Principles of Knowledge Representation and Reasoning
Complexity Theory

Bernhard Nebel, Malte Helmert and Stefan Wölfl

Albert-Ludwigs-Universität Freiburg

April 29, 2008
Principles of Knowledge Representation and Reasoning

April 29, 2008 — Complexity Theory

Motivation

Reminder: Basic Notions

- Algorithms and Turing Machines
- Problems, Solutions, and Complexity
- Complexity Classes P and NP
- Upper and Lower Bounds
- Polynomial Reductions
- NP-Completeness

Beyond NP

- The Class co-NP
- The Class PSPACE
- Other Classes

Oracle TMs and the Polynomial Hierarchy

- Oracle Turing-Machines
- Turing Reduction
- Complexity Classes Based on OTMs
- QBF
Motivation for Using Complexity Theory

- Complexity theory can answer questions on how easy or hard a problem is.
- Gives hints on what algorithms could be appropriate, e.g.:
  - Algorithms for polynomial-time problems are usually easy to design.
  - For NP-complete problems, backtracking and local search work well.
- Gives hints on what type of algorithm will (most probably) not work:
  - For problems that are believed to be harder than NP-complete ones, simple backtracking will not work.
- Gives hint on what sub-problems might be interesting.
Algorithms and Turing Machines

- We use Turing machines as formal models of algorithms.
- This is justified, because:
  - we assume that Turing machines can compute all computable functions.
  - the resource requirements (in term of time and memory) of a Turing machine are only polynomially worse than other models.
- The regular type of Turing machine is the deterministic one: DTM (or simply TM).
- Often, however, we use the notion of nondeterministic TMs: NDTM.
A problem is a set of pairs \((I, A)\) of strings in \(\{0, 1\}^*\).  

\(I\): Instance;  
\(A\): Answer.  

If \(A \in \{0, 1\}\): decision problem  

A decision problem is the same as a formal language: namely the set of strings formed by the instances with answer 1.  

An algorithm decides (or solves) a problem if it computes the right answer for all instances.  

The complexity of an algorithm is a function  

\[ T : \mathbb{N} \rightarrow \mathbb{N}, \]

measuring the number of basic steps (or memory requirement) the algorithm needs to compute an answer depending on the size of the instance.  

The complexity of a problem is the complexity of the most efficient algorithm that solves this problem.
Complexity Classes P and NP

Problems are categorized into complexity classes according to the requirements of computational resources:

- The class of problems decidable on deterministic Turing machines in polynomial time: \( P \)
- Problems in \( P \) are assumed to be efficiently solvable (although this might not be true if the exponent is very large)
- In practice, this notion appears to be more often reasonable than not
- The class of problems decidable on non-deterministic Turing machines in polynomial time: \( NP \)
- More classes are definable using other resource bounds on time and memory
Upper and Lower Bounds

- **Upper bounds** (membership in a class) are usually easy to prove:
  - provide an algorithm
  - show that the resource bounds are respected

- **Lower bounds** (hardness for a class) are usually difficult to show:
  - the technical tool here is the polynomial reduction (or any other appropriate reduction)
  - show that some hard problem can be reduced to the problem at hand
Polynomial Reductions

- Given two languages $L_1$ and $L_2$, $L_1$ can be \textit{polynomially reduced to} $L_2$, written $L_1 \leq_p L_2$, iff there exists a polynomially computable function $f$ such that

$$x \in L_1 \text{ iff } f(x) \in L_2$$

- It cannot be harder to decide $L_1$ than $L_2$
- $L$ is \textit{hard} for a class $C$ ($C$-hard) iff all languages of this class can be reduced to $L$.
- $L$ is \textit{complete} for $C$ ($C$-complete) iff $L$ is $C$-hard and $L \in C$. 
NP-complete Problems

- A problem is **NP-complete** iff it is **NP-hard** and in **NP**.
- Example: **SAT** – the satisfiability problem for propositional logic – is NP-complete (Cook/Karp)
- Membership is obvious, hardness follows because computations on a NDTM correspond to satisfying truth-assignments of certain formulae
Beyond NP  The Class co-NP

The Complexity Class co-NP

- Note that there is some asymmetry in the definition of NP:
  - It is clear that we can decide SAT by using a NDTM with polynomially bounded computation
  - There exists an accepting computation of polynomial length iff the formula is satisfiable
  - What if we want to solve UNSAT, the complementary problem?
  - It seems necessary to check all possible truth-assignments!

- Define co-$C = \{ L | \Sigma^* - L \in C \}$, provided $\Sigma$ is our alphabet
- co-NP = $\{ L | \Sigma^* - L \in NP \}$
- For example UNSAT, TAUT $\in$ co-NP!
- Note: P is closed under complement, i.e.,

\[ P \subseteq NP \cap co-NP \]
Beyond NP  The Class PSPACE

PSPACE
There are problems even more difficult than NP and co-NP.

Definition ((N)PSPACE)
PSPACE (NPSPACE) is the class of decision problems that can be decided on deterministic (non-deterministic) Turing machines using only polynomially many tape cells.

Some facts about PSPACE:
- PSPACE is closed under complements (as all other deterministic classes)
- PSPACE is identical to NPSPACE (because non-deterministic Turing machines can be simulated on deterministic TMs using only quadratic space)
- NP \(\subseteq\) PSPACE (because in polynomial time one can “visit” only polynomial space, i.e., NP \(\subseteq\) NPSPACE)
- It is unknown whether NP \(\neq\) PSPACE, but it is believed that this is true.
PSPACE-completeness

Definition (PSPACE-completeness)
A decision problem (or language) is PSPACE-complete, if it is in PSPACE and all other problems in PSPACE can be polynomially reduced to it. Intuitively, PSPACE-complete problems are the “hardest” problems in PSPACE (similar to NP-completeness). They appear to be “harder” than NP-complete problems from a practical point of view.

An example for a PSPACE-complete problem is the NDFA equivalence problem:

**Instance**: Two non-deterministic finite state automata $A_1$ and $A_2$.

**Question**: Are the languages accepted by $A_1$ and $A_2$ identical?
Other Complexity Classes . . .

- There are complexity classes **above** PSPACE (EXPTIME, EXPSPACE, NEXPTIME, DEXPTIME . . .)
- there are (infinitely many) classes **between** NP and PSPACE (the polynomial hierarchy defined by oracle machines)
- there are (infinitely many) classes **inside** P (circuit classes with different depths)
- and for most of the classes **we do not know** whether the containment relationships are **strict**
Oracle Turing Machines

- An **Oracle Turing machine ((N)OTM)** is a Turing machine (DTM, NDTM) with the possibility to query an **oracle** (i.e., a different Turing machine **without resource restrictions**) whether it accepts or rejects a given string.

- **Computation by the oracle does not cost anything!**

- **Formalization:**
  - A tape onto which strings for the oracle are written,
  - A yes/no answer from the oracle depending on whether it accepts or rejects the input string.

- **Usage of OTMs answers what-if questions:** What if we could solve the oracle-problem efficiently?
Turing Reductions

- **OTMs** allow us to define a more general type of reduction.
- **Idea:** The “classical” reduction can be seen as calling a subroutine once.
- $L_1$ is **Turing-reducible** to $L_2$, symbolically $L_1 \leq_T L_2$, if there exists a poly-time OTM that decides $L_1$ by using an oracle for $L_2$.
- Polynomial reducibility implies Turing reducibility, but not vice versa!
- NP-hardness and co-NP-hardness with respect to Turing reducibility are equivalent!
- Turing reducibility can also be applied to general search problems!
Complexity Classes Based on Oracle TMs

1. $P^{NP}$ = decision problems solved by poly-time DTM with an oracle for a decision problem in NP.

2. $NP^{NP}$ = decision problems solved by poly-time NDTM with an oracle for a decision problem in NP.

3. co-$NP^{NP}$ = complements of decision problems solved by poly-time NDTM with an oracle for a decision problem in NP.

4. $NP^{NP^{NP}}$ = ...

... and so on
Example

- Consider the **Minimum Equivalent Expression (MEE) problem**:

  **Instance**: A well-formed Boolean formula $\phi$ using the standard connectives (not $\leftrightarrow$) and a nonnegative integer $K$.

  **Question**: Is there a well-formed Boolean formula $\phi'$ that contains $K$ or fewer literal occurrences and that is logical equivalent to $\phi$?

- This problem is NP-hard (wrt. to Turing reductions).
- It does not appear to be NP-complete
- We could guess a formula and then use a SAT-oracle

  $\text{MEE} \in \text{NP}^\text{NP}$. 
The Polynomial Hierarchy

The complexity classes based on OTMs form an infinite hierarchy.

The polynomial hierarchy PH

\[
\begin{align*}
\Sigma_0^p &= P \\
\Sigma_{i+1}^p &= NP^{\Sigma_i^p} \\
\Pi_0^p &= P \\
\Pi_{i+1}^p &= \text{co-}\Sigma_{i+1}^p \\
\Delta_0^p &= P \\
\Delta_{i+1}^p &= P^{\Sigma_i^p}
\end{align*}
\]

\[
\begin{align*}
\text{PH} &= \bigcup_{i \geq 0} (\Sigma_i^p \cup \Pi_i^p \cup \Delta_i^p) \subseteq \text{PSPACE} \\
\text{NP} &= \Sigma_1^p \\
\text{co-NP} &= \Pi_1^p
\end{align*}
\]
Quantified Boolean Formulae: Definition

- If $\phi$ is a propositional formula, $P$ is the set of Boolean variables used in $\phi$ and $\sigma$ is a sequence of $\exists p$ and $\forall p$, one for every $p \in P$, then $\sigma \phi$ is a QBF.

- A formula $\exists x \phi$ is true if and only if $\phi[\top/x] \lor \phi[\bot/x]$ is true. (Equivalently, $\phi[\top/x]$ is true or $\phi[\bot/x]$ is true.)

- A formula $\forall x \phi$ is true if and only if $\phi[\top/x] \land \phi[\bot/x]$ is true. (Equivalently, $\phi[\top/x]$ is true and $\phi[\bot/x]$ is true.)

- This definition directly leads to an AND/OR tree traversal algorithm for evaluating QBF.
Quantified Boolean Formulae: Definition

The evaluation problem of QBF generalizes both the *satisfiability* and *validity/tautology problems* of propositional logic. The latter are respectively *NP-complete* and *co-NP-complete* whereas the former is *PSPACE-complete*.

**Example**
The formulae $\forall x \exists y (x \leftrightarrow y)$ and $\exists x \exists y (x \land y)$ are true.

**Example**
The formulae $\exists x \forall y (x \leftrightarrow y)$ and $\forall x \forall y (x \lor y)$ are false.
The Polynomial Hierarchy: Connection to QBF

Truth of QBFs with prefix $\exists \forall \exists \ldots$ is $\Sigma^p_i$-complete.

Truth of QBFs with prefix $\forall \exists \forall \ldots$ is $\Pi^p_i$-complete.

Special cases corresponding to SAT and TAUT:
The truth of QBFs with prefix $\exists x^1_1 \ldots x^1_n$ is $\text{NP}= \Sigma^p_1$-complete.
The truth of QBFs with prefix $\forall x^1_1 \ldots x^1_n$ is $\text{co-NP}= \Pi^p_1$-complete.
Literature

M. R. Garey and D. S. Johnson. 

C. H. Papadimitriou. 
*Computational Complexity.* 