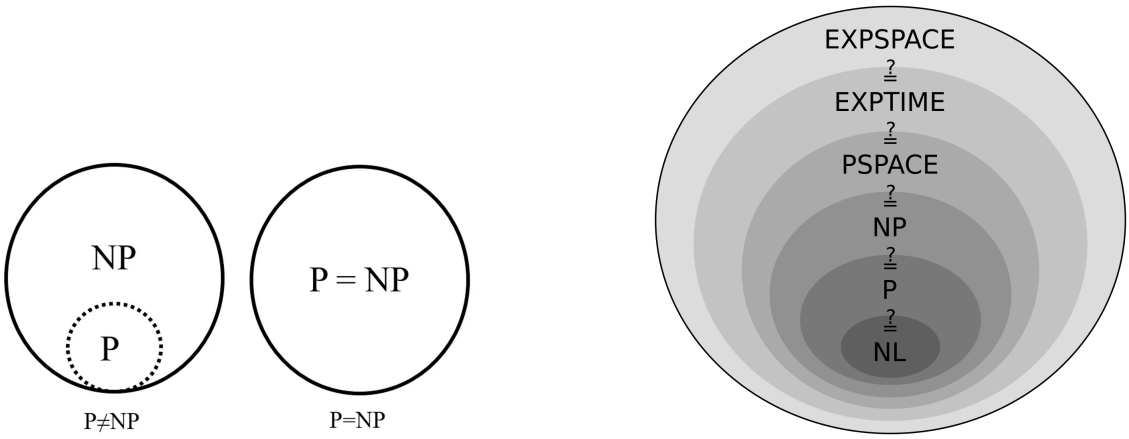


Figure 1: P and NP sets [5] and complexity classes [4]

- examples:
  - if an NP-complete language is in P, then NP = P
  - if a P-complete language is in L, then P = L
  - if a P-complete language is in NL, then P = NL
  - no EXP-complete language can be in P

Table method – time complexity

- table method to understand time complexity
- Turing Machine M decides language L on an input x
- computation on input x as $|x|^k \times |x|^k$ computation table

$\rightarrow$ $|x|^k$ is the time bound

- rows i are the time steps (from 0 to $|x|^k - 1$)
- columns j are the string positions
- T_{ij} represents the content of position j of the string of M at time i (after i steps)

Example:

- Turing Machine M deciding palindromes in time n^2

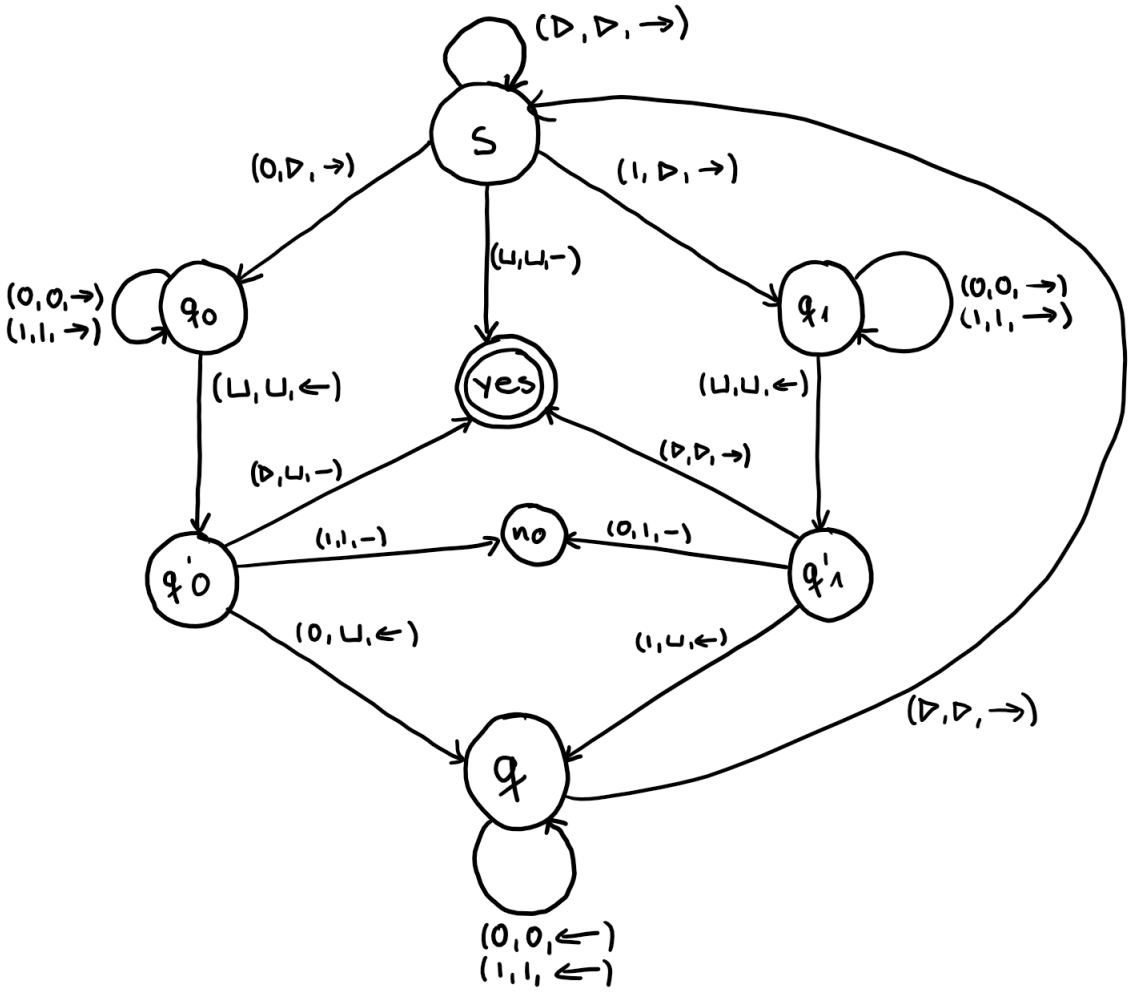


Figure 2: Binary palindrome Turing Machine M contructed from the definition in example 2.3 in the book [7]

- edges of the Turing machine:
  (input symbol σ, symbol to replace input symbol on tape with, direction in which the head moves)

$p \in K, \quad \sigma \in \Sigma$		$\delta(p, \sigma)$
s	0	$(q_0, \triangleright, \rightarrow)$
s	1	$(q_1, \triangleright, \rightarrow)$
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
s	$\sqcup$	(“yes”, $\sqcup$, $-$)
q_0	0	$(q_0, 0, \rightarrow)$
q_0	1	$(q_0, 1, \rightarrow)$
q_0	$\sqcup$	$(q_0', \sqcup, \leftarrow)$
q_1	0	$(q_1, 0, \rightarrow)$
q_1	1	$(q_1, 1, \rightarrow)$
q_1	$\sqcup$	$(q_1', \sqcup, \leftarrow)$

$p \in K, \quad \sigma \in \Sigma$		$\delta(p, \sigma)$
q_0'	0	$(q, \sqcup, \leftarrow)$
q_0'	1	(“no”, 1, $-$)
q_0'	$\triangleright$	(“yes”, $\sqcup$, $\rightarrow$)
q_1'	0	(“no”, 1, $-$)
q_1'	1	$(q, \sqcup, \leftarrow)$
q_1'	$\triangleright$	(“yes”, $\triangleright$, $\rightarrow$)
q	0	$(q, 0, \leftarrow)$
q	1	$(q, 1, \leftarrow)$
q	$\triangleright$	$(s, \triangleright, \rightarrow)$

Figure 3: Definition of binary palindrome Turing Machine M in example 2.3 in the book [7]

On input $x = 0110$ it results in the following computaion table:

▷	0_s	1	1	0	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1_{q_0}	1	0	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1_{q_0}	0	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1	0_{q_0}	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1	0	$⊔_{q_0}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1	$0_{q_0'}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1_q	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1_q	1	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	$▷_q$	1	1	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1_s	1	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	1_{q_1}	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	1	$⊔_{q_1}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	$1_{q_1'}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	$▷_q$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	$⊔_s$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	"yes"	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔

Figure 4: Computation table of binary palindrome Turing Machine M on input $x = 0110$ in the book [7]

- σ is the current input symbol
- q is the current state
- ⊔ is the clank symbol
- ▷ is the first symbol

Questions:
Is the Turing Macheine constructed correctly? Does the *no*-state need a double circle, because it halts or does only the accepting state get a double circle?

P-completeness of CircuitValue

Problem: CircuitValue

The CircuitValue Problem is the problem of computing the output of a given Boolean circuit on a given input.

In terms of time complexity, it can be solved in linear time (topological sort).

- P-complete

Proof idea:

- CircuitValue is in P (prerequisite for being P-complete)
- show: any language $L \in$ P can be reduced to CircuitValue
- L is decided by Turing Machine M in polynomial time (n^k)
- show: there is a reduction R and an input x to M
  R puts out a circuit C without variable gates, whose value is true only if M accepts x
- computation table T of M
- if $i = 0$: value of $T_{i,j}$ is the jth symbol of x or a ⊔

- if $j = 0$: value of $T_{i,j}$ is a $\triangleright$
- if $j = |x|^k - 1$: value of $T_{i,j}$ is a $\sqcup$

$\triangleright$	0_s	1	1	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_{q_0}	1	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1_{q_0}	0	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	0_{q_0}	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	0	$\sqcup_{q_0}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1	$0_{q_0'}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1	1_q	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_q	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright_q$	1	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	1_s	1	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	1_{q_1}	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	1	$\sqcup_{q_1}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	$1_{q_1'}$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright_q$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	$\sqcup_s$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$
$\triangleright$	$\triangleright$	$\triangleright$	"yes"	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$

Figure 5: Initial rows marked on previous palindrome example table

- value of $T_{i,j}$ is the content of position j of the string on time i

$\rightarrow$ depends on the same position and neighbor position in the previous steo $i-1$: $T_{i-1,j-1}, T_{i-1,j}, T_{i-1,j+1}$

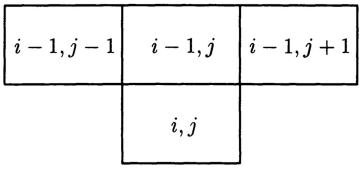


Figure 6: Value of $T_{i,j}$ depends on values of $T_{i-1,j-1}, T_{i-1,j}, T_{i-1,j+1}$

- $T_{i-1,j-1}, T_{i-1,j}, T_{i-1,j+1} \in \Sigma \Rightarrow$ cursor not at position $j-1, j, j+1$, this concludes to $T_{i,j} = T_{i-1,j}$

- set Γ contains all symbols that can appear on the table (symbols from Σ or symbol state combinations)
- encode each symbol $\sigma \in \Sigma$ as a vector $s = (s_1, ..., s_m)$
  - the entries of the vector $(s_1, ..., s_m)$ are either 0 or 1
  - the vector hast $m = \log |\Gamma|$ entries
- computation table can now be constructed as a table $S_{i,j,l}$ with binary entries (and $0 \leq i \leq n^k - 1$ and $0 \leq j \leq n^k - 1$ and $1 \leq l \leq m$)
- each binary entry S_{ijl} depends on previous entries $S_{i-1,j-1,l'}, S_{i-1,j,l'}, S_{i-1,j+1,l'}$, where l' ranges over 1 to m
- there are m Boolean functions $F_1, ..., F_m$ such that

$$S_{i,j,l} = F_l(S_{i-1,j-1,1}, ..., S_{i-1,j-1,m}, S_{i-1,j+1,1}, ..., S_{i-1,j+1,m})$$

- every boolean function can be renderes as a boolean circuit

$\rightarrow$ there is a boolean circuit C with $3m$ inputs and m outputs that computes $T_{i,j}$ with given $T_{i-1,j-1}, T_{i-1,j}, T_{i-1,j+1}$

- reduction R from L to CircuitValue
- for each input x, $R(x)$ consists of $(|x|^k - 1) \cdot (|x|^k - 2)$ copies of the circuit C
  $\rightarrow$ one for each entry in $T_{i,j}$
- call $C_{i,j}$ the (i,j)th copy of C
- $C_{i,j}$ depends on $C_{i-1,j-1}, C_{i-1,j}, C_{i-1,j+1}$ (if $i \geq 0$)
- first row and first and last column known
- other entries based on these entries
- output circuit of $R(x)$ is $C_{|x|^k-1,1}$
  (final step and string position 1, assuming that string position 1 contains *yes* or *no*)
- value of circuit $R(x)$ is only true if $x \in L$ because of equivalence of table structure
- $R(x)$ is in $\log n$ space

CircuitValue without NOT remains P-complete:

- AND, OR, NOT gate in circuit
- monotone circuit: does not have NOT gates
- move NOT downwards with DeMorgan's law (put negation into bracktes and exchange operator) until inputs are changed

MonotoneCircuitValue is P-complete.

CircuitSat is NP-complete

Problem: CircuitSat

The circuit satisfiability problem (CircuitSat) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true.

Input: a Boolean circuit C

Question: Is there a truth assignment which makes C output the value true?

- cook's theorem: Sat is NP-complete

Proof idea:

- CircuitSat $\in$ NP
- CircuitSat reduces to Sat
- to show: all languages in NP can be reduced to CircuitSat
- let $L \in$ NP
- describe a reduction $R(x)$ which constructs a circuit for each input string x such that $x \in L$ only if $R(x)$ is satisfiable
- Turing Machine M which decides L nondeterministically in time n^k exists becaus $L \in$ NP

- M accepst x only if $x \in L$
- reminder: Boolean circuits are build like binary trees with only two children to each node
- tranfer M into the structure of a binary tree with adding more states:

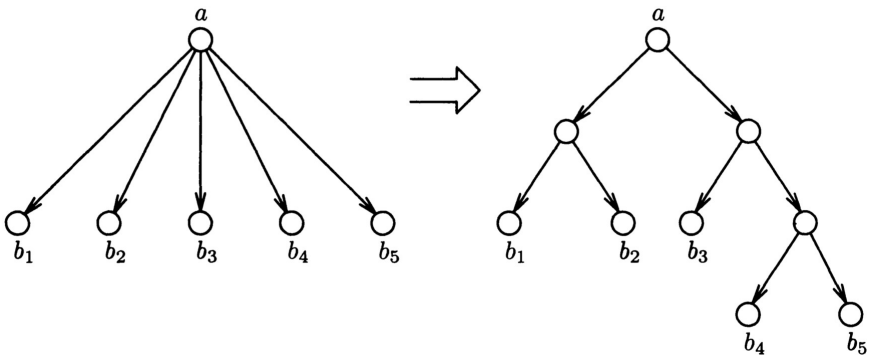


Figure 7: Transformation of M [7]

- computation table not really possible because computation in nondeterministic machine happens in parallel paths
  $\rightarrow$ fix a sequence of chioces c
- define computation table $T(M, x, c)$
- top row and first and last column rentries are known
- other entries $T_{i,j}$ depend on entries in $T_{i-1,j-1}, T_{i-1,j}, T_{i-1,j+1}$ and chice c of previous step
- circuit C has $3m+1$ instead of $3m$ entries because on entry is the nondeterministic choice
- value of circuit $R(x)$ is only true if $x \in L$ because of equivalence of table structure
- $R(x)$ is in $\log n$ space

Relation between complexity classes

$N \subseteq NL \subseteq NC \subseteq P \subseteq NP \subseteq PSPACE$

NP-complete problems

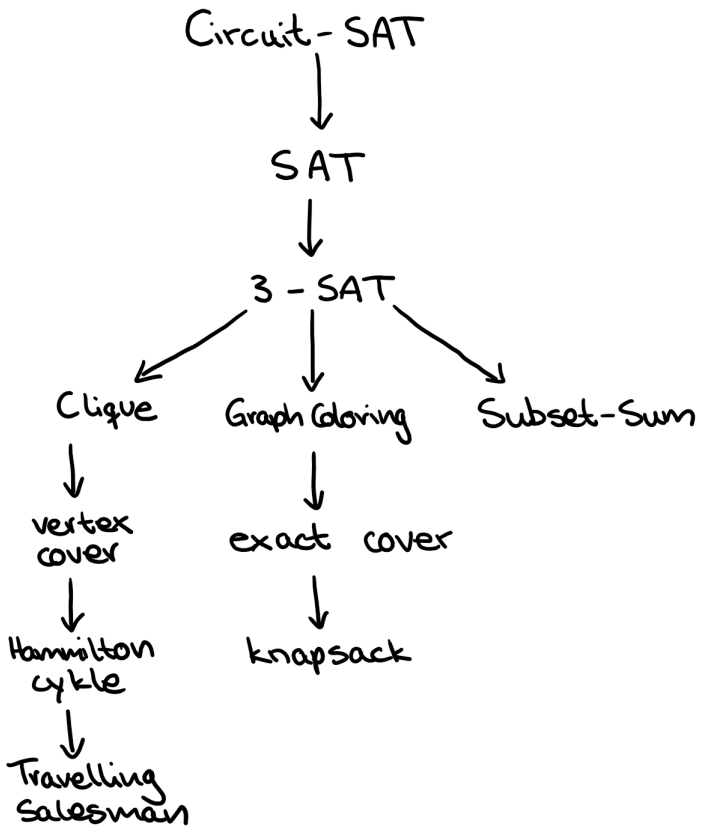


Figure 8: NP-complete problems in relation

- k-SAT for $k \geq 3$ is NP-complete

Circuit-SAT

The circuit satisfiability problem (Circuit-SAT) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true.

SAT

The SAT (satisfiability) problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. [9]

3-SAT

Like the SAT problem, 3-SAT is determining the satisfiability of a formula in CNF where each clause is limited to at most three literals.

Clique

The CLIQUE problem is the problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph.

VertexCover

In graph theory, a VERTEXCOVER (sometimes NODECOVER) of a graph is a set of vertices that includes at least one endpoint of every edge of the grap

HamiltonCycle

A HAMILTONCYCLE is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.

TravellingSalesman

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

GraphColoring

In graph theory, graph coloring is a special case of graph labeling. It is an assignment of colors to elements of a graph subject to certain constraints.

ExactCover

Given a collection S of subsets of set X, an exact cover is the subset S^* of S such that each element of X is contained is exactly one subset of S^*.

Knapsack

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

SubsetSum

The SUBSETSUM problem involves determining whether or not a subset from a list of integers can sum to a target value. For example, consider the list of $nums = [1, 2, 3, 4]$. If the target is 7 , there are two subsets that achieve this sum: $\{3, 4\}$ and $\{1, 2, 4\}$.

P-complete problems

- CIRCUITVALUE
- LINEARPROGRAMMING
- HORNSAT

CircuitValue

The CircuitValue Problem is the problem of computing the output of a given Boolean circuit on a given input.
In terms of time complexity, it can be solved in linear time (topological sort).
The problem is closely related to the Sat (Boolean Satisfiability) problem which is complete for NP and its complement, which is complete for co-NP.

LinearProgramming

LinearProgramming is a method to achieve the best outcome (such as maximum profit or lowest cost) in a mathematical model whose requirements are represented by linear relationships.

HornSAT

HornSAT is the problem of deciding whether a given set of propositional Horn clauses is satisfiable or not.
A Horn clause is a clause (a disjunction of literals) with at most one positive literal.

NL problems

- 2-Sat
- Reachability

2-Sat

Like the SAT and 3-SAT problem, 2-SAT is determining the satisfiability of a formula in CNF where each clause is limited to at most two literals.

Reachability

Given a graph G and two nodes $n_1, n_2 \in V$, is there path from n_1 to n_2?
A graph $G = (V, E)$ is a finite set V of nodes and a set E of edges as node pairs.

Reachability can be nondeterministically solved in space $\log n$.

L problems

- 1-Sat

1-Sat

Like the SAT and 3-SAT problem, 2-SAT is determining the satisfiability of a formula in CNF where each clause is limited to at most one literal.

References

[1] Martin Berglund. *Lecture notes in Computational Complexity.*

[2] *Chomsky's Normal Form (CNF).* Website. https://www.javatpoint.com/automata-chomskys-normal-form, opened on 26.09.2022.

[3] *Circuit satisfiability problem – Proof of NP-Completeness.* Website. https://en.wikipedia.org/wiki/Circuit_satisfiability_problem, opened on 25.11.2022.

[4] *Complexity classes diagram image source.* https://en.wikipedia.org/wiki/Complexity_class.

[5] *Image source: P-NP sets.* https://www.techno-science.net/actualite/np-conjecture-000-000-partie-denouee-N21607.html.

[6] klaus-joern Lange. "The Boolean Formula Value Problem as Formal Language". In: (Jan. 2012). DOI: 10.1007/978-3-642-31644-9_9.

[7] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[8] A.J. Kfoury Robert N. Moll Michael A. Arbib. *An Introduction to Formal Language Theory.* Springer-Verlag, 1988.

[9] Prof. Dr. Thomas Schwentick. *Lecture notes in Grundbegriffe der theoretischen Informatik.* https://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Archiv/Bachelor_LA_GyGe_Inf_Modellv/_Module/INF-BfP-GTI/index.html.

Pina Kolling
piko0011

Summary: Lecture 7

Summary for the chapters *9.1* and *9.2*. [5, 1]

NP-Completeness

NP

Class of lanugages decided by nonderteministic Turing machines in polynomial time.
Most problems are in NP.

NP-completeness:

- easiest problems among those we do not know how to solve efficiently
- if P≠NP can be proven: exact border of efficient solvability is found
- best bet for proving P=NP: show that some NP-complete problem is P
- Until then, the NP-complete problems are the least likely ones in NP to be efficiently solved
- Where is the line between P and NP?

Lanugage L

$L = \{x : (x, y) \in R \textit{ for some } y\}$
L gets an input x and finds a y with $((x, y) \in R$ and the relation $R \subseteq \Sigma^* \times \Sigma^*$.

For example: Is there a satisfying assignment (y) for a formular (x)?
Polynomially decidable:

- R is polynomially decidable if there is da deterministic Turing machine deciding the language L in polynomial time
- then the relation R (not the language L) is polynomially decidable

Polynomially balanced:

- R is polynomially balanced if $(x, y) \in R$ implies $|y| \leq |x|^k$ for some $k \geq 1$
  $\rightarrow$ length of the second component is bounded by a polynomial in the length of the first
- then the relation R (not the language L) is polynomially balanced

NP

The language $L \subseteq \Sigma^*$ is in NP only if there is a polynomially decidable and polynomially balanced relation R such that $L = \{x : (x, y) \in R \textit{ for some } y\}$.

Proof idea:

- L is decided by a nondeterministic Turing Machine M
- M guesses a y for an input x (y is of length at most $|x|^k$)
  $\rightarrow$ polynomially balanced
- M decides whether y encodes an accepting computation on x in linear time
  $\rightarrow$ polynomially decidable

Succinct certificate (for NP-complete problems)

- *yes* instance of x has a polynomial witness y (certificate)
- *no* instances don't have such a certificate
- Examples:
  - SAT: certificate is the truth assignment
  - HAMILTONPATH: certificate is the hamilton path of a graph

Typical problems in NP

- sometimes the optimum needs to be found
- sometimes any object that fits the specification is enough
- constraints can be added to optimization problems

3Sat is NP-complete

SAT

The SAT (satisfiability) problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. [7]

3SAT

Like the SAT problem, 3SAT is determining the satisfiability of a formula in CNF where each clause is limited to at most three literals.

- kSAT with $k \geq 1$ is a special case of SAT

Reduction from SAT to 3SAT: [6]

- the reduction replaces each clause with a set of clauses, each having exactly three literals
- rewrite the clauses of of the input
- example:

$$\begin{aligned}&(x_1) \wedge (x_1 \vee \bar{x_2}) \wedge (x_2 \vee x_3 \vee x_5) \wedge (x_1 \vee x_4 \vee \bar{x_6} \vee \bar{x_7}) \wedge (x_1 \wedge x_2 \wedge \bar{x_3} \vee x_5 \vee x_7) \\ \equiv\ &(x_1 \vee x_1 \vee x_1) \wedge (x_2 \vee x_3 \vee x_5)\end{aligned}$$

3Sat with more retrictions

3SAT remains NP-complete even for expressions in which each variable is restricted to appear at most three times and each literal at most twice.

Proof idea:

- if a variable appears more often than twice:
  introduce new variables and make sure (with the introduction of new clauses) that they have the same truthvalue as the original variable

Pina Kolling
piko0011

2Sat in P (graph construction)

2-Sat

Like the SAT and 3-SAT problem, 2-SAT is determining the satisfiability of a formula in CNF where each clause is limited to at most two literals.

- let ψ be an instance of 2SAT (clauses with two literals each)
- construct formular ψ as graph
- the nodes are the variables (node for a and $\neg a$)
- for clauses $(\neg a \vee b) \equiv a \to b$: edge from the node a to the node b
- paths in G are implications (implication is transitiv)
- ψ is unsatisfiable only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in G

Proof idea:

- the transitivity of the implication is proven
- ψ is unsatisfiable only if there is a variable x such that there are paths from x to $\neg x$ and from $\neg x$ to x in G

- there are paths from x to $\neg x$ and from $\neg x$ to x
- path from x to $\neg x$:
  - transitivity of implication

    leads to clause $x \to \neg x \equiv \neg x \vee \neg x \equiv \neg x$
- path from $\neg x$ to x
  - transitivity of implication

    leads to clause $\neg x \to x \equiv \neg\neg x \vee x \equiv x \vee x \equiv x$
- the two clauses are connected with an logic and (because formula is in CNF) which leads to $\neg x \wedge x$ which is unsatisfiable

- there are no paths from any x to $\neg x$ and back x: no edge is changing from true to false or the other way around
- whenever a node is assigned to a value, all the successors are assigned to the same value and there can be no edge from true to false or from false to true

Pina Kolling
piko0011

2Sat in NL

2SAT is in NL.

Proof idea:

- NL is closed under complement
- show: unsatisfiable expressions can be recognized in NL
- nondeterministically guessing a variable x and check for paths between x and $\neg x$ and back

MaxSat is NP-complete

Max2Sat

MAX2SAT is the problem of determining the maximum number of clauses, of a given Boolean formula in CNF with a maximum of 2 variables per clause, that can be made true by an assignment of truth values to the variables of the formula.
MAX2SAT is an optimization problem.

MAX2SAT is NP-complete.

Example:

$$(x)(y)(z)(w)$$
$$(\neg x \vee \neg y)(\neg y \vee \neg z)(\neg z \vee \neg x)$$
$$(x \vee \neg w)(y \vee \neg w)(z \vee \neg w)$$

x	y	z	w	$(\neg x \vee \neg y)$	$(\neg y \vee \neg z)$	$(\neg z \vee \neg x)$	$(x \vee \neg w)$	$(y \vee \neg w)$	$(z \vee \neg w)$	
0	0	0	0	1	1	1	1	1	1	6
0	0	0	1	1	1	1	0	0	0	4
0	0	1	0	1	1	1	1	1	1	7
0	0	1	1	1	1	1	0	0	1	6
0	1	0	0	1	1	1	1	1	1	7
0	1	0	1	1	1	1	0	1	0	6
0	1	1	0	1	0	1	1	1	1	7
0	1	1	1	1	0	1	0	1	1	7
1	0	0	0	1	1	1	1	1	1	7
1	0	0	1	1	1	1	1	0	0	6
1	0	1	0	1	1	0	1	1	1	7
1	0	1	1	1	1	0	1	0	1	7
1	1	0	0	0	1	1	1	1	1	7
1	1	0	1	0	1	1	1	1	0	7
1	1	1	0	0	0	0	1	1	1	6
1	1	1	1	0	0	0	1	1	1	7

- not all clauses can be satisfied

Proof idea:

- any truth assignment that satisfies $x \vee y \vee z$ satisfies 6 or 7 clauses
- the remaining truth assignments $(\neg x \wedge \neg y \wedge \neg z)$ satisfy 4 or 6 clauses
- reduction from 3SAT to MAXSAT because $x \vee y \vee z$ needs to be satisfied
- instance ψ of 3SAT
- instance $R(\psi)$ of MAXSAT

- for each clause $C_i = \alpha \vee \beta \vee \gamma$ of ψ: add the original clauses to $R(\psi)$ (x, y, z, w are replaced by $\alpha, \beta, \gamma, w_i$)
- clauses of $R(\psi)$ which are corresponding to a clause of ψ are called a group
- if ψ has m clauses, $R(\psi)$ has $10m$ clauses
- set constraint $K = 7$ satisfied clauses

- goal can be achieved in $R(\psi)$ only if ψ is satisfiable
- suppose $7m$ clauses can be satisfied in $R(\psi)$:
  - each group can satisfy at most seven clauses
  - there are m groups

  $\rightarrow$ seven clauses must be satisfied in each group

- any assignment that satisfies all clauses in ψ can be turned into one that satisfies $7m$ clauses in $R(\psi)$ (by defining the truth value of w_i)

NaeSat is NP-complete

NaeSat

Like SAT, NaeSat consists of a collection of Boolean variables and clauses. NaeSat requires that the values in each clause are not all equal to each other (in other words, at least one is true, and at least one is false). [4]

Max2Sat is NP-complete.
Proof idea:

- reduction from CircuitSat to Max2Sat
  $\rightarrow$ similar to reduction from CircuitSat to Sat
- add literal z to all clauses that have one or two literals
- if a truth assignment T follows the NaeSat-conditions, its complement $\bar{T}$ does it too
- in one of the truth assignments: z has the value false
- because of the definition of the gates: true and false included in following clauses

Reduction CIRCUIT SAT to SAT

Problem: CIRCUIT SAT

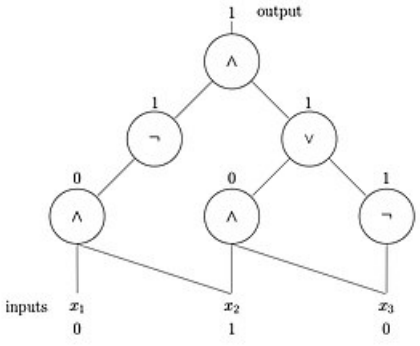
The circuit satisfiability problem (CIRCUIT SAT) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true.

- CIRCUIT SAT can be reduced to SAT
  $\rightarrow$ demonstrates CIRCUIT SAT is not significantly harder that SAT
- given a circuit C, construct a boolean expression $R(C)$ such that $R(C)$ is satisfiable if and only if C is satisfiable
- the variables of $R(C)$ are those of C plus g for each gate g of C

- clauses of $R(C)$:
  - g is a variable gate x:
    Add clauses $(\neg g \vee x)$ and $(g \vee \neg x) \equiv g \Leftrightarrow x$
  - g is a *true* gate:
    Add clause $(g) \rightarrow g$ must be true to make $R(C)$ true
  - g is a *false* gate:
    Add clause $(\neg g) \rightarrow g$ must be false to make $R(C)$ true
  - g is a $\neg$ gate with predecessor gate h:
    Add clauses $(\neg g \vee \neg h)$ and $(g \vee h) \equiv g \Leftrightarrow \neg h$
  - g is a $\vee$ gate with predecessor gates h and h':
    Add clauses $(\neg h \vee g)$, $(\neg h' \vee g)$ and $(h \vee h' \vee \neg g)$, meaning: $g \Leftrightarrow (h \vee h')$
  - g is a $\wedge$ gate with predecessor gates h and h':
    Add clauses $(\neg g \vee h)$, $(\neg g \vee h')$, and $(\neg h \vee \neg h' \vee g)$, meaning: $g \Leftrightarrow (h \wedge h')$
  - g is the output gate:
    Add clause (g), meaning: g must be true to make $R(C)$ true

[3]

Example:



- g as variable gate: x_1 x_2 x_3
- g as $\wedge$ gate: $(x_1 \wedge x_2)$ $(x_2 \wedge x_3)$
- g as $\neg$ gate: $(\neg x_3)$ $\neg(x_1 \wedge x_2)$
- g as $\vee$ gate: $(x_2 \wedge x_3) \vee \neg x_3$
- g as $\wedge$ gate: $\neg(x_1 \wedge x_2) \wedge ((x_2 \wedge x_3) \vee \neg x_3)$

Figure 1: Boolean circuit example [2]

References

[1] Martin Berglund. *Lecture notes in Computational Complexity.*

[2] *Image source: Boolean Circuit.* https://upload.wikimedia.org/wikipedia/en/thumb/d/df/Three_input_Boolean_circuit.jpg/300px-Three_input_Boolean_circuit.jpg.

[3] Prof. Yuh-Dauh Lyuu. *Lecture slides on Reductions.* https://www.csie.ntu.edu.tw/~lyuu/complexity/2004/c_20041020.pdf. 2004.

[4] Cristopher Moore and Stephan Mertens. *The nature of computation.* OUP Oxford, 2011.

[5] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

[6] Swagato Sanyal. *Reduction from SAT to 3SAT.* https://cse.iitkgp.ac.in/~palash/2018AlgoDesignAnalysis/SAT-3SAT.pdf, last opened: 29.11.22.

[7] Prof. Dr. Thomas Schwentick. *Lecture notes in Grundbegriffe der theoretischen Informatik.* https://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Archiv/Bachelor_LA_GyGe_Inf_Modellv/_Module/INF-BfP-GTI/index.html.

Pina Kolling
piko0011

Summary: Lecture 8

Summary for the chapters *9.3* and *9.4*. [3, 1]

Undirected graph

An undirected graph G is a pair of sets (V, E) where V is the finite set of nodes and E is a set of unordered pairs in V that are symmetric:

$$\forall\, i, j \in E, i \neq j :\ (i, j) \in E \Rightarrow (j, i) \in E$$

IndependentSet

IndependentSet

Input: An undirected Graph $G = (V, E)$ and a number k.
Question: Is there a set $I \subseteq V$ of $k = |I|$ nodes with no edges in between? (IndependentSet)

3SAT

Like the Sat problem, 3Sat is determining the satisfiability of a formula in CNF where each clause is limited to at most three literals.

IndependentSet is NP-complete.

Proof idea:

- triangle construction: any independent set can contain at most one node of the triangle

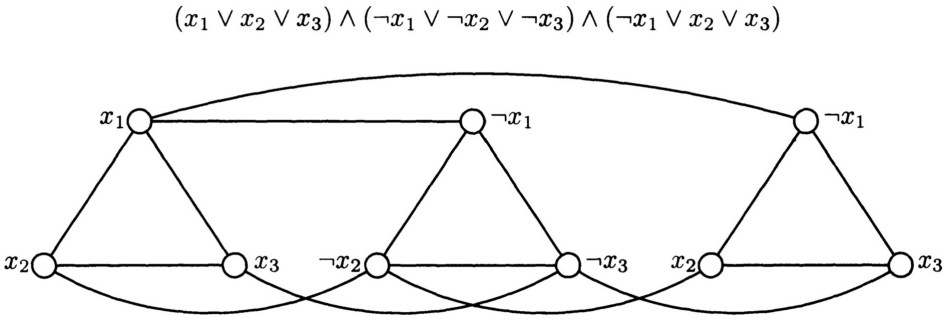


Figure 1: Graph with triangles [3]

- consider only graphs whose nodes can be partitioned in m disjoint triangles
  $\rightarrow$ independent set can contain at moast m nodes (one from each triangle)
- reduction from 3Sat to IndependentSet
- construct graph of formula ϕ:
  - each literal as a node
  - clauses as triangles
  - edges between nodes in different triangles if they correspond to the same literal (negated)
  - $K = m$ (m clauses)

- given: instance ϕ of 3SAT with m clauses $C_1, ..., C_m$
- each clause $C_i = (\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$ (with α as boolean variables or negation of those)
- reduction R constructs a graph: $R(\phi) = (G, K)$ where $K = m$ and $G = (V, E)$
- nodes $V = \{v_{ij} : \ i = 1, ..., m; j = 1, 2, 3\}$
  nodes for each of the m clauses (i) for each of the 3 literals (j)
- edges $E = \{[v_{ij}, v_{ik}] : \ i = 1, ..., m; j \neq k\} \cup \{[v_{ij}, v_{lk}] : \ i \neq l, \alpha ij = \neg\alpha_{lk}\}$
  edges between the nodes in one clause (triangle edges)
  edges between nodes with the same corresponding literal, but negated

- there is an indepentent set I of K nodes in G only if ϕ is satisfiable
- I must contain a node from each triangle
- negated literals are connected: I cannot contain a literal and its negation
- I is a truth assignment of ϕ:
  - true literals: nodes in I
  - one true literal per clause

HamiltonPath is NP-complete

HamiltonPath

A HAMILTONPATH is a path in a graph that visits each node exactly once.

HAMILTONPATH is NP-complete.

Proof idea:

- reduction from 3SAT to HAMILTONPATH
- given: formula ϕ in CNF with n variales $x_1, ..., x_n$ and m clauses $C_1, ..., C_m$ with each 3 variables
- construct a graph $R(\phi)$ that has a hamilton path only if ϕ is satisfiable:
- boolean variables:
  - choice between true and false
  - all occurences of x must have the same value (and $\neg x$ the opposite)
  - use *choice* gadget (like flip flop)
- XOR:

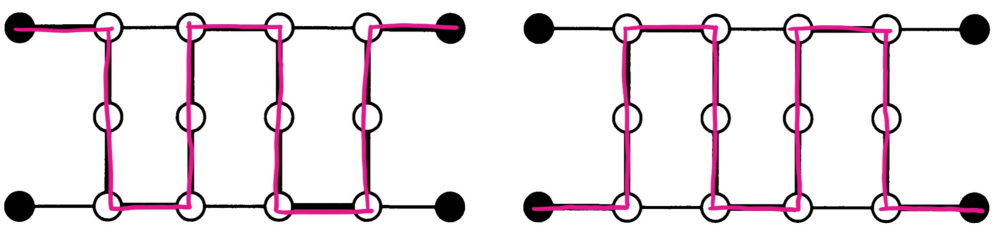


Figure 2: XOR subgraph from the book [3] with the relevant edges marked additionally

  - use *consistency* gadget
  - because of hamilton path: there are only two ways to traverse through this sub graph (as shown above)
  - leads to exclusive or (XOR)

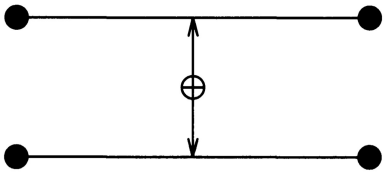


Figure 3: XOR connecting two independent edges (*consistency* gadget) [3]

- clauses:
  - triangles for clause construction
  - one side for each literal
  - if literal is false: hamilton path traveres triangle side
  - at least one literal need to be true: else all three edges of triangle will be traversed and this is not a hamilton path
- put everything together as graph G:
  - G has n copies of the *choice* gadget as a chain (one for each variable)

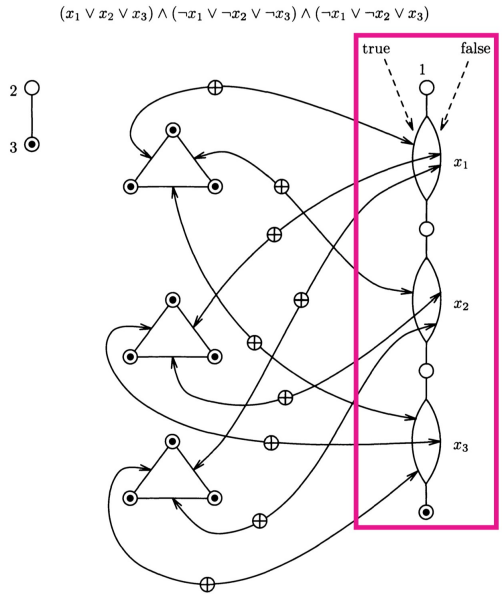


Figure 4: *Choice* gadgets marked in graph from the book [3]

- G has m triangles (one for each clause) with edges for each clause in the triangle

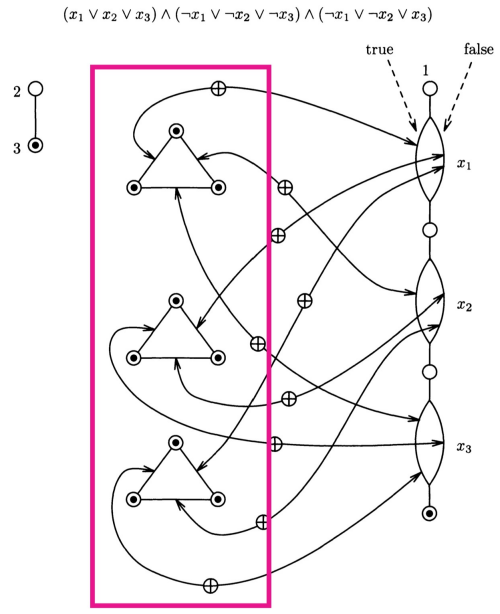


Figure 5: Clauses marked in graph from the book [3]

- finally all $3m$ nodes of the triangles, the last node of the chain of *choice* gadgets and a new node 3 are connected with all possible edges
- a single node 2 is connectes to the node 3

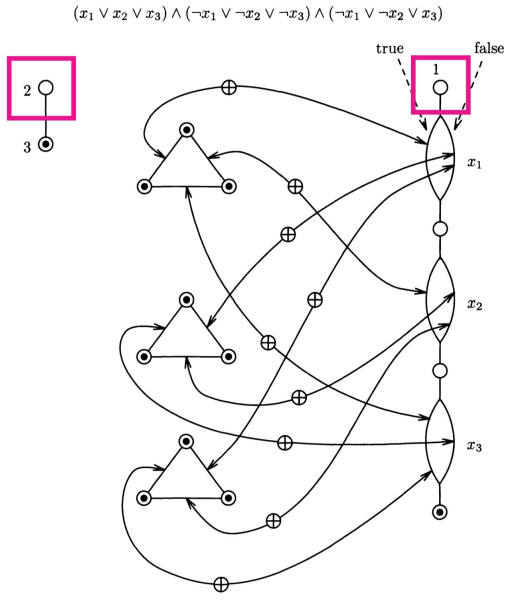


Figure 6: Nodes 1 and 2 marked (start and end node) in graph from the book [3]

- graph has a hamilton path only if ϕ has a satisfying truth assignment
- for hamilton path: start node is node 1 and end node is node 2
- from node 1 it must traverse one of the parallel edges of the *choice* gadget for the first variable
- exclusive ors must be traversed
- whole chain of *choice* gadgets will be traversed
  $\rightarrow$ in this way a truth assignment T is created

- then the triangles are traversed and it ends up in node 2 if there is a hamilton path and ϕ is satisfyable

TSP(D)

TSP(D)

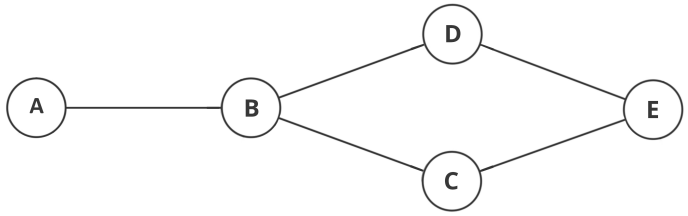
TSP(D) is a decision version of TSP.
Input: A $n \times n$ distance matrix and a bound $B \in \mathbb{N}$
Question: Is there a round tour of length $\leq B$ that visits all *cities*?

TSP(D) is NP-complete.

Proof idea:

- reduce from HamiltonPath to Tsp
- given: graph G with n nodes
- design: matrix d_{ij} and a budget B of nodes with $B = |V| + 1$ such that there is a tour of length B or less only if the G has a hamilton path
- d_{ij} usually contains the distance from city i to city j
- n cities: one node for each city in the graph
  $\rightarrow n$ nodes
- distance between two cities i and j is 1 if there is an edge $[i, j]$ and 2 otherwise

Example:



	A	B	C	D	E
A	–	1	2	2	2
B	1	–	1	1	2
C	2	1	–	2	1
D	2	1	2	–	1
E	2	2	1	1	–

Figure 7: Corresponding table to the graph

- undirected: distances are symmetric, leads to $d_{ij} = d_{ji}$
- set limit to $B = |V| + 1 = 6$
- $\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}$ is as small as possible
- π is a permutation

The following sum for the example can at most be 6:

$$\begin{aligned}
\text{A to B:}\quad & d_{\pi(0),\pi(1)} = 1 \\
\text{B to C:}\quad & d_{\pi(1),\pi(2)} = 1 \\
\text{C to E:}\quad & d_{\pi(2),\pi(3)} = 1 \\
\text{E to D:}\quad & d_{\pi(3),\pi(4)} = 1 \\
\text{D to A:}\quad & d_{\pi(4),\pi(0)} = 2 \\
& \sum = 6
\end{aligned}$$

- the answer to the TSP problem is *yes*
- the graph contains a hamilton path

Knapsack

Knapsack

Given is a set of items with a weight and a value. The task is to choose which items to include so that the total weight is less than the given limit of the knapsack and the total value is as large as possible.

Recalling my Dynamic Programming knowledge for the KNAPSACK problem
The formula for the dynamic programming of this problem is the following:

$$Opt(i,j) = \begin{cases} 0, & \text{for } 0 \leq j \leq size \\ Opt(i-1,j), & \text{for } j < w[i] \\ max\{Opt(i-1,j), v[i] + Opt(i-1, j-w[i])\}, & \text{else} \end{cases}$$

- if $0 \leq j \leq w$: there are no objects which could be put into the bag
- second case: object i does not fit into the bag and the optimal solution is found with the objects from index 1 to $i-1$
- else: object i is either part of the optimal solution or it consists out of the objects 1 to $i-1$

Table fill-out example:

- left: table of the values and weight of each item with index i is shown on the left
- right: table that gets filled in with a dynamic programming approach of the formula above, the maximum bag size is 7
- first column in the table represent the objects and the last row the weight
- entries in the table show the maximum value
- maximum value for the size 7 is 10.

i	$w[i]$	$v[i]$
1	1	1
2	3	4
3	2	3
4	4	6
5	6	8

5	0	1	3	4	6	7	9	10
4	0	1	3	4	6	7	9	10
3	0	1	3	4	5	7	8	8
2	0	1	1	4	5	5	5	5
1	0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7

ExactCoverBy3Sets

Given a set X, with $|X| = 3q$ (so, the size of X is a multiple of 3), and a collection C of 3-element subsets of X. Can we find a subset C' of C where every element of X occurs in exactly one member of C'? (So, C' is an exact cover of X). [2]

Example EXACTCOVERBY3SETS:

- suppose X was $\{1,2,3,4,5,6\}$

- if C was $\{\{1,2,3\},\{2,3,4\},\{1,2,5\},\{2,5,6\},\{1,5,6\}\}$ then $C' = \{\{2,3,4\},\{1,5,6\}\}$ is an exact cover because each element in X appears exactly once
- if instead, C was $\{\{1,2,3\},\{2,4,5\},\{2,5,6\}\}$, then any C' will not be an exact cover (all 3 subsets need to cover all elements in X exactly once)
- note that an exact cover C' will contain exactly q elements. [2]

KNAPSACK is NP-complete.

Proof idea:

- special case of KNAPSACK where $v_i = w_i$ and $K = W$
- K is the goal value (sum of values v_i needs to be greater than K)
- W is the maximum bag size (sum of weights w_i has to be smaller than W)
- given: set of n integers $w_i, ..., w_n$ and an integer K
- find: subset of the given integers that sums up exactly to K

- reduce from EXACTCOVERBY3SETS to KNAPSACK
- given: instance $\{S_1, S_2, ..., S_n\}$ of EXACTCOVERBY3SETS
- Are there disjoint sets among the given ones that cover the set $X = \{1, 2, ..., 3q\}$?
- given sets as bit vectors $\{0,1\}^{3q}$
- find a subset of these integers that add up to $K = 2^n - 1$ (find the all-ones vector)
- but: binary integer addition is diferent from set union
- example: $3 + 5 + 7 = 15 \equiv 0011 + 0101 + 0111 = 1111$ but the corresponding sets are $\{3,4\},\{2,4\},\{2,3,4\}$
- $\{3,4\},\{2,4\},\{2,3,4\}$ are not disjoint and not a exact cover of $\{1,2,3,4\}$
- vectors not in base 2 but in base $n+1$
  set S_i becomes then $w_i = \sum_{j \in S_i} (n+1)^{3m-j}$
- there is a set of integer that adds up to K only if there is an exact cover among $\{S_1, S_2, ..., S_n\}$

References

[1] Martin Berglund. *Lecture notes in Computational Complexity.*

[2] *Exact Cover by 3-Sets.* `https://npcomplete.owu.edu/2014/06/10/exact-cover-by-3-sets/`, last opened: 03.12.22.

[3] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

Pina Kolling
piko0011

Summary: Lecture 9

Summary for the chapter *10.3.* [6, 2]

Function problems

Function problem

Finding a specific solution to a problem if possible, else return *no*.

In other words: A function problem is defined by a binary relation $R(x, y)$. For every input x, an algorithm that solves the problem must output a y such that $R(x, y)$. If there is no such y, the answer must be *no*.

- focus so far: languages deciding decision problems
- give *yes* or *no* as answer
- now: focus on finding a solution:
  - find satisfying truth assignment for a boolean expression
  - find optimal tour for TSP
  - → function problems

- decision problems are helpful for negative results of function problems
- complexity of the decision problem helps to specify the complexity of the corresponding function problem

SAT and FSAT

SAT

The SAT (satisfiability) problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. [7]

FSAT

The FSAT (satisfiability) problem is a function problem.

Given a boolean expression ϕ.
If ϕ is satisfiable, return a satisfying truth assignment and otherwise return *no*.

- for input ϕ there might be no satisying truth assignment $\phi \notin$ SAT
  - return *no*
- for input ϕ there might be more than one satisying truth assignment
  - return any satisfying truth assignment
- if SAT can be solved in polynomial time, FSAT can be solved in polynomial time, too

Algorithm for FSAT:

- expression ϕ with variables $x_1, ..., x_n$

- ask if ϕ is satisfiable ($\phi \in$ SAT):
  - if *no*: stop and return *no*
  - if *yes*: come up with satisying truth assignment
    - consider two expressions: $\phi[x_1 = \text{true}]$ and $\phi[x_1 = \text{false}]$
    - check which one is satisfiable ($\phi[x_1 = \text{true}] \in$ SAT or $\phi[x_1 = \text{false}] \in$ SAT) (if both are, chose one)
    - substitute the value of x_1 in ϕ
    - continue with x_2
    - at most $2n$ calls to find the satisfying truth assignment

Algorithm for FSAT as pseudo code:

An Algorithm for FSAT Using SAT

```
1:  t := ϵ; {Truth assignment.}
2:  if φ ∈ SAT then
3:     for i = 1, 2, ..., n do
4:        if φ[x_i = true] ∈ SAT then
5:           t := t ∪ {x_i = true};
6:           φ := φ[x_i = true];
7:        else
8:           t := t ∪ {x_i = false};
9:           φ := φ[x_i = false];
10:       end if
11:    end for
12:    return t;
13: else
14:    return "no";
15: end if
```

Figure 1: FSAT algorithm as pseudo code [5]

Self-reducibilty

A function problem reduces to its corresponding decision problem.

- SAT is self-reducable

Questions:

- Book [6]:

  FSAT draws its difficulty precisely from the possibility that there may be no truth assignment satisfying the given expression.

→ Why is the difficulty coming from the possibility that there might be no truth assignment? It would in the first check of $\phi \in$ SAT return *no* and terminate? Is it because it takes longer for SAT to return *no* on every computation than it takes if a *yes* is found and all variables are substituted and checked with SAT?

Pina Kolling
piko0011

TSP and TSP(D)

TSP(D)

Given a list of cities and the distances between each pair of cities.

Is there a possible route of length k that visits each city exactly once and returns to the origin city?

TSP

Given a list of cities and the distances between each pair of cities.

What is the shortest possible route that visits each city exactly once and returns to the origin city?

- solve Tsp with an algorithm for Tsp(D)
- find optimum cost C of the tour with binary search (between 0 and 2^n) and using Tsp(D)
- change one intercity distance at a time to $C + 1$ to check if it is part of the optimal tour
  - call Tsp(D) C and the new distance entry
  - if *no*: restore old entry
    $\rightarrow$ distance is part of the tour
- after n^2 calls only entries of the distance matrix are $\leq C$ that are used for the optimum tour

FP and FNP

Lanugage L

$L = \{x : (x, y) \in R \textit{ for some } y\}$
L gets an input x and finds a y with $((x, y) \in R$ and the relation $R \subseteq \Sigma^* \times \Sigma^*$.

NP

The language $L \subseteq \Sigma^*$ is in NP only if there is a polynomially decidable and polynomially balanced relation R such that $L = \{x : (x, y) \in R \textit{ for some } y\}$.

Relationship between decision and function problems:

- L is a lanuage in NP
  - **Decision problem:**
    There is a string y with $R(x, y)$ only if $x \in L$.
  - **Function problem:**
    Given x, find a string y such that $R(x, y)$ if it exists, else return *no*.

FNP

Class of all function problems asspciated with languages in NP.

FP

FP is the subclass of FNP that contains function problems, that can be solved in polynomial time.

Examples:

- FSat is in FNP but expected to be in FP
- HornSat is in FP
- BipartiteGraph is in FP

Reductions between function problems

Reductions between function problems

A function problem A reduces to a function problem B if the following holds:

- R and S are string functions, x and z are strings
- If x is an instance of A then $R(x)$ is an instance of B.
- If z is a correct output of $R(x)$, then $S(z)$ is a correct output of x.

- R produces an instance $R(x)$ of the function problem B
- $S(z)$ is an constructed output for x from any correct output z of $R(x)$
- translate answers back to the original problem
- reduction is a pair (R, S):
  - R translates input x to input x'
  - S translates result z' to result z
- a function problem A is complete for a class FC if it is in FC and all problems in that class reduce to A
- FP and FNP are closed under reduction
- reductions of function problems compose

How to prove FP = FNP?

- FP = FNP only if P = NP

→ to prove the theorem above: show that Sat $\in P$ implies FSat $\in FP$

- this can be shown by constructing a satisfying truth assignment
- Sat$'$ is a formular φ plus an assignment that satisfies φ
- assignment as clauses that connects the single variables or their negation with $\wedge$
- algorithm for FSat with the help of Sat already described above

Cryptography

Cryptography argument [1, 6, 7]:

- P vs. NP problem is an unsolved problem
- currently clear: a correct solution to an NP problem can be checked for correctness in polynomial time
- experts wish NP problems to remain almost unsolvable because of cryptography
- complexity in cryptography is not only desirable, but necessary
- important to know that most encryption methods used today are based solely on the fact that the effort to *guess* the key is too high
  → problem of *guessing* is an NP problem
- proof of the solvability of NP problems means the end of all currently used encryption methods

→ cryptographic argument: if P=NP, no safe encoding exists

Total FNP

Total functions

Function problems in FNP that are quaranteed to never return *no* are called total problems.

In other words: A problem R in FNP is called total if for every input string x there is at least one string y such that $R(x, y)$.

- total problems sound like they are injective (for every input exists at least one output)

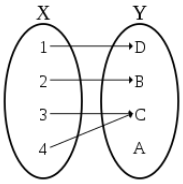


Figure 2: Visualization of injection [3]

FACTORING

Given an integer N.
Find its prime decomposition $N = p_1^{k_1}, p_2^{k_2}, ..., p_m^{k_m}$ together with the primality certificates of $p_1, ..., p_m$.

- FACTORING is a total function problem

Primality certificate:

- a primality certificate is a formal proof that a number is prime
- allows a number to be rapidly checked without having to run an expensive or unreliable primality test

Example [4]:

- the factors of 15 are 3 and 5
- the factoring problem is to find 3 and 5 when given 15

- prime factorization requires splitting an integer into factors that are prime numbers
- every integer has a unique prime factorization
- FACTORING is in FNP
- no known polynomial algorithm for FACTORING

TFNP

The sucblass of FNP that contains all total functions problems is denoted as TFNP.

Questions:

- Can the terms *total function*, *total problem* and *total function problem* be used interchangeably?

References

[1] Dr Datenschutz (Website). *P vs. NP: Ein Geschenk der Informatik an die Mathematik*. Last opened 11.11.2022. URL: https://www.dr-datenschutz.de/p-vs-np-ein-geschenk-der-informatik-an-die-mathematik/#:~:text=Hierbei%20werden%20von%20einem%20Computer,effizient%20l%C3%B6sen%20lassen%20oder%20nicht..

[2] Martin Berglund. *Lecture notes in Computational Complexity*.

[3] *Image source: Injective*. https://en.wikipedia.org/wiki/Injective_function.

[4] RSA Laboratories. *What is the factoring problem?* Website. Last opened 06.12.2022. URL: http://security.nknu.edu.tw/crypto/faq/html/2-3-3.html.

[5] Prof. Yuh-Dauh Lyuu. *Lecture slides on Complexity*. https://www.csie.ntu.edu.tw/~lyuu/complexity/2010/20101130.pdf. 2010.

[6] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[7] Prof. Dr. Thomas Schwentick. *Lecture notes in Grundbegriffe der theoretischen Informatik*. https://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Archiv/Bachelor_LA_GyGe_Inf_Modellv/_Module/INF-BfP-GTI/index.html.

Pina Kolling
piko0011

Summary: Lecture 10

Summary for the chapters *11.1* up to page 245 and *11.2* (page 258 optional). [5, 1]

Randomized algorithms

Algorithms based on randomization.
(The algorithm employs a degree of randomness as part of its logic or procedure.)

Bipartite matching

Bipartite Graph

A graph $G = (U, V, E)$ is called bipartite if the vertices can be divided into two disjoint and independent sets U and V. (There are no edges between two elements of U or two elements of V).

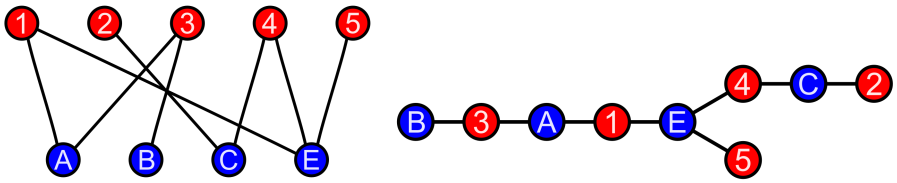


Figure 1: Examples of bipartife graphs with U and V marked in red and blue [2]

Problem: BipartiteMatching

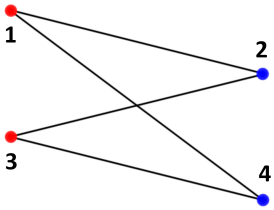
Given: Bipartite graph $G = (U, V, E)$.

Is there a perfect matching $M \subseteq E$ such that for any two edges (u, v) and (u', v') in M $u \neq u'$ and $v \neq v'$.

I other words: A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. The matching M is called perfect if for every node in V there is some edge in M. [3, 5, 4]

- construct bipartite graph with n nodes as $n \times n$ matrix A
- the element $A_{i,j}$ is a variable $x_{i,j}$ if $(i, j) \in E$
- the element $A_{i,j}$ is 0 if $(i, j) \notin E$

Example:



$$A = \begin{pmatrix} 0 & x_{1,2} & 0 & x_{1,4} \\ x_{2,1} & 0 & x_{2,3} & 0 \\ 0 & x_{3,2} & 0 & x_{3,4} \\ x_{4,1} & 0 & x_{4,3} & 0 \end{pmatrix}$$

Questions:
Is EVERY node cotained in the subset M when it is a perfect matching? Does a perfect matching then only exist with an even number of vertices and $|U| = |V|$?

Pina Kolling
piko0011

Determinant calculation

Leibniz-formula:

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \qquad |A| = a{\cdot}e{\cdot}i{+}b{\cdot}f{\cdot}g{+}c{\cdot}d{\cdot}h{-}g{\cdot}e{\cdot}c{-}h{\cdot}f{\cdot}a{-}i{\cdot}d{\cdot}b$$

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad |B| = 1{\cdot}5{\cdot}9{+}2{\cdot}6{\cdot}7{+}3{\cdot}4{\cdot}8{-}7{\cdot}5{\cdot}3{-}8{\cdot}6{\cdot}1{-}9{\cdot}4{\cdot}2 = 0$$

$$|A| = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} A_{i,\pi(i)}$$

- $\sigma(\pi)$ decides if + or −
- leads to $n!$ summands
- Example: $n = 3$
  $3! = 6$ summands
  6 permutations for π

  $+1$: (1 2 3) (2 3 1) (3 1 2)

  -1: (3 2 1) (2 1 3) (1 3 2)

  $\rightarrow |A| = A_{1,1} \cdot A_{2,2} \cdot A_{3,3} + A_{2,1} \cdot A_{3,2} \cdot A_{1,3} + ...$

Gaussian elimination:

- Gauß algorithm for solving LSE (linear systems of equations)
- allowed operations:
  - addition of rwos
  - subtraction of rows
  - multiply row with integer x
  - divide row by integer x
  - switch to rows
- wanted: upper triangular form (all entries below the diagonal 0)
- determinant is the product of the diagonal entries
- Example:

$$\begin{pmatrix} 1 & 3 & 2 & 5 \\ 1 & 7 & -2 & 4 \\ -1 & -3 & -2 & 2 \\ 0 & 1 & 6 & 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 4 & -4 & -1 \\ 0 & 0 & 0 & 7 \\ 0 & 1 & 6 & 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 4 & -4 & -1 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 7 & 2\frac{1}{4} \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 2 & 5 \\ 0 & 4 & -4 & -1 \\ 0 & 0 & 7 & 2\frac{1}{4} \\ 0 & 0 & 0 & 7 \end{pmatrix}$$

Figure 2: Examples gaussian elimination [5]

$|A| = 1 \cdot 4 \cdot 7 \cdot 7 = 196$

Symbolic Determinants

Symbolic matrix:

- matrix with variables instead of numerical entries
- Example:

$$\begin{pmatrix} x & w & z \\ z & x & w \\ y & z & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} x & w & z \\ 0 & \frac{x^2-zw}{x} & \frac{wx-z^2}{x} \\ 0 & \frac{zx-wy}{x} & -\frac{zy}{x} \end{pmatrix} \Rightarrow \begin{pmatrix} x & w & z \\ 0 & \frac{x^2-zw}{x} & \frac{wx-z^2}{x} \\ 0 & 0 & -\frac{yz(xz-xw)+(zx-wy)(wx-z^2)}{x(x^2-zw)} \end{pmatrix}$$

Figure 3: Examples gaussian elimination with symbolic matrix [5]

- subdeterminants have exponentially many terms
- wanted result: know if determinant is 0

Monte Carlo algorithm

Check if determinant of symbolic matrix is 0:

- use arbitrary integers for the variables
  $\rightarrow$ numerical matrix
- calculate determinant of numerical matrix:
  - if not 0:
    determinant of symbolic matrix is not 0
  - if 0:
    determinant of symbolic matrix is probably 0
    - numbers could be chosen such that the numrical determinant is 0 even though the symbolic one is not 0

Monte Carlo algorithm

Randomized algorithm for deciding if a graph G has a perfect matching with calculating the determinant of the corresponding matrix A to G.

- choose m random integers $i_1, ..., i_m$ between 0 and $2m$
- compute the determinant $|A|(i_1, ..., i_m)$ with the Gaussian elimnation
- if $|A|(i_1, ..., i_m) \neq 0$ reply *G has a perfect matching*
- if $|A|(i_1, ..., i_m) = 0$ reply *G has probably no perfect matching*

- if perfect matching found: decision is reliable and final
- if perfect matching not found: possibility of false negative

Monte Carlo algorithm

(Algorithm above) decides whether a symbolic matrix is **not** indentically to zero.

Reducing chance of false negatives:

- perform many independent experiments
- chose each time random integers (independently)
- repeat k times the evaluation of the determinant of the symbolic matrix
  - answer always zero:
    chance that G hs no perfect matching is higher $(1 - (\frac{1}{2})^k)$
  - answer different from zero once:
    perfect matching exists

Monte Carlo algorithm:

- Monte Carlo algorithm has no false positives
- probability of false negatives is bounded
- time needed always polynomieal

Randomized complexity classes

Monte Carlo Turing Machine

A polynomial Monte Carlo Turing Machine M that decides a language L is a nondeterministic Turing Machine with exactly two choices in each step and the following conditions:

- if $x \in L$ then at least half of the computations on x halt with *yes*
- if $x \notin L$ then all computations halt with *no*

- randomized algorithms can be modeled with an ordinary non-deterministic Turing Machine with different interpretation of the meaning of accepting the input
- no false positive answers+
- probability of false negatives is at most $\frac{1}{2}$

RP

Complexity class of all languages that are decided with polynomial Monte Carlo Turing Machines is denoted as RP.

- RP lies between P and NP ($P \subseteq RP \subseteq NP$)

ZPP

LasVegas algorithm

The LasVegas algorithm as a Monte Carlo algorithm and its complement (one has no false positives and one has no false negatives). It runs k independent experiments on both and the right answer will come up. (Either a positive answer from the one with no false positives or a negative answer from the one with no false negatives.)

- RP: no false positive answers but false negative answers possible
- coRP: no false negative answers but false positive answers possible
- RP ∩ coRP seems interesting:
  - problem in this class has two Monte Carlo algorithms:
    - one has no false positives
    - one has no false negatives
  - run enough experiments on both: right answer will come up
    (either a positive answer from the one with no false positives or a negative answer from the one with no false negatives)
  - correct answer will be known for sure
  - execute both algorithms independent k times: probability that the correct answer is not obtained is $\frac{1}{2^k}$

  → called LasVegas algorithm

ZPP

RP ∩ coRP

Complexity class of all languages that are decided with LasVegas algorithms is denoted as RP.

PP

Problem: MAJSAT

Given: Boolean expression φ

Is it true that the majority of the 2^n truth assignments to its variables satisfy it?

- MajSat is probably not in NP
- MajSat is probably not in RP
- MajSat is in PP

PP

The class PP contains all languages L, such that there is a nondeterminsic polynomial Turing Machine M such that for all inputs x:

$x \in L$ only if more than half of the computations end up accepting (*yes*).

M decides L by majority.

- PP is a syntactic class (not a semantic class)
- NP ⊆ PP

Syntactic class:

- has a complete language L
- P and NP

Semantic class:

- no complete problems
- there is no easy way to tell whether a machine always halts with a certified output

BPP

Answers (*yes* and *no*) are correct with probability $\frac{3}{4}$.

- unclear whether BPP $\subseteq$ NP
- BPP is closed under complement: coBPP = BPP
- BPP is a semantic class

Relations between randomized complexity classes

Figure 4: Relations between classes from the lecture slides [1]

- unclear whether BPP $\subseteq$ NP

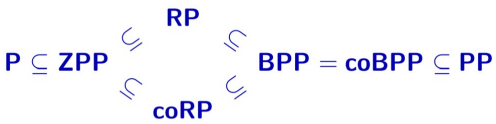


Figure 5: Relations between classes from the lecture slides [1]

- RP $\subseteq$ BPP:
  - every language in RP has a BPP algorithm:
    - run algorithm twice to assure that the probability of false negatives is less than $\frac{1}{4}$
    - probability of false positives is 0 ($0 \leq \frac{1}{4}$)
- BPP $\subseteq$ PP:
  - majority of PP: $> \frac{1}{2}$
  - majority of BPP: $> \frac{3}{4}$

$\rightarrow$ RP $\subseteq$ BPP $\subseteq$ PP

References

[1] Martin Berglund. *Lecture notes in Computational Complexity.*

[2] *Bipartite graph image source.* `https://en.wikipedia.org/wiki/Bipartite_graph`.

[3] GeeksforGeeks. *Maximum Bipartite Matching.* `https://www.geeksforgeeks.org/maximum-bipartite-matching/`, last opened: 09.12.22.

[4] Swastik Kopparty. *Bipartite Graphs and Matchings.* `https://sites.math.rutgers.edu/~sk1233/courses/graphtheory-F11/matching.pdf`, last opened 09.12.22. 2011.

[5] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

Pina Kolling
piko0011

Summary: Lecture 11

Summary for the chapter *13.1* up to page 307. [1, 2]

Title

Content

Approximation

- to solve function problems
- optimization problems: find less than perfect answers
- ϵ for how many percent are *perfect*
- PTAS: how much do you care about the result? algorithm will adapt time to this

$$\frac{c(M(x)) - \mathrm{OPT}(x)}{c(M(x))} \leq \epsilon$$

Node Cover: Decision problem

Given: Graph $G = (V, E)$ and $k \in \mathbb{N}$

Is there a subset $V' \subseteq V$ with $|V'| \leq k$ such that every edge $e \in E$ contains a node from the subset V'?

Node Cover: Optimization problem

Given: Graph $G = (V, E)$ and $k \in \mathbb{N}$

Something else :D

Maximum Cut: Decision problem

Given: Graph $G = (V, E)$ and $k \in \mathbb{N}$

Is there a subset $V' \subseteq V$ such that there are at least k edges between $V \setminus V'$ and V'?

Maximum Cut: Optimization problem

Given: Graph $G = (V, E)$

Find the subset $V' \subseteq V$ such that there is the maximum number of edges between $V \setminus V'$ and V'?

TODO
Questions:

References

[1] Martin Berglund. *Lecture notes in Computational Complexity.*

[2] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.