Binghamton University

EngiNet™

State University of New York

Thomas J. Watson

School of Engineering and Applied Science

WARNING

All rights reserved. No part of the course materials used in the instruction of this course may be reproduced in any form or by any electronic or mechanical means, including the use of information storage and retrieval systems, without written approval from the copyright owner.

©2005 Binghamton University
State University of New York

CS 575

Design and Analysis of Computer Algorithms

Professor Michal Cutler

Lecture 22

November 22, 2005

This class

- The theory of NP-completeness
  - Tractable and intractable problems
  - The classes P, NP and NPC

Classifying problems

- Classify problems as tractable or intractable.
- Problem is *tractable* if there **exists at least one** polynomial bound algorithm that solves it.
- An algorithm is *polynomial bound* if its worst case growth rate can be bound by a polynomial $p(n)$ in the size n of the problem

$$p(n) = a_n n^k + \ldots + a_1 n + a_0 \text{ where } k \text{ is a constant}$$

- Note: $O(\lg n) \subset O(n)$, $O(n \lg n) \subset O(n^2)$

Intractable problems

- Problem is *intractable* if it is not tractable.
  - **All** algorithms that solve the problem are not polynomial bound.
  - They have a worst case growth rate $f(n)$ which cannot be bound by a polynomial p(n) in the size n of the problem.
- For intractable problems the bounds are:

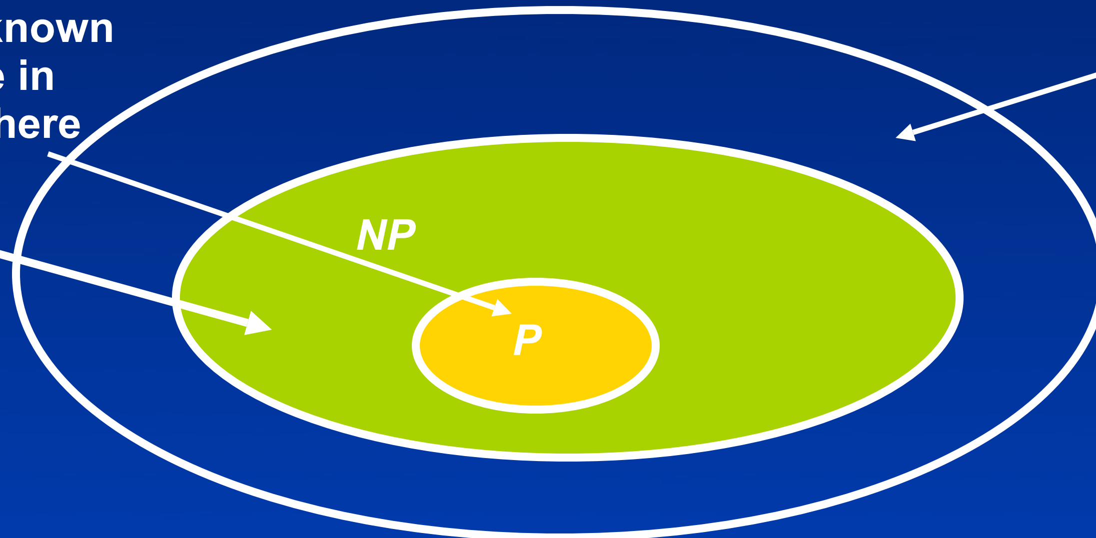
$$f(n) = c^n \text{, or } n^{\log n}\text{, etc.}$$

Why is this classification useful?

- If problem is intractable, no point in trying to find an efficient algorithm
- All algorithms will be too slow for **large inputs**.

Intractable problems

- Turing showed some problems are so hard that no algorithm can solve them (undecidable): Halting problem
- Other researchers showed some decidable problems from automata, mathematical logic, etc. are intractable: Presburger Arithmetic



Hard practical problems

- There are many practical problems for which no one has yet found a polynomial bound algorithm.
- Examples: traveling salesperson, 0/1 knapsack, graph coloring, bin packing etc.
- Most design automation problems such as testing and routing.
- Many networks, database and graph problems.

The theory of NP completeness

- The theory of NP-completeness enables showing that these problems are at least as hard as *NP-complete* problems
- Practical implication of knowing problem is NP-complete is that it is **probably** intractable (whether it is or not has not been proved yet)
- So any algorithm that solves it will probably be very slow for large inputs

We need to discuss

- Decision problems
- Converting optimization problems into decision problems
- The relationship between an optimization problem and its decision version
- The class P
- Verification algorithms
- The class NP
- The concept of polynomial transformations
- The class of NP-complete problems

Decision Problems

Decision Problems

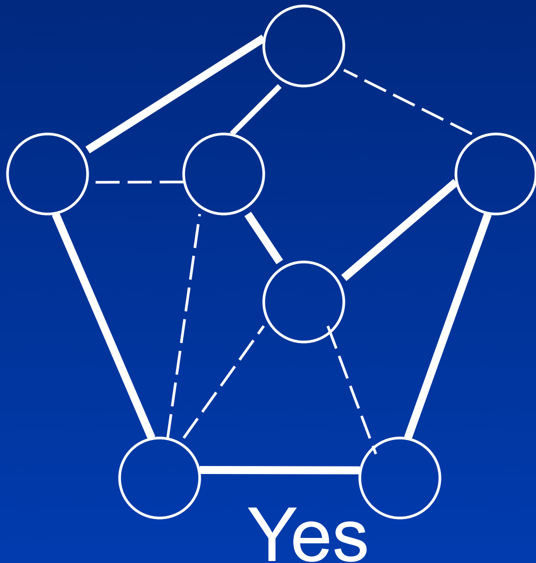
- A *decision* problem answers *yes or no* for a given input

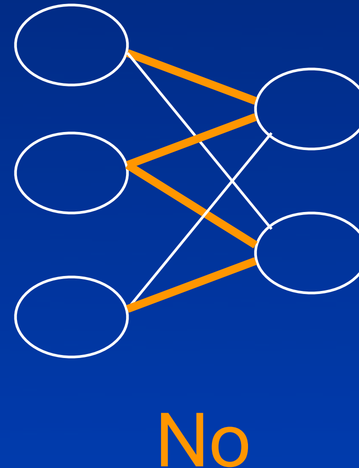
Decision Problems

- A *decision* problem answers *yes or no* for a given input
- Examples:
  - Given a graph *G* Is there a path from *s* to *t* of length at most *k*?
  - Does graph *G* contain a Hamiltonian cycle?
  - Given a graph *G* is it bipartite?
  - Given a bipartite graph does it contain a perfect match?
  - Can G be colored with at most k colors?
  - Given a flow network with supply and demand

A decision problem: HAMILTONIAN-CYCLE

- A *Hamiltonian cycle* of a graph *G* is a cycle that includes each vertex of the graph exactly once.
- Problem: Given a graph G, does G have a Hamiltonian cycle?





Converting to decision problems

- Optimization problems can be converted to decision problems (typically) by adding a bound B on the value to optimize, and asking the question:
  - Is there a solution whose value is at most B? (for a minimization problem)
  - Is there a solution whose value is at least B? (for a maximization problem)

An optimization problem: traveling salesman

- Given:
  - A finite set $C=\{c_1,\ldots,c_m\}$ of cities,
  - A distance function $d(c_i, c_j)$ of nonnegative numbers.
- Find the length of the **minimum** distance tour which includes every city exactly once

A decision problem for traveling salesman (TS)

- Given a finite set $C=\{c_1,\ldots,c_m\}$ of cities, a distance function $d(c_i, c_j)$ of nonnegative numbers and a bound B
- Is there a tour of all the cities (in which each city is visited exactly once) with total length **at most B**?
- There is no known polynomial bound algorithm for TS.

Relationship between optimization and decision version

- Have a solution to the optimization problem:
  - Find the optimal solution
  - Compare optimal solution to the bound B and answer “yes” or “no”.
- Example: Does G contain path from s to t of length <=B?

  D← Dijkstra(s) //length of shortest path from s to t
  return (D[t]<=B)

- **Tractable optimization problem → tractable decision problem**

Relationship between optimization and decision version

- **If the decision problem is "hard" → optimization problem is also "hard"**
  - Why? If the optimization was easy then the decision problem is easy.

The class P

The class P

- P is the class of decision problems that have polynomial bounded algorithms

The class P

- P is the class of decision problems that have polynomial bounded algorithms
- Problem in P
  - Given a weighted graph G, is there a spanning tree of weight at most B?
  - Is graph G a tree?
  - Is graph G acyclic?
  - Is G connected?
  - Can G be colored with two colors?
  - Is array A sorted?

Encoding and Time Complexity

- Run time is a function of problem size.
- An encoding is used to represent a problem instance.
- The size of a problem instance is the number of characters in a (reasonable) encoding of its input
- **The chosen encoding affects the time complexity of an algorithm!**

Effect of encoding on run time

- Assume: input to an algorithm is an integer *k*, and the algorithm's complexity is $\Theta(k)$
- A unary encoding of input k has size = k.
- A binary encoding of k has size = $\lfloor \lg k \rfloor + 1$.
- Unary encoding would make the algorithm linear
- Binary encoding shows the algorithm to be exponential (size <= lg k +1, $k >= 2^{size-1}$)
- We prefer to consider this algorithm exponential!

What is a reasonable encoding?

- Numbers should not have unary representation
- Binary, octal, decimal, ect, are reasonable
- Representation should not have “padding” to artificially increase problem size

Is the algorithm polynomial bound?

- The sieve algorithm for finding whether a number *n* is prime divides it by 2, 3, …, $\sqrt{n}$
- It is not polynomial bound

$\sqrt{n}$

- Notice that there is no difference in complexity if the encoding is in binary or in decimal.

Other definitions of P

- The class P can be defined as a class of formal languages:
  - P = { L | there exists an polynomial bounded algorithm A that decides L}
  - P = { L | L is accepted by a polynomial time algorithm}
- We can prove the second definition given the first

The goal of verification algorithms

- The goal of a verification algorithm for a problem is to verify a 1 answer for a given problem instance (i.e., if the answer is 1 the verification algorithm verifies this answer)

- The inputs to the verification algorithm are:
  - the original input (problem instance) and
  - a *certificate* (some representation of a solution if it exists)

Verification Algorithms

- A *verification algorithm*, takes a problem instance x and *answers* 1, if there **exists** a certificate y such that the answer for x with certificate y is 1

Polynomial bound verification algorithms

- Given a decision problem *d*.

- A verification algorithm for *d* is *polynomial bound*
  if given an *input x* to *d*, there exists a "short" certificate *y*,
  such that ($|y|=O(|x|^c)$) where c is a constant,
  and a polynomial bound algorithm A(x, y) that verifies an answer 1 for d with input x

Note: |y| is the size of the certificate, |x| is the size of the input

A verification algorithm for TS

- Given a problem instance x for TS and a certificate *y*
  - Check that *y* is indeed a cycle that includes every vertex exactly once
  - Compute the length of the cycle
  - Verify that the length of the cycle is at most B
- Is the size of y polynomial in the size of x?
- Is the verification algorithm polynomial?

The class NP

- NP is the class of decision problems for which there is a polynomial bounded verification algorithm
- It can be shown that:
  - all decision problems in P, and
  - decision problems such as traveling salesman, knapsack, bin pack, are also in NP

The relation between P and NP

- $P \subseteq NP$

- If a decision problem is solvable in polynomial time, a polynomial time verification algorithm can easily be designed that *ignores the certificate* and answers “yes” for all inputs with the answer “yes”.

The relation between P and NP

- **PATH**: Does G (non-negative weights) contain path from *s* to *t* of length <=B?

```
verPath(G, s, certificate)
    D← Dijkstra(s)
    if (D[t]<=B)
        return "yes"
```

The relation between P and NP (cont)

- It is not known whether P = NP.
- Problems in P can be *solved* "quickly"
- Problems in NP can be *verified* "quickly".
- It is probably easier to verify a solution than to solve a problem.
- Many researchers believe that P and NP are not the same class.

Reducibility

- We will study the relative hardness of languages through polynomial-time reducibility.
- A problem Q can be reduced to another problem Q' if any instance of Q can be rephrased as an instance of Q', such that the solution to the instance of Q' provides a solution to the instance of Q.

Example

- The problem of solving linear equations reduces to the problem of solving quadratic equations.
- Given an instance (a, b) to solve the linear equations problem $ax + b = 0$, we can create an instance (0, a, b) to solve the quadratic equations problem $0x^2+ax+b=0$, whose solution provides a solution to our original problem.

Polynomial reductions

- **Motivation**: The definition of NP-completeness uses the notion of *polynomial reductions* of one problem π_1 to another problem π_2, written as

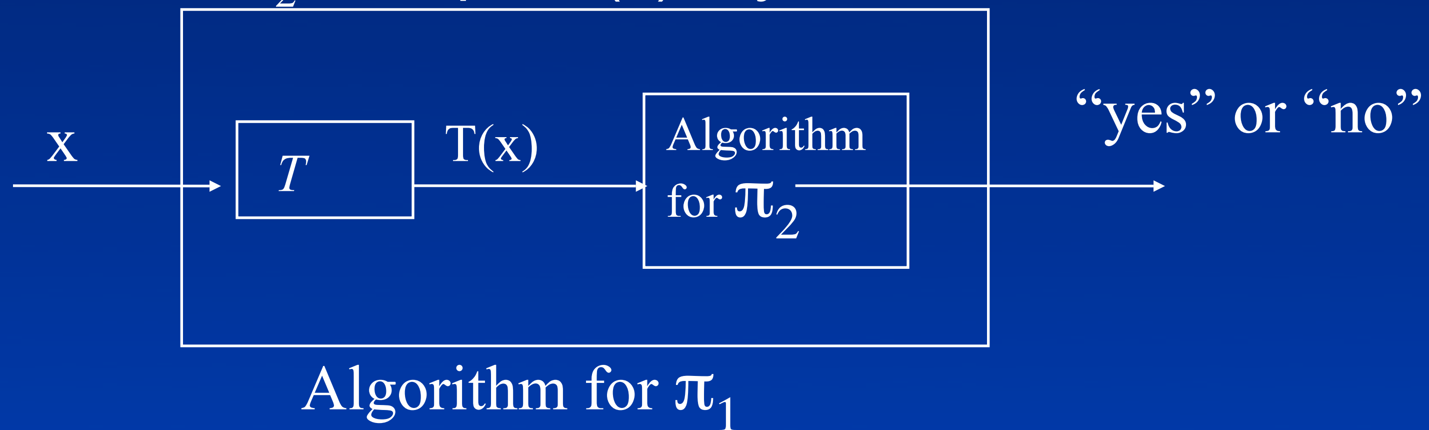
$$\pi_1 \propto \pi_2$$

- Let T be a function that converts any input x for decision problem π_1 into input T(x) for decision problem π_2

Polynomial reductions

T(x) is a *polynomial reduction* from π_1 to π_2 if:

1. T can be computed in polynomial bounded time
2. The answer to π_1 for input x is *yes* if and only if the answer to π_2 for input T(x) is *yes*.



Implications of polynomial reductions

- **Theorem**: If $\pi_1 \propto \pi_2$ and π_2 is in P, then π_1 is in P
- Proof outline: We need to sum:

1. The number of steps to convert x to T(x) and
2. The number of steps needed by algorithm A for π_2 to provide the answer for T(x).

- Then, we need to show that the sum is bound by a polynomial in the size n of x.

Proof

- Let $p(n) = O(n^c)$ be a polynomial bound on the computation of T
- Let $q(m) = O(m^{c'})$ be a polynomial bound on algorithm A for π_2

1. The conversion needs $O(n^c)$ steps where $|x|=n$.
2. The size of the problem generated by T(x) is at most p(n) characters (assuming that at each step the program for T writes a symbol).
3. So algorithm A with input of at most p(n) symbols does at most $q(p(n)) = O((n^c)^{c'})$ steps
4. The total work to transform x to T(x) and then apply A to get the correct answer, is at most p(n)+q(p(n)) = O

NP-completeness

- A language L (or problem corresponding to the language) is *NP-complete* if

1. $L \in NP$ and
2. $L' \propto L$ for every $L' \in NP$.

- The class of NP-complete problems (languages) is called NPC.

Theorem

1. If any problem in NPC is polynomial-time solvable, then P = NP.
2. If any problem in NP is not polynomial-time solvable, then all problems in NPC are not polynomial-time solvable.

Proof

1. Given: $L \in P$ and $L \in NPC$

   For any $L' \in NP$ $L' \propto L$ (definition of NPC).

   So by last theorem, $L' \in P$ and $P = NP$.

2. Given: $L \in NP$ and $L \notin P$

   Let $L' \in NPC$. Assume that $L' \in P$. Then $L \propto L'$ (definition of NPC) and thus by last theorem, $L \in P$.

   Contradiction. So $L' \notin P$ and all NPC problems are not polynomial time solvable

Proving NP completeness

- Cook showed CNF-SATISFIABILITY is NP-complete.
- To prove any other problem NP-complete, we then just have to reduce CNF- SATISFIABILITY to that problem and show that problem lies in the class NP.

Theorem: polynomial reducibility is transitive

If L1 $\propto$ L2 and L2 $\propto$ L3 then L1 $\propto$ L3

Proof outline:

- There is a polynomially computable function
  - that reduces all inputs $x \in$ L1 into an input T1(x) $\in$ L2
  - that reduces all inputs $y \in$ L2 into an input T2(y) $\in$ L3
- Applying T1 to x and T2 to T1(x) reduces any instance $x \in$ L1 to an instance T2(T1(x)) $\in$ L3, and this conversion requires a polynomial number of steps

Shortcut for NP-completeness Proofs

- To prove a language *L* is NP-complete:

Prove $L \in NP$.

Choose $L' \in NPC$, and show $L' \propto L$

- $L \in NP$. We will show that every $M \in NP$ satisfies $M \propto L$, and thus L is NP-complete
  - Let $M \in NP$. $M \propto L'$ (definition of NPC), and $L' \propto L$ (proved by us). So by transitivity $M \propto L$