CSE 421
Introduction to Algorithms
Winter 2000

NP-Completeness
(Chapter 11)

Easy Problems vs. Hard Problems

Easy - problems whose worst case running
time is bounded by some polynomial in the
size of the input.

Easy = Efficient

Hard - problems that cannot be solved
efficiently.

The class P

Definition: $P =$ set of problems solvable by
computers in polynomial time.
i.e. $T(n) = O(n^k)$ for some $k$.

• These problems are sometimes called
tractable problems.

Examples: sorting, SCC, matching, max flow,
shortest path, MST.

Is P a good definition of efficient?

Is $O(n^{100})$ efficient? Is $O(10^n)$ efficient?

Is $O(2^n)$ really so bad?

So we have:

$P = \text{“easy” = efficient = tractable = solvable in polynomial-time.}$

Decision Problems

• Technically, we will restrict our discussion to
decision problems - problems that have an
answer of either yes or no.

• Most problems can be easily converted to
decision problems:
  – Example: Instead of looking for the size of the
    shortest path from $s$ to $t$ in a graph $G$, we ask:
    “Is there a path from $s$ to $t$ of length $\leq k$?”
  – If we know how to solve the decision problem,
    then we can usually solve the original problem.

Examples of Decision Problems in P

Big Flow

Given: graph $G$ with edge lengths, vertices $s$ and $t$, integer $k$.

Question: Is there an $s$-$t$ flow of length $\geq k$?

Small Spanning Tree

Given: weighted undirected graph $G$, integer $k$.

Question: Is there a spanning tree of weight $\leq k$?
Decision problem as a Language-recognition problem

- Let $U$ be the set of all possible inputs to the decision problem.
- $L \subseteq U = \text{the set of all inputs for which the answer to the problem is yes.}$
- We call $L$ the language corresponding to the problem. (problem = language)
- The decision problem is thus:
  - to recognize whether or not a given input belongs to $L = \text{the language recognition problem.}$

The class NP

Definition: NP = set of problems solvable by a non-deterministic algorithm in polynomial time.

Another way of saying this:

NP = The class of problems whose solution can be verified in polynomial time.
NP = "non-deterministic polynomial"

Examples: all of problems in P plus: SAT, TSP, Hamiltonian cycle, bin packing, vertex cover.

Complexity Classes

Complexity Classes

NP

P

NP = Polynomial-time verifiable

P = Polynomial-time solvable

Verifying Solutions

Given a problem and a potential solution, verify if the solution is correct in polynomial time.

In general, guess a solution, and then check if the guess is correct in polynomial time.

Examples of Problems in NP

Vertex Cover

A vertex cover of $G$ is a set of vertices such that every edge in $G$ is incident to at least one of these vertices. Example:

Question: Given a graph $G$, integer $k$, determine whether $G$ has a vertex cover containing $\leq k$ vertices?
Verify: Given a set of $\leq k$ vertices, does it cover every edge? (Guess and check in polynomial time.)

Examples of Problems in NP

Satisfiability (SAT)

A Boolean formula in conjunctive normal form (CNF) is satisfiable if there exists a truth assignment of 0's and 1's to its variables such that the value of the expression is 1. Example:

$S = (x \cdot y \cdot z) + (\neg x \cdot y \cdot z) + (\neg x \cdot \neg y \cdot z)$

Question: Given a Boolean formula in CNF, is it satisfiable?
Verify: Given a truth assignment, does it satisfy the formula? (Guess and check in polynomial time.)
Problems in P can also be verified in polynomial-time

**Shortest Path**: Given a graph $G$ with edge lengths, is there a path from $s$ to $t$ of length $\leq k$?

**Verify**: Given a path from $s$ to $t$, is its length $\leq k$?

**Small Spanning Tree**: Given a weighted undirected graph $G$, is there a spanning tree of weight $\leq k$?

**Verify**: Given a spanning tree, is its weight $\leq k$?

Nondeterminism

- A **nondeterministic algorithm** has all the “regular” operations of any other algorithm available to it.
- In addition, it has a powerful primitive, the **nd-choice primitive**.
- The nd-choice primitive is associated with a fixed number of choices, such that each choice causes the algorithm to follow a different computation path.

Nondeterminism (cont.)

- A **nondeterministic algorithm** consists of an interleaving of regular deterministic steps and uses of the nd-choice primitive.
- We require that:
  - The algorithm have at least one “good” path sequence of choices for every $x \in L$.
  - For all $x \not\in L$, we reach a reject outcome in all paths.

We say that a nondeterministic algorithm recognizes a language $L$ if:

Given an input $x$, it is possible to convert each nd-choice encountered during the execution of the algorithm into a real choice such that the outcome of the algorithm will be to accept $x$, iff $x \in L$.

The class NP-complete

**Definition**: NP-complete = set of problems in NP that (we are pretty sure) cannot be solved in polynomial time.

These are thought of as the hardest problems in the class NP.

**Interesting fact**: If any one NP-complete problem could be solved in polynomial time, then all NP-complete problems could be solved in polynomial time.

Complexity Classes

NP = Polynomial-time verifiable
P = Polynomial-time solvable
NP-Complete = “Hardest” problems in NP
The class NP-complete (cont.)

- Hundreds of important problems have been shown to be NP-complete.

**Interesting Fact:** The general belief is that there is no efficient algorithm for any NP-complete problem, but no proof of that belief is known.

**Examples:** SAT, clique, vertex cover, Hamiltonian cycle, TSP, bin packing.

Complexity Classes of Problems

Does $P = NP$?

- This is an open question.
- To show that $P = NP$, we have to show that every problem that belongs to NP can be solved by a polynomial time deterministic algorithm.
- No one has shown this yet.
- (It seems unlikely to be true.)

Dealing with NP-complete Problems

**What if I think my problem is not in $P$?**

Here is what you do:
1) Prove your problem is NP-complete.
2) Come up with an algorithm to solve the problem approximately.

I will cover (1) this week, Larry will cover (2) next week.

Reductions: a useful tool

**Definition:** To reduce A to B means to figure out how to solve A, given a subroutine solving B.

**Example:** reduce MEDIAN to SORT
Solution: sort, then select $(n/2)$ th

**Example:** reduce SORT to FIND_MAX
Solution: FIND_MAX, remove it, repeat

**Example:** reduce MEDIAN to FIND_MAX
Solution: transitivity: compose solutions above.
More Examples of reductions

**Example:** reduce BIPARTITE_MATCHING to MAX_FLOW

Is there a matching of size k? Is there a flow of size k?

\[
\begin{array}{c}
u \\ v \\ f \\ s \\ t \\ \text{All capacities } = 1
\end{array}
\]

Polynomial-Time Reductions

**Definition:** Let \( L_1 \) and \( L_2 \) be two languages from the input spaces \( U_1 \) and \( U_2 \).

We say that \( L_1 \) is **polynomially reducible** to \( L_2 \) if there exists a polynomial-time algorithm \( f \) that converts each input \( u_1 \in U_1 \) to another input \( u_2 \in U_2 \) such that \( u_1 \in L_1 \) iff \( u_2 \in L_2 \).

\[
\begin{align*}
u_1 \in L_1 & \iff f(u_1) \in L_2
\end{align*}
\]

Polynomial-Time Reductions (cont.)

**Define:** \( A \leq_p B \) “\( A \) is polynomial-time reducible to \( B \)”, iff there is a polynomial-time computable function \( f \) such that:

\[
x \in A \iff f(x) \in B
\]

“complexity of \( A \) \leq complexity of \( B \)+ complexity of \( f \)”

\[
\begin{align*}
(1) \quad A \leq_p B \text{ and } B \in P & \implies A \in P \\
(2) \quad A \leq_p B \text{ and } A \notin P & \implies B \notin P \\
(3) \quad A \leq_p B \text{ and } B \leq_p C & \implies A \leq_p C \text{ (transitivity)}
\end{align*}
\]

Using an Algorithm for \( B \) to Decide \( A \)

**Algorithm to decide \( A \)**

\[
x \xrightarrow{\text{Algorithm to compute } f(x)} f(x) \xrightarrow{\text{Algorithm to decide } B} f(x) \in B? \quad x \in A?
\]

“If \( A \leq_p B \), and we can solve \( B \) in polynomial time, then we can solve \( A \) in polynomial time also.”

More Definitions

**Definition:** Problem \( B \) is **NP-hard** if every problem in NP is polynomially reducible to \( B \).

**Definition:** Problem \( B \) is **NP-complete** if:

1. \( B \) belongs to NP, and
2. \( B \) is NP-hard.
Proving a problem is NP-complete

- Technically, for condition (2) we have to show that every problem in NP is reducible to B. (yikes!) This sounds like a lot of work.
- For the very first NP-complete problem (SAT) this had to be proved directly.
- However, once we have one NP-complete problem, then we don’t have to do this every time.
- Why? Transitivity.

Re-stated Definition

**Lemma 11.3:** Problem $B$ is NP-complete if:
1. $B$ belongs to NP, and
2. $A$ is polynomial-time reducible to $B$, for some problem $A$ that is NP-complete.

That is, to show (2) given a new problem $B$, it is sufficient to show that SAT or any other NP-complete problem is polynomial-time reducible to $B$.

Usefulness of Transitivity

Now we only have to show $L' \leq_p L$, for some problem $L'$ is NP-complete, in order to show that $L$ is NP-hard. Why is this equivalent?

1. Since $L' \in NP$-complete, we know that $L'$ is NP-hard. That is: $\forall L'' \in NP$, we have $L'' \leq_p L'$
2. If we show $L' \leq_p L$, then by transitivity we know that: $\forall L'' \in NP$, we have $L'' \leq_p L$.

Thus $L$ is NP-hard.

The growth of the number of NP-complete problems

- Steve Cook (1971) showed that SAT was NP-complete.
- Richard Karp (1972) found 24 more NP-complete problems.
- Today there are hundreds of known NP-complete problems.
  - Garey and Johnson (1979) is a good source of NP-complete problems.

SAT is NP-complete

**Cook’s theorem:** SAT is NP-complete

**Satisfiability (SAT)**
A Boolean formula in conjunctive normal form (CNF) is satisfiable if there exists a truth assignment of 0’s and 1’s to its variables such that the value of the expression is 1. Example:

$S = (x \land y \land z) \lor (x \land y \land z) \lor (x \land \neg y \land \neg z)$

Example above is satisfiable. (We can see this by setting $x=1$, $y=1$ and $z=0$.)

SAT is NP-complete

Rough idea of proof:

1. **SAT is in NP** because we can guess a truth assignment and check that it satisfies the expression in polynomial time.
2. **SAT is NP-hard** because ....
SAT is NP-hard

- A Turing machine (even a nondeterministic one) and all of its operations on a given input can be "described" by a Boolean expression.
- That is, the expression will be satisfiable iff the Turing machine will terminate in an accepting state for the given input.
- Therefore, any NP algorithm can be described by an instance of a SAT problem.
- Thus: Cook's theorem: SAT is NP-complete.

How do you prove problem A is NP-complete?

1) Prove A is in NP: show that given a solution, it can be verified in polynomial time.
2) Prove that A is NP-hard:
   a) Select a known NP-complete problem B.
   b) Describe a polynomial time computable algorithm that computes a function f, mapping every instance of B to an instance of A.
   (that is: \( B \leq_p A \))
   c) Prove that every yes-instance of B maps to a yes-instance of A, and every no-instance of B maps to a no-instance of A.
   d) Prove that the algorithm computing \( f(B) \) runs in polynomial time.

Proof that problem A is NP-complete

1) Prove A is in NP: "Given a possible solution to A, I can verify its correctness in polynomial time."
2) Prove that A is NP-hard:
   a) "I will reduce known NP-complete problem B to A."
   b) "Let b be an arbitrary instance of problem B. Here is how you convert b to an instance \( a \) of problem A."
      Note: this method must work for ANY instance of B.
   c) "If \( a \) is a 'yes'-instance, then this implies that \( b \) is also a 'yes'-instance. Conversely, if \( b \) is a 'yes'-instance, then this implies that \( a \) is also a 'yes'-instance."
   d) "The conversion from \( B \) to \( A \) runs in polynomial time because…."

NP-complete problem: Vertex Cover

Input: Undirected graph \( G = (V, E) \), integer \( k \).
Output: True iff there is a subset \( C \) of \( V \) of size \( \leq k \) such that every edge in \( E \) is incident to at least one vertex in \( C \).

Example: Vertex cover of size \( \leq 2 \).

NP-complete problem: Clique

Input: Undirected graph \( G = (V, E) \), integer \( k \).
Output: True iff there is a subset \( C \) of \( V \) of size \( \geq k \) such that all vertices in \( C \) are connected to all other vertices in \( C \).

Example: Clique of size \( \geq 4 \)

NP-complete problem: Satisfiability (SAT)

Input: A Boolean formula in CNF form.
Output: True iff there is a truth assignment of 0s and 1s to the variables such that the value of the expression is 1.

Example: Formula \( S \) is satisfiable with the truth assignment \( x=1, y=1 \) and \( z=0 \).
\[
S = (x \lor y \lor z) \land (\neg x \lor y \lor z) \land (\neg x \lor y \lor z)
\]
**NP-complete problem: 3-Coloring**

**Input:** An undirected graph \( G = (V,E) \).

**Output:** True iff there is an assignment of colors to the vertices in \( G \) such that no two adjacent vertices have the same color. (using only 3 colors)

**Example:**

```
  o ---- o ---- o
     |    |    |
     o ---- o ---- o
```

**NP-complete problem: Knapsack**

**Input:** set of objects with weights and values, a maximum weight that can be carried and a desired value. (see p. 357 in Manber)

**Output:** True iff there is a subset of the objects with (total weight \( \leq \) allowable weight) and (total value \( \geq \) desired value).

**Example:** Items: \( \{a, b, c\} \), size(a)=3, size(b)=6, size(c)=4
value(a)=$30, value(b)=$24, value(c)=$18
Max weight = 10, Desired value = $50.

**Answer:** yes, \{a, b\}

**NP-complete problem: Partition**

**Input:** Set of items \( S \), each with an associated size. The sum of the items’ sizes is \( 2k \).

**Output:** True iff there is a subset of the items whose sizes add up to \( k \).

**Example:** \( S = \{2,3,1,10,4,6\} \). Is there a subset of items that sums to 13? (yes)