An Overview of Complexity Theory

5DV037 — Fundamentals of Computer Science

Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

What is Complexity Theory

- Until this point, the focus has been on <u>what</u> can be done with a particular computing model.
- Attention is now turned to <u>how efficiently</u> tasks can be performed.
  - Time resources required (time complexity)
  - Space resources required (space complexity)
- There are three levels at which these question may be asked:

Algorithm analysis: How well does a given algorithm perform a given task?

  - How efficient is quicksort?

Problem complexity: What is the best performance possible for a given problem?

  - How efficient is the best possible sorting algorithm?

Complexity theory: How can different problems in general be classified in terms of complexity?

  - How does the complexity of sorting compare to that of finding minimum spanning trees?

Complexity Measures for Computations on TMs

- Turing machines provide an ideal framework for formulating abstract complexity theory.
- The number of steps which such a machine takes in performing a computation is inherent in the model.
  - Just count the number of transitions..
  - the length of the computation from initial configuration to the halt configuration.
- The size of the input is the length of the input string.
- These parameters are independent of the problem and independent of the representation of the input.
- Other models of computation do not always provide such flexibility.

A Review of “Big-Oh” Notation

- Typically, the performance of an algorithm is measured in terms of the size n of the input.
- Time or space usage may be measured; here time will be chosen since it is the most common resource to be so measured.
- Recall: An algorithm is $O(f(n))$ If there is:
  - a constant $k > 0$, and
  - an $n_0 \in \mathbb{N}$, such that:
  - for all $n \geq n_0$, the algorithm runs in at most $k \cdot f(n)$ time units.

Example: A “good” sorting algorithm runs in time $O(n \cdot \log(n))$.

  - The parameter n measures the number of elements to be sorted.
  - The time is measured in terms of some primitive execution units of the computer (assign, compare, add, *etc.*).

- This model may be used for *worst-case*, *average-case*, and *best-case* time.

Limitations of the Problem-Specific Approach

- This model works well when comparing different algorithms for the same problem.
- However, it requires modification to be useful in comparing different problems.
- Consider the assumptions made in modelling the sorting problem:
  - Each element in the input sequence is of a fixed size.
  - Operations such as comparison take fixed time regardless of the size of the elements which are to be compared.
- These assumptions must fail as n becomes sufficiently large and the input consists of distinct elements.
- Other problems may use other assumptions.
  - ➵ Such assumptions make it difficult to compare the complexity of algorithms for different problems.
  - Particularly, the techniques to be developed transform one problem to another..
  - and this requires a uniform method of problem encoding.

Low-Level Measurement of Complexity

- In order to compare algorithms for different problems, a lower-level notion of complexity is appropriate.
- This model is based upon the ubiquitous DTM.
- The size of the input is measured by the length of the representation as a string in the input alphabet Σ.
  - This may be larger than the conventional length.

    Example; In a list of numbers to be sorted, the number m will require $\log(m)$ bits in binary notation, rather than a constant size regardless of m.
- The number of steps which an operation requires is measured by the number of steps that the implementing DTM takes.
  - This may be larger than the conventional programming-language convention.

    Example; The time required to compare two numbers will be proportional to the lengths of the representations of those numbers, rather than a constant.

Reasonable Encodings

- A further issue is that algorithms may be made to look better than they really are through the use of clever encoding.

Example: Encode numbers in unary and implement addition as concatenation.

Example: Encode numbers as their prime factors and implement multiplication as factor-by-factor addition.

- Both of these encoding schemes are “unreasonable” because they do not work with standard representations which may be used in many different problems.
- To obtain uniform results across diverse problems, and to ensure that transformations of one problem to another are meaningful, it is necessary that the encodings abide by certain constraints.

Structured Strings

- It is usually required that all algorithms employ encodings based upon *structured strings*, which are defined as follows.

  Numbers: Any string of 0's and 1's (possibly preceded by a minus sign) is a structured string which represents a number in base two.

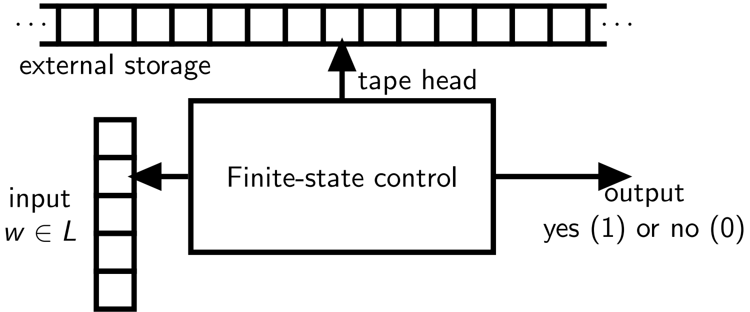
  Names: If σ is a structured string, then so too is $[\sigma]$, which represents a *name* encoded by σ.

  Lists: If $\sigma_1, \sigma_2, \ldots, \sigma_k$ are structured strings, then so too is $\langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$, representing the corresponding *list* or *tuple*.

- This is enough to encode problem instances for most problems of interest.
- Since numbers, tuples, and names are encoded in a standard way, comparison of input size for different problems becomes feasible.
- Note that this approach will not generally result in the "standard" encoding for specific problems, such as sorting.

Dependence upon the Specific Model of Turing Machine

- The Church-Turing thesis provides a common upper bound on what a computing machine can do.
- However, it says nothing about complexity.
- Different models of computer can and do have vastly different complexities for a given algorithm.
- To reconcile this, the standard definition of abstract complexity is based upon a multi-tape Turing machine.
- In particular, the input in on a different tape than the working memory.



Problem Classes of the Form $DTIME(T(n))$

- A *complexity function* is any function $f : \mathbb{N} \rightarrow \mathbb{R}$ (here $\mathbb{R}$ is the real numbers)
  - which is *eventually nonnegative* in the sense that there is an $n_0 \in \mathbb{N}$
  - such that for any $n \geq n_0$, $f(n) \geq 0$.
- Fix the input alphabet to be $\{0, 1\}$.
- Given a complexity function f, define $DTIME(f(n))$ to be the set of all languages (or decision problems) which can be decided on a multitape DTM in $O(f(n))$ steps, with n representing Length(w).
- The name *DTIME* stands for *deterministic time*.
- Some authors use the notation $TIME(f(n))$ instead.
- Some authors view $DTIME(f(n))$ to mean those problems which can be solved in at most $f(n)$ steps on a multitape DTM for every input of length at most n (with no requirement that n be large and with no scaling by a constant).

Relative Complexity for Different Models of DTM

- How dependent is this notion upon the particular model of DTM?

Theorem: Suppose that a given problem P may be solved in at most $f(n)$ steps for $DTIME(f(n))$ for some complexity function f.

  - Then P may be solved on a DTM with only one tape in at most $(f(n))^2$ steps. $\square$

- In other words, the "slowdown" in going from a multitape DTM to a single-tape DTM is at most square in the original complexity.

Example: If a given problem may be solved in at most $(\text{Length}(w))^3$ steps on a multitape DTM, then it may be solved on a single-tape DTM in at most $(\text{Length}(w))^6$ steps.

- For the purposes of the framework to be developed, this is not of major importance, as will be seen next.

The Problem Class $\mathcal{P}$

- Define
$$\mathcal{P} = \bigcup_{i \in \mathbb{N}} DTIME(n^i)$$
- $\mathcal{P}$ is the set of all decision problems which can be solved in polynomial time on a DTM.
- It is also said that $\mathcal{P}$ is the set of problems which may be solved in *deterministic polynomial time*.
- Note that the $f(n) \rightsquigarrow f(n)^2$ "slowdown" for multi-tape to single-tape DTMs does not affect the membership of this class.
- It would be the same were the definition of $DTIME(f(n))$ for single-tape machines.

Keep in mind: Everything is decidable; this is about complexity, not about halting!

Which Problems Are in $\mathcal{P}$?

- Membership in the class $\mathcal{P}$ is often taken as the gold standard for whether or not a given problem admits a *tractable* solution or not.
- Unfortunately, for many problems of immense practical importance, no (deterministic) polynomial-time algorithm is known.
- Yet, it has never been proven that no such algorithm can exist.
- The focus of this discussion is to try to understand this situation better.
- Many problems which fall into this class exhibit a unique behavior:
  - Very efficient algorithms (typically $O(n)$) exist for *verifying* that a candidate solution is correct.
  - The best known algorithms for *finding* a solution are exponential $O(2^n)$ or nearly so.
  - Some examples will illustrate this situation.

Example — Satisfiability of Boolean Expressions

- Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables.
- A *truth assignment* to X is a mapping $h : X \to \{0, 1\}$.
- x_i is *true* for h if $h(x_i) = 1$, and *false* for h if $h(x_i) = 0$.
- The *Boolean expressions over X*, denoted $\mathrm{BE}(X)$, are built up from from X in the usual way, using $\neg$, $\vee$, and $\wedge$.

Examples:

$$\varphi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg(\neg x_3 \wedge x_4))$$
$$\varphi_2 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg(\neg x_3 \vee x_4))$$

- The truth assignment $h : X \to \{0, 1\}$ extends to Boolean expressions in the obvious way $\bar{h} : \mathrm{BE}(X) \to \{0, 1\}$.
- The formula φ is *satisfiable* if there is a truth assignment h for which $\bar{h}(\varphi) = 1$.

Examples: φ_1 is satisfiable with $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$ or $(0, 1, 0, 0)$.
φ_2 is unsatisfiable.

- This general problem (as X ranges over all finite sets of variables) is known as SAT (satisfiability of Boolean expressions).

Finding vs. Verifying a Solution

- It is easy to verify that a proposed solution is valid:

Example: Verify that $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$ satisfies
$\varphi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \wedge \neg x_2) \wedge (\neg(\neg x_3 \wedge x_4))$.

- $(1 \vee 0) \wedge (\neg 1 \vee 0 \vee \neg 0) \wedge (1 \vee \neg 0 \vee \neg 0) \wedge (\neg 1 \vee \neg 0) \wedge (\neg(\neg 0 \wedge 0)) =$
$(1 \vee 0) \wedge (0 \vee 0 \vee 1) \wedge (0 \vee 1 \vee 1) \wedge (\neg(1 \wedge 0)) = (1) \wedge (1) \wedge (1) \wedge (\neg 0) =$
$(1) \wedge (1) \wedge (1) \wedge (1) = 1$.
- Such verification can be performed in at most quadratic time on a multi-tape DTM (better on a random-access machine).
- However, in order to
  - (a) find a solution, or to
  - (b) determine that no solution exists,

  no approach which is substantially better than exhaustive search is known.
- For a formula with n variables, the number of possibilities is 2^n.
  - ➽ Determining unsatisfiability has exponential complexity in the worst case.

Other Problems which Have Similar Properties

- Many important problems exhibit these properties:
  - Verification of a candidate solution is fast (typically no worse than $O(n^2)$)
  - The best known algorithms for finding a solution are exponential.

Example: The 0/1 Knapsack decision problem:

  - A *knapsack* with capacity M.
  - A set E of objects, with each object a having a *weight* w_a and a *value* v_a.
  - A goal total value (or profit) P.
- Find a subset $S \subseteq E$ with:
  - value at least P: $\sum_{a \in S} \mathrm{v}_a \geq P$.
  - weight at most M: $\sum_{a \in S} \mathrm{w}_a \leq M$.
- Application: Optimization of resource usage.
  - Value = profit.
  - Weight = resource usage by the given object.
  - Capacity = total amount of resources available.

Other Problems which Have Similar Properties — 2

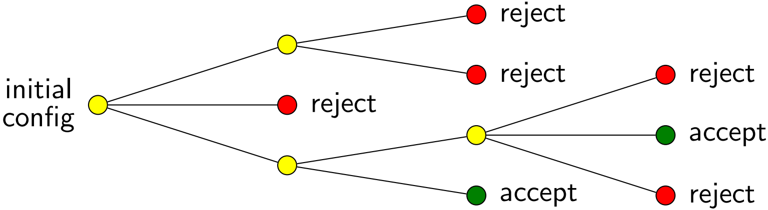
- Graph problems:
  - vertex cover
  - clique
  - Hamiltonian circuit
- Allocation problems:
  - partition
  - three-dimensional matching.
- Plus thousands of others which have arisen over the years.
- All share this same property:
  - Easy to verify a candidate solution.
  - No known way which is substantially better in the worst case than exhaustive search (exponential complexity) to find a solution.
- But no one has ever been able to show that they are not in $\mathcal{P}$ either.

Common Properties of These Problems

- All of these problems have two properties in common.
    - Each can be solved efficiently on a <u>nondeterministic</u> TM.
    - They may each be transformed to the other efficiently (*i.e.*, in polynomial time).
- These properties will now be examined more closely, and their implications assessed.

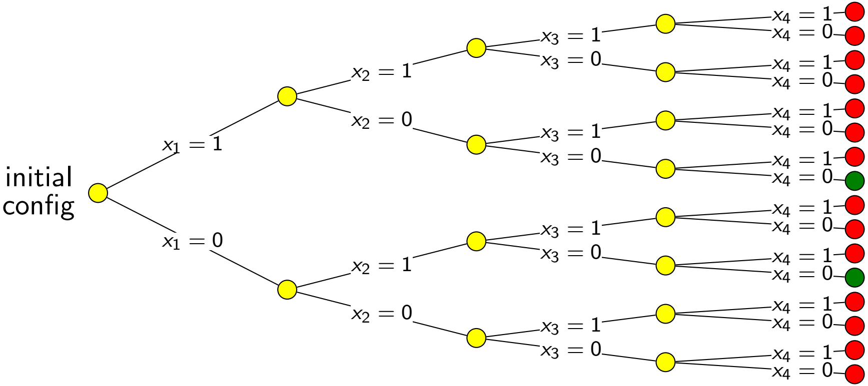
Problem Solving Using Nondeterministic Turing Machines

- Recall that a nondeterministic TM (NDTM) can have many parallel or alternative branches of execution.
- A string is accepted (or a problem answer is "yes") if <u>some</u> branch ends in an accepting state.
- A string is rejected (or a problem answer is "no") if <u>all</u> branches end in a rejecting state.
- Only *deciders* are considered; failure to halt is not a possibility.



Nondeterministic Solution of Satisfiability

- For a NDTM which tests for satisfiability of Boolean expressions, and a four-variable formula such as
  $\varphi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \wedge \neg x_2) \wedge (\neg(\neg x_3 \wedge x_4))$
- the alternatives of the machine will appear as shown below.
- Each path may be run in quadratic time, so the nondeterministic complexity is $O(n^2)$ (on an NDTM; better on a random-access machine).



Problem Classes of the Form $NTIME(T(n))$

- The definition of *NTIME* is similar to that of *DTIME*, but for nondeterministic machines.
- Given a complexity function f, define $NTIME(f(n))$ to be the set of all languages (or decision problems) which can be decided on a multitape NDTM in $O(f(n))$ steps, with n representing Length(w).
- The name *NTIME* stands for *nondeterministic time*.
- Some authors view $NTIME(f(n))$ to mean those problems which can be solved in at most $f(n)$ steps on a multitape NDTM for every input of length at most n (with no requirement that n be large and with no scaling by a constant).
- An $f(n) \rightsquigarrow f(n)^2$ "slowdown" for multi-tape to single-tape NDTMs exists, in analogy to the DTM case.

The Problem Class $\mathcal{NP}$

- The definition of $\mathcal{NP}$ is similar to that of $\mathcal{P}$, but using *NTIME* instead of *DTIME*:

$$\mathcal{NP} = \bigcup_{i \in \mathbb{N}} NTIME(n^i)$$

- $\mathcal{NP}$ is the set of all decision problems which can be solved in polynomial time on a nondeterministic TM (NDTM).
- It is also said that $\mathcal{NP}$ is the set of problems which may be solved in *nondeterministic polynomial time*.
- Note that the $f(n) \rightsquigarrow f(n)^2$ "slowdown" for multi-tape to single-tape DTMs does not affect the membership of this class.
- Think of $\mathcal{NP}$ as the set of decision problems which may be solved in polynomial time under the model of computation in which:
  - Unbounded branching of alternatives is allowed; and
  - Success of one branch is equivalent to success (a "yes" answer).

The Question of $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$

- Clearly $\mathcal{P} \subseteq \mathcal{NP}$, since every DTM may be regarded as an NDTM.

Question: What about the reverse inclusion, $\mathcal{NP} \subseteq \mathcal{P}$?

- It might seem "obvious" that this cannot be the case.
- Checking a solution is "obviously" less complex than determining whether a solution exists (within exponentially many possibilities).
- Many computer scientists feel that this is the case.
- But (up to polynomial-time equivalence) is it?
- Despite the practical experience, no one has ever been able to come close to showing that this is the case.
- It is perhaps the most famous and important open problem in theoretical computer science.

The Idea of $\mathcal{NP}$-Completeness

- There is a further dimension to this story.
- Most of the decision problems of the form:
  - it is easy to test a given solution for correctness; but
  - no algorithm in $\mathcal{P}$ is known for finding such a solution
- are equivalent in a very compelling way.
- If an algorithm in $\mathcal{P}$ could be found for finding solutions to one of these problems, then ...
- ... such an algorithm could be found for all such problems.
- This problems in this class are called $\mathcal{NP}$-complete, and the class is denoted $\mathcal{NPC}$.
- This important issue warrants a closer look.

Polynomial-Time Reduction

- A *reduction* of decision problem P_1 to decision problem P_2 is a computable function which maps instances of P_1 into instances of P_2 and which preserves "yes" and "no".
- This needs to be made a bit more precise.
- View a decision problem as a pair $P = (\mathrm{Inst}(P), \rho_P)$, in which
  - $\mathrm{Inst}(P)$ is the set of *instances* of P; and
  - $\rho_P : \mathrm{Inst}(P) \to \{0, 1\}$ is the function which gives the answer "yes" or "no" for each instance.

Example: For the problem SAT:

- $\mathrm{Inst}(\mathsf{SAT})$ is the set of all Boolean expressions (over finite sets of variables);
- ρ_{SAT} sends the Boolean expression φ to 1 if it is satisfiable, and 0 if it is not.

Polynomial-Time Reduction — 2

- Formally, a *reduction* of P_1 to P_2 is a computable function $e : \mathrm{Inst}(P_1) \to \mathrm{Inst}(P_2)$ which makes the following diagram commute:

$$
\begin{tikzcd}
\mathrm{Inst}(P_1) \arrow[rr, "\rho_{P_1}"] \arrow[d, "e"'] & & \{0,1\} \\
\mathrm{Inst}(P_2) \arrow[urr, "\rho_{P_2}"'] & &
\end{tikzcd}
$$

- This means that both paths from $\mathrm{Inst}(P_2)$ to $\{0,1\}$ yield the same result.
- Think of using e as a subroutine in a decider for P_2 in order to decide P_1.
- The reduction e is *polynomial* or *tractable* if there exists a DTM which computes it.
- Write

$$P_1 \propto P_2$$

just in case there is a polynomial reduction from P_1 to P_2.
- In this case, say that P_1 *polynomially reduces* (or *tractably reduces*) to P_2.

Example — Conjunctive Normal Form

- A *literal* is a Boolean expression of the form x or $\neg x$, with x a variable.
- A *clause* is an expression of the form $(\ell_1 \vee \ell_2 \vee \ldots \vee \ell_k)$ in which each ℓ_i is a literal.
- A Boolean formula is in *conjunctive normal form* (*CNF*) if it is a conjunction of clauses; *i.e.*,

$$(\ell_{11} \vee \ell_{12} \vee \ldots \vee \ell_{1k_1}) \wedge (\ell_{21} \vee \ell_{22} \vee \ldots \vee \ell_{2k_2}) \wedge \ldots \wedge (\ell_{m1} \vee \ell_{m2} \vee \ldots \vee \ell_{mk_m})$$

Example:

$$\varphi_1 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg(\neg x_3 \wedge x_4))$$

is not in CNF, while

$$\varphi_1' = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_4)$$

is in CNF.

- A Boolean formula in CNF is in *3-conjunctive normal form* (*3CNF*) if each clause contains at most three literals.

Example: φ_1' above is in 3CNF.

- The corresponding satisfiability problems are called CNF-SAT and 3CNF-SAT.

Example — Reduction of CNF-SAT to 3CNF-SAT

Proposition: CNF-SAT $\propto$ 3CNF-SAT.

Proof: It it suffices to give a reduction on clauses.

- This will be illustrated for a clause of five literals.
- The clause $(\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 \vee \ell_5)$ is satisfiable iff the conjunction $(\ell_1 \vee \ell_2 \vee y_1) \wedge (\ell_3 \vee \neg y_1 \vee y_2) \wedge (\ell_4 \vee \ell_5 \vee \neg y_2)$ is.
- The y_i's are new variables.
- This idea extends in a natural way, and may be performed in deterministic polynomial time. $\square$

Warning: You may have learned how to transform any Boolean expression into one in CNF is another course.

- This transformation is <u>not</u> polynomial.
- However, SAT $\propto$ CNF-SAT.

Formalization of $\mathcal{NP}$-Completeness and the Class $\mathcal{NPC}$

- A problem P is called *$\mathcal{NP}$-complete* if
  - (a) it is in $\mathcal{NP}$; and
  - (b) for every other problem $P' \in \mathcal{NP}$, $P' \propto P$.
- The collection of all $\mathcal{NP}$-complete problems is denoted $\mathcal{NPC}$.
- Intuitively, an $\mathcal{NP}$-complete problem is a "hardest" problem within $\mathcal{NP}$.

Question: Do $\mathcal{NP}$-complete problems exist?

Answer: Yes, there are many of them.

- The fundamental $\mathcal{NP}$-complete problem is SAT.
- This is known as *Cook's theorem*.

Cook's Theorem

Theorem (Stephen A. Cook, 1971): SAT $\in \mathcal{NPC}$; *i.e.*, the problem SAT is $\mathcal{NP}$-complete.

Proof idea: Let $P \in \mathcal{NP}$, and let M be a (single-tape) NDTM which solves P in nondeterministic polynomial time.

- Write a huge logical expression which describes the behavior of M for a given input $A \in \mathrm{Inst}(P)$.
- This expression uses propositions of the following forms:

$$\begin{aligned} C(i,j,t) = 1 &\quad\Leftrightarrow\quad \text{tape cell } i \text{ contains symbol } j \text{ at time } t. \\ S(k,t) = 1 &\quad\Leftrightarrow\quad M \text{ is in state } q_k \text{ at time } t. \\ H(i,t) = 1 &\quad\Leftrightarrow\quad \text{the tape head is scanning cell } i \text{ at time } t. \end{aligned}$$

- The parameters i, j, k, and t are bounded in value, so these are just (parameterized) propositions.
- The expression may be generated in deterministic polynomial time.
- The logical expression describing the behavior of M is satisfiable iff A is true for P (the answer is "yes"). $\square$

Implications of Cook's Theorem

Corollary: If SAT $\in \mathcal{P}$, then every $P \in \mathcal{NP}$ is also in $\mathcal{P}$. $\Box$

- In other words, if SAT $\in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
- Over the years, thousands of other important (and not so important) problems have also been shown to be $\mathcal{NP}$-complete, including:
  - CNF-SAT and 3CNF-SAT,
  - the discrete knapsack problem,
  - the other problems on the list presented earlier.
- If any one of these problems could be shown to be in $\mathcal{P}$, then they would all be in $\mathcal{P}$.
- Still, no one has been able to do this.

Question: Are there problems in $\mathcal{NP}$ which are not in $\mathcal{NPC}$?

Answer: Excluding trivial problems (always "yes" or always "no")...

  - ... a positive answer would imply that $\mathcal{P} \neq \mathcal{NP}$.

- Nobody knows.

$\mathcal{NP}$-Incompleteness $\Rightarrow \mathcal{P} \neq \mathcal{NP}$

- Let Idprob $= (\{0,1\}, \mathbf{1})$ be the *identity problem* with $\mathbf{1} : \{0,1\} \to \{0,1\}$ the identity function.

Observation: If $P \in \mathcal{P}$, then $P \propto$ Idprob.

Proof: Let $P = (\mathsf{Inst}(P), \rho_P)$ be any problem in $\mathcal{NP}$, and consider the diagram below.

$$\begin{tikzcd}
\mathsf{Inst}(P) \arrow[rr, "\rho_P"] \arrow[d, "e = \rho_P"'] & & \{0,1\} \\
\{0,1\} \arrow[urr, "\mathbf{1}"'] & &
\end{tikzcd}$$

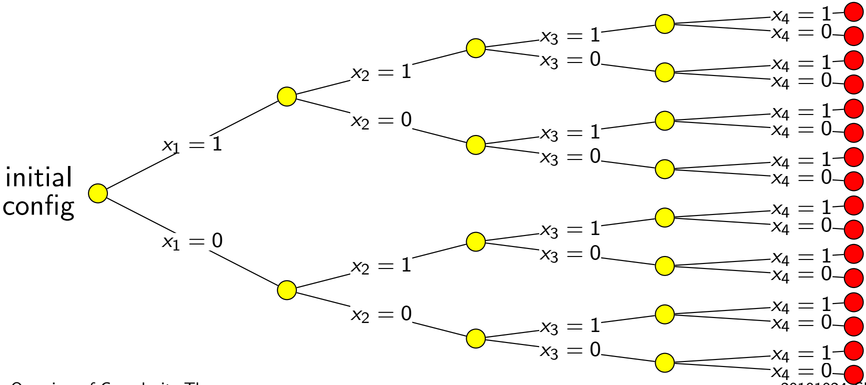
- If $\mathcal{P} = \mathcal{NP}$, then every $P \propto$ Idprob for every $P \in \mathcal{NP}$; *i.e.*, Idprob is $\mathcal{NP}$-complete.
- From this it follows that, if $\mathcal{P} = \mathcal{NP}$, then any *nontrivial* decision problem which is in $\mathcal{NP}$ is $\mathcal{NP}$-complete if
- A decision problem is *nontrivial* if it is true for some of its instances and false for others.

Co-$\mathcal{NP}$ Problems

- In contrast to that of $\mathcal{P}$, the definition of $\mathcal{NP}$ is asymmetric.

Example: Consider the problem SAT again.

- If a Boolean expression φ is satisfiable, this may be discovered in nondeterministic polynomial time..
- However, to establish unsatisfiability requires that all possibilities fail.
- The branching behavior of the NDTM does not appear to help.



Co-$\mathcal{NP}$ Problems — 2

- The *complement* $\overline{P}$ of a decision problem just switches 0 and 1 (or "yes" and "no").
- $\rho_{\overline{P}} = 1 - \rho_P$.

Example: Unsatisfiability of Boolean expressions is the complement of satisfiability.

- A problem P is in co-$\mathcal{NP}$ if its complement $\overline{P} \in \mathcal{NP}$.
- As illustrated, co-$\mathcal{NP}$ problems are "intuitively" more difficult than problems which are in $\mathcal{NPC}$.
- However ...

Theorem: If there is a problem $P \in \mathcal{NP}$ with $\overline{P} \notin \mathcal{NP}$, then $\mathcal{P} \neq \mathcal{NP}$.

- So, if it could be shown, for example, that <u>un</u>satisfiability of Boolean expressions cannot be solved in <u>non</u>deterministic polynomial time, then $\mathcal{P} \neq \mathcal{NP}$.

$\mathcal{NP}$-Hard Problems

- The terminology $\mathcal{NP}$-hard is used in two distinct but related ways.
- It is used to describe decision problems which are at least as hard as $\mathcal{NP}$-complete problems.
  - In this sense, all complements of $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard.
- It is used to describe optimization (and other) problems which arise from decision problems in $\mathcal{NP}$.

Example: The 0/1 Knapsack *optimization* problem.

  - A *knapsack* with capacity M.
  - A set E of objects, with each object a having a *weight* w_a and a *value* v_a.

- Find the most value which can be placed in the knapsack without exceeding the capacity.
- Rather than asking to meet a target value, find the most valuable configuration.

For More Information

- The following notes, from a course on the analysis of algorithms, present a somewhat more formal and complete look at some of the topics of these slides.

  `http://www8.cs.umu.se/~hegner/Courses/TDBC91/H08/Slides/cmplxthy9.pdf`

- The slides from the whole course may be found here:

  `http://www8.cs.umu.se/~hegner/Courses/TDBC91/H08/Slides/index.html`