An Overview of Computational Complexity

Chap. 14

Summary

- While the issue on TM was only whether or not something could be done in *principle*, the focus shifts to *how well things can be done in practice*.
- The 2nd part of the theory of computation is introduced: *complexity theory*, an extensive topic that deals with the *efficiency of computation*.
- Some typical issues in complexity theory, including the *role of nondeterminism*.

Learning Objectives

- Explain the concept of *Computational Complexity* as it relates to Turing Machines.
- Describe *deterministic* and *nondeterministic solutions* to the *SAT problem*.
- Determine if a *Boolean expression in CNF* is *satisfiable*.
- Describe the efficiency of standard TMs that simulate two-tape machines and of those that simulate nondeterministic machines.
- Define the Complexity classes *P* and *NP*, as well as the relationship between P and NP.
- Explain the concepts of *intractability* and *NP-completeness.*
- List some well-known *NP-complete problems.*
- Discuss the significance and status of the *P = NP?* Question.

Efficiency of Computation

- *Computational complexity* is
  the study of the *efficiency of algorithms*.
- When studying the (computing) time of an algorithm (= the time complexity of algorithm), the following *assumptions* are made:
  - The algorithm will be modeled by a Turing Machine.
  - The size of the problem will be denoted by n.
  - When analyzing an algorithm, the focus is on its *general behavior*, particularly as the size of the problem increases.
- A computation has Time-complexity *T(n)* if it can be completed in no more than T(n) moves on some TM.

Turing Machine Models and Complexity

- Although different models of TMs are equivalent, the efficiency of a computation can be affected by the *number of tapes* available and by whether it is *deterministic* or *nondeterministic*.
- Consider the *Satisfiability Problem* (SAT): Given a Boolean expression *e* in Conjunctive Normal Form (CNF), find an *assignment of values* to the Boolean variables that makes *e* to be *true.*
- For example, the expression $e_1 = (x_1' \vee x_2) \wedge (x_1 \vee x_3)$ is true when $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$. Note: $x_1' = \neg x_1$
- However, the expression $e_2 = (x_1 \vee x_2) \wedge x_1' \wedge x_2'$ is not satisfiable, i.e. no solution exists.

Solving the Satisfiability Problem

- A deterministic algorithm would take all possible values for the n variables and evaluate the expression for each combination.
- Since there are 2^n possibilities, the *deterministic* solution has *exponential time complexity*.
- A *nondeterministic* algorithm would guess the value of each of the n variables at each step and evaluate each of the 2^n possibilities simultaneously, thus resulting in an O(n) algorithm.
- *There is no known nonexponential deterministic algorithm for solving the SAT problem.*

Simulation of a Two-Tape Machine

- Theorem 14.1: If a *two-tape* TM can carry out a computation in n steps, the computation can be simulated by a standard TM in $O(n^2)$ moves.
- To simulate the two-tape computation, the standard TM would
  - Keep a description of the two-tape machine on its tape.
  - For each two-tape move, search the entire active area of its tape.
- After n moves, the active area has a length of at most $O(n)$, so the entire simulation takes $O(n^2)$ moves.

Simulation of a Nondeterministic TM (NTM)

- Theorem 14.2: If a Nondeterministic TM can carry out a computation in n steps, the computation can be carried out by a standard TM in $O(k^{an})$ moves, where k and a are independent of n.

Proof) To simulate the nondeterministic computation, the standard machine would *keep track of all possible configurations*, *searching* and *updating* the entire active area of its tape.

If k is the maximum branching factor for the NT, after n steps there are at most k^n possible configurations, and the length of each configuration is $O(n)$.

Therefore, to simulate one move, the standard machine must search an active area of length $O(nk^n)$.

If a configuration grows, the rest of the tape's content have to be moved. If each k^n configuration grows in a single move, the complete process requires $O(n^3k^{2n})$ moves. Since it's dominated by $O(k^{3n})$, the computation is done by STM in $O(k^{an})$,

Language Families and Complexity Classes

- Language classification: TM vs. Time complexity of a language
- Definition 14.1:

  A TM M *decides* a language L in time T(n)

  if $\forall w \in$ L with |w|=n is decided in T(n) moves.

  If M is nondeterministic, $\forall$w$\in$L, there is *at least one sequence* of moves of length less than or equal to T(|w|) that leads to acceptance, and that TM halts on all inputs in time T(|w|).

- Definition 14.2:

  A language L is said to be a member of the class DTIME(T(n)) if there exists a *deterministic* multitape-TM that decides L in time O(T(n)).

  A language L is said to be a member of the class NTIME(T(n))

  if there exists a *nondeterministic* multitape-TM that decides L in time O(T(n)).

Language Families and Complexity Classes

- Relation b/t the complexity classes, such as

  $\text{DTIME}(T(n)) \subseteq \text{NTIME}(T(n))$ and $T_1(n) = O(T_2(n))$

  implies $\text{DTIME}(T_1(n)) \subseteq \text{DTIME}(T_2(n))$
- An order of $T(n)$ increases, we take more languages progressively.
- Theorem 14.3:

  For every integer $k \geq 1$, $\text{DTIME}(n^k) \subset \text{DTIME}(n^{k+1})$
- Example 14.3:

Every regular language can be recognized by a DFA in time proportional to the length of the input, n. Thus, $L_{REG} \subseteq \text{DTIME}(n)$.

But DTIME (n) includes much more than L_{REG}.

e.g.) In Example 13.7, CFL $\{a^n b^n \mid n \geq 0\}$ can be recognized by a two-tape machine in time $O(n)$.

Language Families and Complexity Classes

- Example 14.4: The non-CFL $L=\{ww \mid w \in \{a,b\}^*\}$ is NTIME(n).

We can recognize strings in L by the following algorithm.

1. Copy the input from the input file to tape 1. Nondeterministically guess the middle of this string.

2. Copy the second part to tape 2.

3. Compare the symbols on tape 1 and tape 2 one by one.

All steps can be done in O(|w|) time, so $L \in$ NTIME(n).

$L \in$ DTIME (n) if we find the middle of a string in O(n) time.

1. We look at each symbol on tape 1, keeping a count on tape 2, but counting only every second symbol. This gives us one half of the length of input string.
2. Use the count on tape 2 to find the starting cell of the 2^{nd} -half of input on tape 1. Copy the 2^{nd} -half of input from its starting cell to the end of input to tape 2.
3. Finally, match tape 1 with tape 2 symbol by symbol.

Since all steps require O(n) moves, a total of time complexity is O(n).

Language Families and Complexity Classes

- Example 14.5:

$L_{CF} \subseteq \text{DTIME}(n^3)$ $\quad$ $L_{CF} \subseteq \text{NTIME}(n)$

Proof) $L_{CF} \subseteq \text{DTIME}(n^3)$ -- $O(n^3)$.

– Please see the proof of membership algorithm for CFG in CYK algorithm (slide of chap.8, book sec.6.3).

Considerations Affecting Complexity Classes and Languages

- As there exists an infinite number of properly nested complexity classes DTIME(n^k), k = 1, 2, ...,

  it's difficult to classify languages by the complexity classes associated with the corresponding TM acceptors.

- Since the particular model of TM used affects the complexity of the associated algorithms, it is difficult to determine which variation to use as the best model of an actual computer.

- The efficiency differences between *deterministic* and *nondeterministic* algorithms can be much more significant than differences between alternative deterministic algorithms involving different numbers of available tapes.

  -- Th^m 14.2

The Complexity Classes P and NP

- There are two famous complexity classes associated with languages: *P* and *NP*
- P is the *set of all languages that are accepted by some Deterministic TM in polynomial time*, without any regard to the degree of the polynomial:

$$P = \bigcup_{i \geq 1} DTIME(n^i)$$

- NP is the *set of all languages that are accepted by some Nondeterministic TM in polynomial time:*

$$NP = \bigcup_{i \geq 1} NTIME(n^i)$$

- P $\subseteq$ NP.

The Relationship Between P and NP

- Obviously, $P \subseteq NP$.
- What is not known is whether P is a proper subset of NP, in other words,

  is $P \subset NP$ or $P = NP$?

- While it is generally believed that there are languages in NP which are not in P, no one has yet found a conclusive example.
- Because of its significance on the feasibility of certain computations, this question remains *the most fundamental unresolved problem in theoretical computer science!!!*

Intractability

- A P problem b/t realistic computation and unrealistic computation.
- A problem is *intractable* if it has such high resource requirements that practical solutions are unrealistic, although the problem may be computable in principle.
- Algorithms for solving intractable problems consume an *extraordinary amount of time* for nontrivial values of n on any computer available now or in the foreseeable future.
- According to the *Cook-Karp Thesis*,

  a problem in P is tractable, and

  a problem not in P is intractable.

Some NP Problems

- The following problems, among others, can be solved *Nondeterministically in Polynomial time (NP)*:
  - The *Satisfiability Problem (SAT∈ NP)*
  - The *Hamiltonian Path Problem (HAMPATH ∈ NP)*:

    Given an *undirected graph* with n vertices, find a *simple path that passes through all the vertices exactly once*. – no cycle.
  - The *Clique Problem (CLIQ ∈ NP)*:

    Given an *undirected graph* with n vertices, find a *subset of k vertices* s.t. there is an *edge between every pair of vertices in the subset.*
- These problems have *deterministic solutions* with *exponential time complexity*, but no deterministic polynomial solution has been found.

Some NP Problems

The Satisfiability (SAT) Problem:

Example 14.6:

- Suppose a CNF expression has length n, with m different literals. Since clearly $m < n$, we can take n as the problem size:

  $i.e.$ n = # of clauses(constraints), m = # of symbols/variables.

- Next, we must encode the CNF expression as a string for a TM. For instance, by taking $\Sigma=\{x,\wedge,\vee,(,),-,0,1\}$ and encoding the subscript i of x_i as a binary number: e.g.) x_3 to x11.

- Since the subscript $i \leq m$, the length of any subscript $\leq \log_2 m$.

- As a consequence, the maximum encoded length of an *n-symbol* CNF is *O(n log n).*: $m < n$ because $n \leq 2^m - 1$.

- Generate a trial solution for the variables -- it can be done in O(n) time, nondeterministically. This trial solution is then substituted into the input string. This can be done in O(n^2log n) time*. The entire process therefore can be done in O(n^2logn) or O(n^3) time, and SAT ∈ NP.

3SAT Problem (Sipser's)

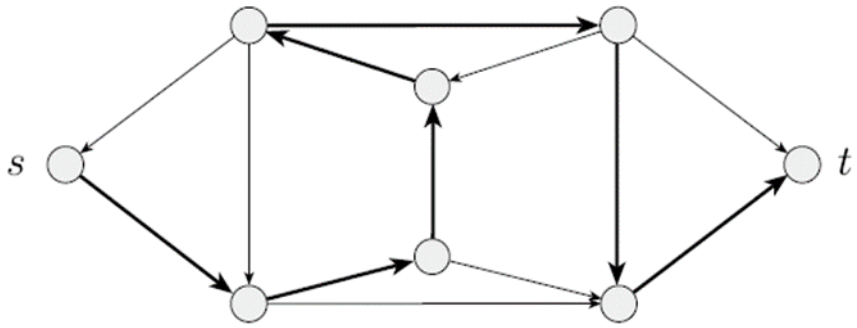
- The special case of the Satisfiability problem where all formulas are in 3-Conjunctive Normal Form (CNF).
  - 3CNF is a Boolean formula in *Conjunctive Normal Form (CNF)* whose clauses consist of at most 3 literals: $\bigwedge_i (\bigvee_j x_j)$
  - A Boolean formula is in *Conjunctive Normal Form (CNF)* if it comprises the *clauses* connected with $\wedge$ (logical and).
  - A *Clause* is a formula of the literals connected with $\vee$ (logical or).
  - A literal is a Boolean variable or a negated Boolean variable: x or $\neg x$.
  - E.g.)

    $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$

- 3SAT = { <ϕ> | ϕ is a satisfiable 3CNF formula. }

Example: Hamiltonian Path Problem (in Sipser's)

- A *Hamiltonian path* in a directed graph G is a path that goes through each node of G exactly once.
- Hamiltonian path problem consists of testing whether a directed graph G contains a Hamiltonian path connecting two specified nodes.
- HAMPATH ={ <G, s, t> |G is a directed graph with a Hamiltonian path from *s* to *t* }
- It is easy to obtain an exponential algorithm to solve HAMPATH, however, nobody knows whether a polynomial one exists.



An NTM deciding HAMPATH

N_1 ="On input string <G, s, t> where G is a directed graph containing nodes *s, t*:

1. Write a list of m numbers $v_1, .., v_m$

   where m is the number of nodes in G.

   Each v_i is nondeterministically selected between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = v_1$ and $t = v_m$. If either fails, *reject*.
4. For each i, $1 \leq i \leq m$, check whether (v_i, v_{i+1}) is an edge of G.

   If any are not, *reject*. Otherwise, *accept*."

- Each stage clearly runs in a nondeterministic polynomial time, thus so does N_1.

Some NP-Problem

- Example 14.8: Clique Problem

Let G be an undirected graph with vertices v_1, v_2, ..., v_n.

A k-clique is a subset $V_k \subseteq 2^V$, such that there is an edge between every pair of vertices $v_i, v_j \in V_k$. The clique problem (CLIQ) is to decide if, for a given *k*, G has a *k*-clique.

A deterministic search can examine all the elements of 2^V.

– doable but exponential time complexity.

A nondeterministic algorithm just guesses the correct subset.

The representation of the graph and the checking are similar to the previous example, so we claim without further elaboration that the clique problem can be solved in $O(n^4)$ time and that CLIQ $\in$ NP.

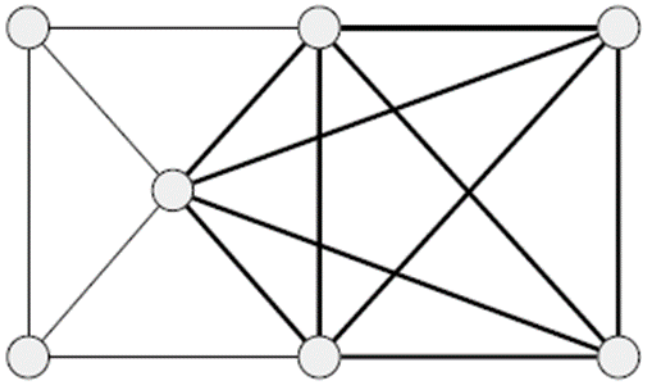
There are many other such problems with the same characteristics.

1. All problems are in NP and have simple nondeterministic solutions.

2. All problems have deterministic solutions with exponential time complexity, but it is not known if they are tractable.

Example in NP: Undirected Graphs and Cliques

- A *clique* in an undirected graph is a *subgraph*
  wherein every two nodes are *connected by an edge*.
- A *k*-clique is a clique containing *k* nodes.
- The clique problem is to determine whether a graph contains a clique of a specified size. Let
  CLIQUE = {<G, *k*> | G is an undirected graph with a *k*-clique}.
- Example: 5-clique



Example: Undirected Graphs and Cliques

- Theorem 7.24: CLIQUE $\in$ NP.

Proof Idea: The clique is the certificate.

- Proof) The following is a verifier V for CLIQUE:

V = "On input <<G, *k*>, *c*>:

1. Test whether *c* is a set of *k* nodes in G.
2. Test whether G contains all edges connecting nodes in *c*.
3. If both pass, *accept*. Otherwise, *reject*."

- Alternative Proof)

Construct a polynomial time NTM N that decides CLIQUE:

N = "On input <G, *k*> where G is an undirected graph:

1. Nondeterministically select a subset C of *k* nodes of G.
2. Test whether all nodes in C are connected
   and whether G contains all edges connecting nodes in C.
3. If yes, *accept*. Otherwise, *reject.*"

Example in NP: *SUBSET-SUM* Problem

- A set of k integers $x_1, .., x_k$ and a target number t are given.
- Determine whether this set contains a subset that adds up to t.

SUBSET-SUM

$= \{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}$, and for some

$C = \{y_1, \ldots, y_l\} \subseteq$ S, we have $\sum y_i = t. \}$

- Example: $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ SUBSET-SUM since 4+21= 25.
- Note: S = $\{x_1, \ldots, x_k\}$ and C = $\{y_1, \ldots, y_l\}$ may be multisets;
  i.e. elements may be repeated.
- Cf) Subarray-Sum problem
- Theorem 7.25: SUBSET-SUM $\in$ NP.

Example: *SUBSET-SUM* Problem

- Theorem 7.25: SUBSET-SUM $\in$ NP.

Proof Idea: The subset is the certificate.

- Proof) The following is a verifier V for SUBSET-SUM:

  V = "On input <<*S*, *t*>, *c*>:

  1. Test whether c is a collection of numbers that sum to *t*.
  2. Test whether S contains all the numbers in *C*.
  3. If both pass, *accept*. Otherwise, *reject*."

- Alternative Proof) Give the NTM N that decides SUBSET-SUM:

  N = "On input <*S*, *t*>:

  1. Nondeterministically select a subset C of numbers in *S*.
  2. Test whether C is a collection of numbers that sum to *t*.
  3. If the test passes, *accept*. Otherwise, *reject*."

Polynomial-Time Reduction

- Since NP problems have similar characteristics, it is convenient to determine if they can be reduced to each other.
- Definition 14.3: A language L_1 is *polynomial-time reducible* to another language L_2 if there exists a *Deterministic Turing Machine* that can *transform* any string $w_1 \in L_1$ to $w_2 \in L_2$ so that
  - The *transformation* can be completed *in polynomial time*, and
  - w_1 is in L_1 if and only if w_2 is in L_2.
  - Notation: $L_1 \leq_P L_2$

- i.e. If L_1 and L_2 are languages over Σ_1 and Σ_2, a *polynomial-time reduction* from L_1 to L_2 is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ satisfying two conditions
  1. $\forall w \in \Sigma_1^*, \quad w \in L_1$ if and only if $f(w) \in L_2$
  2. f can be computed in polynomial time
     i.e., there is a DTM with polynomial time complexity that computes f.

Polynomial-Time Reduction (cont.)

- Polynomial-time Reducibility is Transitive:
  - If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$
- So, If $L_1 \leq_P L_2$ and $L_2 \leq_P P$, then $L_1 \leq_P P$

- As we show that a language is decidable by reducing it to another decidable language, we can show that a language is in P by reducing it to another P problem:
  - In the case of decidability, we only needed the reduction to be computable.
  - Here, we need the reduction function to be computable in *polynomial time*.

- Consider 3SAT, a modified version of the SAT problem in which each clause can have at most 3 literals;
  - The SAT problem is polynomial-time reducible to 3SAT.
  - The 3SAT problem is polynomial-time reducible to CLIQUE.

NP-Completeness

- Some problems have been identified as being as complex as any other problem in NP.
- Definition 14.4: A language (or problem) *L* is *NP-Complete* if
  - $L \in$ NP, and
  - For all $L_1 \in$ NP is *polynomial-time reducible to L*: $L1 \leq_P L$
- This definition is very significant because,

  if a Deterministic Polynomial-Time algorithm is found for any NP-complete problem, then every language in NP is also in P.

  (because it can be polynomial-time reducible to another language in NP)
- *If so, P = NP!!*

  i.e. If $L \in$ *NP-complete* and $L \in P$, then *P = NP!*
- Theorem 14.5: The Satisfiability Problem is NP-complete.

NP-Completeness

- Theorem 14.5: The SAT Problem is NP-complete.
- Proof idea: For every configuration sequence of a TM, one can construct a CNF expression that is satisfiable

  iff there is a sequence of configurations leading to acceptance.

  The details are omitted. – The topics of theory of complexity.

- Proving NP-Completeness by Reduction: in Goddard's

  If S is NP-complete, $T \in NP$ and $S \leq_P T$, then T is NP-complete.

- Consider 3SAT, a modified version of the SAT problem in which each clause can have at most 3 literals;
  - $SAT \leq_P 3SAT$.
  - $3SAT \leq_P CLIQUE$.

Cook-Levin Theorem: SAT is NP-Complete.

- SAT is fundamental.

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof Idea)

- Show that SAT $\in$ NP -- done. (slide#18 -- cf) $O(2^n)$ in DTM)
- Show that any language A $\in$ NP is polynomial time reducible to SAT.
- The reduction of A takes an NTM N and a string *w* and produces a Boolean formula ϕ that simulates the NTM N that decides A operating on w.
- If N accepts, ϕ has a satisfying assignment that corresponds to that computation; if N doesn't accept, no assignment satisfies ϕ.
- Therefore, $w \in$ A iff ϕ is satisfiable.

- For actual proof, see the slides #43 – at the end of this chapter.

3SAT Problem (from Sipser's)

- Idea of Reduction from SAT to 3SAT.
  - Let $f(w)$ involve the variables in w as well as new ones
  - The trick is to incorporate the new variables so that
    - For every satisfying truth assignment to the variables of w, some assignment to the new variables makes $f(w)$ true
    - For every non-satisfying assignment to the variables of w, no assignment to the new variables makes $f(w)$ true

3SAT Problem (from Sipser's)

- The special case of the Satisfiability problem where all formulas are in 3-Conjunctive Normal Form (CNF).
  - 3CNF is a Boolean formula in *Conjunctive Normal Form (CNF)* whose clauses consist of at most 3 literals: $\bigwedge_i (\bigvee_j x_j)$
  - A Boolean formula is in *Conjunctive Normal Form (CNF)* if it comprises the *clauses* connected with $\wedge$ (logical and).
  - A *Clause* is a formula of the literals connected with $\vee$ (logical or).
  - A literal is a Boolean variable or a negated Boolean variable: x or $\neg x$.
  - E.g.)

    $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$

- 3SAT = { <ϕ> | ϕ is a satisfiable 3CNF formula. }

3SAT Problem (Sipser's)

- Theorem 7.32: 3SAT is polynomial time reducible to CLIQUE,

where CLIQUE = {<G, *k*>| G is an undirected graph with *k*-clique}.

Proof Idea)

- The polynomial time reduction *f*: 3SAT $\rightarrow$ CLIQUE, (3SAT $\leq_p$ CLIQUE) converts formulas to graphs.
- In the constructed graphs, cliques of a specified size, *k*, correspond to satisfying assignments of the formula.
- Structures within the graph are designed to mimic the behavior of the variables and clauses.

Proof) Let ϕ be a 3CNF formula with *k*-clauses:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_{\backslash k} \vee c_k).$$

The reduction *f* generates the string ⟨G, *k*⟩, where G is an undirected graph as follows:

3SAT Problem (cont.)

- Theorem 7.32: 3SAT is polynomial time reducible to CLIQUE

Proof, cont.) The reduction *f* generates the string $\langle G, k \rangle$:

Nodes of G:

- The nodes in G are organized in *k*-groups of 3 nodes, called the triples, $t_1, \ldots, t_k$.
- Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause.
- Label each node of G with its corresponding literal in ϕ.

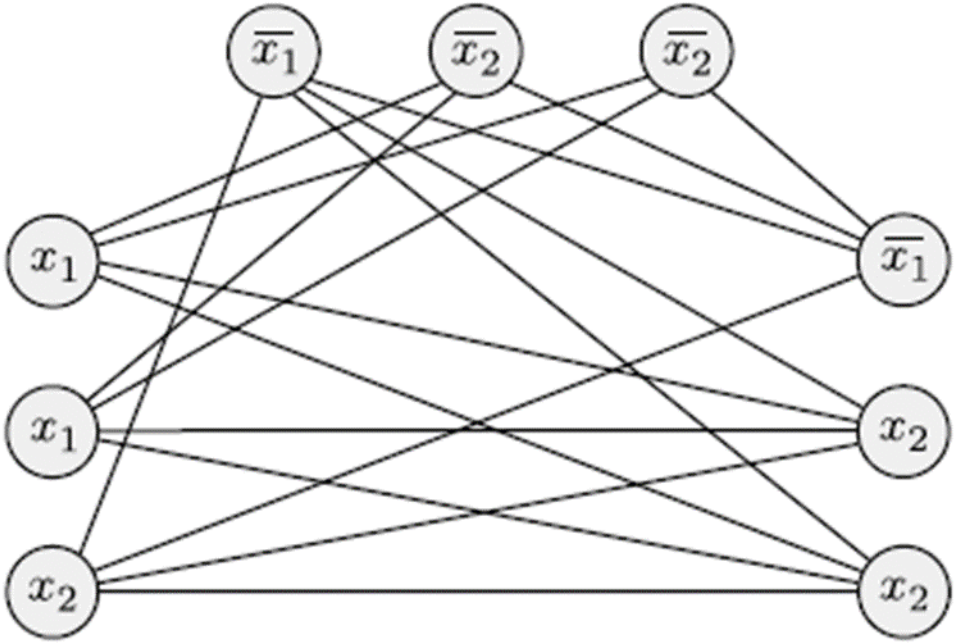
Edges of G:

- Edges of G *connect all* but two types of pairs of nodes in G.
  - No edge between nodes in the same triple.
  - No edge between two nodes with contradictory labels, such as in x_i and $\neg x_i$.

From 3SAT to CLIQUE (cont.)

- 3SAT formula ϕ in the transformed in the graph.

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2).$$



3SAT $\rightarrow$ CLIQUE (cont.)

ϕ is satisfiable iff G has a *k*-CLIQUE.

Proof) $\rightarrow$) Suppose that ϕ has a satisfying assignment.

- At least one literal is true in every clause (required by $\vee$)
- The nodes of G are grouped into triples.

  In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more literals are true in some clause, we select the true literal arbitrarily.
- The nodes just selected form a k-clique; number of nodes is k (i.e. k clauses in ϕ) and each pair of selected nodes are connected by the construction of G.
- Selected node are not from the same triplet, by construction;

  They could not have contradictory labels because otherwise the associated labels would be both true in the satisfying assignment.

Hence G contains a k-clique.

CLIQUE → 3SAT

ϕ is satisfiable iff G has a *k*-CLIQUE.

←) *k*-clique in G implies ϕ is satisfiable. Suppose G has a *k*-clique.

- No two of the clique nodes can occur in the same triplet because nodes in the same triplet are not connected. So, each of the *k*-triplets contains one of the *k*-clique nodes.
- Assign truth value to the variables of ϕ so that each literal labeling a clique node is made true. This is possible because two nodes with contradictory labels are not connected.
- This assignment satisfies the formula ϕ. Since each node corresponds to a clause that has a true value in it the clause is true; since clauses are connected by $\wedge$, the formula ϕ is true.

Therefore:

- If *CLIQUE* is solvable in polynomial time, so is *3SAT*. Q.E.D.

NP-Hardness (from Sipser's)

- Definition Q7.34: A language B is NP-hard if every A $\in$NP is polynomial time reducible to B:

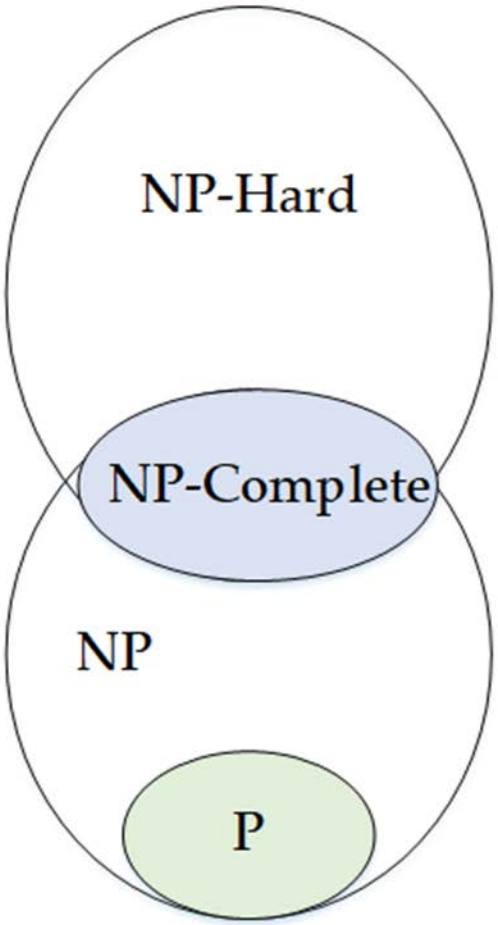
  $\forall A \in NP,\ A \leq_P B \Rightarrow$ B is NP-hard.

  even though B may not be in NP itself.

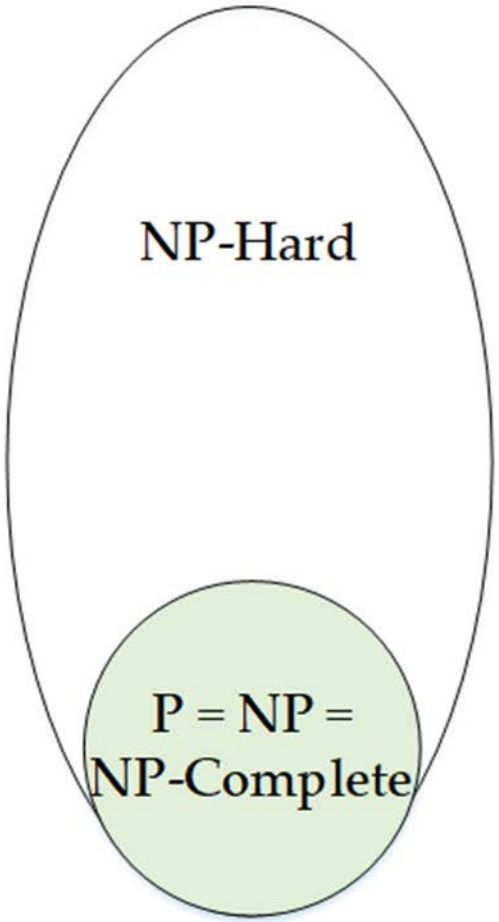
  - At least as hard as the hardest problems in NP.
- Cf.) Definition 7.34: A language B is *NP-complete*

  if $B \in NP$ and $\forall A \in NP,\ A \leq_P B$
- In Church-Turing Thesis, the language

  D = { $\langle p \rangle$ | *p* is a polynomial in several variables having an integral root} is undecidable.
- D is NP-hard. -- So, you must show that all problems in NP are polynomial time reducible to D.
- Lemma: If A $\in$ NP and A is NP-hard, then A is NP-complete.



If P ≠ NP



If P = NP

Exmple: NP-Complete Problems

- SAT, 3SAT, HAMPATH, CLIQUE, SUBSET-SUM Problems
- VERTEX-COVER Problem:
  - In an undirected graph G, is G contains a vertex cover of a specified size k?

    where a vertex cover of G is a subset of the nodes where every edge of G touches one of those nodes.
- PARTITION Problem
- BIN-PACKING Problem:
  - The decision problem of packing the items of different volumes into a finite number of bins – NP-complete problem.
  - The optimization problem of packing the items of different volumes into a finite number of bins in a way of minimizing the number of bins – NP-hard.
- 0-1 KNAPSACK Problem:
  - Combinatorial optimization problem of deciding the number of items of different weights and their values in a way of maximizing the total value.

    – NP-Hard problem
  - The decision problem of deciding the items within the total value – NP-Complete.

An Open Question: P = NP?

- Computer scientists continue to look for an *efficient (deterministic, polynomial-time) algorithm* that can be applied to all NP problems, therefore concluding that P = NP.
- On the other hand, if a proof is found that *any of the NP-complete problems* is *intractable*, then we can conclude that $P \subset NP$ and that many interesting problems are not practically solvable.
- In spite of our best efforts, *no efficient algorithm* has been found for any NP-complete problem, so our conjecture is that $P \neq NP$.
- However, until a proof is found,

  P = NP? remains the fundamental open question in complexity theory!

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof)

1. SAT $\in$ NP: A polynomial time NTM can guess an assignment to the variables of a given formula ϕ and accept if the assignment satisfies ϕ.
2. Let A $\in$ NP: Show that A is polynomial time reducible to SAT.

   For an NTM N that decides A in O(n^k) time for some constant k,

   construct a formula ϕ that simulates N.

   The construction of ϕ is based on organizing the computation

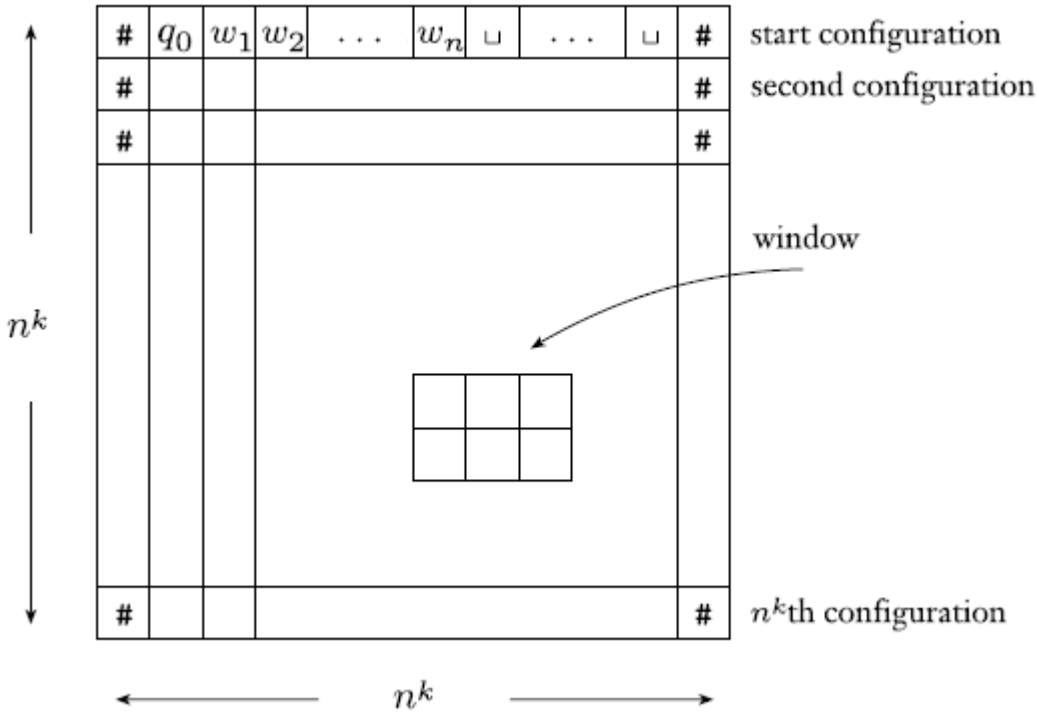
   performed by N into an $n^k \times n^k$ tableau.

   - Rows are the configurations of a branch of the computation of N on input *w*.
   - Each configuration starts and ends with #.
   - A tableau is *accepting* if any of its rows is an accepting configuration.

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Configurations Tableau of N



Observations:

(1) Each configuration starts and ends with a # symbol.

(2) A tableau is accepting if any of its rows is an accepting configuration.

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Accepting Tableau(*N*, *w*):

- Every accepting tableau for *N* and *w* correspond to an accepting computation branch of *N* on *w*.
- Problem of determining whether *N* accepts *w* is equivalent to the problem of determining whether an accepting tableau for *N* on *w* exists.

Polynomial time $f : A \rightarrow SAT$

- On input *w*, *f* produces ϕ_w.
- Variables of ϕ_w: Let N = (Q, Σ, Γ, δ, q_0, q_a, q_r), and C = Q $\cup$ Γ $\cup$ {#}.

  For each $1 \leq i, j \leq n^k \wedge s \in C$ we have a variable $x_{i,j,s}$ in ϕ_w.
- Cells: Each of the $(n^k)^2$ entries of a tableau is called a cell.

  For every state $\forall s \in C$, $x_{i,j,s} = 1$ if cell[*i*, *j*] = s.
- Formula $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$ corresponds to Tableau(*N*, *w*).

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Assignment-Tableau(N, w) Correspondence:

- First, ensure that the tableau contains exactly one symbol per cell.
- Thus, ϕ_{cell} must guarantee that exactly one variable is true for each cell:
  1. at least one variable that is associated with a cell is true, by:

$$\bigvee_{s \in C} x_{i,j,s}$$

  2. no more than one variable is true for each cell, i.e. for each pair of variables at least one is false, which is represented by

$$\bigwedge_{s,t \in C, s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t})$$

- A satisfying assignment which makes true one variable for each cell is specified by

$$\phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} [(\bigvee_{s \in C} x_{i,j,s}) \wedge (\bigwedge_{s,t \in C, s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}))]$$

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Tableau(*N, w*) must be an Accepting tableau:

An accepting tableau is specified by ϕ_{start}, ϕ_{move}, ϕ_{accept} :

- ϕ_{start} ensures that the 1st row of the tableau is the starting configuration of N on w by the equality:

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$$

- ϕ_{accept} guarantees that an accepting configuration occurs in the tableau: i.e. q_a must be in one of the cells:

$$\phi_{accept} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_a}$$

- ϕ_{move} guarantees that each row of the tableau correspond to a configuration that legally follows the preceding row's configuration according the N's transition rules.

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Legal Windows:

- A 2×3 window of cells is *legal* if that window does not violate the actions specified by N's transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.
- Consider the transitions:

$$\delta(q_1, a) = \{ (q_1, b, R)\}, \ \delta(q_1, b) = \{ (q_2, c, L), (q_2, a, R) \}.$$

- Examples of legal windows for this machine:

(a) $\delta(q_1, b) = (q_2, c, L)$

a	q_1	b
q_2	a	c

(b) $\delta(q_1, b) = (q_2, a, R)$

a	q_1	b
a	a	q_2

(c) $\delta(q_1, a) = (q_1, b, R)$

a	a	q_1	a
a	a	b	q_1

(d) a head wasn't in the adjacent location of the window.

#	b	a
#	b	a

(e) $\delta(q_1, b) = \{(q_2, c, L), \}$

a	b	a	q_1	b
a	b	q_2	c	w_i

(f) $\delta(q_1, b) = \{(q_2, c, L), \}$

w_i	q_1	b	b	b
q_2	w_i	c	b	b

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Legal Windows:

- Windows (a)+(b): legal because the transition allows N to move this way.
- (c): legal because with q_1 appearing on the right side of the top row, we don't know what symbol the head is over.
- (d): legal because top and bottom are identical, which could happen if the head weren't adjacent to the location of the window.
- (e): legal because state q_1 reading a b might have been immediately to the right of the top row and would have moved to the left.
- (f): legal because state q_1 might have been immediately to the left of the top row changing b to c and moving left.

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Illegal Windows:

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

- (a): Illegal because the central symbol cannot be changed without an adjacent state.
- (b): Illegal because the transition function states that

  b gets changed to *c*, not to *a*.
- (c): Illegal because two states appear in the bottom row.

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Construction of ϕ_{move}:

Claim 7.41: If the top row of the tableau is the start configuration and every window is legal then each row is a configuration that legally follows the configuration represented by the preceding row.

- Each window contains six cells, which may be set in a fixed number of ways to yield a legal window, according to N's transition function
- ϕ_{move} says that the setting of these six cells is legal by

$$\phi_{move} = \bigwedge_{1 \le i < n^k 1 < j < n^k} (window[i,j] \text{ is legal})$$

  in which a window[*i,j*] has a cell[*i,j*] as the upper central position.

- *window[i,j] is legal* $\Leftrightarrow$

$$\bigvee_{a_1,\ldots,a_6} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

  where a_1, a_2, .., a_6 are the contents of the six cells.

The Cook-Levin Theorem: SAT is NP-Complete

Theorem 7.37 (Cook-Levin Theorem): SAT is NP-complete.

Proof: cont.) Complexity of the Reduction:

- Tableau is $n^k \times n^k$ and thus contains n^{2k} cells; each has |C|= l variables associated with it where l depends on N. Hence, the total number of variables O(n^{2k}).
- Estimating the size of the four components of ϕ_w:
  - ϕ_{cell} contains a fixed fragment of ϕ_w for each cell of the tableau, so its size is O(n^{2k})
  - ϕ_{start} has a fixed fragment for each cell in the top row, so has size O(n^k).
  - ϕ_{move} and ϕ_{accept} contain a fixed fragment for each cell of the tableau, so their size is O(n^{2k}).
  - Therefore the total size of ϕ_w is polynomial in n.
- Each component of ϕ_w can be produced in polynomial time.
- Thus, we can construct a polynomial time reduction from *N* into ϕ_w .
- Therefore, it concludes the proof of Cook-Levin Theorem. Q.E.D.

The Cook-Levin Theorem: SAT is NP-Complete

The size of the four components of ϕ_w: $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$

- ϕ_{cell} contains a fixed fragment of ϕ_w for each cell: $O(n^{2k})$

$$\phi_{cell} = \wedge_{1 \le i,j \le n^k}[(\vee_{s \in C} x_{i,j,s}) \wedge (\wedge_{s,t \in C, s \ne t}(\neg x_{i,j,s} \vee \neg x_{i,j,t}))]$$

- ϕ_{start} has a fixed fragment for each cell in the top row: $O(n^k)$.

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$$

- ϕ_{move} and ϕ_{accept} contain a fixed fragment for each cell: $O(n^{2k})$.

$$\phi_{move} = \bigwedge_{1 \le i < n^k 1 < j < n^k} (window[i,j] \text{ is legal})$$

window[i,j] is legal $\Leftrightarrow$

$$\bigvee_{a_1,\ldots,a_6} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

$$\phi_{accept} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_a}$$

Thus, the total size of ϕ_w is polynomial in n: $O(n^{k1})$.

The Cook-Levin Theorem: SAT is NP-Complete

Conclusion:

- SAT is an NP-complete problem since
  - it is NP and
  - we can polynomially reduce any arbitrary NP problem into it.
- Showing that other problems are NP-complete:
  - generally much simpler than proving the above theorem,
  - Show a polynomial reduction to known NP-complete problem, such as SAT.
- SAT was the first problem to be proven to be NP-complete.
- All problems in NP are at most as difficult as SAT.
- Providing a polynomial algorithm to SAT solves the P versus NP dilemma.
- Thus, SAT is fundamental problem to NP-Complete!