CSE408
Vertex Cover and Bin Packing

Lecture # 37

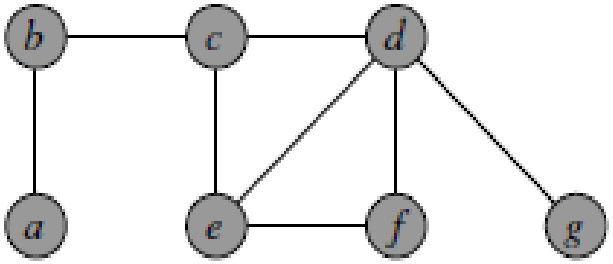
The vertex-cover problem

A *vertex cover* of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

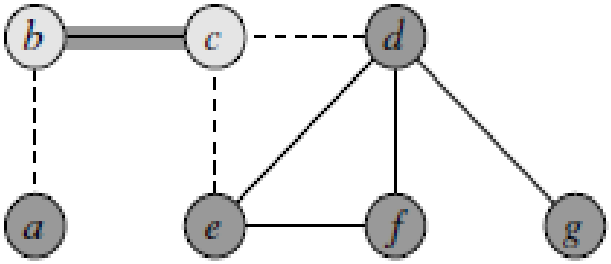
APPROX-VERTEX-COVER(G)

```
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4        do let (u, v) be an arbitrary edge of E'
5           C ← C ∪ {u, v}
6           remove from E' every edge incident on either u or v
7  return C
```

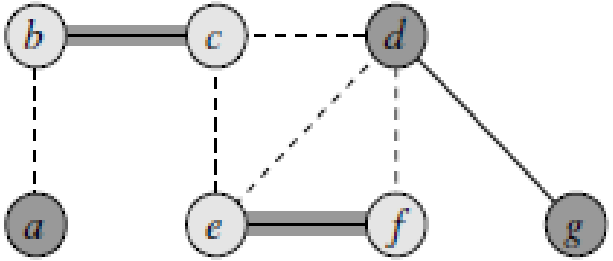
Example



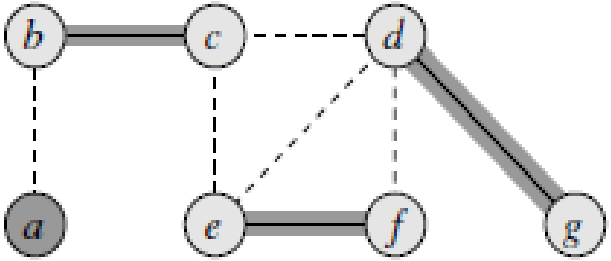
(a)



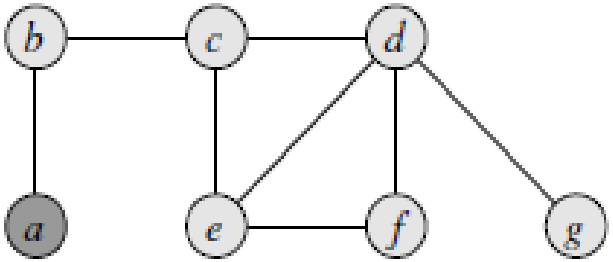
(b)



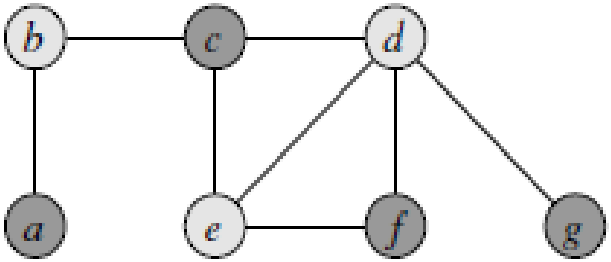
(c)



(d)



(e)



(f)

The set cover problem

An instance $(X, \mathcal{F})$ of the *set-covering problem* consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

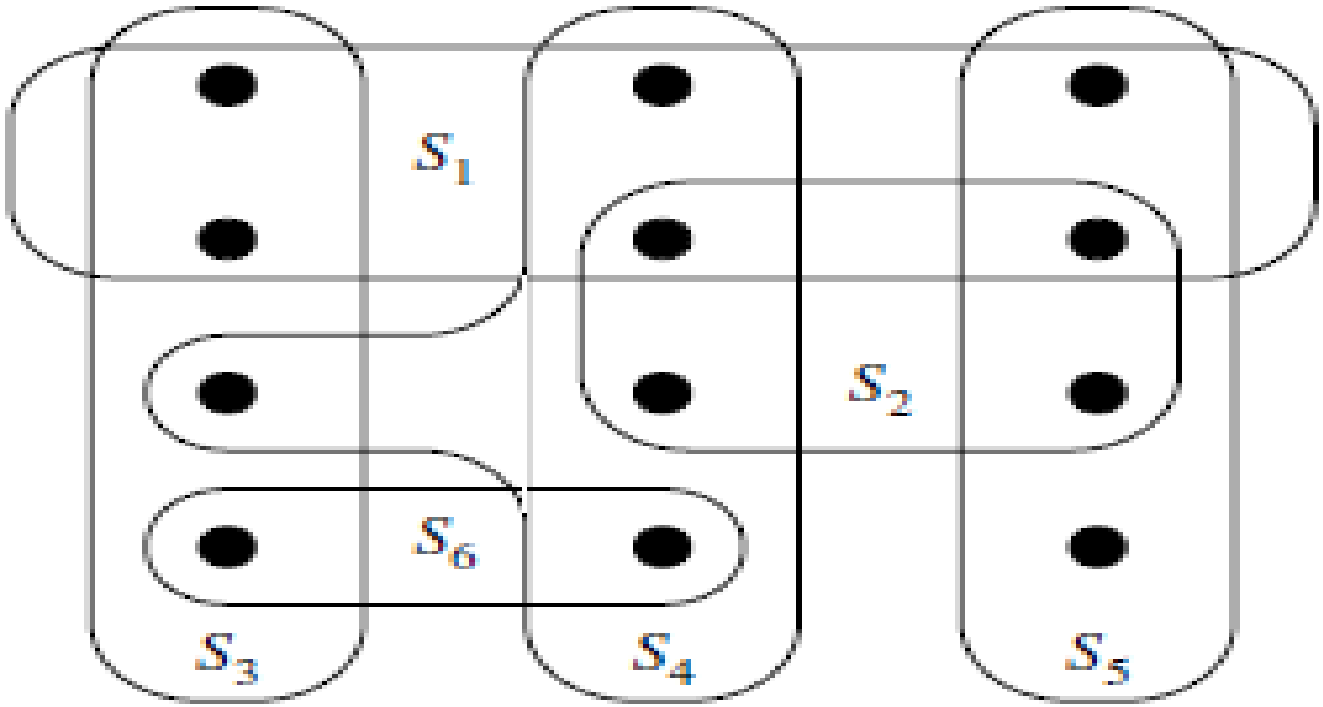


Figure 35.3 An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$. The greedy algorithm produces a cover of size 4 by selecting the sets S_1, S_4, S_5, and S_3 in order.

A greedy approximation algorithm

The greedy method works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered.

Set cover algo

GREEDY-SET-COVER$(X, \mathcal{F})$

1 $U \leftarrow X$
2 $\mathcal{C} \leftarrow \emptyset$
3 while $U \neq \emptyset$
4 do select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5 $U \leftarrow U - S$
6 $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
7 return $\mathcal{C}$

Bin Packing

- In the **bin packing problem**, objects of different volumes must be packed into a finite number of bins or containers each of volume *V* in a way that minimizes the number of bins used. In computational complexity theory, it is a combinatorial NP-hard problem.
- There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and so on.
- They have many applications, such as filling up containers, loading trucks with weight capacity constraints, creating file backups in removable media and technology mapping in Field-programmable gate array semiconductor chip design.

- The bin packing problem can also be seen as a special case of the cutting stock problem.
- When the number of bins is restricted to 1 and each item is characterised by both a volume and a value, the problem of maximising the value of items that can fit in the bin is known as the knapsack problem.
- Despite the fact that the bin packing problem has an NP-hard computational complexity, optimal solutions to very large instances of the problem can be produced with sophisticated algorithms.
- In addition, many heuristics have been developed: for example, the **first fit algorithm** provides a fast but often non-optimal solution, involving placing each item into the first bin in which it will fit.
- It requires $\Theta(n \log n)$ time, where n is the number of elements to be packed.

Bin packing

- The algorithm can be made much more effective by first sorting the list of elements into decreasing order (sometimes known as the first-fit decreasing algorithm), although this still does not guarantee an optimal solution, and for longer lists may increase the running time of the algorithm.
- It is known, however, that there always exists at least one ordering of items that allows first-fit to produce an optimal solution.[1]
- An interesting variant of bin packing that occurs in practice is when items can share space when packed into a bin.
- Specifically, a set of items could occupy less space when packed together than the sum of their individual sizes.

- This variant is known as VM packing[2] since when virtual machines (VMs) are packed in a server, their total memory requirement could decrease due to pages shared by the VMs that need only be stored once.
- If items can share space in arbitrary ways, the bin packing problem is hard to even approximate.
- However, if the space sharing fits into a hierarchy, as is the case with memory sharing in virtual machines, the bin packing problem can be efficiently approximated.

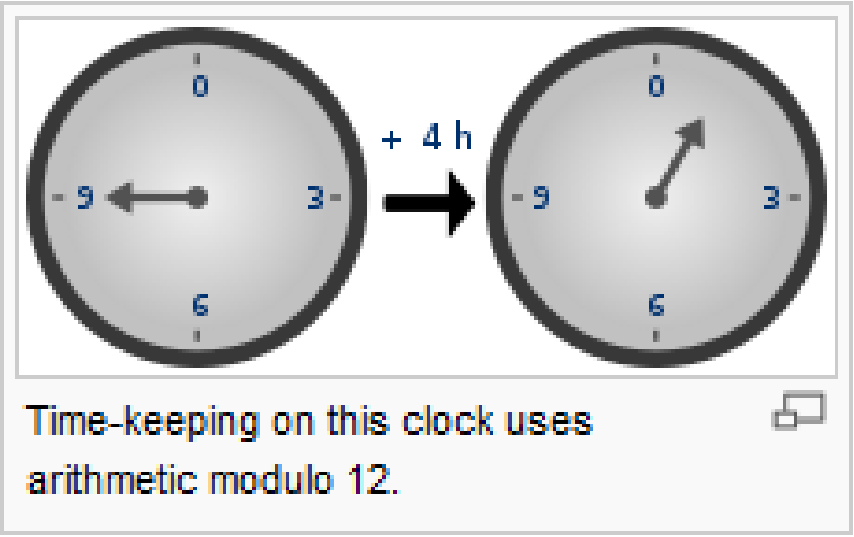
Thank You !!!

CSE408
Modular Arithmetic & Chinese Remainder Theorem

Lecture # 38

Modular Arthmetic

- In mathematics, **modular arithmetic** (sometimes called **clock arithmetic**) is a system of arithmetic for integers, where numbers "wrap around" upon reaching a certain value—the **modulus**.



Time-keeping on this clock uses arithmetic modulo 12.

Modular Arthmetic

- Modular arithmetic can be handled mathematically by introducing a congruence relation on the integers that is compatible with the operations of the ring of integers: addition, subtraction, and multiplication. For a positive integer *n*, two integers *a* and *b* are said to be **congruent modulo** *n*, written:

$$a \equiv b \pmod{n},$$

- if their difference *a* − *b* is an integer multiple of *n* (or *n* divides *a* − *b*). The number *n* is called the **modulus** of the congruence.

- The properties that make this relation a congruence relation (respecting addition, subtraction, and multiplication) are the follow $a_1 \equiv b_1 \pmod{n}$
- If $a_2 \equiv b_2 \pmod{n}$,
- And
  - $a_1 + a_2 \equiv b_1 + b_2 \pmod{n}$
- then:
  - $a_1 - a_2 \equiv b_1 - b_2 \pmod{n}$

- It should be noted that the above two properties would still hold if the theory were expanded to include all real numbers, that is if were not necessarily all integers. The next property, however, would fail if these variables were
  - $a_1 a_2 \equiv b_1 b_2 \pmod{n}$.

Chinese Theorem

- The **Chinese remainder theorem** is a result about congruences in number theory and its generalizations in abstract algebra. It was first published in the 3rd to 5th centuries by Chinese mathematician Sun Tzu.
- In its basic form, the Chinese remainder theorem will determine a number *n* that when divided by some given divisors leaves given remainders.
- For example, what is the lowest number *n* that when divided by 3 leaves a remainder of 2, when divided by 5 leaves a remainder of 3, and when divided by 7 leaves a remainder of 2?
- A common introductory example is a woman who tells a policeman that she lost her basket of eggs, and that if she makes three portions at a time out of it, she was left with 2, if she makes five portions at a time out of it, she was left with 3, and if she makes seven portions at a time out of it, she was left with 2.
- She then asks the policeman what is the minimum number of eggs she must have had. The answer to both problems is 23.

- Suppose $n_1, n_2, \ldots, n_k$ are positive integers that are pairwise coprime. Then, for any given sequence of integers $a_1, a_2, \ldots, a_k$, there exists an integer x solving the following system of simultaneous congruences.

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

Furthermore, all solutions x of this system are congruent modulo the product, $N = n_1 n_2 \ldots n_k$.

Hence $x \equiv y \pmod{n_i}$ for all $1 \leq i \leq k$, if and only if $x \equiv y \pmod{N}$.

Example

Sometimes, the simultaneous congruences can be solved even if the n_i's are not pairwise coprime. A solution x exists if and only if:

$$a_i \equiv a_j \pmod{\gcd(n_i, n_j)} \qquad \text{for all } i \text{ and } j$$

All solutions x are then congruent modulo the least common multiple of the n_i.

Brute Force Technique:-

For example, consider the problem of finding an integer x such that

$$x \equiv 2 \pmod 3$$
$$x \equiv 3 \pmod 4$$
$$x \equiv 1 \pmod 5$$

A brute-force approach converts these congruences into sets and writes the elements out to the product of 3×4×5 = 60 (the solutions modulo 60 for each congruence):

x ∈ {2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59, ...}

x ∈ {3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, ...}

x ∈ {1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, ...}

To find an x that satisfies all three congruences, intersect the three sets to get:

x ∈ {11, ...}

Which can be expressed as

$$x \equiv 11 \pmod{60}$$

Theorem 31.27 (Chinese remainder theorem)
Let $n = n_1 n_2 \cdots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) , \tag{31.23}$$

where $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$, and

$$a_i = a \bmod n_i$$

for $i = 1, 2, \ldots, k$. Then, mapping (31.23) is a one-to-one correspondence (bijection) between $\mathbf{Z}_n$ and the Cartesian product $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$. Operations performed on the elements of $\mathbf{Z}_n$ can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$\begin{aligned} a &\leftrightarrow (a_1, a_2, \ldots, a_k) , \\ b &\leftrightarrow (b_1, b_2, \ldots, b_k) , \end{aligned}$$

then

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k) , \tag{31.24}$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k) , \tag{31.25}$$

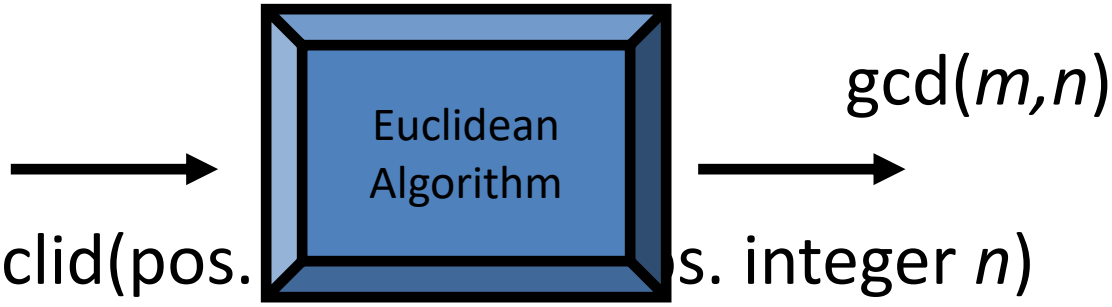
$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k) . \tag{31.26}$$

Thank You !!!

CSE408
GCD and Optimization Problem

Lecture # 39

Euclidean Algorithm



integer euclid(pos. s. integer *n*)

```
    x = m, y = n
    while(y > 0)
        r = x  mod y
        x = y
        y = r
    return x
```

Euclidean Algorithm. Example

gcd(33,77):

Step	$r = x \bmod y$	x	y
0	–	33	77

Euclidean Algorithm. Example

gcd(33,77):

Step	$r = x$ **mod** y	x	y
0	–	33	77
1	33 **mod** 77 = 33	77	33

Euclidean Algorithm. Example

gcd(33,77):

Step	$r = x$ **mod** y	x	y
0	–	33	77
1	33 **mod** 77 = 33	77	33
2	77 **mod** 33 = 11	33	11

Euclidean Algorithm. Example

gcd(33,77):

Step	$r = x$ **mod** y	x	y
0	–	33	77
1	33 **mod** 77 = 33	77	33
2	77 **mod** 33 = 11	33	11
3	33 **mod** 11 = 0	11	0

Euclidean Algorithm. Example

gcd(244,117):

Step	$r = x \bmod y$	x	y
0	–	244	117

Euclidean Algorithm. Example

gcd(244,117):

Step	$r = x$ **mod** y	x	y
0	–	244	117
1	244 **mod** 117 = 10	117	10

Euclidean Algorithm. Example

gcd(244,117):

Step	$r = x$ **mod** y	x	y
0	–	244	117
1	244 **mod** 117 = 10	117	10
2	117 **mod** 10 = 7	10	7

Euclidean Algorithm. Example

gcd(244,117):

Step	$r = x$ **mod** y	x	y
0	–	244	117
1	244 **mod** 117 = 10	117	10
2	117 **mod** 10 = 7	10	7
3	10 **mod** 7 = 3	7	3

Euclidean Algorithm. Example

gcd(244,117):

Step	$r = x$ **mod** y	x	y
0	–	244	117
1	244 **mod** 117 = 10	117	10
2	117 **mod** 10 = 7	10	7
3	10 **mod** 7 = 3	7	3
4	7 **mod** 3 = 1	3	1

Euclidean Algorithm. Example

gcd(244,117):

Step	$r = x$ **mod** y	x	y
0	–	244	117
1	244 **mod** 117 = 10	117	10
2	117 **mod** 10 = 7	10	7
3	10 **mod** 7 = 3	7	3
4	7 **mod** 3 = 1	3	1
5	3 **mod** 1=0	1	0

By definition ➔ 244 and 117 are rel. prime.

Euclidean Algorithm Correctness

The reason that Euclidean algorithm works is gcd(*x*,*y*) is not changed from line to line. If *x'*, *y'* denote the next values of *x* , *y* then:

gcd(*x'*,*y'*) = gcd(*y*, *x* **mod** *y*)

= gcd(*y*, *x* + *qy*) (the useful fact)

= gcd(*y*, *x*) (subtract *y* -multiple)

= gcd(*x*,*y*)

Optimization Problem

- In mathematics and computer science, an **optimization problem** is the problem of finding the *best* solution from all feasible solutions. Optimization problems can be divided into two categories depending on whether the variables are continuous or discrete.
- An optimization problem with discrete variables is known as a **combinatorial optimization problem**. In a combinatorial optimization problem, we are looking for an object such as an integer, permutation or graph from a finite (or possibly countable infinite) set.

Thank You !!!

CSE408
Complexity Classes

Lecture # 40

NP-Completeness

- Poly time algorithm: input size n (in some encoding), worst case running time – $O(n^c)$ for some constant c.
- Three classes of problems
  - P: problems solvable in poly time.
  - NP: problems verifiable in poly time.
  - NPC: problems in NP and as hard as any problem in NP.

NP-Completeness (verifiable)

- Verifiable in poly time: given a certificate of a solution, could verify the certificate is correct in poly time.
- Examples (their definitions come later):
  - Hamiltonian-cycle, given a certificate of a sequence $(v_1, v_2, \ldots, v_n)$, easily verified in poly time.
  - 3-CNF, given a certificate of an assignment 0s, 1s, easily verified in poly time.
  - (so try each instance, and verify it, but 2^n instances)
- Why not defined as “solvable in exponential time?” or “Non Poly time”?

NP-Completeness (why NPC?)

- A problem $p \in$ NP, and any other problem $p' \in$ NP can be translated as p in poly time.
- So if p can be solved in poly time, then all problems in NP can be solved in poly time.
- All current known NP hard problems have been proved to be NPC.

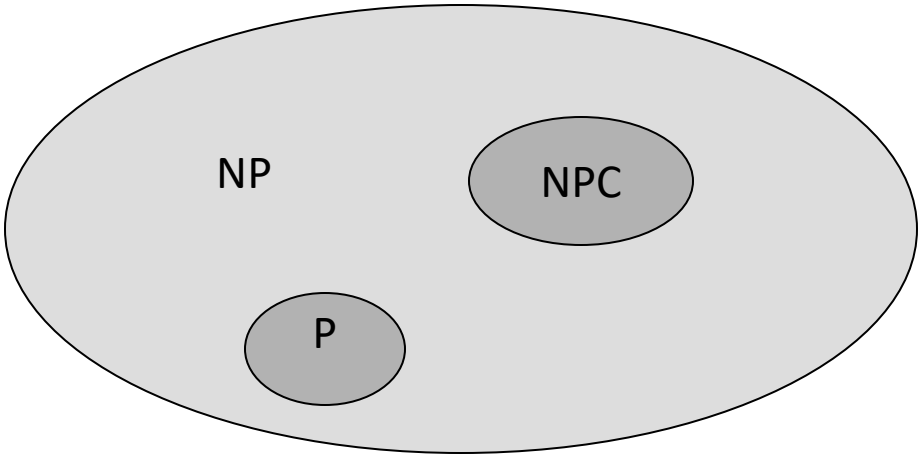
Relation among P, NP, NPC

- $P \subseteq NP$ (Sure)
- $NPC \subseteq NP$ (sure)
- $P = NP$ (or $P \subset NP$, or $P \neq NP$) ???
- $NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???
- $P \neq NP$: one of the deepest, most perplexing open research problems in (theoretical) computer science since 1971.

Arguments about P, NP, NPC

- No poly algorithm found for any NPC problem (even so many NPC problems)
- No proof that a poly algorithm cannot exist for any of NPC problems, (even having tried so long so hard).
- Most theoretical computer scientists believe that NPC is intractable (i.e., hard, and $P \neq NP$).

View of Theoretical Computer Scientists on P, NP, NPC



$P \subset NP$, $NPC \subset NP$, $P \cap NPC = \varnothing$

Why discussion on NPC

- If a problem is proved to be NPC, a good evidence for its intractability (hardness).
- Not waste time on trying to find efficient algorithm for it
- Instead, focus on design approximate algorithm or a solution for a special case of the problem
- Some problems looks very easy on the surface, but in fact, is hard (NPC).

Decision VS. Optimization Problems

- Decision problem: solving the problem by giving an answer “YES” or “NO”
- Optimization problem: solving the problem by finding the optimal solution.
- Examples:
  - SHORTEST-PATH (optimization)
    - Given *G*, *u*,*v*, find a path from *u* to *v* with fewest edges.
  - PATH (decision)
    - Given *G*, *u*,*v*, and *k*, whether exist a path from *u* to *v* consisting of at most *k* edges.

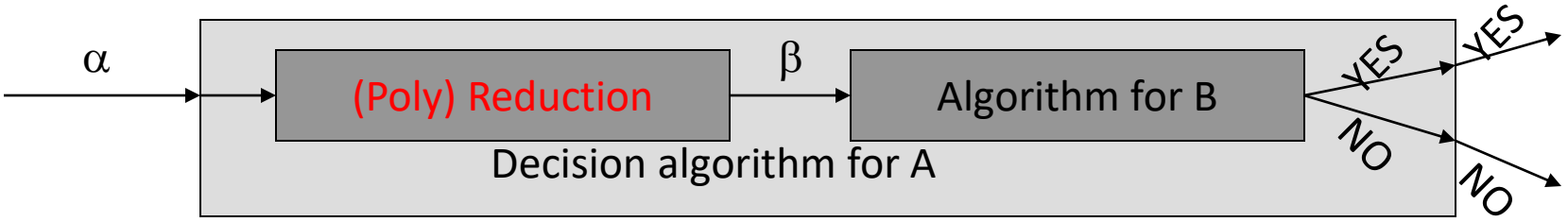
Decision VS. Optimization Problems (Cont.)

- Decision is easier (i.e., no harder) than optimization
- If there is an algorithm for an optimization problem, the algorithm can be used to solve the corresponding decision problem.
  - Example: SHORTEST-PATH for PATH
- If optimization is easy, its corresponding decision is also easy. Or in another way, if provide evidence that decision problem is hard, then the corresponding optimization problem is also hard.
- NPC is confined to decision problem. (also applicable to optimization problem.)
  - Another reason is that: easy to define reduction between decision problems.

(Poly) reduction between decision problems

- Problem (class) and problem instance
- Instance α of decision problem A and instance β of decision problem B
- A reduction from A to B is a transformation with the following properties:
  - The transformation takes poly time
  - The answer is the same (the answer for α is YES if and only if the answer for β is YES).

Implication of (poly) reduction



1. If decision algorithm for B is poly, so does A.
   A is no harder than B (or B is no easier than A)

2. If A is hard (e.g., NPC), so does B.

3. How to prove a problem B to be NPC ??

   (at first, prove B is in NP, which is generally easy.)

   3.1 find a already proved NPC problem A
   3.2 establish an (poly) reduction from A to B

Question: What is and how to prove the first NPC problem?

Circuit-satisfiability problem.

Discussion on Poly time problems

- $\Theta(n^{100})$ vs. $\Theta(2^n)$
  - Reasonable to regard a problem of $\Theta(n^{100})$ as intractable, however, very few practical problem with $\Theta(n^{100})$.
  - Most poly time algorithms require much less.
  - Once a poly time algorithm is found, more efficient algorithm may follow soon.
- Poly time keeps same in many different computation models, e.g., poly class of serial random-access machine ≡ poly class of abstract Turing machine ≡ poly class of parallel computer (#processors grows polynomially with input size)
- Poly time problems have nice closure properties under addition, multiplication and composition.

Encoding impact on complexity

- The problem instance must be represented in a way the program (or machine) can understand.
- General encoding is “binary representation”.
- Different encoding will result in different complexities.
- Example: an algorithm, only input is integer k, running time is $\Theta(k)$.
  - If k is represented in *unary*: a string of k 1s, the running time is $\Theta(k) = \Theta(n)$ on length-n input, poly on n.
  - If k is represented in *binary:* the input length $n = \lfloor \log k \rfloor + 1$, the running time is $\Theta(k) = \Theta(2^n)$, exponential on n.
- Ruling out *unary*, other encoding methods are same.

Examples of encoding and complexity

- Given integer *n*, check whether *n* is a composite.
- Dynamic programming for subset-sum.

Class P Problems

- Let n= the length of binary encoding of a problem (i.e., input size), $T(n)$ is the time to solve it.
- A problem is *poly-time solvable* if $T(n) = O(n^k)$ for some constant k.
- Complexity class **P**=set of problems that are *poly-time solvable*.

Poly Time Verification

- PATH problem: Given <G,*u*,*v*,*k*>, whether exists a path from *u* to *v* with at most *k* edges?
- Moreover, also given a path *p* from *u* to *v*, verify whether the length of *p* is at most *k*?
- Easy or not?

Of course, very easy.

Poly Time Verification, encoding, and language

- Hamiltonian cycles
  - A simple path containing every vertex.
  - HAM-CYCLE={<G>: G is a Hamiltonian graph, i.e. containing Hamiltonian cycle}.
  - Suppose ***n*** is the length of **encoding** of G.
  - HAM-CYCLE can be considered as a **Language** after encoding, i.e. a subset of $\sum^*$ where $\sum=\{0,1\}^*$.
- The naïve algorithm for determining HAM-CYCLE runs in $\Omega(m!)=\Omega(2^m)$ time, where m is the number of vertices, $m \approx n^{1/2}$.
- However, given an ordered sequence of m vertices (called "certificate"), let you verify whether the sequence is a Hamiltonian cycle. Very easy. In $O(n^2)$ time.

Class NP problems

- For a problem *p*, given its certificate, the certificate can be verified in poly time.
- Call this kind of problem an NP one.
- Complement set/class: Co-NP.
  - Given a set S (as a universal) and given a subset A
  - The complement is that S-A.
  - When NP problems are represented as languages (i.e. a set), we can discuss their complement set, i.e., Co-NP.

Relation among P, NP and co-NP={L: L $\in$NP where L= $\overline{\sum^*-L}$}

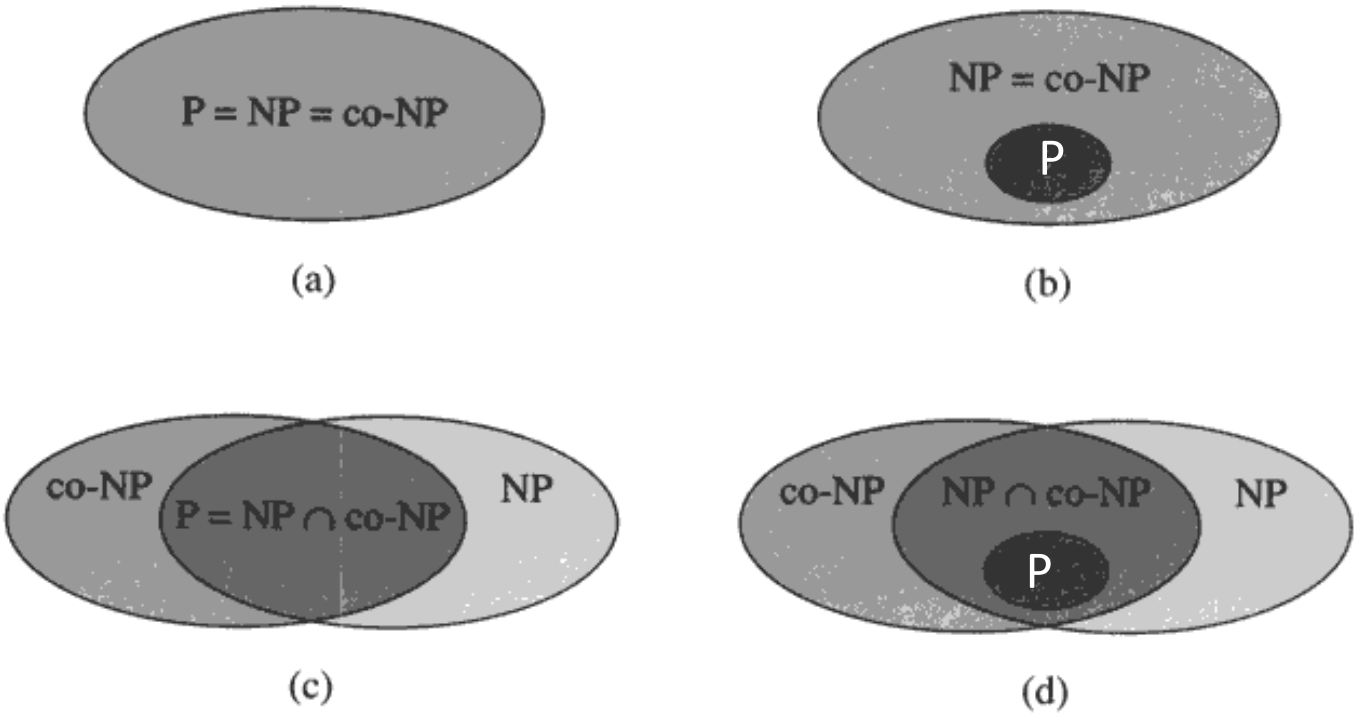


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** P = NP = co-NP. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. **(c)** P = NP ∩ co-NP, but NP is not closed under complement. **(d)** NP ≠ co-NP and P ≠ NP ∩ co-NP. Most researchers regard this possibility as the most likely.

NP-completeness and Reducibility

- A (class of) problem P_1 is poly-time reducible to P_2 , written as $P_1 \leq_p P_2$ if there exists a poly-time function f: $P_1 \rightarrow P_2$ such that for any instance of $p_1 \in P_1$, p_1 has "YES" answer if and only if answer to $f(p_1)$ ($\in P_2$) is also "YES".
- *Theorem 34.3*: (page 985)
  - For two problems P_1, P_2, if $P_1 \leq_p P_2$ then $P_2 \in P$ implies $P_1 \in P$.

NP-completeness and Reducibility (cont.)

- A problem p is NP-complete if
  1. $p \in NP$ and
  2. $p' \leq_p p$ for every $p' \in NP$.

  (if p satisfies 2, then p is said NP-hard.)

Theorem 34.4 (page 986)

if any NP-compete problem is poly-time solvable, then P=NP. Or say: if any problem in NP is not poly-time solvable, then no NP-complete problem is poly-time solvable.

First NP-complete problem—Circuit Satisfiability (problem definition)

- Boolean combinational circuit
  - Boolean combinational elements, wired together
  - Each element, inputs and outputs (binary)
  - Limit the number of outputs to 1.
  - Called *logic gates*: NOT gate, AND gate, OR gate.
  - *true table*: giving the outputs for each setting of inputs
  - true assignment: a set of boolean inputs.
  - satisfying assignment: a true assignment causing the output to be 1.
  - A circuit is satisfiable if it has a satisfying assignment.

Circuit Satisfiability Problem: definition

- Circuit satisfying problem: given a boolean combinational circuit composed of AND, OR, and NOT, is it stisfiable?
- CIRCUIT-SAT={<C>: C is a satisfiable boolean circuit}
- Implication: in the area of computer-aided hardware optimization, if a subcircuit always produces 0, then the subcircuit can be replaced by a simpler subcircuit that omits all gates and just output a 0.

Two instances of circuit satisfiability problems

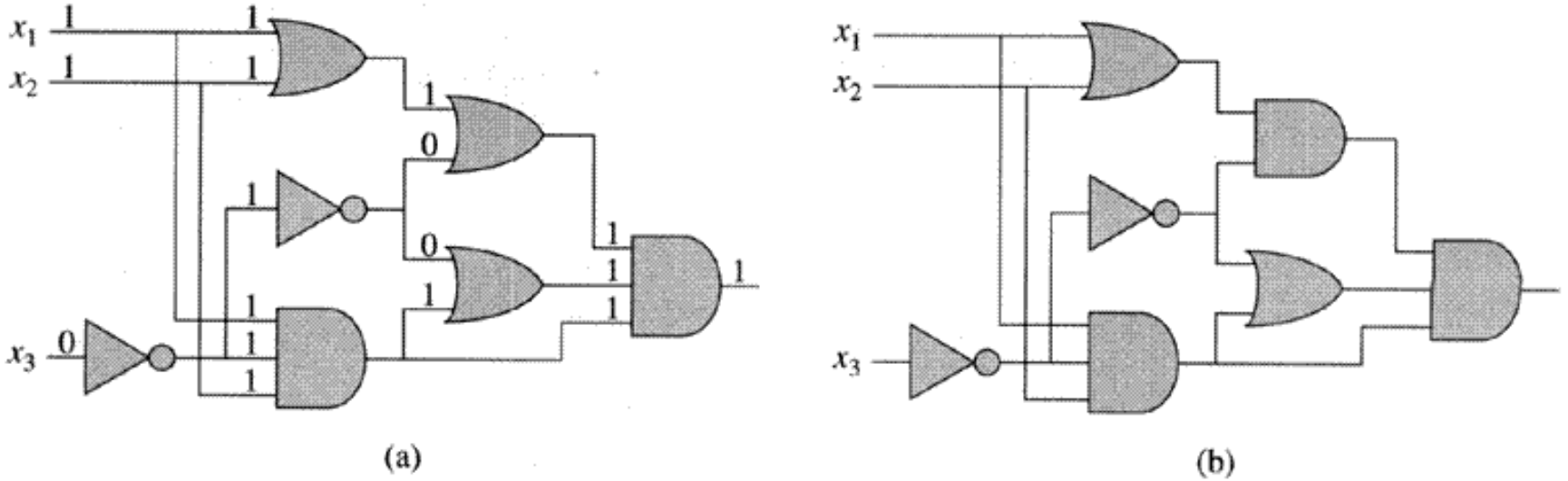


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Solving circuit-satisfiability problem

- Intuitive solution:
  - for each possible assignment, check whether it generates 1.
  - suppose the number of inputs is k, then the total possible assignments are 2^k. So the running time is $\Omega(2^k)$. When the size of the problem is $\Theta(k)$, then the running time is not poly.

Circuit-satisfiability problem is NP-complete

- *Lemma 34.5*:(page 990)
  - CIRCUIT-SAT belongs to NP.
- Proof: CIRCUIT-SAT is poly-time verifiable.
  - Given (an encoding of) a CIRCUIT-SAT problem C and a certificate, which is an assignment of boolean values to (all) wires in C.
  - The algorithm is constructed as follows: just checks each gates and then the output wire of C:
    - If for every gate, the computed output value matches the value of the output wire given in the certificate and the output of the whole circuit is 1, then the algorithm outputs 1, otherwise 0.
    - The algorithm is executed in poly time (even linear time).
- An alternative certificate: a true assignment to the inputs.

Circuit-satisfiability problem is NP-complete (cont.)

- *Lemma 34.6:* (page 991)
  - CIRCUIT-SAT is NP-hard.
- Proof: Suppose X is *any problem* in NP
  - construct a poly-time algorithm *F* maps every problem instance *x* in X to a circuit C=*f*(*x*) such that the answer to *x* is YES if and only if C$\in$CIRCUIT-SAT (is satisfiable).

Circuit-satisfiability problem is NP-hard (cont.)

- Since $X \in NP$, there is a poly-time algorithm A which verifies X.
- Suppose the input length is n and Let $T(n)$ denote the worst-case running time. Let k be the constant such that $T(n)=O(n^k)$ and the length of the certificate is $O(n^k)$.

Circuit-satisfiability problem is NP-hard (cont.)

- Idea is to represent the computation of A as a sequence of configurations, $c_0, c_1,\ldots,c_i,c_{i+1},\ldots,c_{T(n)}$, each c_i can be broken into
  - (program for A, program counter PC, auxiliary machine state, input x, certificate y, working storage) and
  - c_i is mapped to c_{i+1} by the combinational circuit M implementing the computer hardware.
  - The output of A: 0 or 1– is written to some designated location in working storage. If the algorithm runs for at most $T(n)$ steps, the output appears as one bit in $c_{T(n)}$.
  - Note: $A(x,y)$=1 or 0.

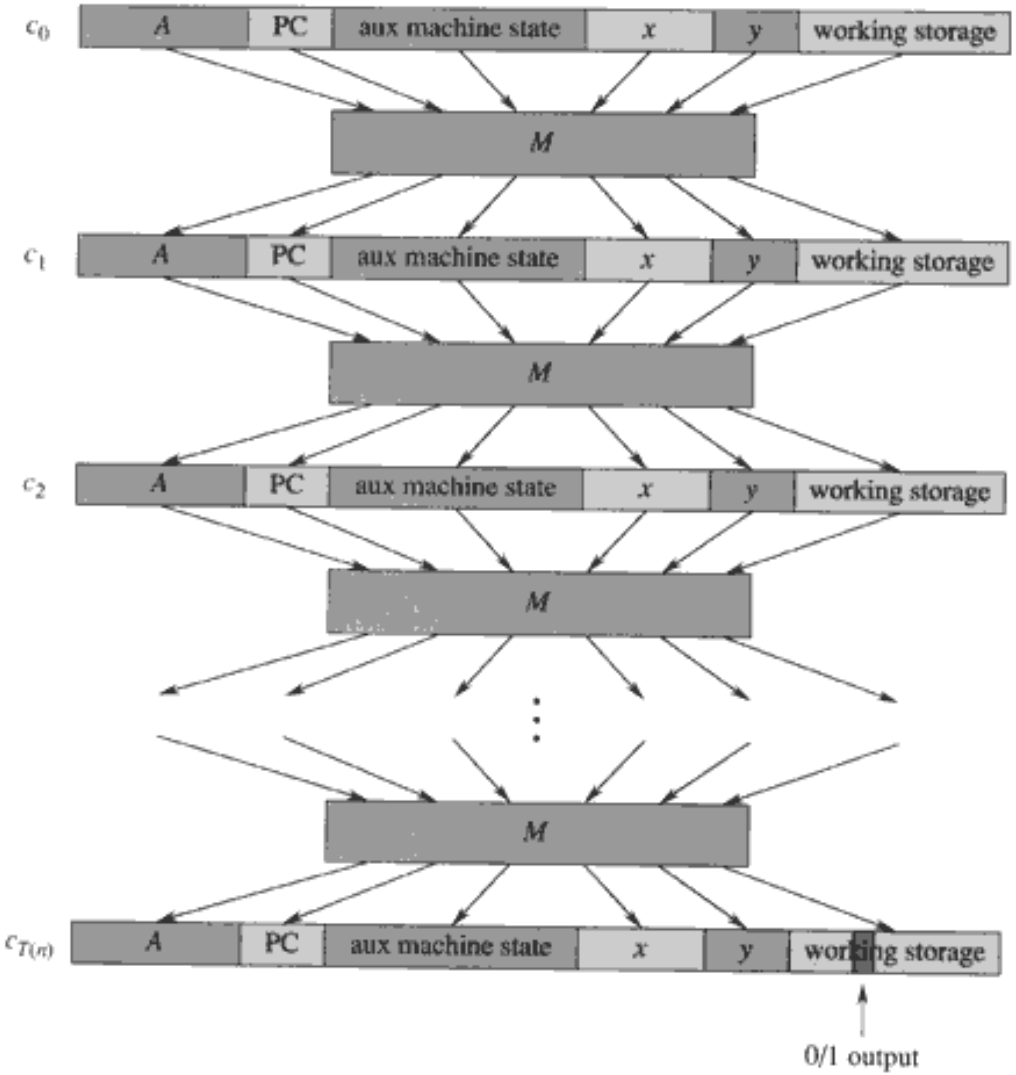


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. Each configuration is mapped to the next configuration by a boolean combinational circuit M. The output is a distinguished bit in the working storage.

Circuit-satisfiability problem is NP-hard (cont.)

- The reduction algorithm F constructs a single combinational circuit C as follows:
  - Paste together all $T(n)$ copies of the circuit M.
  - The output of the ith circuit, which produces c_i, is directly fed into the input of the $(i+1)$st circuit.
  - All items in the initial configuration, except the bits corresponding to certificate y, are wired directly to their known values.
  - The bits corresponding to y are the inputs to C.
  - All the outputs to the circuit are ignored, except the one bit of $c_{T(n)}$ corresponding to the output of A.

Circuit-satisfiability problem is NP-hard (cont.)

- Two properties remain to be proven:
  - F correctly constructs the reduction, i.e., C is satisfiable if and only if there exists a certificate y, such that $A(x,y)=1$.
    - $\Leftarrow$Suppose there is a certificate y, such that $A(x,y)=1$. Then if we apply the bits of y to the inputs of C, the output of C is the bit of $A(x,y)$, that is $C(y)= A(x,y) =1$, so C is satisfiable.
    - $\Rightarrow$Suppose C is satisfiable, then there is a y such that $C(y)=1$. So, $A(x,y)=1$.
  - F runs in poly time.

Circuit-satisfiability problem is NP-hard (cont.)

- F runs in poly time.
  - Poly space:
    - Size of x is n.
    - Size of A is constant, independent of x.
    - Size of y is $O(n^k)$.
    - Amount of working storage is poly in n since A runs at most $O(n^k)$.
    - M has size poly in length of configuration, which is poly in $O(n^k)$, and hence is poly in n.
    - C consists of at most $O(n^k)$ copies of M, and hence is poly in n.
    - Thus, the C has poly space.
  - The construction of C takes at most $O(n^k)$ steps and each step takes poly time, so F takes poly time to construct C from x.

CIRCUIT-SAT is NP-complete

- In summary
  - CIRCUIT-SAT belongs to NP, verifiable in poly time.
  - CIRCUIT-SAT is NP-hard, every NP problem can be reduced to CIRCUIT-SAT in poly time.
  - Thus CIRCUIT-SAT is NP-complete.

NP-completeness proof basis

- *Lemma 34.8* (page 995)
  - If X is a problem (class) such that $P' \leq_p X$ for some $P' \in$ NPC, then X is NP-hard. Moreover, if $X \in$ NP, then $X \in$ NPC.
- Steps to prove X is NP-complete
  - Prove $X \in$ NP.
    - Given a certificate, the certificate can be verified in poly time.
  - Prove X is NP-hard.
    - Select a known NP-complete P'.
    - Describe a transformation function *f* that maps every instance *x* of P' into an instance *f*(*x*) of X.
    - Prove *f* satisfies that the answer to $x \in$ P' is YES if and only if the answer to $f(x) \in$ X is YES for all instance $x \in$ P'.
    - Prove that the algorithm computing *f* runs in poly-time.

NPC proof –Formula Satisfiability (SAT)

- SAT definition
  - n boolean variables: $x_1, x_2,\ldots, x_n$.
  - M boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge,\vee,\neg,\rightarrow,\leftrightarrow,\ldots$ and
  - Parentheses.
- A SAT ϕ is satisfiable if there exists an true assignment which causes ϕ to evaluate to 1.
- SAT=$\{< \phi >: \phi$ is a satifiable boolean formula$\}$.
- The historical honor of the first NP-complete problem ever shown.

SAT is NP-complete

- *Theorem 34.9:* (page 997)
  - SAT is NP-complete.
- Proof:
  - SAT belongs to NP.
    - Given a satisfying assignment, the verifying algorithm replaces each variable with its value and evaluates the formula, in poly time.
  - SAT is NP-hard (show CIRCUIT-SAT$\leq_p$ SAT).

SAT is NP-complete (cont.)

- CIRCUIT-SAT$\leq_p$ SAT, i.e., any instance of circuit satisfiability can be reduced in poly time to an instance of formula satisfiability.
- Intuitive induction:
  - Look at the gate that produces the circuit output.
  - Inductively express each of gate's inputs as formulas.
  - Formula for the circuit is then obtained by writing an expression that applies the gate's function to its input formulas.

- Unfortunately, this is not a poly reduction
  - Shared formula (the gate whose output is fed to 2 or more inputs of other gates) cause the size of generated formula to grow exponentially.

SAT is NP-complete (cont.)

- Correct reduction:
  - For every wire x_i of C, give a variable x_i in the formula.
  - Every gate can be expressed as $x_o \leftrightarrow (x_{i_1} \theta\, x_{i_2} \theta \ldots \theta\, x_{i_l})$
  - The final formula ϕ is the AND of the circuit output variable and conjunction of all clauses describing the operation of each gate. (example Figure 34.10)
- Correctness of the reduction
  - Clearly the reduction can be done in poly time.
  - C is satisfiable if and only if ϕ is satisfiable.
    - If C is satisfiable, then there is a satisfying assignment. This means that each wire of C has a well-defined value and the output of C is 1. Thus the assignment of wire values to variables in ϕ makes each clause in ϕ evaluate to 1. So ϕ is 1.
    - The reverse proof can be done in the same way.

Example of reduction of CIRCUIT-SAT to SAT

$$\phi = x_{10} \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_4 \leftrightarrow \neg x_3)$$

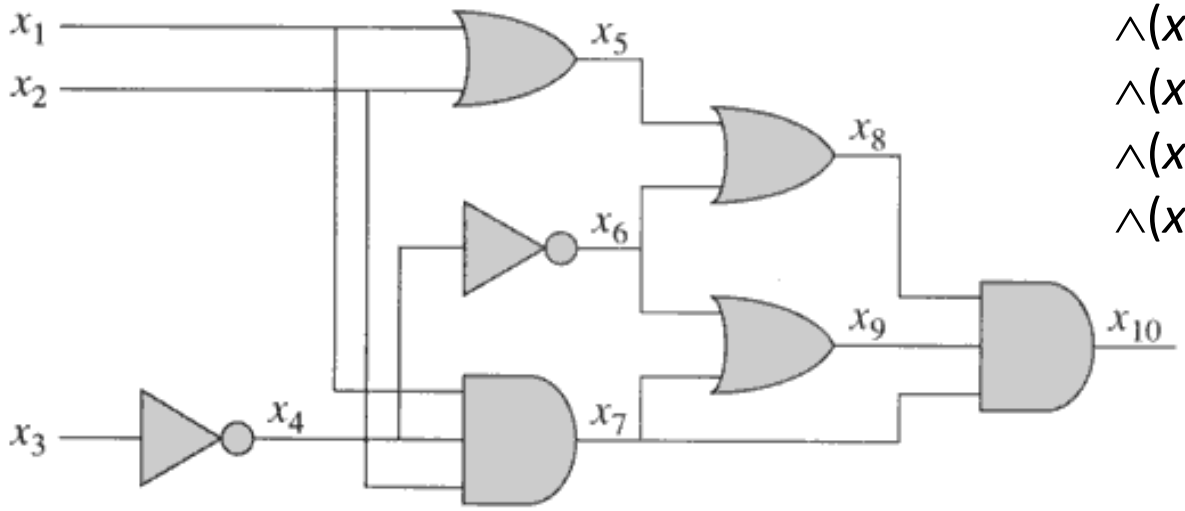


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

INCORRECT REDUCTION: $\phi = x_{10} = x_7 \wedge x_8 \wedge x_9 = (x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7)$
$= (x_1 \wedge x_2 \wedge x_4) \wedge ((x_1 \vee x_2) \vee \neg x_4) \wedge (\neg x_4 \vee (x_1 \wedge x_2 \wedge x_4)) = \ldots.$

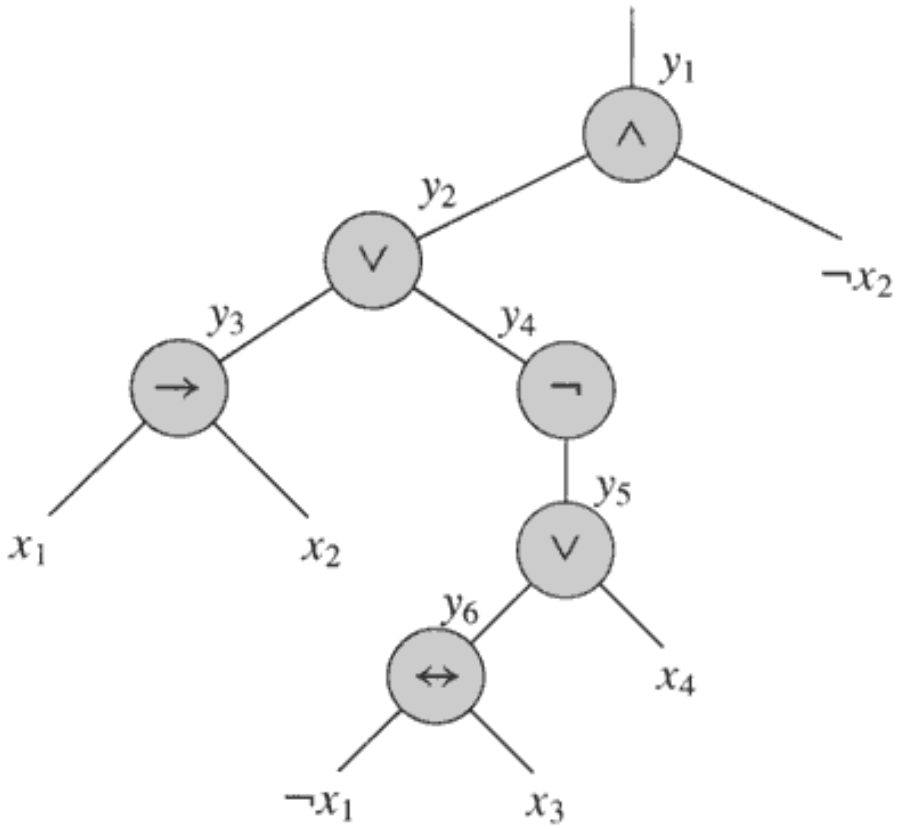
NPC Proof –3-CNF Satisfiability

- 3-CNF definition
  - A *literal* in a boolean formula is an occurrence of a variable or its negation.
  - CNF (Conjunctive Nornal Form) is a boolean formula expressed as AND of clauses, each of which is the OR of one or more literals.
  - 3-CNF is a CNF in which each clause has exactly 3 distinct literals (a literal and its negation are distinct)
- 3-CNF-SAT: whether a given 3-CNF is satiafiable?

3-CNF-SAT is NP-complete

- Proof: 3-CNF-SAT $\in$NP. Easy.
  - 3-CNF-SAT is NP-hard. (show SAT $\leq_p$3-CNF-SAT)
    - Suppose ϕ is any boolean formula, Construct a binary 'parse' tree, with literals as leaves and connectives as internal nodes.
    - Introduce a variable y_i for the output of each internal nodes.
    - Rewrite the formula to ϕ' as the AND of the root variable and a conjunction of clauses describing the operation of each node.
    - The result is that in ϕ', each clause has at most three literals.
    - Change each clause into conjunctive normal form as follows:
      - Construct a true table, (small, at most 8 by 4)
      - Write the disjunctive normal form for all true-table items evaluating to 0
      - Using DeMorgan law to change to CNF.
    - The resulting ϕ'' is in CNF but each clause has 3 or less literals.
    - Change 1 or 2-literal clause into 3-literal clause as follows:
      - If a clause has one literal l, change it to $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$.
      - If a clause has two literals $(l_1 \vee l_2)$, change it to $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$.

Binary parse tree for $\phi=((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$



$$\begin{aligned}\phi' = {} & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Disjunctive Normal Form:

$\phi_i' = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2)$
$\vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

Conjunctive Normal Form:

$\phi_i'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2)$
$\wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$

Figure 34.12 The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

3-CNF is NP-complete

- ϕ and reduced 3-CNF are equivalent:
  - From ϕ to ϕ' , keep equivalence.
  - From ϕ' to ϕ'' , keep equivalence.
  - From ϕ'' to final 3-CNF, keep equivalence.
- Reduction is in poly time,
  - From ϕ to ϕ' , introduce at most 1 variable and 1 clause per connective in ϕ.
  - From ϕ' to ϕ'' , introduce at most 8 clauses for each clause in ϕ'.
  - From ϕ'' to final 3-CNF, introduce at most 4 clauses for each clause in ϕ''.

NP-completeness proof structure

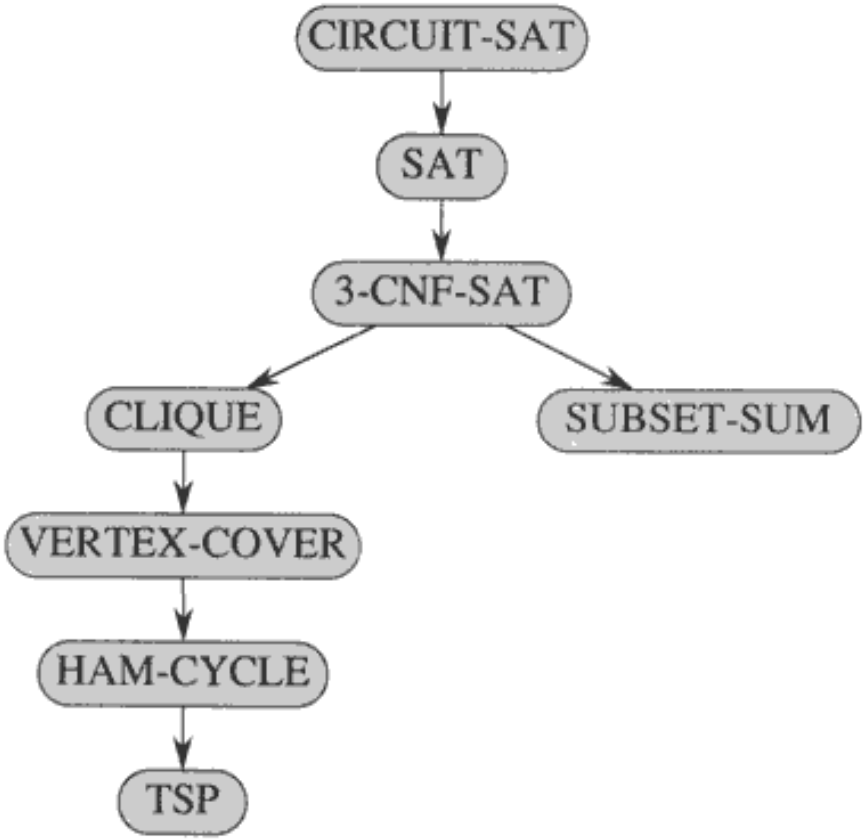


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

NPC proof -- CLIQUE

- Definition: a clique in an undirected graph G=(V,E) is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E, i.e., a clique is a complete subgraph of G.
- Size of a clique is the number of vertices in the clique.
- Optimization problem: find the maximum clique.
- Decision problem: whether a clique of given size *k* exists in the graph?
- CLIQUE={<G,*k*>: G is a graph with a clique of size *k*.}
- Intuitive solution: ???

CLIQUE is NP-complete

- *Theorem 34.11*: (page 1003)
  - CLIQUE problem is NP-complete.
- Proof:
  - CLIUEQE $\in$NP: given G=(V,E) and a set V'$\subseteq$V as a certificate for G. The verifying algorithm checks for each pair of $u,v \in V'$, whether $\langle u,v \rangle \in E$. time: $O(|V'|^2|E|)$.
  - CLIQUE is NP-hard:
    - show 3-CNF-SAT $\leq_p$CLUQUE.
    - The result is surprising, since from boolean formula to graph.

CLIQUE is NP-complete

- Reduction from 3-CNF-SAT to CLUQUE.
  - Suppose $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a boolean formula in 3-CNF with k clauses.
  - We construct a graph G=(V,E) as follows:
    - For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$, place a triple of v_1^r, v_2^r, v_3^r into V
    - Put the edge between two vertices v_i^r and v_j^s when:
      - $r \neq s$, that is v_i^r and v_j^s are in different triples, and
      - Their corresponding literals are consistent, i.e, l_i^r is not negation of l_j^s .
  - Then ϕ is satisfiable if and only if G has a clique of size k.

$\phi=(x_1\vee\neg x_2\vee\neg x_3)\wedge(\neg x_1\vee x_2\vee x_3)\wedge(x_1\vee x_2\vee x_3)$ and its reduced graph G

$C_1=x_1\vee\neg x_2\vee\neg x_3$ ...

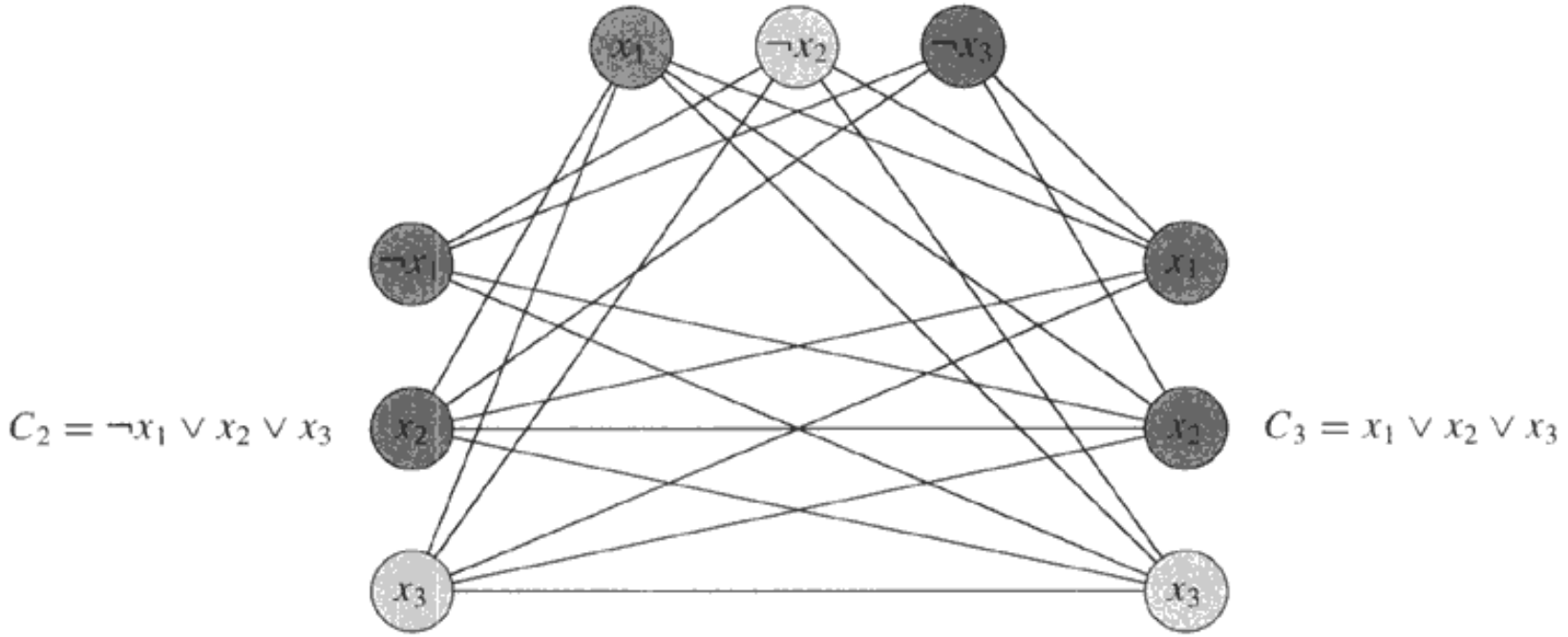


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 may be either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

CLIQUE is NP-complete

- Prove the above reduction is correct:
  - If ϕ is satisfiable, then there exists a satisfying assignment, which makes at least one literal in each clause to evaluate to 1. Pick one this kind of literal in each clause. Then consider the subgraph V' consisting of the corresponding vertex of each such literal. For each pair $v_i^r, v_j^s \in$V', where $r \neq s$. Since l_i^r, l_j^s are both evaluated to 1, so l_i^r is not negation of l_j^s, thus there is an edge between v_i^r and v_j^s. So V' is a clique of size k.
  - If G has a clique V' of size k, then V' contains exact one vertex from each triple. Assign all the literals corresponding to the vertices in V' to 1, and other literals to 1 or 0, then each clause will be evaluated to 1. So ϕ is satisfiable.
- It is easy to see the reduction is in poly time.
- The reduction of an instance of one problem to a specific instance of the other problem.

Traveling-salesman problem is NPC

- TSP={<G,*c*,*k*>:

  G=(V,E) is a complete graph,

  c is a function from $V \times V \rightarrow Z$,

  $k \in Z$, and G has a traveling salesman tour with cost at most *k*.}

- *Theorem 34.14*: (page 1012)
  - TSP is NP-complete.

TSP is NP-complete

- TSP belongs to NP:
  - Given a certificate of a sequence of vertices in the tour, the verifying algorithm checks whether each vertex appears once, sums up the cost and checks whether at most *k*. in poly time.
- TSP is NP-hard (show HAM-CYCLE$\leq_p$TSP)
  - Given an instance G=(V,E) of HAM-CYCLE, construct a TSP instance <G',c,0) as follows (in poly time):
    - G'=(V,E'), where E'={<*i*,*j*>: *i*,*j*$\in$ V and *i*$\neq$*j*} and
    - Cost function *c* is defined as *c*(*i*,*j*)=0 if (*i*,*j*) $\in$E, 1, otherwise.
  - If G has a hamiltonian cycle *h*, then *h* is also a tour in G' with cost at most 0.
  - If G' has a tour *h*' of cost at most 0, then each edge in *h*' is 0, so each edge belong to E, so *h*' is also a hilmitonian cycle in G.

Subset Sum is NPC

- SUNSET-SUM={<S,*t*>: S is a set of integers and there exists a S'$\subseteq$S such that $t=\sum_{s\in S'}s$.}
- *Theorem 34.15*: (page 1014)
  - SUBSET-SUM is NP-complete.

SUBSET-SUM is NPC

- SUBSET-SUM belongs to NP.
  - Given a certificate S', check whether *t* is sum of S' can be finished in poly time.
- SUBSET-SUM is NP-hard (show 3-CNF-SAT$\leq_p$SUBSET-SUM).

SUBSET-SUM is NPC

- Given a 3-CNF formula $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ with literals $x_1, x_2, \ldots, x_n$. Construct a SUBSET-SUM instance as follows:
  - Two assumptions: no clause contains both a literal and its negation, and either a literal or its negation appears in at least one clause.
  - The numbers in S are based on 10 and have $n+k$ digits, each digit corresponds to (or is labeled by) a literal or a clause.
  - Target t=1…1||4…4 (n 1's and k 4's)
  - For each literal x_i, create two integers:
    - $v_i = 0\ldots01_{(i)}0\ldots0||0\ldots01_{(l)}0\ldots01_{(w)}0\ldots0$, where x_i appears in $C_l,\ldots,C_w$.
    - $v_i' = 0\ldots01_{(i)}0\ldots0||0\ldots1_{(m)}0\ldots\ldots01_{(p)}0\ldots0$, where $\neg x_i$ appears in $C_m,\ldots,C_p$. .
    - Clearly, v_i and v_i' can not be equal in right k digits, moreover all v_i and v_i' in S are distinct.
  - For each clause C_j, create two integers:
    - $s_j = 0\ldots0||0\ldots01_{(j)}0\ldots0$,
    - $s_j' = 0\ldots0||0\ldots02_{(j)}0\ldots0$.
    - all s_j and s_j' are called "slack number". Clearly, all s_j and s_j' in S are distinct.
  - Note: the sum of digits in any one digit position is 2 or 6, so when there is no carries when adding any subset of the above integers.

SUBSET-SUM is NPC

- The above reduction is done in poly time.
- The 3-CNF formula ϕ is satisfiable if and only if there is a subset S' whose sum is *t*.
  - suppose ϕ has a satisfying assignment.
    - Then for $i=1,\ldots,n$, if $x_i=1$ in the assignment, then v_i is put in S', otherwise, then v_i' is put in S'.
    - The digits labeled by literals will sum to 1.
    - Moreover, for each digit labeled by a clause C_j and in its three literals, there may be 1, 2, or 3 assignments to be 1. correspondingly, both s_j and s_j' or s_j', or s_j is added to S' to make the sum of the digit to 4.
    - So S' will sum to 1…14…4.
  - Suppose there is a S' which sums to 1…14…4. then S' contains exact one of v_i and v_i' for $i=1,\ldots,n$. if $v_i \in$ S', then set $x_i=1$, otherwise, $v_i' \in$ S', then set $x_i=0$. It can be seen that this assignment makes each clause of ϕ to evaluate to 1. so ϕ is satisfiable.

Thank You !!!

Master Theorem:
Practice Problems and Solutions

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Practice Problems

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T(n/2) + n^2$

2. $T(n) = 4T(n/2) + n^2$

3. $T(n) = T(n/2) + 2^n$

4. $T(n) = 2^n T(n/2) + n^n$

5. $T(n) = 16T(n/4) + n$

6. $T(n) = 2T(n/2) + n \log n$

[1] most of the time, $k = 0$

7. $T(n) = 2T(n/2) + n/\log n$

8. $T(n) = 2T(n/4) + n^{0.51}$

9. $T(n) = 0.5T(n/2) + 1/n$

10. $T(n) = 16T(n/4) + n!$

11. $T(n) = \sqrt{2}T(n/2) + \log n$

12. $T(n) = 3T(n/2) + n$

13. $T(n) = 3T(n/3) + \sqrt{n}$

14. $T(n) = 4T(n/2) + cn$

15. $T(n) = 3T(n/4) + n\log n$

16. $T(n) = 3T(n/3) + n/2$

17. $T(n) = 6T(n/3) + n^2\log n$

18. $T(n) = 4T(n/2) + n/\log n$

19. $T(n) = 64T(n/8) - n^2\log n$

20. $T(n) = 7T(n/3) + n^2$

21. $T(n) = 4T(n/2) + \log n$

22. $T(n) = T(n/2) + n(2 - \cos n)$

Solutions

1. $T(n) = 3T(n/2) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = T(n/2) + 2^n \implies \Theta(2^n)$ (Case 3)
4. $T(n) = 2^n T(n/2) + n^n \implies$ Does not apply (a is not constant)
5. $T(n) = 16T(n/4) + n \implies T(n) = \Theta(n^2)$ (Case 1)
6. $T(n) = 2T(n/2) + n \log n \implies T(n) = n \log^2 n$ (Case 2)
7. $T(n) = 2T(n/2) + n/\log n \implies$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)
8. $T(n) = 2T(n/4) + n^{0.51} \implies T(n) = \Theta(n^{0.51})$ (Case 3)
9. $T(n) = 0.5T(n/2) + 1/n \implies$ Does not apply ($a < 1$)
10. $T(n) = 16T(n/4) + n! \implies T(n) = \Theta(n!)$ (Case 3)
11. $T(n) = \sqrt{2}T(n/2) + \log n \implies T(n) = \Theta(\sqrt{n})$ (Case 1)
12. $T(n) = 3T(n/2) + n \implies T(n) = \Theta(n^{\lg 3})$ (Case 1)
13. $T(n) = 3T(n/3) + \sqrt{n} \implies T(n) = \Theta(n)$ (Case 1)
14. $T(n) = 4T(n/2) + cn \implies T(n) = \Theta(n^2)$ (Case 1)
15. $T(n) = 3T(n/4) + n \log n \implies T(n) = \Theta(n \log n)$ (Case 3)
16. $T(n) = 3T(n/3) + n/2 \implies T(n) = \Theta(n \log n)$ (Case 2)
17. $T(n) = 6T(n/3) + n^2 \log n \implies T(n) = \Theta(n^2 \log n)$ (Case 3)
18. $T(n) = 4T(n/2) + n/\log n \implies T(n) = \Theta(n^2)$ (Case 1)
19. $T(n) = 64T(n/8) - n^2 \log n \implies$ Does not apply ($f(n)$ is not positive)
20. $T(n) = 7T(n/3) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
21. $T(n) = 4T(n/2) + \log n \implies T(n) = \Theta(n^2)$ (Case 1)
22. $T(n) = T(n/2) + n(2 - \cos n) \implies$ Does not apply. We are in Case 3, but the regularity condition is violated. (Consider $n = 2\pi k$, where k is odd and arbitrarily large. For any such choice of n, you can show that $c \geq 3/2$, thereby violating the regularity condition.)