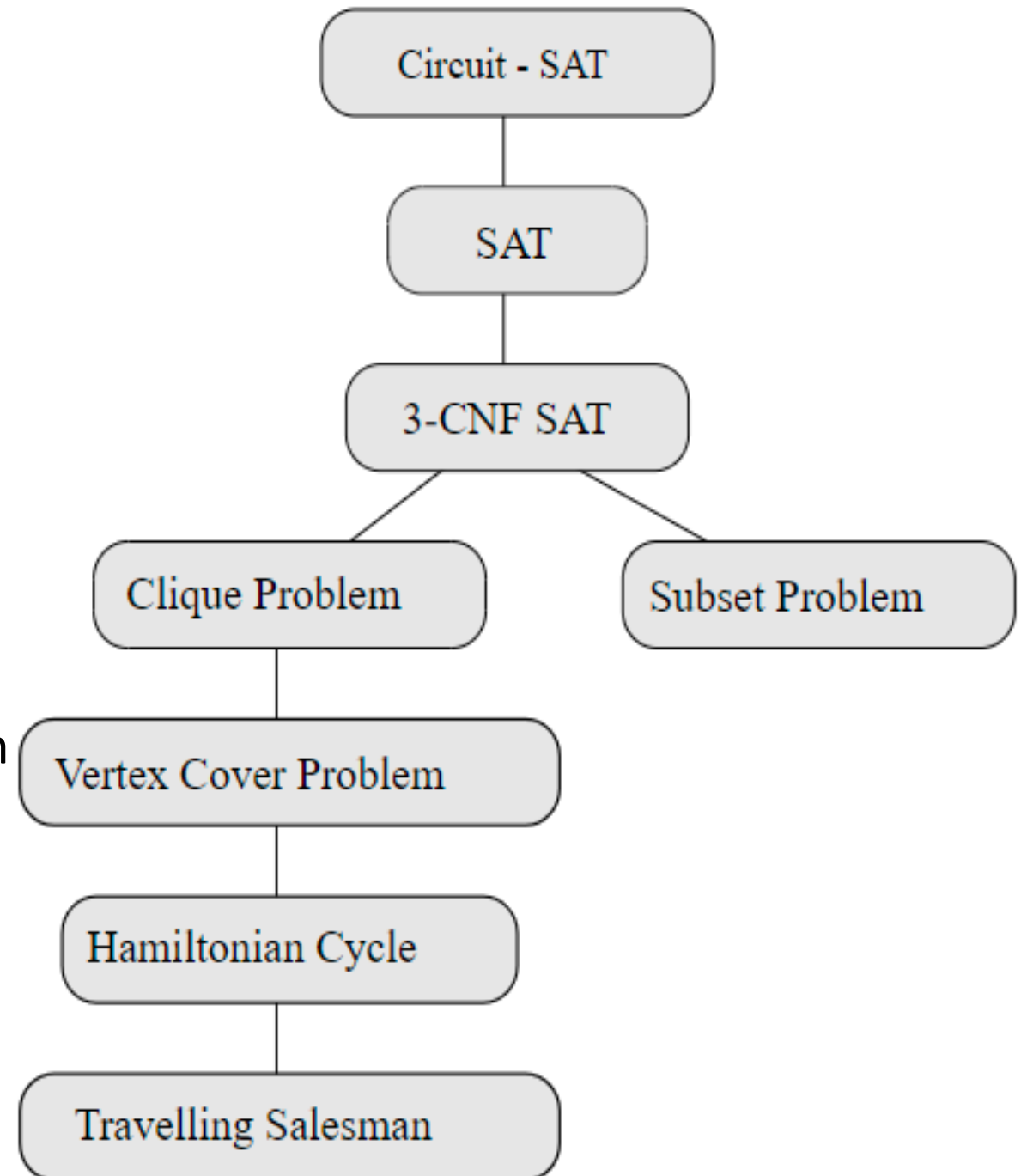


# Design and Analysis of Algorithm

Mahesh Shirole,  
VJTI, Mumbai-19.

# NP-Complete Problems

- The **circuit-satisfiability** problem is NP-complete
- Now we can prove other NP problems are NP-complete
- The hierarchy of the problems how they are reduced to prove NP-completeness
- All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT



# NP-completeness proofs

- We proved that the **circuit-satisfiability** problem is NP-complete by a direct proof that  $L \leq_p \text{CIRCUIT-SAT}$  for every language  $L \in \text{NP}$
- **Lemma:** If  $L$  is a language such that  $L' \leq_p L$  for some  $L' \in \text{NPC}$ , then  $L$  is NP-hard. If, in addition,  $L \in \text{NP}$ , then  $L \in \text{NPC}$
- **Proof:** Since  $L'$  is NP-complete, for all  $L'' \in \text{NP}$ , we have  $L'' \leq_p L'$ . By supposition,  $L' \leq_p L$ , and thus by transitivity, we have  $L'' \leq_p L$ , which shows that  $L$  is NP-hard. If  $L \in \text{NP}$ , we also have  $L \in \text{NPC}$
- In other words, by reducing a known **NP-complete** language  $L'$  to  $L$ , we implicitly **reduce every language in NP** to  $L$
- A method for proving that a language  $L$  is NP-complete:
  1. Prove  $L \in \text{NP}$
  2. Select a known NP-complete language  $L'$
  3. Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$
  4. Prove that the function  $f$  satisfies  $x \in L'$  if and only if  $f(x) \in L$  for all  $x \in \{0, 1\}^*$
  5. Prove that the algorithm computing  $f$  runs in polynomial time

# SAT (Boolean formula) NP-complete

- We formulate the (formula) satisfiability problem in terms of the language SAT as follows
  1.  $n$  boolean variables:  $x_1, x_2, \dots, x_n$ ;
  2.  $m$  boolean connectives: any boolean function with one or two inputs and one output, such as  $\wedge$  (AND),  $\vee$  (OR),  $\neg$  (NOT),  $\rightarrow$  (implication),  $\leftrightarrow$  (if and only if); and
  3. parentheses. (Without loss of generality, we assume that there are no redundant parentheses, i.e., a formula contains at most one pair of parentheses per boolean connective.)
- Encode a boolean formula  $\phi$  in a length that is polynomial in  $n+m$
- A truth assignment for a boolean formula  $\phi$  is a set of values for the variables of  $\phi$
- A satisfying assignment is a truth assignment that causes it to evaluate to 1
- A formula with a satisfying assignment is a satisfiable formula

# SAT (Boolean formula) NP-complete

- The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms,
- $\text{SAT} = \{ \langle \Phi \rangle : \Phi \text{ is a satisfiable boolean formula} \}$

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$ , since

$$\begin{aligned}\phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1,\end{aligned}$$

and thus this formula  $\phi$  belongs to SAT.

# SAT (Boolean formula) NP-complete

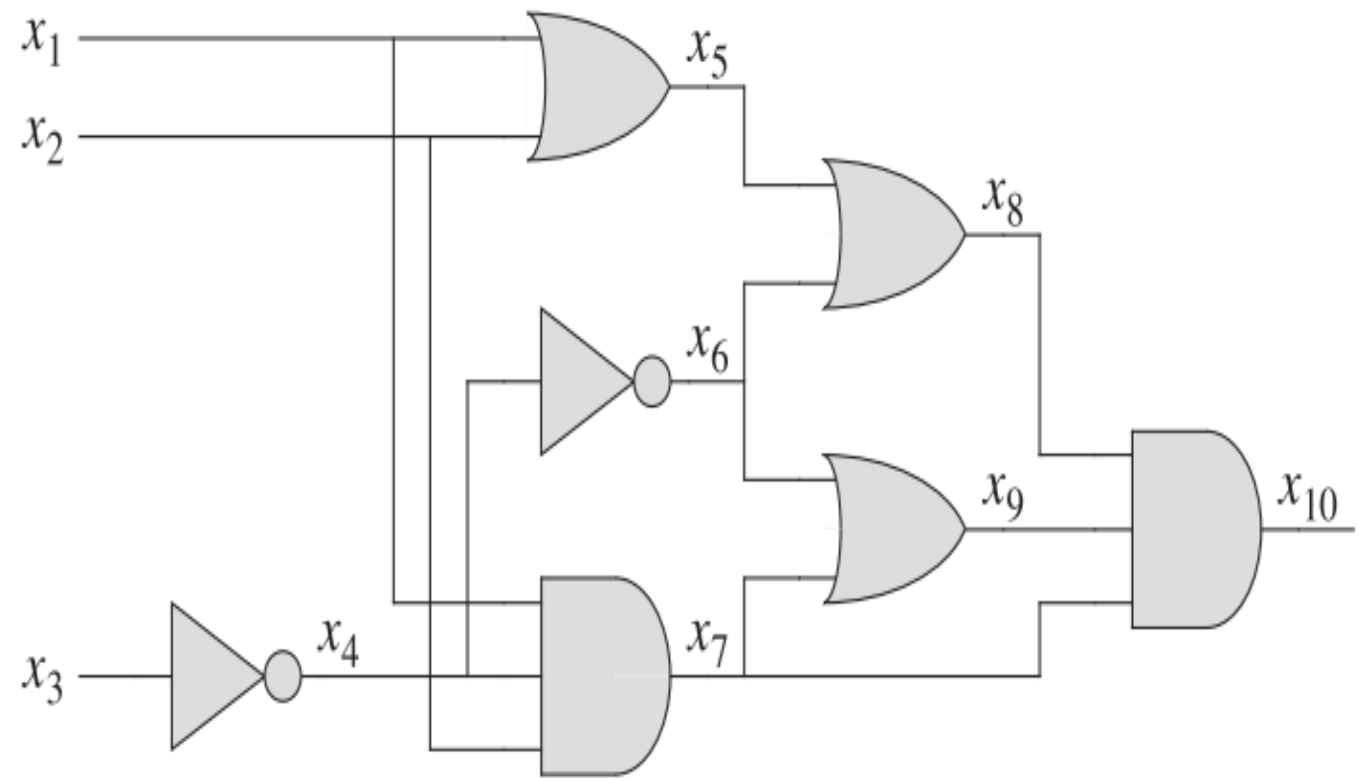
- **Theorem:** Satisfiability of boolean formulas is NP-complete
- Theorem says that a polynomial-time algorithm is unlikely to exist for SAT problem
  - A formula with  $n$  variables has  $2^n$  possible assignments. If the length of  $\langle \Phi \rangle$  is polynomial in  $n$ , then checking every assignment requires  $\Omega(2^n)$  time, which is superpolynomial in the length of  $\langle \Phi \rangle$
- **Step1:** Prove  $SAT \in NP$ 
  - A certificate consisting of a satisfying assignment for an input formula  $\Phi$  can be verified in polynomial time
  - The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression
  - This task is easy to do in polynomial time
  - If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable

# SAT (Boolean formula) NP-complete

- **Step 2: Select a known NP-complete language  $L'$** 
  - Show that  $\text{CIRCUIT-SAT} \leq_p \text{SAT}$
- **Step 3: Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$** 
  - For each wire  $x_i$  in the circuit  $C$ , the formula  $\Phi$  has a variable  $x_i$
  - We can express how each gate operates as a small formula involving the variables of its incident wires
  - We call each of these small formulas a *clause*
  - The formula  $\Phi$  produced by the reduction algorithm is the AND of the circuit output variable with the conjunction of clauses describing the operation of each gate
- **Step 4: Prove that the function  $f$  satisfies  $x \in L'$  if and only if  $f(x) \in L$  for all  $x \in \{0, 1\}^*$**
- **Step 5: Prove that the algorithm computing  $f$  runs in polynomial time**
  - Given a circuit  $C$ , it is straightforward to produce such a formula  $\Phi$  in polynomial time
  - If  $C$  has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1
  - Therefore, when we assign wire values to variables in  $\Phi$ , each clause of  $\Phi$  evaluates to 1, and thus the conjunction of all evaluates to 1
  - Conversely, if some assignment causes  $\Phi$  to evaluate to 1, the circuit  $C$  is satisfiable by an analogous argument

# Example SAT reduction

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





# 3-CNF satisfiability

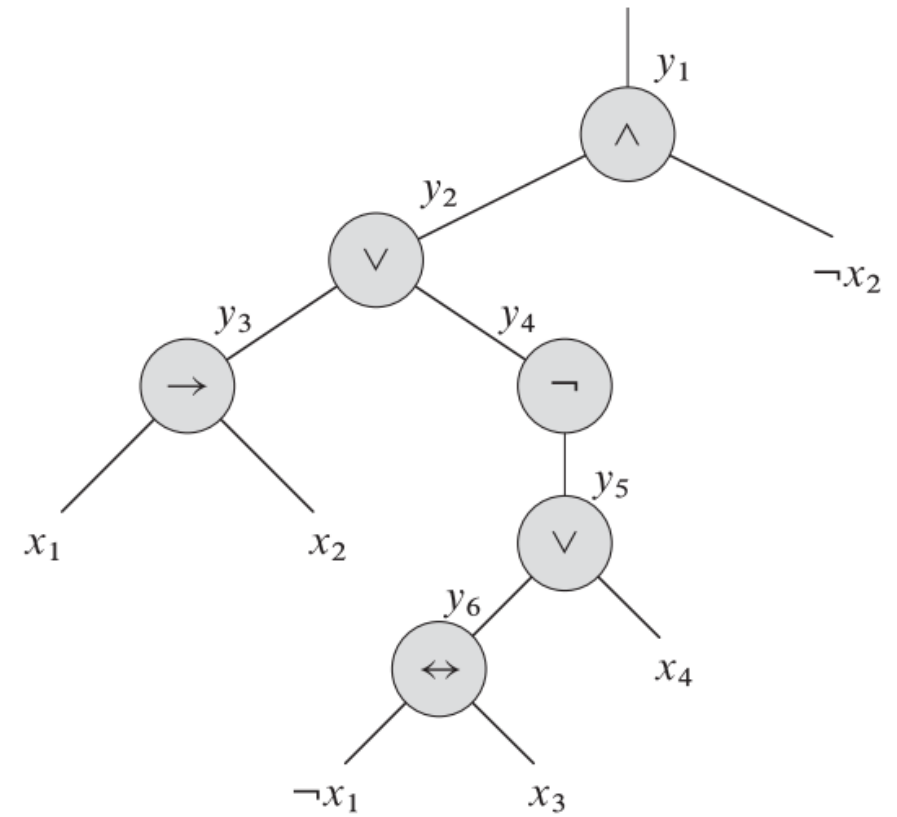
- We define 3-CNF satisfiability using the following terms
  - A *literal* in a boolean formula is an occurrence of a variable or its negation
  - A boolean formula is in *conjunctive normal form*, or *CNF*, if it is expressed as an *AND* of clauses, each of which is the *OR* of one or more literals
  - A boolean formula is in *3-conjunctive normal form*, or *3-CNF*, if each clause has exactly three distinct literals
- For example, the following boolean formula is in 3-CNF
  - $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$
- In 3-CNF-SAT, we are asked whether a given boolean formula  $\phi$  in 3-CNF is satisfiable

# 3-CNF satisfiability

- **Theorem:** Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete
- The above theorem says that a polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form
- **Step1:** Prove **3-CNF  $\in$  NP**
  - A certificate consisting of a satisfying assignment for an input formula  $\phi$  can be verified in polynomial time
  - The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression
  - This task is easy to do in polynomial time
  - If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable

# 3-CNF satisfiability

- **Step 2: Select a known NP-complete language  $L'$** 
  - Show that  $SAT \leq_p 3\text{-CNF-SAT}$
- **Step 3: Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$** 
  - First, we construct a binary “parse” tree for the input formula  $\phi$ , with literals as leaves and connectives as internal nodes
  - We can now think of the binary parse tree as a circuit for computing the function
  - We introduce a variable  $y_i$  for the output of each internal node. Then, we rewrite the original formula  $\phi$  as the AND of the root variable and a conjunction of clauses describing the operation of each node
  - Observe that the formula  $\phi'$  thus obtained is a conjunction of clauses  $\phi'_i$ , each of which has at most 3 literals



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

# 3-CNF satisfiability

- **Step 3: Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$**

- The second step of the reduction converts each clause  $\phi'$  into conjunctive normal form
- We construct a truth table for  $\phi'_i$  into conjunctive normal form by evaluating all possible assignments to its variables
- Each row of the truth table consists of a possible assignment of the variables of the clause, together with the value of the clause under that assignment
- Using the truth-table entries that evaluate to 0, we build a formula in disjunctive normal form (or DNF)—an OR of ANDs—that is equivalent to  $\neg\phi'_i$
- We then negate this formula and convert it into a CNF formula  $\phi''_i$  by using **DeMorgan's laws** for propositional logic to complement all literals, change ORs into ANDs, and change ANDs into ORs.

$$\begin{aligned}\neg(a \wedge b) &= \neg a \vee \neg b, \\ \neg(a \vee b) &= \neg a \wedge \neg b,\end{aligned}$$

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

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) .$$

Negating and applying DeMorgan's laws, we get the CNF formula

$$\begin{aligned}\phi''_1 &= (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ &\quad \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) ,\end{aligned}$$

which is equivalent to the original clause  $\phi'_1$ .

# 3-CNF satisfiability

- **Step 3: Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$** 
  - The third and final step of the reduction further transforms the formula so that each clause has exactly 3 distinct literals
  - We construct the final 3-CNF formula  $\Phi'''$  from the clauses of the CNF formula  $\Phi''$
  - For each clause  $C_i$  of  $\Phi''$ , we include the following clauses in  $\Phi'''$ :
    - If  $C_i$  has 3 distinct literals, then simply include  $C_i$  as a clause of  $\Phi'''$
    - If  $C_i$  has 2 distinct literals, that is, if  $C_i = (l_1 \vee l_2)$ , where  $l_1$  and  $l_2$  are literals, then include  $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$  as clauses of  $\Phi'''$ . The literals  $p$  and  $\neg p$  merely fulfil the syntactic requirement that each clause of  $\Phi'''$  has exactly 3 distinct literals
    - If  $C_i$  has just 1 distinct literal  $l$ , then include  $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$  as clauses of  $\Phi'''$ . Regardless of the values of  $p$  and  $q$ , one of the four clauses is equivalent to  $l$ , and the other 3 evaluate to 1

# 3-CNF satisfiability

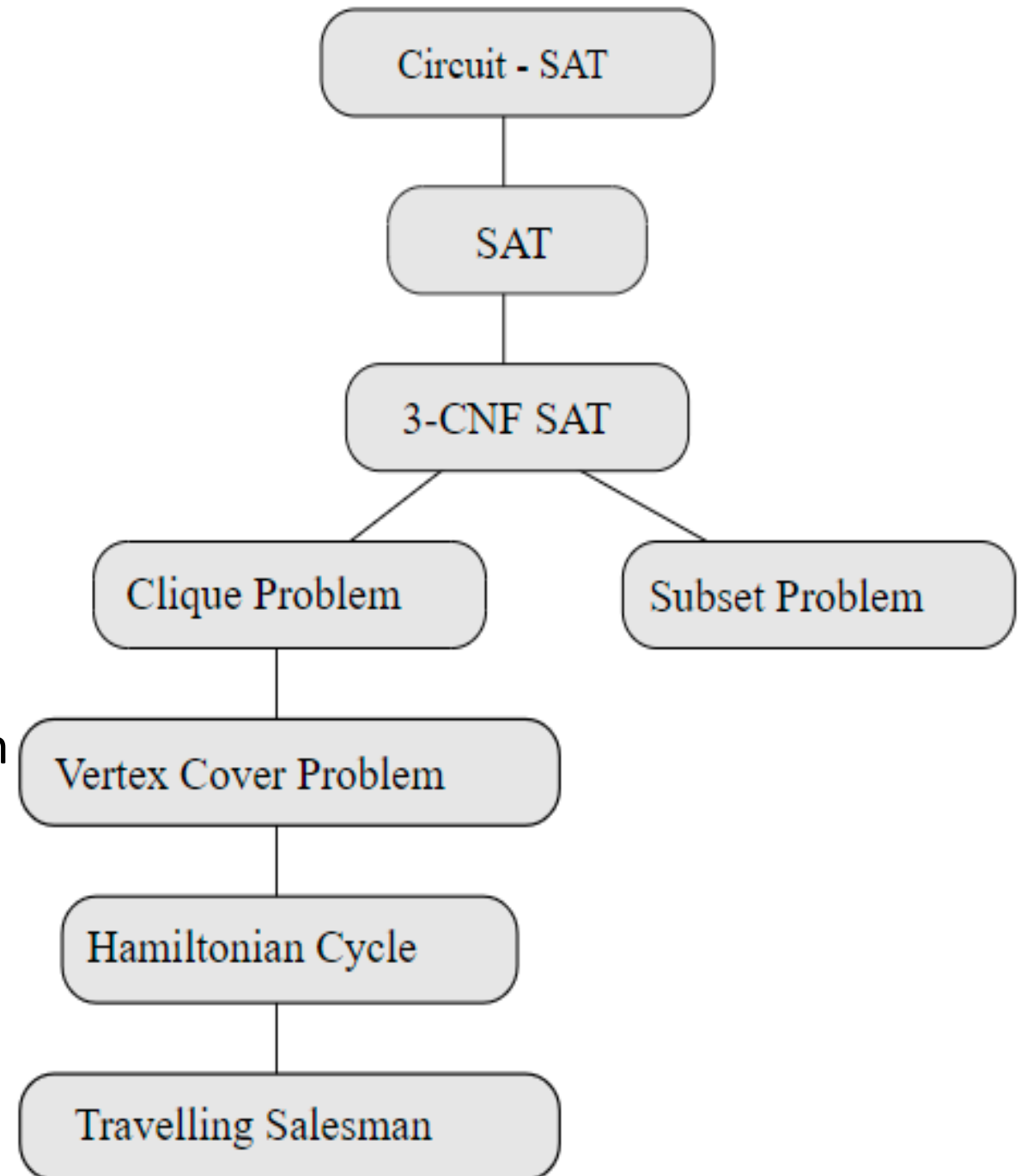
- **Step 4: Prove that the function  $f$  satisfies  $x \in L'$  if and only if  $f(x) \in L$  for all  $x \in \{0, 1\}^*$** 
  - The reduction from CIRCUIT-SAT to SAT, the construction of  $\phi'$  from  $\phi$  in the first step preserves satisfiability
  - The second step produces a CNF formula  $\phi''$  that is algebraically equivalent to  $\phi'$
  - The third step produces a 3-CNF formula  $\phi'''$  that is effectively equivalent to  $\phi''$ , since any assignment to the variables  $p$  and  $q$  produces a formula that is algebraically equivalent to  $\phi''$
- **Step 5: Prove that the algorithm computing  $f$  runs in polynomial time**
  - Constructing  $\phi'$  from  $\phi$  introduces at most 1 variable and 1 clause per connective in  $\phi$
  - Constructing  $\phi''$  from  $\phi'$  can introduce at most 8 clauses into  $\phi''$  for each clause from  $\phi'$ , since each clause of  $\phi'$  has at most 3 variables, and the truth table for each clause has at most  $2^3 = 8$  rows
  - The construction of  $\phi'''$  from  $\phi''$  introduces at most 4 clauses into  $\phi'''$  for each clause of  $\phi''$
  - Thus, the size of the resulting formula  $\phi'''$  is polynomial in the length of the original formula
  - Each of the constructions can easily be accomplished in polynomial time

# Design and Analysis of Algorithm

Mahesh Shirole,  
VJTI, Mumbai-19.

# NP-Complete Problems

- The **circuit-satisfiability** problem is NP-complete
- Now we can prove other NP problems are NP-complete
- The hierarchy of the problems how they are reduced to prove NP-completeness
- All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT





# NP-completeness proofs

- We proved that the **circuit-satisfiability** problem is NP-complete by a direct proof that  $L \leq_p \text{CIRCUIT-SAT}$  for every language  $L \in \text{NP}$
- **Lemma:** If  $L$  is a language such that  $L' \leq_p L$  for some  $L' \in \text{NPC}$ , then  $L$  is NP-hard. If, in addition,  $L \in \text{NP}$ , then  $L \in \text{NPC}$
- **Proof:** Since  $L'$  is NP-complete, for all  $L'' \in \text{NP}$ , we have  $L'' \leq_p L'$ . By supposition,  $L' \leq_p L$ , and thus by transitivity, we have  $L'' \leq_p L$ , which shows that  $L$  is NP-hard. If  $L \in \text{NP}$ , we also have  $L \in \text{NPC}$
- In other words, by reducing a known **NP-complete** language  $L'$  to  $L$ , we implicitly **reduce every language in NP** to  $L$
- A method for proving that a language  $L$  is NP-complete:
  1. Prove  $L \in \text{NP}$
  2. Select a known NP-complete language  $L'$
  3. Describe an algorithm that computes a function  $f$  mapping every instance  $x \in \{0, 1\}^*$  of  $L'$  to an instance  $f(x)$  of  $L$
  4. Prove that the function  $f$  satisfies  $x \in L'$  if and only if  $f(x) \in L$  for all  $x \in \{0, 1\}^*$
  5. Prove that the algorithm computing  $f$  runs in polynomial time

# The subset-sum problem

- In the subset-sum problem, we are given a finite set  $S$  of positive integers and an integer target  $t > 0$ . We ask whether there exists a subset  $S' \subseteq S$  whose elements sum to  $t$ .
- For example,
  - Consider  $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$  and  $t = 138457$ , then
  - A solution subset  $S_0 = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$

As usual, we define the problem as a language:

$$\text{SUBSET-SUM} = \{ \langle S, t \rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s \} .$$

# The subset-sum problem

- **Theorem:** The subset-sum problem is NP-complete

*Proof* To show that SUBSET-SUM is in NP, for an instance  $\langle S, t \rangle$  of the problem, we let the subset  $S'$  be the certificate. A verification algorithm can check whether  $t = \sum_{s \in S'} s$  in polynomial time.

- We now show that  **$3\text{-CNF-SAT} \leq_p \text{SUBSET-SUM}$**
- Given a 3-CNF formula  $\Phi$  over variables  $x_1, x_2, \dots, x_n$  with clauses  $C_1, C_2, \dots, C_k$ , each containing exactly three distinct literals, the reduction algorithm constructs an instance  $\langle S, t \rangle$  of the subset-sum problem such that  $\Phi$  is satisfiable if and only if there exists a subset of  $S$  whose sum is exactly  $t$

# The subset-sum problem

- Assumptions
  - No clause contains both a variable and its negation
  - Each variable appears in at least one clause
- The reduction creates two numbers in set  $S$  for each variable  $x_i$  and two numbers in  $S$  for each clause  $C_j$
- We create numbers in **base 10**, where each number contains  $n+k$  digits and each digit corresponds to either one variable or one clause
- We construct set  $S$  and target  $t$  as follows
- We label each digit position by either a variable or a clause. The least significant  $k$  digits are labeled by the clauses, and the most significant  $n$  digits are labeled by variables

# The subset-sum problem

- The target  $t$  has a **1** in each digit labelled by a **variable** and a **4** in each digit labelled by a **clause**
- For each variable  $x_i$ , set  $S$  contains two integers  $v_i$  and  $v'_i$
- Each of  $v_i$  and  $v'_i$  has a **1** in the digit labelled by  $x_i$  and **0**s in the other variable digits
- If literal  $x_i$  appears in clause  $C_j$ , then the digit labelled by  $C_j$  in  $v_i$  contains a **1**
- If literal  $\neg x_i$  appears in clause  $C_j$ , then the digit labelled by  $C_j$  in  $v'_i$  contains **1**
- All other digits labelled by clauses in  $v_i$  and  $v'_i$  are **0**
- For each clause  $C_j$ , set  $S$  contains two integers  $s_j$  and  $s'_j$
- Each of  $s_j$  and  $s'_j$  has **0**s in all digits other than the one labelled by  $C_j$
- For  $s_j$ , there is a **1** in the  $C_j$  digit, and  $s'_j$  has a **2** in this digit

		$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$v_1$	=	1	0	0	1	0	0	1
$v'_1$	=	1	0	0	0	1	1	0
$v_2$	=	0	1	0	0	0	0	1
$v'_2$	=	0	1	0	1	1	1	0
$v_3$	=	0	0	1	0	0	1	1
$v'_3$	=	0	0	1	1	1	0	0
$s_1$	=	0	0	0	1	0	0	0
$s'_1$	=	0	0	0	2	0	0	0
$s_2$	=	0	0	0	0	1	0	0
$s'_2$	=	0	0	0	0	2	0	0
$s_3$	=	0	0	0	0	0	1	0
$s'_3$	=	0	0	0	0	0	2	0
$s_4$	=	0	0	0	0	0	0	1
$s'_4$	=	0	0	0	0	0	0	2
$t$	=	1	1	1	4	4	4	4

# The subset-sum problem

- Note that the greatest sum of digits in any one digit position is 6, which occurs in the digits labeled by clauses
- Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits
- We can perform the *reduction in polynomial time*
- The set  **$S$**  contains  **$2n + 2k$**  values, each of which has  **$n$**   **$+ k$**  digits, and the time to produce each digit is polynomial in  **$n + k$**
- The target  **$t$**  has  **$n + k$**  digits, and the reduction produces each in constant time

		$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$v_1$	=	1	0	0	1	0	0	1
$v'_1$	=	1	0	0	0	1	1	0
$v_2$	=	0	1	0	0	0	0	1
$v'_2$	=	0	1	0	1	1	1	0
$v_3$	=	0	0	1	0	0	1	1
$v'_3$	=	0	0	1	1	1	0	0
$s_1$	=	0	0	0	1	0	0	0
$s'_1$	=	0	0	0	2	0	0	0
$s_2$	=	0	0	0	0	1	0	0
$s'_2$	=	0	0	0	0	2	0	0
$s_3$	=	0	0	0	0	0	1	0
$s'_3$	=	0	0	0	0	0	2	0
$s_4$	=	0	0	0	0	0	0	1
$s'_4$	=	0	0	0	0	0	0	2
$t$	=	1	1	1	4	4	4	4

**Figure 34.19** The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is  $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ , where  $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ ,  $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ , and  $C_4 = (x_1 \vee x_2 \vee x_3)$ . A satisfying assignment of  $\phi$  is  $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$ . The set  $S$  produced by the reduction consists of the base-10 numbers shown; reading from top to bottom,  $S = \{1001001, 1000110, 1000001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$ . The target  $t$  is 1114444. The subset  $S' \subseteq S$  is lightly shaded, and it contains  $v'_1$ ,  $v'_2$ , and  $v_3$ , corresponding to the satisfying assignment. It also contains slack variables  $s_1$ ,  $s'_1$ ,  $s'_2$ ,  $s_3$ ,  $s_4$ , and  $s'_4$  to achieve the target value of 4 in the digits labeled by  $C_1$  through  $C_4$ .

# The subset-sum problem

- We now show that the 3-CNF formula  $\Phi$  is satisfiable if and only if there exists a subset  $S' \subseteq S$  whose sum is  $t$
- First, suppose that  $\Phi$  has a satisfying assignment
- For  $i = 1, 2, \dots, n$ , if  $x_i = 1$  in this assignment, then include  $v_i$  in  $S'$ . Otherwise, include  $v'_i$
- For each variable-labeled digit, the sum of the values of  $S'$  must be 1, which matches those digits of the target  $t$
- Because each clause is satisfied, the clause contains some literal with the value 1
- Therefore, each digit labeled by a clause has at least one 1 contributed to its sum by a  $v_i$  or  $v'_i$  value in  $S'$ .
- In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause labeled digit has a sum of 1, 2, or 3 from the  $v_i$  or  $v'_i$  value in  $S'$
- We achieve the target of 4 in each digit labeled by clause  $C_j$  by including in  $S'$  the appropriate nonempty subset of slack variables  $\{s_i, s'_i\}$

		$x_1$	$x_2$	$x_3$	$C_1$	$C_2$	$C_3$	$C_4$
$v_1$	=	1	0	0	1	0	0	1
$v'_1$	=	1	0	0	0	1	1	0
$v_2$	=	0	1	0	0	0	0	1
$v'_2$	=	0	1	0	1	1	1	0
$v_3$	=	0	0	1	0	0	1	1
$v'_3$	=	0	0	1	1	1	0	0
$s_1$	=	0	0	0	1	0	0	0
$s'_1$	=	0	0	0	2	0	0	0
$s_2$	=	0	0	0	0	1	0	0
$s'_2$	=	0	0	0	0	2	0	0
$s_3$	=	0	0	0	0	0	1	0
$s'_3$	=	0	0	0	0	0	2	0
$s_4$	=	0	0	0	0	0	0	1
$s'_4$	=	0	0	0	0	0	0	2
$t$	=	1	1	1	4	4	4	4



# The subset-sum problem

- Now, suppose that there is a subset  $S' \subseteq S$  that sums to  $t$
- The subset  $S'$  must include exactly one of  $v_i$  and  $v'_i$  for each  $i = 1, 2, \dots, n$ , for otherwise the digits labeled by variables would not sum to 1
- If  $v_i \in S'$ , we set  $x_i = 1$
- Otherwise,  $v'_i \in S'$ , we set  $x_i = 0$
- We claim that every clause  $C_j$ , for  $j = 1, 2, \dots, k$ , is satisfied by this assignment
- To prove this claim, note that to achieve a sum of 4 in the digit labeled by  $C_j$ , the subset  $S'$  must include at least one  $v_i$  and  $v'_i$  value that has a 1 in the digit labeled by  $C_j$ , since the contributions of the slack variables  $s_j$  and  $s'_j$  together sum to at most 3.
- If  $S'$  includes a  $v_i$  that has a 1 in  $C_j$ 's position, then the literal  $x_i$  appears in clause  $C_j$
- Since we have set  $x_i = 1$  when  $v_i \in S'$ , clause  $C_j$  is satisfied
- If  $S'$  includes a  $v'_i$  that has a 1 in  $C_j$ 's position, then the literal  $\neg x_i$  appears in clause  $C_j$
- Since we have set  $x_i = 0$  when  $v'_i \in S'$ , clause  $C_j$  is again satisfied
- Thus, all clauses of  $\Phi$  are satisfied, which completes the proof



# Design and Analysis of Algorithm (R4IT2007S)

Mahesh Shirole

VJTI, Mumbai-19

# Approximation Algorithms

- Many problems of practical significance are NP-complete
- We don't know how to find an optimal solution in polynomial time
- There are at least three ways to get around NP-completeness
  - First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory
  - Second, we may be able to isolate important special cases that we can solve in polynomial time
  - Third, we might come up with approaches to find *near-optimal* solutions in polynomial time (either in the worst case or the expected case)
- In practice, near optimality is often good enough
- ***Approximation algorithms*** return near-optimal solutions

# Design and Analysis of Algorithm

Mahesh Shirole

VJTI, Mumbai-19

# Approximation Algorithms

- Many problems of practical significance are NP-complete
- We don't know how to find an optimal solution in polynomial time
- There are at least three ways to get around NP-completeness
  - First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory
  - Second, we may be able to isolate important special cases that we can solve in polynomial time
  - Third, we might come up with approaches to find *near-optimal* solutions in polynomial time (either in the worst case or the expected case)
- In practice, near optimality is often good enough
- ***Approximation algorithms*** return near-optimal solutions

# Performance ratios for approximation algorithms

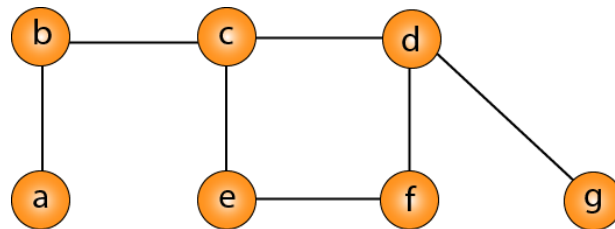
- Depending on the problem, we may define an optimal solution as
  - one with maximum possible cost (maximization problem)
  - one with minimum possible cost (minimization problem)
- We say that an algorithm for a problem has an **approximation ratio** of  $\rho(n)$  if, for any input of size  $n$ , the cost  $C$  of the solution produced by the algorithm is within a factor of  $\rho(n)$  of the cost  $C^*$  of an optimal solution

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n)$$

- The definitions of the approximation ratio and of a  $\rho(n)$  - approximation algorithm apply to both minimization and maximization problems
- For a maximization problem,  $0 < C \leq C^*$ , and the ratio  $C^*/C$  gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution
- For a minimization problem,  $0 < C^* \leq C$ , and the ratio  $C/C^*$  gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution

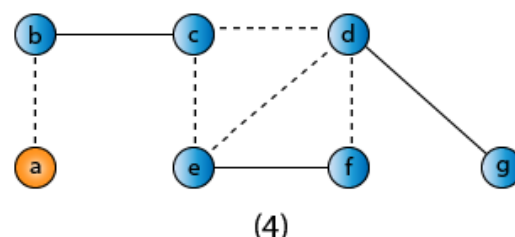
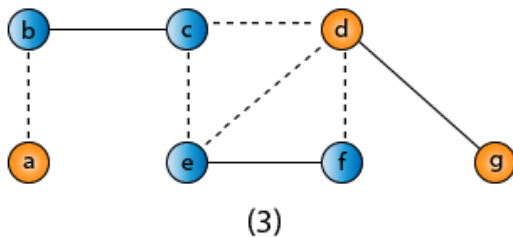
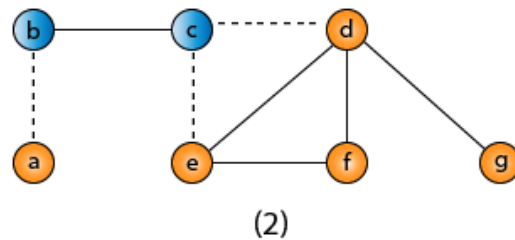
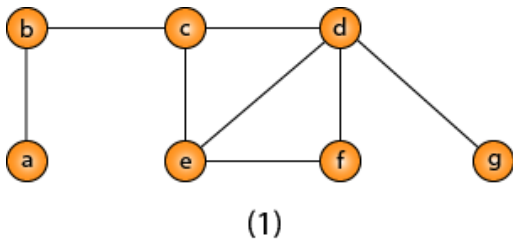
# The vertex-cover problem

- 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)
- That is, each vertex “covers” its incident edges, and a vertex cover for  $G$  is a set of vertices that covers all the edges in  $E$
- The size of a vertex cover is the number of vertices in it
- The **vertex-cover problem** is to find a vertex cover of minimum size in a given undirected graph
- We call such a vertex cover an **optimal vertex cover**. This problem is the optimization version of an NP-complete decision problem



# Approximate Vertex Cover Algorithm

- Approximation algorithm takes as input an undirected graph  $G$  and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover



APPROX-VERTEX-COVER( $G$ )

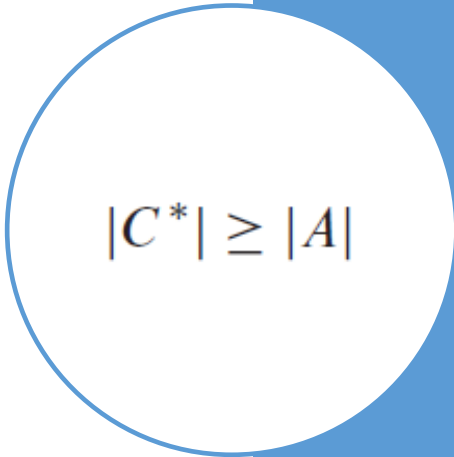
```
1  $C = \emptyset$ 
2  $E' = G.E$ 
3 while  $E' \neq \emptyset$ 
4     let  $(u, v)$  be an arbitrary edge of  $E'$ 
5      $C = C \cup \{u, v\}$ 
6     remove from  $E'$  every edge incident on either  $u$  or  $v$ 
7 return  $C$ 
```

The running time of this algorithm is  $O(V + E)$ , using adjacency lists to represent  $E'$

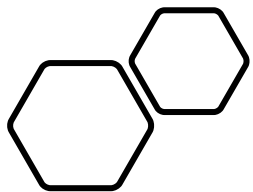
## *Theorem:* APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

- **Proof:** APPROX-VERTEX-COVER runs in polynomial time
- The set  $\mathbf{C}$  of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in  $\mathbf{G.E}$  has been covered by some vertex in  $\mathbf{C}$
- APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover
- let  $\mathbf{A}$  denote the set of edges that **line 4** of APPROX-VERTEX-COVER picked
- In order to cover the edges in  $\mathbf{A}$ , any vertex cover—in particular, an optimal cover  $\mathbf{C}^*$ —must include at least one endpoint of each edge in  $\mathbf{A}$
- No two edges in  $\mathbf{A}$  share an endpoint, since once an edge is picked in **line 4**, all other edges that are incident on its endpoints are deleted from  $\mathbf{E}'$  in **line 6**
- Thus, no two edges in  $\mathbf{A}$  are covered by the same vertex from  $\mathbf{C}^*$ , and we have the lower bound on the size of an optimal vertex cover

$$|\mathbf{C}^*| \geq |\mathbf{A}|$$


$$|\mathbf{C}^*| \geq |\mathbf{A}|$$





**Theorem:** APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

- Each execution of **line 4** picks an edge for which neither of its endpoints is already in  $C$ , yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned
- Combining equations (35.2) and (35.3), we obtain  $|C| \leq 2|C^*|$  thereby proving the theorem

$$|C| = 2|A|$$

(35.3)

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| \end{aligned}$$

$$|C^*| \geq |A|$$

(35.2)

# The subset-sum problem

- the subset-sum problem is a pair  $(S, t)$ , where  $S$  is a set  $\{x_1, x_2, \dots, x_n\}$  of positive integers and  $t$  is a positive integer
- This decision problem asks whether there exists a subset of  $S$  that adds up exactly to the target value  $t$
- We saw this problem is NP-complete
- In the optimization problem, we wish to find a subset of  $\{x_1, x_2, \dots, x_n\}$  whose sum is as large as possible but not larger than  $t$
- For example, we may have a truck that can carry no more than  $t$  pounds, and  $n$  different boxes to ship, the  $i^{\text{th}}$  of which weighs  $x_i$  pounds
- We wish to fill the truck with as heavy a load as possible without exceeding the given weight limit
- The subset-sum problem is a special case of the *knapsack problem*

# An exponential-time exact algorithm

EXACT-SUBSET-SUM( $S, t$ )

```
1   $n = |S|$ 
2   $L_0 = \langle 0 \rangle$ 
3  for  $i = 1$  to  $n$ 
4       $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$ 
5      remove from  $L_i$  every element that is greater than  $t$ 
6  return the largest element in  $L_n$ 
```

- To implement this algorithm, we could use an iterative procedure that, in iteration  $i$ , computes the sums of all subsets of  $\{x_1, x_2, \dots, x_i\}$ , using as a starting point the sums of all subsets of  $\{x_1, x_2, \dots, x_{i-1}\}$
- The procedure EXACT-SUBSET-SUM takes an input set  $S = \{x_1, x_2, \dots, x_n\}$  and a target value  $t$
- This procedure iteratively computes  $L_i$ , the list of sums of all subsets of  $\{x_1, x_2, \dots, x_i\}$  that do not exceed  $t$ , and then it returns the maximum value in  $L_n$
- If  $L$  is a list of positive integers and  $x$  is another positive integer, then we let  $L + x$  denote the list of integers derived from  $L$  by increasing each element of  $L$  by  $x$
- For example, if  $L = \langle 1; 2; 3; 5; 9 \rangle$ , then  $L+2 = \langle 3; 4; 5; 7; 11 \rangle$
- Use this notation for sets, so that  $S + x = \{s + x : s \in S\}$

# An exponential-time exact algorithm

EXACT-SUBSET-SUM( $S, t$ )

```
1   $n = |S|$ 
2   $L_0 = \langle 0 \rangle$ 
3  for  $i = 1$  to  $n$ 
4       $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$ 
5      remove from  $L_i$  every element that is greater than  $t$ 
6  return the largest element in  $L_n$ 
```

- We also use an auxiliary procedure **MERGE-LISTS**( $L, L'$ ), which returns the sorted list that is the merge of its two sorted input lists  $L$  and  $L'$  with duplicate values removed
- Like the MERGE procedure in merge sort, **MERGE-LISTS** runs in time  $O(|L| + |L'|)$
- To see how EXACT-SUBSET-SUM works, let  $P_i$  denote the set of all values obtained by selecting a (possibly empty) subset of  $\{x_1, x_2, \dots, x_i\}$  and summing its members
- For example, if  $S = \{1, 4, 5\}$ , then
  - $P_1 = \{0, 1\}$  ;
  - $P_2 = \{0, 1, 4, 5\}$  ;
  - $P_3 = \{0, 1, 4, 5, 6, 9, 10\}$

# An exponential-time exact algorithm

EXACT-SUBSET-SUM( $S, t$ )

```
1   $n = |S|$ 
2   $L_0 = \langle 0 \rangle$ 
3  for  $i = 1$  to  $n$ 
4       $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$ 
5      remove from  $L_i$  every element that is greater than  $t$ 
6  return the largest element in  $L_n$ 
```

- Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i)$$

- we can prove by induction on  $i$  that the list  $L_i$  is a sorted list containing every element of  $P_i$  whose value is not more than  $t$
- Since the length of  $L_i$  can be as much as  $2^i$ , **EXACT-SUBSET-SUM** is an exponential-time algorithm in general, although it is a polynomial-time algorithm in the special cases in which  $t$  is polynomial in  $|S|$  or all the numbers in  $S$  are bounded by a polynomial in  $|S|$

# A fully polynomial-time approximation scheme

- We can derive a fully polynomial-time approximation scheme for the subset-sum problem by “trimming” each list  $L_i$  after it is created
- The idea behind trimming is that if two values in  $L$  are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly
- More precisely, we use a trimming parameter  $\delta$  such that  $0 < \delta < 1$
- When we **trim** a list  $L$  by  $\delta$ , we remove as many elements from  $L$  as possible, in such a way that if  $L'$  is the result of trimming  $L$ , then for every element  $y$  that was removed from  $L$ , there is an element  $z$  still in  $L'$  that approximates  $y$ , that is,  $\frac{y}{1 + \delta} \leq z \leq y$

$$\frac{y}{1 + \delta} \leq z \leq y$$

# A fully polynomial-time approximation scheme

- We can think of such a  $z$  as “representing”  $y$  in the new list  $L'$
- For example, if  $\delta = 0.1$  and
- $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$  ; then we can trim  $L$  to obtain
- $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$  ; where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23
- Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element

# A fully polynomial-time approximation scheme

- The following procedure trims list  $L = \langle y_1, y_2, \dots, y_m \rangle$  in time,  $\theta(m)$ , given  $L$  and  $\delta$ , and assuming that  $L$  is sorted into monotonically increasing order
- The output of the procedure is a trimmed, sorted list

TRIM( $L, \delta$ )

```
1  let  $m$  be the length of  $L$ 
2   $L' = \langle y_1 \rangle$ 
3   $last = y_1$ 
4  for  $i = 2$  to  $m$ 
5      if  $y_i > last \cdot (1 + \delta)$       //  $y_i \geq last$  because  $L$  is sorted
6          append  $y_i$  onto the end of  $L'$ 
7           $last = y_i$ 
8  return  $L'$ 
```



# A fully polynomial-time approximation scheme

- This procedure takes as input a set  $S = \{x_1, x_2, \dots, x_n\}$  of  $n$  integers (in arbitrary order), a target integer  $t$ , and an “approximation parameter”  $\epsilon$ , where  $0 < \epsilon < 1$
- It returns a value  $z$  whose value is within a  $1 + \epsilon$  factor of the optimal solution
- Line 2 initializes the list  $L_0$  to be the list containing just the element  $0$
- The **for** loop in lines 3–6 computes  $L_i$  as a sorted list containing a suitably trimmed version of the set  $P_i$ , with all elements larger than  $t$  removed
- Since we create  $L_i$  from  $L_{i-1}$ , we must ensure that the repeated trimming doesn’t introduce too much compounded inaccuracy

APPROX-SUBSET-SUM( $S, t, \epsilon$ )

```
1   $n = |S|$ 
2   $L_0 = \langle 0 \rangle$ 
3  for  $i = 1$  to  $n$ 
4       $L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)$ 
5       $L_i = \text{TRIM}(L_i, \epsilon/2n)$ 
6      remove from  $L_i$  every element that is greater than  $t$ 
7  let  $z^*$  be the largest value in  $L_n$ 
8  return  $z^*$ 
```

# A fully polynomial-time approximation scheme

- As an example, suppose we have the instance  $S = \langle 104, 102, 201, 101 \rangle$  with  $t = 308$  and  $\varepsilon = 0.40$ . The trimming parameter  $\delta$  is  $\varepsilon / 8 = 0.05$
- APPROXSUBSET-SUM computes the following values on the indicated lines:
- The algorithm returns  $z = 302$  as its answer, which is well within  $\varepsilon = 40\%$  of the optimal answer  $307 = 104 + 102 + 101$ ; in fact, it is within 2%

line 2:  $L_0 = \langle 0 \rangle$ ,

line 4:  $L_1 = \langle 0, 104 \rangle$ ,

line 5:  $L_1 = \langle 0, 104 \rangle$ ,

line 6:  $L_1 = \langle 0, 104 \rangle$ ,

line 4:  $L_2 = \langle 0, 102, 104, 206 \rangle$ ,

line 5:  $L_2 = \langle 0, 102, 206 \rangle$ ,

line 6:  $L_2 = \langle 0, 102, 206 \rangle$ ,

line 4:  $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$ ,

line 5:  $L_3 = \langle 0, 102, 201, 303, 407 \rangle$ ,

line 6:  $L_3 = \langle 0, 102, 201, 303 \rangle$ ,

line 4:  $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$ ,

line 5:  $L_4 = \langle 0, 101, 201, 302, 404 \rangle$ ,

line 6:  $L_4 = \langle 0, 101, 201, 302 \rangle$ .