Formale Methoden der Informatik

Block 1: Foundations of Complexity Theory

2. Complexity of Problems and Algorithms

Reinhard Pichler

Institut für Informationssysteme
Arbeitsbereich DBAI
Technische Universität Wien

9 October, 2013

Outline

Complexity Theory

Complexity theory focuses on analyzing the computational complexity of problems. In other words, how much time (number of operations) or space (storage) do we need to solve a given problem?

Recall:

- A problem: an infinite set of possible instances with a question
- A decision problem: the question has a yes/no answer

Example

REACHABILITY:
INSTANCE: A graph (V, E) and nodes $u, v \in V$.
QUESTION: Is there a path in the graph from u to v?

- Other kinds of problems:
  - function problems
  - optimization problems
  - enumeration problems
  - counting problems

Algorithm for **REACHABILITY**

$S := \{u\}$;
repeat
    $S' := S$;
    for all $i \in S'$ **do** {
        for all $j \in V$ **do** {
            if $(i,j) \in E$ **then** $S := S \cup \{j\}$;
        }
    }
until $S = S'$;
if $v \in S$ **then** return *true*
else return *false*;

Questions

How efficient is the algorithm?

How is it affected by

- Programming language?
- Computer architecture?
- Representation of the input?

Efficiency of the REACHABILITY algorithm

- A naive implementation requires cubic time.
- Given certain assumptions, the problem is solvable in linear time w.r.t. the number of edges.

Time and Space

We measure the complexity of problems and efficiency of algorithms in terms of time and space:

Time

Given a program Π and an input I for Π, the running time of Π on I is the number of atomic operations performed during the run of Π on input I.

- The notion of "atomic operation" depends on the programming language and the computer architecture.

Space

Given a program Π and an input I, the space required by Π on I is the number of bits of storage that is used during the run of Π on I.

Time and Space (2)

Question

Does a specific programming language or architecture affect the time/space performance of an algorithm implementation?

- Affects only marginally: a specific choice of a language/architecture gives only a minor improvement in time/space performance.
- Intuitively, "minor improvement" means that:

  An efficient program can be translated into any other language (including SIMPLE) or architecture without loosing efficiency.

- Thus to analyze existence/nonexistence of efficient algorithms it is fine to choose SIMPLE as our programming language.
- More formally, "minor" in this context means polynomial.

Church-Turing Thesis Revisited

Church-Turing Thesis

Any algorithm can be programmed in SIMPLE.

Alternative formulation: Any "reasonable" mathematical model of computer algorithms ends up with a model of computation that is equivalent to the SIMPLE programming language.

Strengthening of the Church-Turing Thesis

Any "reasonable" mathematical model of computer algorithms and their time/space performance ends up with a model of computation and associated time/space cost that is equivalent to the SIMPLE programming language within a polynomial.

Complexity of an Algorithm

Our notion of complexity

We are interested in the asymptotic, worst-case complexity of an algorithm, i.e., a function $f(\,)$, s.t. for every problem instance of size n, the answer can be computed in time/space $\mathrm{O}(f(n))$.

Consequences

We are *not interested* in, e.g.

- average case complexity
- best case complexity
- complexity of "instances that typically occur in practice"
- complexity of a finite number of instances
- ...

Discussion

- Our notion of complexity often leads to useful, practically applicable results like, e.g. "This algorithm is only suitable for very small problem sizes, and improved hardware will not help a lot."
- However, there are examples in practice, where a different notion of complexity would be more appropriate.

Example

Linear Programming = optimization problem of a linear function of variables whose domain is restricted by means of linear inequalities.

- There exist algorithms with polynomial worst-case time complexity.
- Experience shows that the Simplex method (with exponential worst-case complexity) *works well in practice.*

Rates of Growth

Let f, g: $\mathbf{N} \mapsto \mathbf{N}$.

- $f(n) = \mathrm{O}(g(n))$ (f grows as g or slower), if there are positive integers c and n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$
- $f(n) = \Omega(g(n))$, if $g(n) = \mathrm{O}(f(n))$
- $f(n) = \Theta(g(n))$, if $g(n) = \mathrm{O}(f(n))$ and $f(n) = \mathrm{O}(g(n))$.

Example

- If $p(n)$ is a polynomial of degree d, then $p(n) = \Theta(n^d)$.
- If $c > 1$ is an integer and $p(n)$ a polynomial, then $p(n) = \mathrm{O}(c^n)$ but $p(n) \neq \Omega(c^n)$, i.e.,

  any polynomial grows strictly slower than any exponential.
- If $k > 1$ is an integer, then $\log^k n = \mathrm{O}(n)$

Complexity of a Problem

Complexity of a problem =
worst-case complexity of the best possible algorithm for this problem

Complexity Theory

- analyzes the complexity of problems
- classifies problems according to their complexity
- analyzes properties of complexity classes
- analyzes relations between complexity classes
- ...

Negative results (like "you will never find an efficient algorithm for this problem") require formal methods.

The Class P (Problems Solvable in Polynomial Time)

Informally, the class P is the collection of all problems that can be solved in polynomial time in the size of the instance.

Definition

The class P consists of all decision problems $\mathcal{P}$ satisfying the following:

1. there is a program Π that decides $\mathcal{P}$, and Π is such that
2. for all instances I of $\mathcal{P}$, the run time of Π on I is polynomial in $|I|$, i.e. the run time is $\mathrm{O}(|I|^k)$, where k is a constant.

Many problems are in P:

- Recall **REACHABILITY** is solvable in cubic time, and hence is in P.
- Checking if a string matches a context-free grammar is in P.

Some Problems in P

We recall first Boolean circuits:

Boolean circuits

A k-ary Boolean circuit consists of:

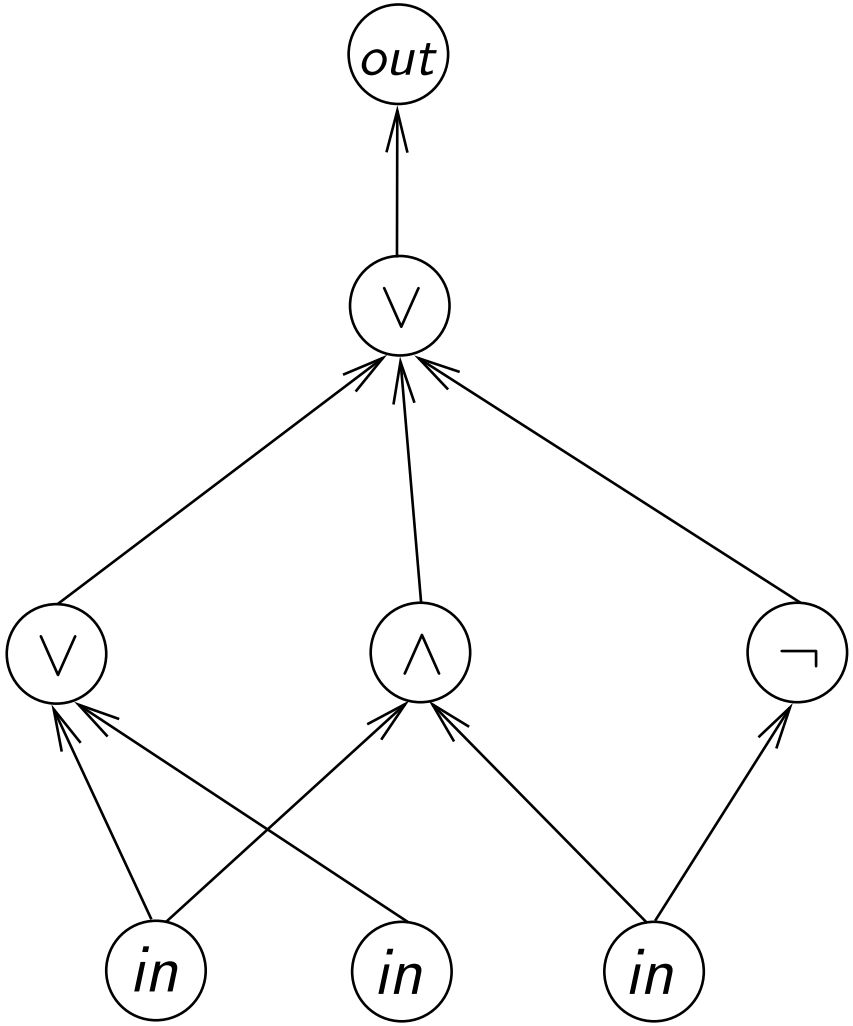
- k input gates $in_1, \ldots, in_k$,
- 1 output gate, and
- *AND* gates, *OR* gates and *NOT* gates such that
  - an *AND* gate outputs *true* iff all its inputs have value *true*,
  - an *OR* gate outputs *true* iff some of its inputs has value *true*, and
  - a *NOT* gate outputs *true* iff its single input has value *false*.

CIRCUIT-EVAL

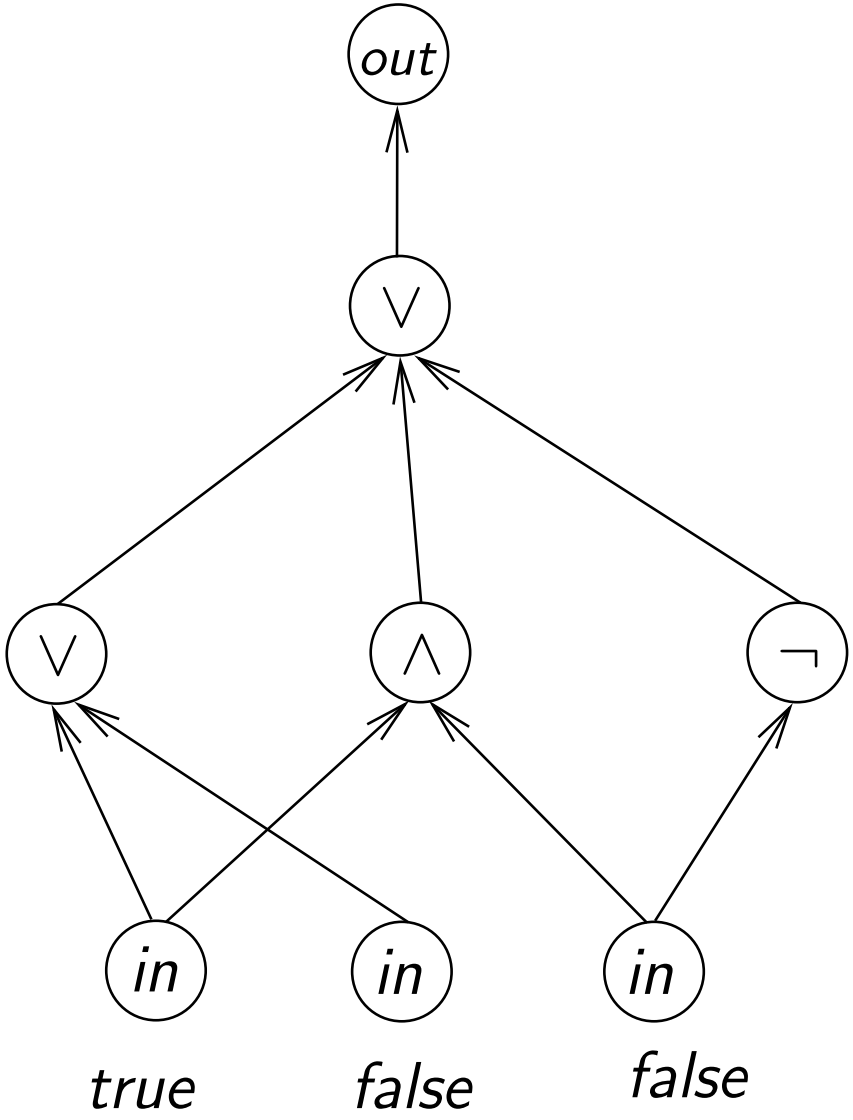
INSTANCE: A Boolean circuit C and a function μ that assigns to each input gate of C the value *true* or *false*.

QUESTION: Does C output *true* given the assignment μ?

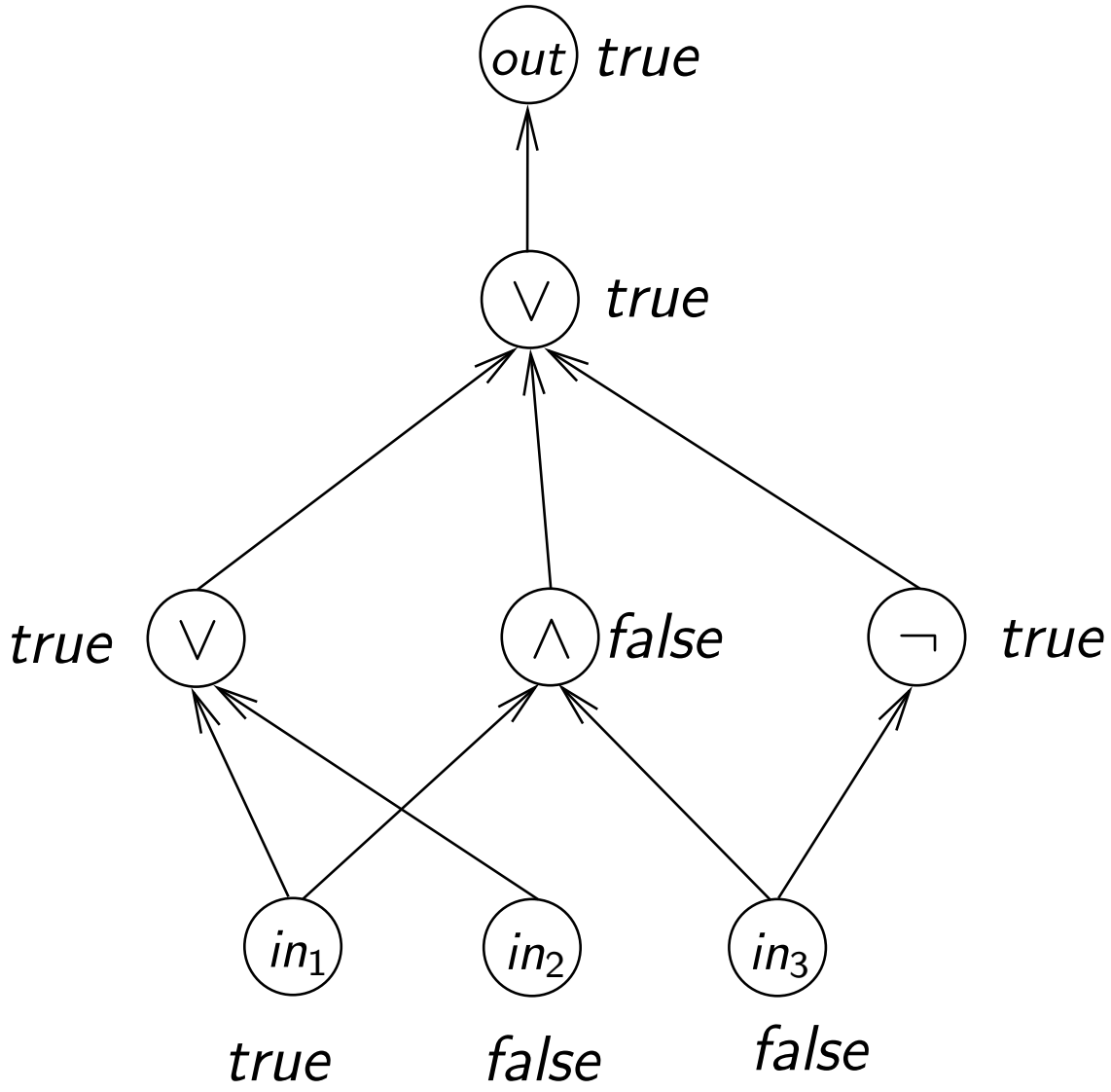
Example Circuit



Example Circuit with an Assignment



Example Circuit Evaluated



CIRCUIT-EVAL is in P

Algorithm **evaluateCircuit**

Input: k-ary Boolean circuit C, and assignment μ
Output: *true* iff C outputs *true* under μ

1. Let $A := \{(in_1, \mu(in_1)), \ldots, (in_k, \mu(in_k))\}$; /* make a copy of μ */
2. Let G be the set of all gates in C;

   The nodes $g \in G$ that occur in A will be called *value-assigned*
3. **while** exists $g \in G$ such that g is not value-assigned **do**
   (i) select one $g \in G$ such that g is not value-assigned but all its input gates are value-assigned (such a gate must exist because C forms a directed acyclic graph where input gates are the only source nodes)
   (ii) add to A the tuple (g, v), where v is the Boolean value determined according to the type of g and the values assigned by A to the input gates of g.
4. **if** $(out, true) \in A$ **then return** *true* **else return** *false*

CIRCUIT-EVAL is in P (2)

We verify that the algorithm **evaluateCircuit** runs in polynomial time in the size of the input:

1. At all times the relation A is of linear size in the size of C.
2. Since each iteration of the steps (i-ii) results in a fresh value-assigned gate, there may only be $|G|$ iterations of the **while** loop.
3. Given a gate $g \in G$, checking whether g is not value-assigned and all its input gates are value-assigned is feasible in quadratic time in the size of G (we may need $|G|$ lookups to A).
4. Due to the observation (3), the step (i) is feasible in cubic time (simply traverse G and make the check described above).
5. Step (ii) is linear in the size of C.
6. Thus overall the algorithm runs in $\mathrm{O}(n^4)$.

Actually, one can devise a linear algorithm, i.e. $\mathrm{O}(n)$ instead of $\mathrm{O}(n^4)$.

Propositional Logic

There is a close relationship between Boolean circuits and formulas of propositional logic (also referred to as Boolean logic).

Symbols

The syntax of propositional logic (= Boolean logic) (i.e. the set of well-formed propositional formulae) is based on the following symbols:

- Boolean variables (or atoms): $X = \{x_1, x_2, \ldots\}$.
- Boolean connectives: $\vee$, $\wedge$, and $\neg$.

Definition

The set of propositional formulae is the smallest set such that

- all Boolean variables are propositional formulae
- if φ_1 and φ_2 are propositional formulae, so are $\neg\varphi_1$, $(\varphi_1 \wedge \varphi_2)$, and $(\varphi_1 \vee \varphi_2)$.

An expression of the form x_i or $\neg x_i$ is called a literal.

Semantics of Propositional Logic

Motivation

How to interpret Boolean formulas?
Observation: Boolean formulas are propositions that are either true or false. They speak about a world where certain atomic propositions (Boolean variables) are either true or false.

Definition

A truth assignment T is a mapping $T : X \to \{true, false\}$.
T is appropriate to a formula φ if $Var(\varphi) \subseteq X$.

$T(\cdot)$ can be inductively extended from variables to arbitrary formulas:

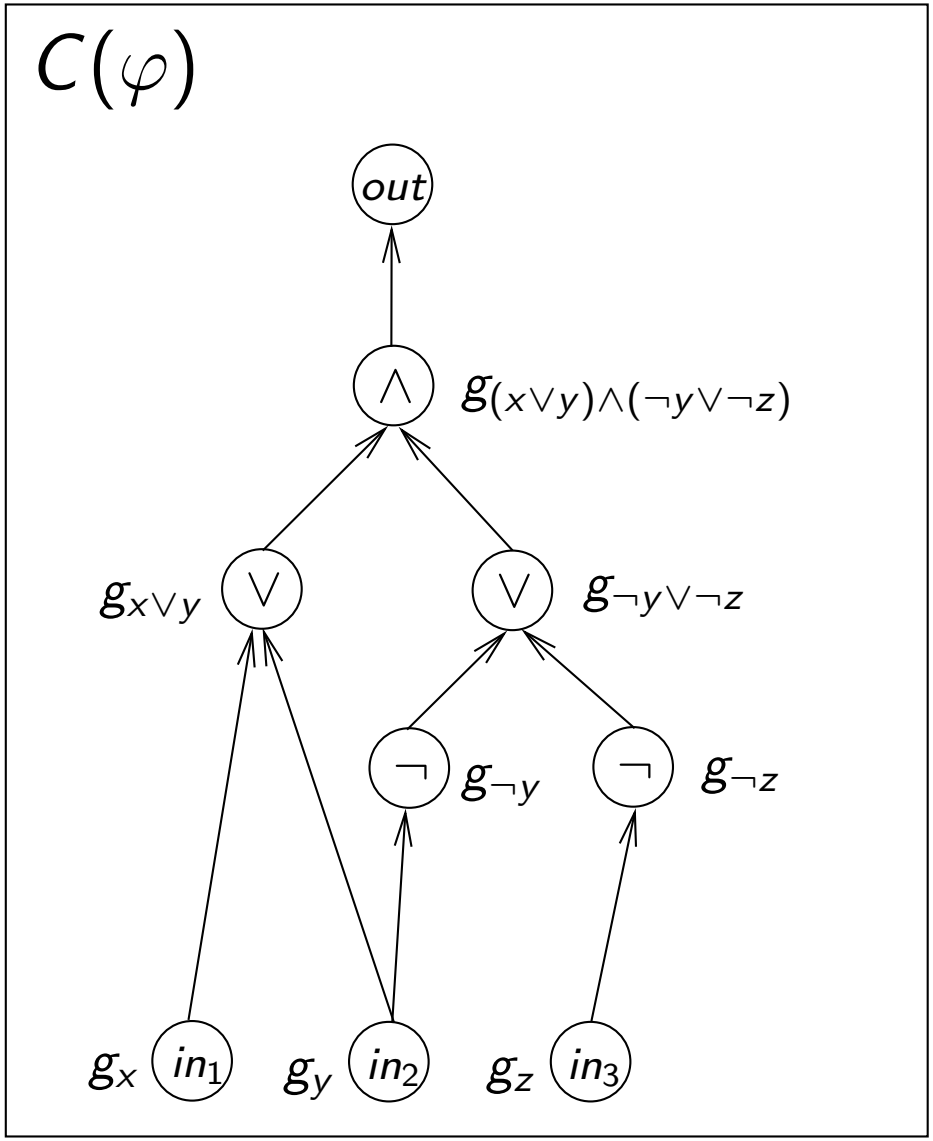
- If $\varphi = \neg\varphi_1$, then $T(\varphi) = true$ iff $T(\varphi_1) = false$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $T(\varphi) = true$ iff $T(\varphi_1) = true = T(\varphi_2)$.
- If $\varphi = \varphi_1 \vee \varphi_2$, then $T(\varphi) = true$ iff $T(\varphi_1) = true$ or $T(\varphi_2) = true$.

Computing $T(\varphi)$ (i.e., given φ and truth assignment T appropriate to it, check if $T(\varphi) = true$ holds), can be done in polynomial time.

Correspondence between Boolean Formulas and Circuits

Example:

$\varphi = (x \vee y) \wedge (\neg y \vee \neg z)$



Discussion

- A problem is considered efficiently solvable if it is in the class P. Recall that this means that there is an algorithm solving the problem such that the rate of growth of the solution time is polynomial w.r.t. the size n of the input, i.e., $\mathrm{O}(n^d)$.
- If there is no polynomial time algorithm available for the problem, then the problem is considered intractable. There is a distinction between
  - intractability (i.e., no efficient algorithm is known and the existence of an efficient algorithm is unlikely – but not formally excluded) $\longrightarrow$ the class NP defined next
  - provable intractability (i.e., the existence of an efficient algorithm is formally excluded) $\longrightarrow$ the class EXPTIME
- Recall that we consider the asymptotic, worst-case performance.

Possible criticism

- Not all polynomial-time algorithms are efficient in practice.
  There are efficient computations that are not polynomial.
  For instance, consider n^{80} vs $2^{\frac{n}{100}}$.
- Average case analysis may be more informative than worst-case.

Conclusion. "*Adopting polynomial time worst-case performance as our criterion of efficiency results in an elegant and useful theory that says something meaningful about practical computation, and would be impossible without this simplification.*"

Satisfiability and Validity

Definition

- A Boolean formula φ is satisfiable iff
  there is a truth assignment T appropriate to φ with $T(\varphi) = \mathit{true}$.
- A Boolean formula φ is valid iff
  for every truth assignment T appropriate to φ , $T(\varphi) = \mathit{true}$.

SAT

INSTANCE: Boolean formula φ.

QUESTION: Is φ satisfiable?

VALIDITY

INSTANCE: Boolean formula φ.

QUESTION: Is φ valid?

Towards the Class NP

Many problems can be analyzed in terms of solutions, proofs, certificates, or witnesses. For example, consider the SAT problem:

- If φ is satisfiable (i.e. is a positive instance of **SAT**), then there is a variable assignment (a solution) that proves the satisfiability of φ.
- If φ is unsatisfiable (i.e. is a negative instance of **SAT**), then a proof of satisfiability cannot be found.

Note that proofs in the case of **SAT** are succinct (i.e. short):

- any assignment can be represented as a subset A of the variables in φ (variables in A are set to *true*; the remaining ones are set to *false*).
- thus an assignment for a formula φ can be encoded in a string of length polynomial (even linear) in the size of φ.

Towards the Class NP

- Observe that given a formula φ and a variable assignment A, it is computationally easy to check if φ evaluates to *true* under A.
  - recall that the evaluation of propositional formulas is in P
- We can decide satisfiability of φ by going through the possible assignments A and efficiently checking whether an assignment makes φ *true*.
- What is the strategy to go through the assignments? Hard to say.
- Naive traversing of all possible assignments would take exponential time:
  - 2^n possible assignments, where n is the number of variables in φ.
- If we are lucky and hit the right assignment soon, the algorithm may terminate within polynomially many steps.

Towards the Class NP

SAT is just an example of a problem belonging to the class **NP**

Intuitively, NP is the class of problems $\mathcal{P}$ such that:

- positive instances I of $\mathcal{P}$ have short solutions (we will call them certificates), and
- given an instance I of $\mathcal{P}$ and a candidate certificate C, it is computationally cheap to check if C certifies that I is a positive instance of $\mathcal{P}$.

"short" and "cheap" means "polynomial in the size of the instance"!

Certificates

Definition (Certificate relation)

Let $\mathcal{P}$ be a decision problem, let $INSTANCES(\mathcal{P})$ denote the set of all instances of $\mathcal{P}$.

A certificate relation for $\mathcal{P}$ is a relation $R \subseteq INSTANCES(\mathcal{P}) \times CERT$, where $CERT$ is a set of finite objects, such that

$$I \in INSTANCES(\mathcal{P}) \text{ is a positive instance of } \mathcal{P}$$

iff

$$\text{there exists } C \in CERT \text{ such that } (I, C) \in R.$$

The Class NP

We formalize the notion of certificates that are short and easy to verify:

Definition

Assume a binary relation R.

We say R is polynomially decidable if there is a polynomial-time algorithm that checks, given a pair v_1, v_2 of objects, whether $(v_1, v_2) \in R$.

We say R is polynomially balanced if $(v_1, v_2) \in R$ implies $|v_2| \leq |v_1|^k$ for some fixed $k \geq 1$.

Definition (The Class NP)

A decision problem $\mathcal{P}$ is in the class NP if there exists a *polynomially balanced and polynomially decidable certificate relation* for $\mathcal{P}$.

Example: **SAT** is in **NP**

Theorem

SAT $\in$ **NP**.

Proof.

To prove that **SAT** $\in$ **NP** we have to define a polynomially balanced and polynomially decidable certificate relation for **SAT**. Simply let

$$R = \{(\varphi, \mu) \mid \text{formula } \varphi \text{ evaluates to } \textit{true} \text{ under assignment } \mu\}.$$

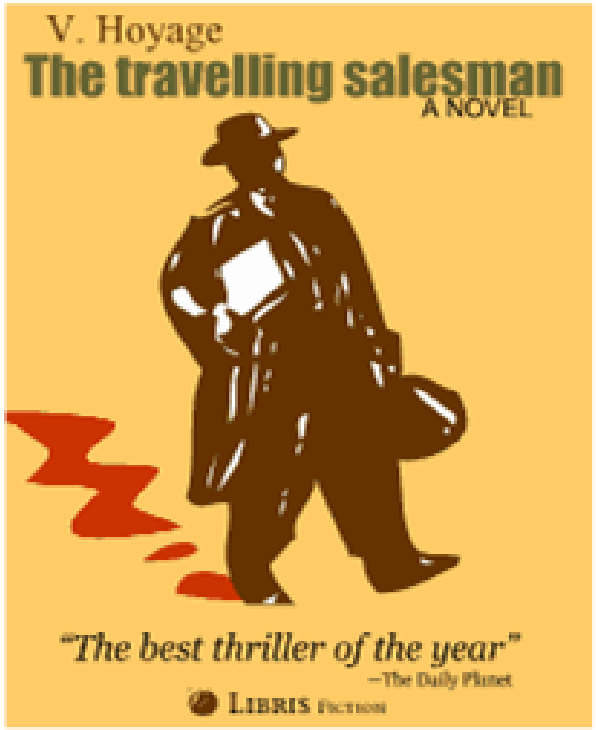
- R is a certificate relation by construction: φ is a positive instance of **SAT** $\Leftrightarrow$ there exists an assignment μ with makes φ evaluate to *true* $\Leftrightarrow$ $(\varphi, \mu) \in R$.
- R is polynomially balanced because each assignment μ for φ can be represented as a subset of variables in φ.
- R is polynomially decidable, because, as we have seen, evaluating the propositional formula φ under μ takes only polynomial time.

NP and Guess & Check Procedures

- Each problem in NP can be seen in terms of a guess & check procedure.
- Assume $\mathcal{P} \in NP$ and let R be the polynomially balanced and polynomially decidable certificate relation for $\mathcal{P}$.
- Intuitively, I is a positive instance of $\mathcal{P}$ iff we can guess a correct certificate for I.
- Since R is polynomially balanced, each guess of a candidate certificate for I is of polynomial size (certainly, there may be exponentially many candidates). Since R is polynomially decidable, a guess can be verified for correctness in polynomial time.
- Observation: there is an interesting discrepancy between the positive and the negative instances of NP problems:
  - A positive instance can be solved in polynomial time if we guess right. $\Rightarrow$ Heuristics are helpful (to navigate in the big search space).
  - A good guessing strategy does not help to solve negative instances.

Decision Problems vs Optimization

- In this lecture we concentrate on decision problems.
- However, the results on decision problem can be transferred to other problems as well.
- Example: Traveling Salesman Problem.



Example: Traveling Salesman Problem

Optimization Problem

TSP

INSTANCE: n cities $1, \ldots, n$ and a nonnegative integer distance d_{ij} between any two cities i and j (such that $d_{ij} = d_{ji}$).

QUESTION:

What is the length of the shortest tour of the cities, i.e.,
there exists a permutation π such that

$$\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)}$$

is as small as possible (where $\pi(n+1) = \pi(1)$)?

Optimization vs. Decision Problem

Decision Problem

TSP(D)
INSTANCE: n cities $1, \ldots, n$ and a nonnegative integer distance d_{ij} between any two cities i and j (such that $d_{ij} = d_{ji}$), and a "budget" B.
QUESTION: Is there a tour of length at most B?

Binary search for solving the optimization problem

- compute $B = n \cdot \max(\{d_{ij} \mid 1 \leq i, j \leq n\})$
- search for solution in interval $[0, B]$:
  Does there exist a tour of length at most $B/2$?
- if 'yes', then continue search in interval $[0, B/2]$
- if 'no', then continue search in interval $[B/2, B]$
- ...

TSP(D) in NP

- **TSP(D)** is another example of a problem in NP:
  - we can guess a candidate tour
  - and check whether the tour visits all the cities and is within the budget.
- All tours are of polynomial length and the verification step is also feasible in polynomial time.
- To see this more formally, we need to define a polynomially balanced and decidable certificate relation for **TSP(D)**.
- For the certificate relation, we simply let R be the set of all pairs (I, T) such that I is an instance of **TSP(D)** and T is a tour that visits all the cities of I and is within the budget of I.
- R is polynomially balanced because each tour T for an instance I is of size linear in $|I|$.
- R is polynomially decidable. Indeed, to check whether (I, T) is included in R it suffices to sum-up the distances in the tour T and check if the result is within the budget of I.

Discussion

Optimization vs. decision problems

- Complexity Theory mostly deals with decision problems: All complexity classes that we encounter in this lecture (and also in the Complexity Theory lecture) are classes of decision problems.
- Every optimization problem has a natural "corresponding" decision problem, i.e.: just introduce an upper bound (for minimization) or a lower bound (for maximization) on the target function.
- Complexity results for decision problems are highly relevant for the corresponding optimization problem: in particular negative results.
- Often (e.g., through the idea of binary search) we may conclude that the decision problem and the optimization problem are either both tractable or both intractable.

Learning Objectives

- Asymptotic, worst-case complexity vs. other notions of complexity
- Basic understanding of growth rates (polynomial vs. exponential)
- The class P
- The class NP
- Tractability vs. intractability
- Optimization vs. decision problem