

# CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science  
Virginia Tech  
Blacksburg, Virginia

Spring 2014

Copyright © 2014 by Clifford A. Shaffer

# Tractable Problems

We would like some convention for distinguishing tractable from intractable problems.

A problem is said to be **tractable** if an algorithm exists to solve it with polynomial time complexity:  $O(p(n))$ .

- It is said to be **intractable** if the best known algorithm requires exponential time.

Examples:

- Sorting:  $O(n^2)$
- Convex Hull:  $O(n^2)$
- Single source shortest path:  $O(n^2)$
- All pairs shortest path:  $O(n^3)$
- Matrix multiplication:  $O(n^3)$

# Tractable Problems (cont)

The technique we will use to classify one group of algorithms is based on two concepts:

- 1 A special kind of reduction.
- 2 Nondeterminism.

# Decision Problems

( $I, S$ ) such that  $S(X)$  is always either “yes” or “no.”

- Usually formulated as a question.

## Example:

- Instance: A weighted graph  $G = (V, E)$ , two vertices  $s$  and  $t$ , and an integer  $K$ .

- Question: Is there a path from  $s$  to  $t$  of length  $\leq K$ ? In this example, the answer is “yes.”

# Decision Problems (cont)

Can also be formulated as a language recognition problem:

- Let  $L$  be the subset of  $I$  consisting of instances whose answer is “yes.” Can we recognize  $L$ ?

The class of tractable problems  $\mathcal{P}$  is the class of languages or decision problems recognizable in polynomial time.

# Polynomial Reducibility

Reduction of one language to another language.

Let  $L_1 \subset I_1$  and  $L_2 \subset I_2$  be languages.  $L_1$  is **polynomially reducible** to  $L_2$  if there exists a transformation  $f : I_1 \rightarrow I_2$ , computable in polynomial time, such that  $f(x) \in L_2$  if and only if  $x \in L_1$ .

We write:  $L_1 \leq_p L_2$  or  $L_1 \leq L_2$ .

# Examples

- CLIQUE  $\leq_p$  INDEPENDENT SET.
- An instance  $I$  of CLIQUE is a graph  $G = (V, E)$  and an integer  $K$ .
- The instance  $I' = f(I)$  of INDEPENDENT SET is the graph  $G' = (V, E')$  and the integer  $K$ , where an edge  $(u, v) \in E'$  iff  $(u, v) \notin E$ .
- $f$  is computable in polynomial time.

# Transformation Example

- $G$  has a clique of size  $\geq K$  iff  $G'$  has an independent set of size  $\geq K$ .
- Therefore,  $\text{CLIQUE} \leq_p \text{INDEPENDENT SET}$ .
- **IMPORTANT WARNING:** The reduction does not **solve** either INDEPENDENT SET or CLIQUE, it merely transforms one into the other.



# Nondeterminism

Nondeterminism allows an algorithm to make an arbitrary choice among a finite number of possibilities.

Implemented by the “nd-choice” primitive:

`nd-choice(ch1, ch2, ..., chj)`

returns one of the choices `ch1, ch2, ...` **arbitrarily**.

Nondeterministic algorithms can be thought of as “correctly guessing” (choosing nondeterministically) a solution.

# Nondeterminism

Nondeterminism allows an algorithm to make an arbitrary choice among a finite number of possibilities.

Implemented by the “nd-choice” primitive:

`nd-choice(ch1, ch2, ..., chj)`

returns one of the choices `ch1, ch2, ...` **arbitrarily**.

Nondeterministic algorithms can be thought of as “correctly guessing” (choosing nondeterministically) a solution.

Alternatively, nondeterministic algorithms can be thought of as running on super-parallel machines that make all choices simultaneously and then reports the “right” solution.

# Nondeterministic CLIQUE Algorithm

```
procedure nd-CLIQUE(Graph G, int K) {  
  VertexSet S = EMPTY;  int size = 0;  
  for (v in G.V)  
    if (nd-choice(YES, NO) == YES) then {  
      S = union(S, v);  
      size = size + 1;  
    }  
  if (size < K) then  
    REJECT;          // S is too small  
  for (u in S)  
    for (v in S)  
      if ((u <> v) && ((u, v) not in E))  
        REJECT;    // S is missing an edge  
  ACCEPT;  
}
```

# Nondeterministic Acceptance

- $(G, K)$  is in the “language” CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.
- Definition of acceptance by a nondeterministic algorithm:
  - ▶ An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.

# Nondeterministic Acceptance

- $(G, K)$  is in the “language” CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.
- Definition of acceptance by a nondeterministic algorithm:
  - ▶ An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.
- An unrealistic model of computation.
  - ▶ There are an exponential number of possible choices, but only one must accept for the instance to be accepted.

# Nondeterministic Acceptance

- $(G, K)$  is in the “language” CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.
- Definition of acceptance by a nondeterministic algorithm:
  - ▶ An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.
- An unrealistic model of computation.
  - ▶ There are an exponential number of possible choices, but only one must accept for the instance to be accepted.
- Nondeterminism is a useful concept
  - ▶ It provides insight into the nature of certain hard problems.

# Class $\mathcal{NP}$

- The class of languages accepted by a nondeterministic algorithm in polynomial time is called  $\mathcal{NP}$ .
- There are an **exponential** number of different executions of nd-CLIQUE on a single instance, but any one execution requires only **polynomial time** in the size of that instance.
- Time complexity of nondeterministic algorithm is greatest amount of time required by any **one** of its executions.

# Class $\mathcal{NP}$ (cont)

## Alternative Interpretation:

- $\mathcal{NP}$  is the class of algorithms that — never mind how we got the answer — can check if the answer is correct in polynomial time.
- If you cannot verify an answer in polynomial time, you cannot hope to find the right answer in polynomial time!



# How to Get Famous

Clearly,  $\mathcal{P} \subset \mathcal{NP}$ .

## Extra Credit Problem:

- Prove or disprove:  $\mathcal{P} = \mathcal{NP}$ .

This is important because there are many natural decision problems in  $\mathcal{NP}$  for which no  $\mathcal{P}$  (tractable) algorithm is known.

# $\mathcal{NP}$ -completeness

A theory based on identifying problems that are as hard as any problems in  $\mathcal{NP}$ .

The next best thing to knowing whether  $\mathcal{P} = \mathcal{NP}$  or not.

A decision problem  $A$  is  $\mathcal{NP}$ -hard if every problem in  $\mathcal{NP}$  is polynomially reducible to  $A$ , that is, for all

$$B \in \mathcal{NP}, \quad B \leq_p A.$$

A decision problem  $A$  is  $\mathcal{NP}$ -complete if  $A \in \mathcal{NP}$  and  $A$  is  $\mathcal{NP}$ -hard.

# Satisfiability

Let  $E$  be a Boolean expression over variables  $x_1, x_2, \dots, x_n$  in conjunctive normal form (CNF), that is, an AND of ORs.

$$E = (x_5 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_3} + x_6).$$

A variable or its negation is called a **literal**.

Each sum is called a **clause**.

SATISFIABILITY (SAT):

- Instance: A Boolean expression  $E$  over variables  $x_1, x_2, \dots, x_n$  in CNF.
- Question: Is  $E$  satisfiable?

# Satisfiability

Let  $E$  be a Boolean expression over variables  $x_1, x_2, \dots, x_n$  in conjunctive normal form (CNF), that is, an AND of ORs.

$$E = (x_5 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_3} + x_6).$$

A variable or its negation is called a **literal**.

Each sum is called a **clause**.

SATISFIABILITY (SAT):

- Instance: A Boolean expression  $E$  over variables  $x_1, x_2, \dots, x_n$  in CNF.
- Question: Is  $E$  satisfiable?

**Cook's Theorem**: SAT is  $\mathcal{NP}$ -complete.

# Proof Sketch

$\text{SAT} \in \mathcal{NP}$ :

- A non-deterministic algorithm **guesses** a truth assignment for  $x_1, x_2, \dots, x_n$  and **checks** whether  $E$  is true in polynomial time.
- It accepts iff there is a satisfying assignment for  $E$ .

# Proof Sketch

SAT  $\in \mathcal{NP}$ :

- A non-deterministic algorithm **guesses** a truth assignment for  $x_1, x_2, \dots, x_n$  and **checks** whether  $E$  is true in polynomial time.
- It accepts iff there is a satisfying assignment for  $E$ .

SAT is  $\mathcal{NP}$ -hard:

- Start with an arbitrary problem  $B \in \mathcal{NP}$ .
- We know there is a polynomial-time, nondeterministic algorithm to accept  $B$ .
- Cook showed how to transform an instance  $X$  of  $B$  into a Boolean expression  $E$  that is satisfiable if the algorithm for  $B$  accepts  $X$ .

# Implications

(1) Since SAT is  $\mathcal{NP}$ -complete, we have not defined an empty concept.

# Implications

- (1) Since SAT is  $\mathcal{NP}$ -complete, we have not defined an empty concept.
- (2) If  $\text{SAT} \in \mathcal{P}$ , then  $\mathcal{P} = \mathcal{NP}$ .



# Implications

- (1) Since SAT is  $\mathcal{NP}$ -complete, we have not defined an empty concept.
- (2) If  $\text{SAT} \in \mathcal{P}$ , then  $\mathcal{P} = \mathcal{NP}$ .
- (3) If  $\mathcal{P} = \mathcal{NP}$ , then  $\text{SAT} \in \mathcal{P}$ .

# Implications

- (1) Since SAT is  $\mathcal{NP}$ -complete, we have not defined an empty concept.
- (2) If  $\text{SAT} \in \mathcal{P}$ , then  $\mathcal{P} = \mathcal{NP}$ .
- (3) If  $\mathcal{P} = \mathcal{NP}$ , then  $\text{SAT} \in \mathcal{P}$ .
- (4) If  $A \in \mathcal{NP}$  and  $B$  is  $\mathcal{NP}$ -complete, then  $B \leq_p A$  implies  $A$  is  $\mathcal{NP}$ -complete.

# Implications

- (1) Since SAT is  $\mathcal{NP}$ -complete, we have not defined an empty concept.
- (2) If  $\text{SAT} \in \mathcal{P}$ , then  $\mathcal{P} = \mathcal{NP}$ .
- (3) If  $\mathcal{P} = \mathcal{NP}$ , then  $\text{SAT} \in \mathcal{P}$ .
- (4) If  $A \in \mathcal{NP}$  and  $B$  is  $\mathcal{NP}$ -complete, then  $B \leq_p A$  implies  $A$  is  $\mathcal{NP}$ -complete.

Proof:

- Let  $C \in \mathcal{NP}$ .
- Then  $C \leq_p B$  since  $B$  is  $\mathcal{NP}$ -complete.
- Since  $B \leq_p A$  and  $\leq_p$  is transitive,  $C \leq_p A$ .
- Therefore,  $A$  is  $\mathcal{NP}$ -hard and, finally,  $\mathcal{NP}$ -complete.

# Implications (cont)

(5) This gives a simple two-part strategy for showing a decision problem  $A$  is  $\mathcal{NP}$ -complete.

(a) Show  $A \in \mathcal{NP}$ .

(b) Pick an  $\mathcal{NP}$ -complete problem  $B$  and show  $B \leq_p A$ .

# $\mathcal{NP}$ -completeness Proof Template

To show that decision problem  $B$  is  $\mathcal{NP}$ -complete:

- ①  $B \in \mathcal{NP}$ 
  - ▶ Give a polynomial time, non-deterministic algorithm that accepts  $B$ .
    - ① Given an instance  $X$  of  $B$ , **guess** evidence  $Y$ .
    - ② **Check** whether  $Y$  is evidence that  $X \in B$ . If so, accept  $X$ .

# $\mathcal{NP}$ -completeness Proof Template

To show that decision problem  $B$  is  $\mathcal{NP}$ -complete:

①  $B \in \mathcal{NP}$

- ▶ Give a polynomial time, non-deterministic algorithm that accepts  $B$ .

① Given an instance  $X$  of  $B$ , **guess** evidence  $Y$ .

② **Check** whether  $Y$  is evidence that  $X \in B$ . If so, accept  $X$ .

②  $B$  is  $\mathcal{NP}$ -hard.

- ▶ Choose a known  $\mathcal{NP}$ -complete problem,  $A$ .
- ▶ Describe a polynomial-time transformation  $T$  of an **arbitrary** instance of  $A$  to a [not necessarily arbitrary] instance of  $B$ .
- ▶ Show that  $X \in A$  if and only if  $T(X) \in B$ .

# 3-SATISFIABILITY (3SAT)

**Instance:** A Boolean expression  $E$  in CNF such that each clause contains exactly 3 literals.

**Question:** Is there a satisfying assignment for  $E$ ?

A special case of SAT.

One might hope that 3SAT is easier than SAT.

# 3SAT is $\mathcal{NP}$ -complete

(1)  $3\text{SAT} \in \mathcal{NP}$ .

```
procedure nd-3SAT(E) {  
  for (i = 1 to n)  
    x[i] = nd-choice(TRUE, FALSE);  
  Evaluate E for the guessed truth assignment.  
  if (E evaluates to TRUE)  
    ACCEPT;  
  else  
    REJECT;  
}
```

nd-3SAT is a polynomial-time nondeterministic algorithm that accepts 3SAT.



# Proving 3SAT $\mathcal{NP}$ -hard

- 1 Choose SAT to be the known  $\mathcal{NP}$ -complete problem.
  - ▶ We need to show that  $\text{SAT} \leq_p 3\text{SAT}$ .
- 2 Let  $E = C_1 \cdot C_2 \cdots C_k$  be any instance of SAT.

Strategy: Replace any clause  $C_i$  that does not have exactly 3 literals with two or more clauses having exactly 3 literals.

Let  $C_i = y_1 + y_2 + \cdots + y_j$  where  $y_1, \cdots, y_j$  are literals.

(a)  $j = 1$

- Replace  $(y_1)$  with

$$(y_1 + v + w) \cdot (y_1 + \bar{v} + w) \cdot (y_1 + v + \bar{w}) \cdot (y_1 + \bar{v} + \bar{w})$$

where  $v$  and  $w$  are new variables.

## Proving 3SAT $\mathcal{NP}$ -hard (cont)

(b)  $j = 2$

- Replace  $(y_1 + y_2)$  with  $(y_1 + y_2 + z) \cdot (y_1 + y_2 + \bar{z})$  where  $z$  is a new variable.

(c)  $j > 3$

- Relace  $(y_1 + y_2 + \dots + y_j)$  with

$$(y_1 + y_2 + z_1) \cdot (y_3 + \bar{z}_1 + z_2) \cdot (y_4 + \bar{z}_2 + z_3) \cdots \\ (y_{j-2} + \bar{z}_{j-4} + z_{j-3}) \cdot (y_{j-1} + y_j + \bar{z}_{j-3})$$

where  $z_1, z_2, \dots, z_{j-3}$  are new variables.

- After replacements made for each  $C_i$ , a Boolean expression  $E'$  results that is an instance of 3SAT.
- The replacement clearly can be done by a polynomial-time deterministic algorithm.

## Proving 3SAT $\mathcal{NP}$ -hard (cont)

- (3) Show  $E$  is satisfiable iff  $E'$  is satisfiable.
- Assume  $E$  has a satisfying truth assignment.
  - Then that extends to a satisfying truth assignment for cases (a) and (b).
  - In case (c), assume  $y_m$  is assigned “true”.
  - Then assign  $z_t, t \leq m - 2$ , true and  $z_k, t \geq m - 1$ , false.
  - Then all the clauses in case (c) are satisfied.

# Proving 3SAT $\mathcal{NP}$ -hard (cont)

- Assume  $E'$  has a satisfying assignment.
- By restriction, we have truth assignment for  $E$ .
  - (a)  $y_1$  is necessarily true.
  - (b)  $y_1 + y_2$  is necessarily true.
  - (c) Proof by contradiction:
    - ★ If  $y_1, y_2, \dots, y_j$  are all false, then  $z_1, z_2, \dots, z_{j-3}$  are all true.
    - ★ But then  $(y_{j-1} + y_{j-2} + \overline{z_{j-3}})$  is false, a contradiction.

We conclude  $\text{SAT} \leq 3\text{SAT}$  and 3SAT is  $\mathcal{NP}$ -complete.

# Tree of Reductions



Reductions go down the tree.

Proofs that each problem  $\in \mathcal{NP}$  are straightforward.

# Perspective

The reduction tree gives us a collection of 12 diverse  $\mathcal{NP}$ -complete problems.

The complexity of all these problems depends on the complexity of any one:

- If any  $\mathcal{NP}$ -complete problem is tractable, then they all are.

This collection is a good place to start when attempting to show a decision problem is  $\mathcal{NP}$ -complete.

Observation: If we find a problem is  $\mathcal{NP}$ -complete, then we should do something other than try to find a  $\mathcal{P}$ -time algorithm.

# SAT $\leq_p$ CLIQUE

- (1) Easy to show CLIQUE in  $\mathcal{NP}$ .
- (2) An instance of SAT is a Boolean expression

$$B = C_1 \cdot C_2 \cdots C_m,$$

where

$$C_i = y[i, 1] + y[i, 2] + \cdots + y[i, k_i].$$

Transform this to an instance of CLIQUE  $G = (V, E)$  and  $K$ .

$$V = \{v[i, j] \mid 1 \leq i \leq m, 1 \leq j \leq k_i\}$$

Two vertices  $v[i_1, j_1]$  and  $v[i_2, j_2]$  are adjacent in  $G$  if  $i_1 \neq i_2$   
AND EITHER  $y[i_1, j_1]$  and  $y[i_2, j_2]$  are the same literal  
OR  $y[i_1, j_1]$  and  $y[i_2, j_2]$  have different underlying variables.  
 $K = m$ .

## SAT $\leq_p$ CLIQUE (cont)

Example:  $B = (x + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot (y + \bar{z})$ .

$K = 3$ .

(3)  $B$  is satisfiable iff  $G$  has clique of size  $\geq K$ .

- $B$  is satisfiable implies there is a truth assignment such that  $y[i, j_i]$  is true for each  $i$ .
- But then  $v[i, j_i]$  must be in a clique of size  $K = m$ .
- If  $G$  has a clique of size  $\geq K$ , then the clique must have size exactly  $K$  and there is one vertex  $v[i, j_i]$  in the clique for each  $i$ .
- There is a truth assignment making each  $y[i, j_i]$  true. That truth assignment satisfies  $B$ .

We conclude that CLIQUE is  $\mathcal{NP}$ -hard, therefore  $\mathcal{NP}$ -complete.



# Co- $\mathcal{NP}$

- Note the asymmetry in the definition of  $\mathcal{NP}$ .
  - ▶ The non-determinism can identify a clique, and you can verify it.
  - ▶ But what if the correct answer is “NO”? How do you verify that?
- Co- $\mathcal{NP}$ : The complements of problems in  $\mathcal{NP}$ .
  - ▶ Is a boolean expression **always** false?
  - ▶ Is there no clique of size  $k$ ?
- It seems unlikely that  $\mathcal{NP} = \text{co-}\mathcal{NP}$ .

# Is $\mathcal{NP}$ -complete = $\mathcal{NP}$ ?

- It has been proved that if  $\mathcal{P} \neq \mathcal{NP}$ , then  $\mathcal{NP}$ -complete  $\neq \mathcal{NP}$ .
- The following problems are not known to be in  $\mathcal{P}$  or  $\mathcal{NP}$ , but seem to be of a type that makes them unlikely to be in  $\mathcal{NP}$ .
  - ▶ GRAPH ISOMORPHISM: Are two graphs isomorphic?
  - ▶ COMPOSITE NUMBERS: For positive integer  $K$ , are there integers  $m, n > 1$  such that  $K = mn$ ?
  - ▶ LINEAR PROGRAMMING

# PARTITION $\leq_p$ KNAPSACK

PARTITION is a special case of KNAPSACK in which

$$K = \frac{1}{2} \sum_{a \in A} s(a)$$

assuming  $\sum s(a)$  is even.

Assuming PARTITION is  $\mathcal{NP}$ -complete, KNAPSACK is  $\mathcal{NP}$ -complete.

# “Practical” Exponential Problems

- What about our  $O(KN)$  dynamic prog algorithm?

# “Practical” Exponential Problems

- What about our  $O(KN)$  dynamic prog algorithm?
- Input size for KNAPSACK is  $O(N \log K)$ 
  - ▶ Thus  $O(KN)$  is exponential in  $N \log K$ .
- The dynamic programming algorithm counts through numbers  $1, \dots, K$ . Takes exponential time when measured by number of bits to represent  $K$ .

# “Practical” Exponential Problems

- What about our  $O(KN)$  dynamic prog algorithm?
- Input size for KNAPSACK is  $O(N \log K)$ 
  - ▶ Thus  $O(KN)$  is exponential in  $N \log K$ .
- The dynamic programming algorithm counts through numbers  $1, \dots, K$ . Takes exponential time when measured by number of bits to represent  $K$ .
- If  $K$  is “small” ( $K = O(p(N))$ ), then algorithm has complexity polynomial in  $N$  and is truly polynomial in input size.
- An algorithm that is polynomial-time if the numbers IN the input are “small” (as opposed to number OF inputs) is called a pseudo-polynomial time algorithm.

# “Practical” Problems (cont)

- Lesson: While KNAPSACK is  $\mathcal{NP}$ -complete, it is often not that hard.
- Many  $\mathcal{NP}$ -complete problems have no pseudo-polynomial time algorithm unless  $\mathcal{P} = \mathcal{NP}$ .

# Coping with $\mathcal{NP}$ -completeness

- (1) Find subproblems of the original problem that have polynomial-time algorithms.
- (2) Approximation algorithms.
- (3) Randomized Algorithms.
- (4) Backtracking; Branch and Bound.
- (5) Heuristics.
  - Greedy.
  - Simulated Annealing.
  - Genetic Algorithms.



# Subproblems

Restrict attention to special classes of inputs.

Examples:

- VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).

# Subproblems

Restrict attention to special classes of inputs.

Examples:

- VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).
- 2-SATISFIABILITY, 2-DIMENSIONAL MATCHING and EXACT COVER BY 2-SETS all have polynomial time algorithms.

# Subproblems

Restrict attention to special classes of inputs.

Examples:

- VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).
- 2-SATISFIABILITY, 2-DIMENSIONAL MATCHING and EXACT COVER BY 2-SETS all have polynomial time algorithms.
- PARTITION and KNAPSACK have polynomial time algorithms if the numbers in an instance are all  $O(p(n))$ .

# Subproblems

Restrict attention to special classes of inputs.

Examples:

- VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).
- 2-SATISFIABILITY, 2-DIMENSIONAL MATCHING and EXACT COVER BY 2-SETS all have polynomial time algorithms.
- PARTITION and KNAPSACK have polynomial time algorithms if the numbers in an instance are all  $O(p(n))$ .
- However, HAMILTONIAN CIRCUIT and 3-COLORABILITY remain  $\mathcal{NP}$ -complete even for a planar graph.

# Backtracking

We may view a nondeterministic algorithm executing on a particular instance as a tree:

- 1 Each edge represents a particular nondeterministic choice.
- 2 The checking occurs at the leaves.

Example:

Each leaf represents a different set  $S$ . Checking that  $S$  is a clique of size  $\geq K$  can be done in polynomial time.

## Backtracking (cont)

Backtracking can be viewed as an in-order traversal of this tree with two criteria for stopping.

- 1 A leaf that accepts is found.
- 2 A partial solution that could not possibly lead to acceptance is reached.

Example:

There cannot possibly be a set  $S$  of cardinality  $\geq 2$  under this node, so backtrack.

Since  $(1, 2) \notin E$ , no  $S$  under this node can be a clique, so backtrack.

# Branch and Bound

- For optimization problems.  
More sophisticated kind of backtracking.
- Use the best solution found so far as a **bound** that controls backtracking.
- Example Problem: Given a graph  $G$ , find a minimum vertex cover of  $G$ .
- Computation tree for nondeterministic algorithm is similar to CLIQUE.
  - ▶ Every leaf represents a different subset  $S$  of the vertices.
- Whenever a leaf is reached and it contains a vertex cover of size  $B$ ,  $B$  is an upper bound on the size of the minimum vertex cover.
  - ▶ Use  $B$  to prune any future tree nodes having size  $\geq B$ .
- Whenever a smaller vertex cover is found, update  $B$ .

# Branch and Bound (cont)

- Improvement:
  - ▶ Use a fast, greedy algorithm to get a minimal (not minimum) vertex cover.
  - ▶ Use this as the initial bound  $B$ .
- While Branch and Bound is better than a brute-force exhaustive search, it is usually exponential time, hence impractical for all but the smallest instances.
  - ▶ ... if we insist on an optimal solution.
- Branch and Bound often practical as an approximation algorithm where the search terminates when a “good enough” solution is obtained.



# Approximation Algorithms

Seek algorithms for optimization problems with a guaranteed bound on the quality of the solution.

VERTEX COVER: Given a graph  $G = (V, E)$ , find a vertex cover of minimum size.

Let  $M$  be a maximal (not necessarily maximum) matching in  $G$  and let  $V'$  be the set of matched vertices.

If  $OPT$  is the size of a minimum vertex cover, then

$$|V'| \leq 2OPT$$

because at least one endpoint of every matched edge must be in **any** vertex cover.

# Bin Packing

We have numbers  $x_1, x_2, \dots, x_n$  between 0 and 1 as well as an unlimited supply of bins of size 1.

Problem: Put the numbers into as few bins as possible so that the sum of the numbers in any one bin does not exceed 1.

Example: Numbers  $3/4, 1/3, 1/2, 1/8, 2/3, 1/2, 1/4$ .

Optimal solution:  $[3/4, 1/8], [1/2, 1/3], [1/2, 1/4], [2/3]$ .

# First Fit Algorithm

Place  $x_1$  into the first bin.

For each  $i, 2 \leq i \leq n$ , place  $x_i$  in the first bin that will contain it.

No more than 1 bin can be left less than half full.

The number of bins used is no more than twice the sum of the numbers.

The sum of the numbers is a lower bound on the number of bins in the optimal solution.

Therefore, first fit is no more than twice the optimal number of bins.

# First Fit Does Poorly

Let  $\epsilon$  be very small, e.g.,  $\epsilon = .00001$ .

Numbers (in this order):

- 6 of  $(1/7 + \epsilon)$ .
- 6 of  $(1/3 + \epsilon)$ .
- 6 of  $(1/2 + \epsilon)$ .

First fit returns:

- 1 bin of [6 of  $1/7 + \epsilon$ ]
- 3 bins of [2 of  $1/3 + \epsilon$ ]
- 6 bins of [ $1/2 + \epsilon$ ]

Optimal solution is 6 bins of [ $1/7 + \epsilon, 1/3 + \epsilon, 1/2 + \epsilon$ ].

First fit is  $5/3$  larger than optimal.

# Decreasing First Fit

It can be proved that the worst-case performance of first-fit is  $17/10$  times optimal.

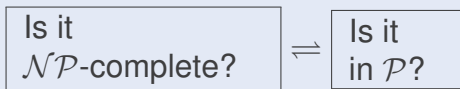
Use the following heuristic:

- Sort the numbers in decreasing order.
- Apply first fit.
- This is called decreasing first fit.

The worst case performance of decreasing first fit is close to  $11/9$  times optimal.

# Summary

- The theory of  $\mathcal{NP}$ -completeness gives us a technique for separating tractable from (probably) intractable problems.
- When faced with a new problem requiring algorithmic solution, our thought process might resemble this scheme:



- Alternately think about each question. Lack of progress on either question might give insights into the answer to the other question.
- Once an affirmative answer is obtained to one of these questions, one of two strategies is followed.

# Strategies

(1) The problem is in  $\mathcal{P}$ .

- This means there are polynomial-time algorithms for the problem, and presumably we know at least one.
- So, apply the techniques learned in this course to analyze the algorithms and improve them to find the lowest time complexity we can.

(2) The problem is  $\mathcal{NP}$ -complete.

- Apply the strategies for coping with  $\mathcal{NP}$ -completeness.
- Especially, find subproblems that are in  $\mathcal{P}$ , or find approximation algorithms.

# Algebraic and Numeric Algorithms

- Measuring cost of arithmetic and numerical operations:
  - ▶ Measure size of input in terms of **bits**.
- Algebraic operations:
  - ▶ Measure size of input in terms of **numbers**.
- In both cases, measure complexity in terms of basic arithmetic operations:  $+$ ,  $-$ ,  $*$ ,  $/$ .
  - ▶ Sometimes, measure complexity in terms of bit operations to account for large numbers.
- Size of numbers may be related to problem size:
  - ▶ Pointers, counters to objects.
  - ▶ Resolution in geometry/graphics (to distinguish between object positions).



# Exponentiation

Given positive integers  $n$  and  $k$ , compute  $n^k$ .

Algorithm:

```
p = 1;
for (i=1 to k)
    p = p * n;
```

Analysis:

- Input size:  $\Theta(\log n + \log k)$ .
- Time complexity:  $\Theta(k)$  multiplications.
- This is **exponential** in input size.

# Faster Exponentiation

Write  $k$  as:

$$k = b_t 2^t + b_{t-1} 2^{t-1} + \dots + b_1 2 + b_0, b \in \{0, 1\}.$$

Rewrite as:

$$k = ((\dots (b_t 2 + b_{t-1}) 2 + \dots + b_2) 2 + b_1) 2 + b_0.$$

New algorithm:

```
p = n;  
for (i = t-1 downto 0)  
    p = p * p * exp(n, b[i])
```

**Analysis:**

- Time complexity:  $\Theta(t) = \Theta(\log k)$  multiplications.
- This is **exponentially** better than before.

# Greatest Common Divisor

- The Greatest Common Divisor (GCD) of two integers is the greatest integer that divides both evenly.
- Observation: If  $k$  divides  $n$  and  $m$ , then  $k$  divides  $n - m$ .
- So,

$$f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n).$$

- Observation: There exists  $k$  and  $l$  such that

$$n = km + l \text{ where } m > l \geq 0.$$

$$n = \lfloor n/m \rfloor m + n \bmod m.$$

- So,

$$f(n, m) = f(m, l) = f(m, n \bmod m).$$

# GCD Algorithm

$$f(n, m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {  
    if (m == 0) return n;  
    return LCF(m, n % m);  
}
```

# Analysis of GCD

- How big is  $n \bmod m$  relative to  $n$ ?

$$\begin{aligned}n \geq m &\Rightarrow n/m \geq 1 \\&\Rightarrow 2 \lfloor n/m \rfloor > n/m \\&\Rightarrow m \lfloor n/m \rfloor > n/2 \\&\Rightarrow n - n/2 > n - m \lfloor n/m \rfloor = n \bmod m \\&\Rightarrow n/2 > n \bmod m\end{aligned}$$

- The first argument must be halved in no more than 2 iterations.
- Total cost: