CS 341: Algorithms

Winter 2014

Kevin Lanctot

Slides by Douglas R. Stinson
with additions by Kevin Lanctot

Topic 1: Introduction

This section mostly reviews material from CS240, i.e. some vocabulary, order notation, some basic formulae, analysis of loops.

Readings for this topic from *Introduction to Algorithms, 3rd ed.*

- Sections 2.1 (loop invariants), 2.2, 2.3, Getting Started
- Chapter 3, Growth of Functions
- Sections 9.1 Minimum and maximum
- Appendices A, C2 and C3

Analysis of algorithms

The word *algorithm* comes from *al-Khowarizmi*, a 9th century Persian mathematician and astronomer who wrote a book, eventually translated into Latin as *Algoritmi de numero Indorum* (al-Khowarizmi's book on Indian numbers),

This book gave us our modern pencil-and-paper algorithms for addition, subtraction, multiplication, and division.

In this course, we study the **design** and **analysis** of algorithms. "Analysis" refers to mathematical techniques for establishing both the **correctness** and **efficiency** of algorithms.

Correctness: We often want a formal proof of correctness of an algorithm we design. This might be accomplished through the use of **loop invariants** and mathematical induction.

Analysis of algorithms–reality check

This course is concerned with the **correctness** and **efficiency** of algorithms. What other aspects of good program design are missing from this approach?

- clarity / readibility / good documentation
- modularity / loose coupling
- security
- reuseability / extensibility / easy to maintain
- user friendly
- use minimal resources (time, memory, critical sections of code), optimized
- elegant (simple solution to a difficult problem)

Analysis of algorithms (cont.)

Efficiency: Given an algorithm A, we want to know how efficient it is. This includes several possible criteria:

- What is the **asymptotic complexity** of algorithm A?
- What is the **exact number** of specified computations done by A?
- How does the **average-case** complexity of A compare to the **worst-case** complexity?
- Is A the most efficient algorithm to solve the given problem? (For example, can we find a **lower bound** on the complexity of **any** algorithm to solve the given problem?)
- Are there problems that cannot be solved efficiently? This topic is addressed in the theory of **NP-completeness**.
- Are there problems that cannot be solved by **any** algorithm? Such problems are termed **undecidable**.

Efficient algorithms–reality check

This course is concerned with designing **efficient** algorithms. Which of the previous criteria are we concerned about in the *real* world?

- Asymptotic complexity captures the scalability of a solution.
- Average case complexity captures the experience of the average user.
- Worse case complexity is useful for realtime applications or when you want the system to be consistently responsive.
- Knowing the lower bound, or that a problem cannot be solved efficiently, or is undecidable, will help you decide if it is worthwhile spending more time trying to improve an algorithm.

Design of algorithms

"Design" refers to **general strategies** for creating new algorithms. If we have good design strategies, then it will be easier to end up with correct and efficient algorithms. Also, we want to avoid using **ad hoc** algorithms that are hard to analyze and understand.

Here are some useful design strategies, many of which we will study:

divide-and conquer

greedy

dynamic programming

depth-first and breadth-first search

local search (not studied in this course)

linear programming (not studied in this course)

Design of algorithms

Another skill we are trying to develop: If you have a challenging problem to solve, what is the best way to approach it?

- focus your attention exclusively on solving the problem
- try solve a simpler version of the problem
- take a break, sleep on it
- compare and contrast with other problems: can we take advantage of the similarities or the differences
- lead a healthy lifestyle
- draw a picture, think graphically
- don't panic
- ask what went wrong with your previous approach and try to correct it

Design of algorithms: a big question.

Computers are currently very good as solving some problems and very poor at solving others when compared to humans, for example, the jellybean challenge.

The jellybean challenge gives a graph illustration of how much faster computers are at arithmetic and Boolean logic. But computers do much worse at natural language processing, creative problem solving, and pattern recognition tasks such facial recognition or recognizing handwriting.

- Why is that?
- What are we missing from the current state of the art?

We currently don't know the answer to this one.

The "Maximum" problem

Problem

Maximum

Instance: *an array A of n integers,*

$$A = [A[1], \ldots, A[n]].$$

Find: *the maximum element in A.*

The **Maximum** problem has an obvious simple solution.

Algorithm: *FindMaximum*$(A = [A[1], \ldots, A[n]])$

$max \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
  do **if** $A[i] > max$
    then $max \leftarrow A[i]$
return (max)

Correctness of FindMaximum

How can we formally prove that *FindMaximum* is correct?

Claim: At the end of iteration i $(i = 2, \ldots, n)$, the current value of max is the maximum element in $[A[1], \ldots, A[i]]$.

The claim can be proven by induction. The base case, when $i = 2$, is obvious.

Now we make an induction assumption that the claim is true for $i = j$, where $2 \leq j \leq n - 1$, and we prove that the claim is true for $i = j + 1$.

If the maximum element is in the first j elements, by the inductive hypothesis max contains this value and the **do** loop preserves this condition.

If the maximum element is $A[j + 1]$, then when $i = j + 1$ the **if** condition in the **do** loop will be true and max will be set to $A[j + 1]$.

When $j = n$ we are done and the correctness of *FindMaximum* is proven.

Analysis of FindMaximum

It is obvious that the complexity of *FindMaximum* is $\Theta(n)$.

Informally, $\Theta(n)$ means a computer takes roughly kn steps to complete the task for an array of size n and for some postive integer k.

More precisely, we can observe that the number of comparisons of array elements done by *FindMaximum* is **exactly** $n-1$.

It turns out that *FindMaximum* is **optimal** with respect to the number of comparisons of array elements.

That is, any algorithm that correctly solves the **Maximum** problem for an array of n elements requires **at least** $n-1$ comparisons of array elements.

How can we prove this assertion?

Answer: *Each element must be involved in one comparison. The first comparison involves two array elements and it takes at least one comparison to compare each new element to the previously processed elements.*

The "Max-Min" problem

Problem

Max-Min

Instance: *an array A of n integers, $A = [A[1], \ldots, A[n]]$.*

Find: *the maximum and the minimum element in A.*

The **Max-Min** problem also has an obvious simple solution.

Algorithm: *FindMaximumAndMinimum*$(A = [A[1], \ldots, A[n]])$

$max \leftarrow A[1]$
$min \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do
- **if** $A[i] > max$ **then** $max \leftarrow A[i]$
- **if** $A[i] < min$ **then** $min \leftarrow A[i]$

return (max, min)

Analysis of FindMaximumAndMinimum

Exercise: Give a formal proof by induction that *FindMaximumAndMinimum* is correct.

The complexity of *FindMaximumAndMinimum* is $\Theta(n)$

More precisely, *FindMaximumAndMinimum* requires $2n - 2$ comparisons of array elements given an array of size n.

The **complexity** is optimal (why?), but there are are algorithms to solve the **Max-Min** problem which require fewer comparisons of array elements than *FindMaximumAndMinimum*.

Note: An algorithm requiring fewer comparisons of array elements **may or may not be faster** than *FindMaximumAndMinimum*.

Is there a simple optimization that we can perform to improve *FindMaximumAndMinimum*? (Hint: consider the two **if** statements.)

An improved algorithm

Algorithm: *ImprovedFindMaximumAndMinimum*(A)

$max \leftarrow A[1]$
$min \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do
- **if** $A[i] > max$
  - **then** $max \leftarrow A[i]$
  - **else**
    - **if** $A[i] < min$
      - **then** $min \leftarrow A[i]$

return (max, min)

Justification: We only need to execute the second **if** statement when the first **if** statement is false.

Analysis of the improved algorithm

The number of comparisons of array elements required by the improved algorithm varies between $n-1$ and $2n-2$. (When do these extreme cases occur?)

It may be of interest to compute the **average** number of comparisons of array elements required by the improved algorithm. The average will be computed over the $n!$ possible **orderings** (i.e., permutations) of n distinct elements.

In iteration i, we perform only one comparison when $A[i]$ is the **maximum** value in $A[1], \ldots, A[i]$; this happens with probability $1/i$.

So we perform one comparison with probability $1/i$, and two comparisons are done with probability $1 - 1/i$. The **average** (i.e., expected) number of comparisons done in iteration i is

$$1 \times \frac{1}{i} + 2 \times \left(1 - \frac{1}{i}\right) = 2 - \frac{1}{i}.$$

Analysis of the improved algorithm (cont.)

Therefore, the expected number of comparisons of array elements is

$$\sum_{i=2}^{n} \left(2 - \frac{1}{i}\right) = 2n - 2 - \Theta(\log n)$$

(this will be justified on a later slide).

Asymptotically, this is essentially $2n$, since $\log n$ is **insignificant** compared to n.

So the improvement is not so great!

A more significant improvement

With some ingenuity, we can actually reduce the number of comparisons of array elements by (roughly) 25%.

Suppose n is even and we consider the elements two at a time. Initially, we compare the first two elements and initialize maximum and minimum values. (**One comparison** is required here.)

Then, each time we compare a new pair of elements, we subsequently compare the larger of the two elements to the current maximum and the smaller of the two to the current minimum. (**Three comparisons** are done here to process two array elements.)

This yields an algorithm requiring a total of $3n/2 - 2$ comparisons.

An optimal (?) algorithm

Algorithm: *OptimalFindMaximumAndMinimum*(A)

comment: assume n is even

if $A[1] > A[2]$ **then** $\begin{cases} max \leftarrow A[1] \\ min \leftarrow A[2] \end{cases}$

else $\begin{cases} max \leftarrow A[2] \\ min \leftarrow A[1] \end{cases}$

for $i \leftarrow 2$ **to** $n/2$

do $\begin{cases} \textbf{if } A[2i-1] > A[2i] \\ \quad \textbf{then} \begin{cases} \textbf{if } A[2i-1] > max \quad \textbf{then } max \leftarrow A[2i-1] \\ \textbf{if } A[2i] < min \quad \textbf{then } min \leftarrow A[2i] \end{cases} \\ \quad \textbf{else} \begin{cases} \textbf{if } A[2i] > max \quad \textbf{then } max \leftarrow A[2i] \\ \textbf{if } A[2i-1] < min \quad \textbf{then } min \leftarrow A[2i-1] \end{cases} \end{cases}$

return (max, min)

Optimality of the previous algorithm

It is possible to **prove** that any algorithm that solves the **Max-Min** problem requires at least $3n/2 - 2$ comparisons of array elements in the worst case.

Therefore the algorithm *OptimalFindMaximumAndMinimum* is in fact **optimal** with respect to the number of comparisons of array elements required.

Problems

Problem: Given a problem instance I for a problem **P**, carry out a particular computational task.

Problem Instance: **Input** for the specified problem.

Problem Solution: **Output** (correct answer) for the specified problem.

Size of a problem instance: Size(I) is a positive integer which is a measure of the size of the instance I.

Algorithms and Programs

Algorithm: An algorithm is a step-by-step process (e.g., described in **pseudocode**) for carrying out a series of computations, given some appropriate input.

Algorithm solving a problem: An Algorithm A **solves** a problem **P** if, for every instance I of **P**, A finds a valid solution for the instance I in finite time.

Program: A program is an **implementation** of an algorithm using a specified computer language.

Running Time

Running Time of a Program: $T_{\mathbf{M}}(I)$ denotes the running time (in seconds) of a program $\mathbf{M}$ on a problem instance I.

Worst-case Running Time as a Function of Input Size: $T_{\mathbf{M}}(n)$ denotes the **maximum** running time of program $\mathbf{M}$ on instances of size n:

$$T_{\mathbf{M}}(n) = \max\{T_{\mathbf{M}}(I) : \text{Size}(I) = n\}.$$

Average-case Running Time as a Function of Input Size: $T^{avg}_{\mathbf{M}}(n)$ denotes the **average** running time of program $\mathbf{M}$ over all instances of size n:

$$T^{avg}_{\mathbf{M}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I:\text{Size}(I)=n\}} T_{\mathbf{M}}(I).$$

Complexity

Worst-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \to \mathbb{R}$. An algorithm A has **worst-case complexity** $f(n)$ if there exists a program $\mathbf{M}$ implementing the algorithm A such that $T_{\mathbf{M}}(n) \in \Theta(f(n))$.

Average-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \to \mathbb{R}$. An algorithm A has **average-case complexity** $f(n)$ if there exists a program $\mathbf{M}$ implementing the algorithm A such that $T^{avg}_{\mathbf{M}}(n) \in \Theta(f(n))$.

Running Time vs Complexity

Running time can only be determined by implementing a program and running it on a specific computer.

Running time is influenced by many factors, including the programming language, processor, operating system, etc.

Complexity (a.k.a. **growth rate**) can be analyzed by high-level mathematical analysis. It is **independent** of the above-mentioned factors affecting running time.

Complexity is a less precise measure than running time since it is asymptotic and it incorporates unspecified constant factors and unspecified lower order terms.

However, if algorithm A has lower complexity than algorithm B, then a program implementing algorithm A will be faster than a program implementing algorithm B for sufficiently large inputs.

Order Notation

O-notation:

$f(n) \in O(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$.

The complexity of f is **not higher** than that of g, (think $f \leq g$).

Ω-notation:

$f(n) \in \Omega(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$.

The complexity of f is **not lower** than that of g (think $f \geq g$).

Θ-notation:

$f(n) \in \Theta(g(n))$ if **there exist** constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$.

Here f and g have the **same complexity** (think $f = g$).

Order Notation (cont.)

o-notation:

$f(n) \in o(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < c\,g(n)$ for all $n \geq n_0$.

Here f has **lower complexity** than g (think $f < g$).

ω-notation:

$f(n) \in \omega(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c\,g(n) < f(n)$ for all $n \geq n_0$.

Here f has **higher complexity** than g, (think $f > g$).

Techniques for Order Notation

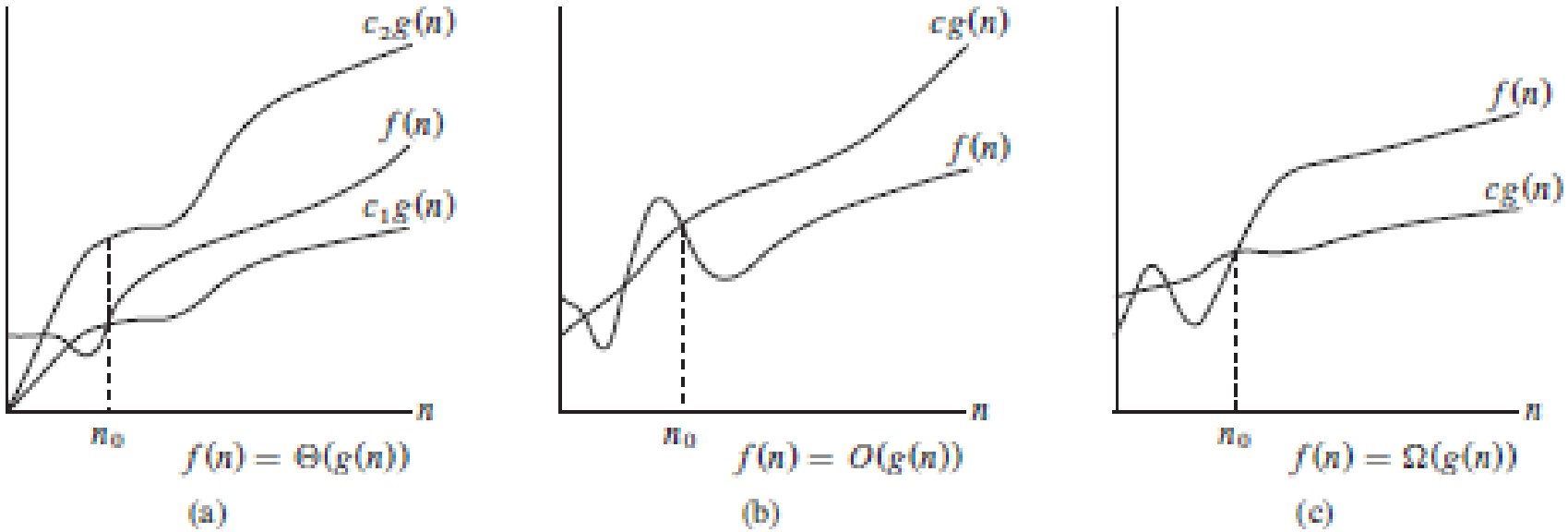
Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty. \end{cases}$$

Order Notation (cont.)



Asymptotic notation can ignore the behaviour of a function, $f(n)$, for low values of n, namely when $n < n_o$.

image from pg 45 of the course text, Figure 3.1

Growth of Functions

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Asymptotic notation captures how quickly a function will grow as a function of n.

table from http://www.csupomona.edu/∼ftang/courses/CS240/lectures/analysis.htm

Relationships between Order Notations

$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

$f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

Proof for the first relationship.

- If $f(n) \in \Theta(g(n))$, **there exist** constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1\, g(n) \leq f(n)$ and $0 \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$.
- Hence $0 \leq g(n) \leq (1/c_1) f(n)$ and $0 \leq (1/c_2) f(n) \leq g(n)$
- Hence $0 \leq (1/c_2)\, f(n) \leq g(n) \leq (1/c_1\, f(n)$ for all $n \geq n_0$.

Algebra of Order Notations

"Maximum" rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Then:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$
$$\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$$
$$\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$$

"Summation" rules:

$$O\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} O(f(i))$$
$$\Theta\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Theta(f(i))$$
$$\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))$$

Sequences

Arithmetic sequence:

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2).$$

Geometric sequence:

$$\sum_{i=0}^{n-1} a\,r^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1. \end{cases}$$

Arithmetic-geometric sequence:

$$\sum_{i=0}^{n-1}(a+di)r^i = \frac{a}{1-r} - \frac{(a+(n-1)d)r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2}$$

provided that $r \neq 1$.

Sequences (cont.)

Harmonic sequence:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

More precisely, it is possible to prove that

$$\lim_{n \to \infty} (H_n - \ln n) = \gamma,$$

where $\gamma \approx 0.57721$ is **Euler's constant**.

Log Formulae

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b x/y = \log_b x - \log_b y$$

$$\log_b 1/x = -\log_b x$$

$$\log_b x^y = y \log_b x$$

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$a^{\log_b c} = c^{\log_b a}$$

$$n! \in \Theta\left(n^{n+1/2} e^{-n}\right)$$

$$\log n! \in \Theta(n \log n)$$

Working with Log Formulae

For example: $\log_b a = \frac{1}{\log_a b}$

It is saying: if $b^x = a$ then $b = a^{1/x}$

Proof:

- Let $\log_b a = x$
- $b^x = a$ (translate back to exponents)
- $x \log_a b = 1$ (take log_a on both sides)
- $x = \frac{l}{\log_a b}$ (divide by $\log_a b$)
- $\log_b a = \frac{1}{\log_a b}$ (substitute for x)

Working with Log Formulae

For example: $a^{\log_b c} = c^{\log_b a}$ (1)

What is it saying?

- Let $\log_b c = x$ and $\log_b a = y$ (2).
- Hence $b^x = c$ and $b^y = a$ (3).
- Substituting (2) into (1) we have $a^x = c^y$ (4).
- Substituting (3) into (4) we have $(b^y)^x = (b^x)^y$.

How would you prove it?

- Taking log_b on both sides of (1) we have: $log_b a^{log_b c} = log_b c^{log_b a}$.
- Which simplifies to $(log_b c)(log_b a) = (log_b a)(log_b c)$.

Techniques for Algorithm Analysis

Two general strategies are as follows:

- Use Θ-bounds **throughout the analysis** and thereby obtain a Θ-bound for the complexity of the algorithm.
- Prove a O-bound and a **matching** Ω-bound **separately** to get a Θ-bound. Sometimes this technique is easier because arguments for O-bounds may use simpler upper bounds (and arguments for Ω-bounds may use simpler lower bounds) than arguments for Θ-bounds do.

Techniques for Loop Analysis

Identify **elementary operations** that require constant time (denoted $\Theta(1)$ time).

The complexity of a loop is expressed as the **sum** of the complexities of each iteration of the loop.

Analyze independent loops **separately**, and then **add** the results: use "maximum rules" and simplify whenever possible.

If loops are nested, start with the **innermost loop** and proceed outwards. In general, this kind of analysis requires evaluation of **nested summations**.

Example of Loop Analysis

Algorithm: *LoopAnalysis1*$(n : integer)$

(1) $sum \leftarrow 0$

(2) **for** $i \leftarrow 1$ **to** n

do { **for** $j \leftarrow 1$ **to** i

do { $sum \leftarrow sum + (i - j)^2$

$sum \leftarrow sum/i$

(3) **return** (sum)

Θ-bound analysis

(1)	$\Theta(1)$
(2)	Complexity of inner **for** loop: $\Theta(i)$
	Complexity of outer **for** loop: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
	Note: $\sum_{i=1}^{n} i = n(n+1)/2$
(3)	$\Theta(1)$
total	$\Theta(n^2)$

Example of Loop Analysis (cont.)

Proving separate O- and Ω-bounds

We focus on the two nested **for** loops (i.e., (2)).

The total number of iterations is $\sum_{i=1}^{n} i$, with $\Theta(1)$ time per iteration.

Upper bound:

$$\sum_{i=1}^{n} O(i) \leq \sum_{i=1}^{n} O(n) = O(n^2).$$

Lower bound:

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2).$$

Since the upper and lower bounds **match**, the complexity is $\Theta(n^2)$.

Lower Bounds from Previous Slide Explained

Key Insight: if solving a version of the problem **P** that involves less work is in $\Omega(n^2)$ then certainly **P** is in $\Omega(n^2)$.

Key Strategy: simplify the problem is such a way that it makes analyzing the asymptotic complexity easier.

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i)$$

The RHS does less work because we've changed the sum "from 1 to n" to "from $n/2$ to n."

$$\sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2)$$

The RHS does less work because it has change from ranging "from $\Omega(n/2)$ to $\Omega(n)$" to being a constant amount of work, $\Omega(n/2)$, each iteration.

Another Example of Loop Analysis

Algorithm: *LoopAnalysis*$(n : integer)$

$sum \leftarrow 0$

for $i \leftarrow 1$ **to** n

do
- $j \leftarrow i$
- **while** $j \geq 1$
  - **do**
    - $sum \leftarrow sum + i/j$
    - $j \leftarrow \left\lfloor \frac{j}{2} \right\rfloor$

return (sum)

Another Example of Loop Analysis

What is the Θ bound on the asymptotic running time of *LoopAnalysis*?

The inner loop runs $\lg i$ times.

The outer loops runs

$$\sum_{i=1}^{n} \lg i = \lg(n!) \in \Theta(n \lg n).$$

Yet Another Example of Loop Analysis

Algorithm: *LoopAnalysis2*$(A : array; n : integer)$

$max \leftarrow 0$
for $i \leftarrow 1$ **to** n
do
  for $j \leftarrow i$ **to** n
  do
    $sum \leftarrow 0$
    for $k \leftarrow i$ **to** j
    do
      $sum \leftarrow sum + A[k]$
      if $sum > max$
        then $max \leftarrow sum$
return (max)

Yet Another Example of Loop Analysis

What is the Θ bound on the asymptotic running time of *LoopAnalysis2*?

Upper bound:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} O(1) < \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} O(1) \in O(n^3).$$

Lower bound:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} \Omega(1) > \sum_{i=1}^{\frac{n}{3}}\sum_{j=\frac{2n}{3}}^{n}\sum_{k=i}^{j} \Omega(1) \in \Omega(\frac{n^3}{27}) = \Omega(n^3).$$

Again, as I move from left to right, I'm doing less work and yet I'm still bounded from below by $\Omega(n^3)$.

Since the upper and lower bounds **match**, the complexity is $\Theta(n^3)$.

Topic 2: Divide-and-Conquer Algorithms

This topic describes the **Divide-and-Conquer** design strategy and investigates the analysis of **recurrence relations**.

Readings for this topic from *Introduction to Algorithms, 3rd ed.*

- Sections 4.2, 4.4-4.6, Divide-and-Conquer Algorithms
- Sections 9.2-9.3, Medians and Order Statistics
- Section 33.4, Finding the closest pair of points

The Divide-and-Conquer Design Strategy

divide: Given a problem instance I, construct one or more smaller problem instances, denoted $I_1, \ldots, I_a$ (these are called **subproblems**). Usually, we want the size of these subproblems to be small compared to the size of I, e.g., half the size.

conquer: For $1 \leq j \leq a$, solve instance I_j recursively, obtaining solutions $S_1, \ldots, S_a$.

combine: Given $S_1, \ldots, S_a$, use an appropriate **combining** function to find the solution S to the problem instance I, i.e., $S \leftarrow \mathit{Combine}(S_1, \ldots, S_a)$.

Design Strategy: Typically recursion is used. Divide is setting up the recursive call(s). Conquer is solving the base case. Combine is processing the results of the recursive call.

Example: Design of Mergesort

Here, a problem instance consists of an array A of n integers, which we want to sort in increasing order. The size of the problem instance is n.

divide: Split A into two subarrays: A_L consists of the first $\lceil \frac{n}{2} \rceil$ elements in A and A_R consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in A.

conquer: Run *Mergesort* on A_L and A_R.

combine: After A_L and A_R have been sorted, use a function *Merge* to merge A_L and A_R into a single sorted array. The merge can be done in time $\Theta(n)$ with a single pass through A_L and A_R. We simply keep track of the "current" element of A_L and A_R, always copying the smaller one into the sorted array.

Mergesort

Algorithm: *Mergesort*$(A : array; n : integer)$

comment: Conquer

if $n = 1$

then $S \leftarrow A$

else
- comment: Divide
- $n_L \leftarrow \lceil \frac{n}{2} \rceil$
- $n_R \leftarrow \lfloor \frac{n}{2} \rfloor$
- $A_L \leftarrow [A[1], \ldots, A[n_L]]$
- $A_R \leftarrow [A[n_L + 1], \ldots, A[n]]$
- $S_L \leftarrow$ *Mergesort*(A_L, n_L)
- $S_R \leftarrow$ *Mergesort*(A_R, n_R)
- comment: Combine
- $S \leftarrow$ *Merge*(S_L, n_L, S_R, n_R)

return (S, n)

Here n, n_L, and n_R are the sizes of the arrays A, A_L, and A_R.

Analysis of Mergesort

Let $T(n)$ denote the time to run *Mergesort* on an array of length n.

divide takes time $\Theta(n)$

conquer takes time $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$

combine takes time $\Theta(n)$

Recurrence relation:

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

Recurrence Relations

It is simpler to replace the Θ's by constant factors c and d. The resulting recurrence relation is called the **exact recurence**.

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

If we then remove the floors, an ceilings, we obtain the **sloppy recurrence**:

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

The exact and sloppy recurrences are identical when n is a power of 2.

We will begin by solving the recurrence when $n = 2^j$ using the **recursion tree method**.

Recursion Tree Method

We construct a **recursion tree**, assuming $n = 2^j$, as follows:

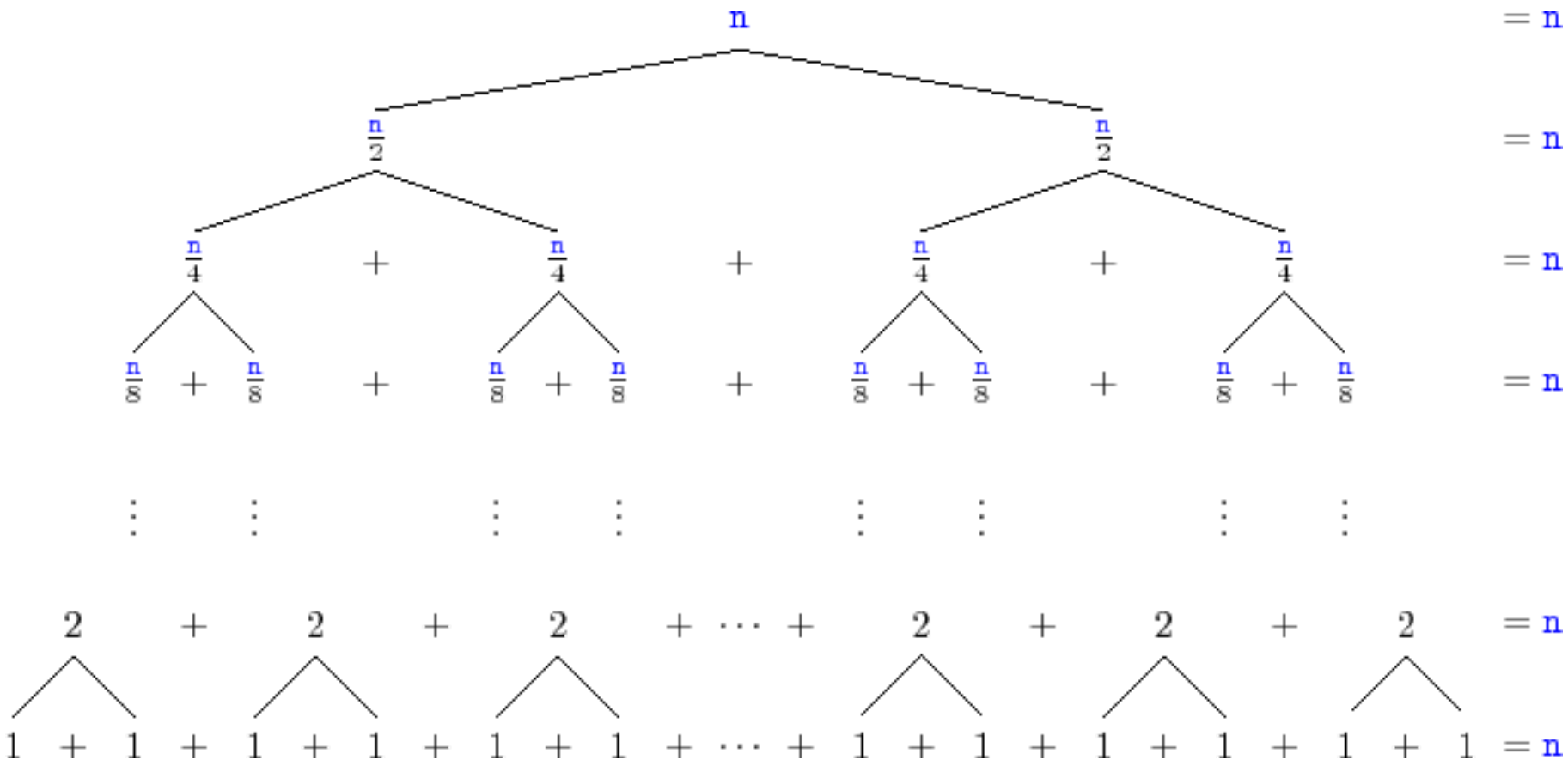
step 1 Start with a one-node tree, say N, which receives the value $T(n)$.

step 2 Grow two children of N. These children, say N_1 and N_2, receive the value $T(n/2)$, and the value of N is updated to be cn.

step 3 Repeat this process recursively, terminating when a node receives the value $T(1) = d$.

step 4 Sum the values on each level of the tree, and then compute the sum of all these sums, i.e. (work per level) x (# of levels), and the result is $T(n)$.

Recursion Tree Method



Total work = (work per level) x (# of levels) = $(c\,n)$ x $(\lg n) \in \Theta(n \lg n)$

image from: http://opendatastructures.org/versions/edition-0.1d/ods-java/node56.html

Recursion Tree Method

If instead of $2T\left(\frac{n}{2}\right)$ we had $4T\left(\frac{n}{2}\right)$ i.e.

$$T(n) = \begin{cases} 4\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

we would get the following

$$T(n) = \sum_{i=1}^{\log_2 n} \frac{4^i}{2^i} cn = cn \sum_{i=1}^{\log_2 n} 2^i \in \Theta(n^2)$$

since $\sum_{i=1}^{\log_2 n} 2^i \in \Theta(2^{\log_2 n}) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Solving the Exact Recurrence

The recursion tree method finds the **exact** solution of the recurrence when $n = 2^j$.

Suppose we express this solution (for powers of 2) as a function of n, using Θ-notation.

The resulting function of n will in fact yield the **complexity** of the solution of the exact recurrence for **all values** of n.

This derivation of the complexity of $T(n)$ not a proof, however. If a rigourous mathematical proof is required, one method is to use induction along with the exact recurrence. Another approach would be to derive upper bounds for an exact power of 2 higher than n and lower bounds for an exact power of two lower than n.

Another approach would be to use the Master Theorem...

Master Theorem

The Master Theorem provides a formula for the solution of many recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

The following is a simplified version of the Master Theorem:

Theorem

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^y). \tag{1}$$

Denote $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x. \end{cases}$$

Master Theorem

a represents the **number of subproblems** you create when you divide and **how wide** the tree gets.

$\frac{n}{b}$ represents the **size of the subproblems** you create when you divide and **how deep** the tree is.

$x = log_b a$ represents **how many leaves and nodes** the tree has, for example

- if $a = 2$ and $b = 2$ then $log_2 2 = 1$ implies $\Theta(n)$ leaves
- if $a = 4$ and $b = 2$ then $log_2 4 = 2$ implies $\Theta(n^2)$ leaves
- if $a = 4$ and $b = 2$ then $log_2 8 = 3$ implies $\Theta(n^3)$ leaves

$\Theta(n^y)$ represents the **complexity** of dividing into subproblems and combining their results, i.e. the complexity of each recursive call.

If $x > y$ the complexity is dominated by the number of nodes.

If $y > x$ the complexity is dominated by the amount of work done by each node.

The Max-Min Problem

Let's design a divide-and-conquer algorithm for the **Max-Min** problem.

Divide: Suppose we split A into two equal-sized subarrays, A_L and A_R.

Conquer: We find the maximum and minimum elements in each subarray recursively, obtaining max_L, min_L, max_R and min_R.

Combine: Then we can easily "combine" the solutions to the two subproblems to solve the original problem instance:

$$max \leftarrow \max\{max_L, max_R\}$$

and

$$min \leftarrow \min\{min_L, min_R\}$$

The Max-Min Problem (cont.)

The recurrence relation describing the complexity of the running time is $T(n) = 2T(n/2) + \Theta(1)$.

The Master Theorem shows that the $T(n) \in \Theta(n)$.

However, we can also count the **exact number** of comparisons done by the algorithm, obtaining the (sloppy) recurrence

$$C(n) = 2C(n/2) + 2, \quad C(2) = 1.$$

For n a power of 2, the solution to this recurrence relation is $C(n) = 3n/2 - 2$, so the divide-and-conquer algorithm is **optimal** for these values of n.

The Max-Min Problem Complexity

Can we use the **recursion tree method** to find the $\Theta(n)$ for the divide-and-conquer algorithm for solving the max-min problem?

Level 0: 1 node of size n, make 2 comparisons.

Level 1: 2 nodes of size $n/2$, make 4 comparisons.

Level 2: 4 nodes of size $n/4$, make 8 comparisons.

Level i: 2^i nodes of size $n/2^i$, make 2^{i+1} comparisons.

The exception is the leaves, just make 1 comparison per node not 2.

Level $(\lg n) - 1$: $n/2$ nodes of size 2, make $n/2$ comparisons (not n).

Idea: take the sum of the whole sequence and subtract off $n/2$.

$$T(n) = 2\left(\sum_{i=1}^{(\log_2 n)-1} 2^i\right) - \frac{n}{2} = 2\frac{2^{\log_2 n} - 1}{2-1} - \frac{n}{2} = 2\frac{n-1}{2-1} - \frac{n}{2} = \frac{3n}{2} - 2$$

Multiprecision Multiplication

Problem

Multiprecision Multiplication

Instance: *Two k-bit positive integers, X and Y, having binary representations*

$$X = [X[k-1], ..., X[0]]$$

and

$$Y = [Y[k-1], ..., Y[0]].$$

Question: *Compute the $2k$-bit positive integer $Z = XY$, where*

$$Z = (Z[2k-1], ..., Z[0]).$$

We are interested in the **bit complexity** of algorithms that solve **Multiprecision Multiplication**, which means that the complexity is expressed as a function of k (the size of the problem instance is $2k$ bits).

Not-So-Fast D&C Multiprecision Multiplication

Algorithm: *NotSoFastMultiply*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

else
$$\begin{cases} Z_1 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_R, k/2) \\ Z_4 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_L, k/2) \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + \textit{LeftShift}(Z_3 + Z_4, k/2) + Z_2 \end{cases}$$

return (Z)

Fast D&C Multiprecision Multiplication

Algorithm: *FastMultiply*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

else
$$\begin{cases} X_T \leftarrow X_L + X_R \\ Y_T \leftarrow Y_L + Y_R \\ Z_1 \leftarrow \textit{FastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{FastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{FastMultiply}(X_T, Y_T, k/2), \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + \textit{LeftShift}(Z_3 - Z_1 - Z_2, k/2) + Z_2 \end{cases}$$

return (Z)

Multiprecision Multiplication Complexity

$$
\begin{array}{rrrr}
 & & & X_L X_R \\
 & & \times & Y_L Y_R \\
\hline
 & & Y_R X_L & Y_R X_R \\
+ & Y_L X_L & Y_L X_R & \\
\hline
 & Z_1 & Z_3 & Z_2
\end{array}
$$

- Normally it would take four multiplications to get the final result: $X_L Y_L$, $X_L Y_R$, $X_R Y_L$, $X_R Y_R$.
- But you can do it in three if you create a term $X_T Y_T$ and subtract off Z_1 and Z_2.
- $Z_1 = X_L Y_L$ and $Z_2 = X_R Y_R$
- $X_T Y_T = X_L Y_L + X_L Y_R + X_R Y_L + X_R Y_R$
- $Z_3 = X_T Y_T - Z_1 - Z_2 = X_L Y_R + X_R Y_L$

Multiprecision Multiplication Complexity

Can we use the **Master Theorem** to find the $\Theta(n)$ for the two algorithms for **Multiprecision Multiplication**?

For *NotSoFastMultiply* the recurrence is

$$T(n) = 4\,T\left(\frac{n}{2}\right) + \Theta(n^1).$$

Here $x = \log_2 4 = 2$, $y = 1$ so the algorithm is in $\Theta(n^2)$.

For *FastMultiply* the recurrence is

$$T(n) = 3\,T\left(\frac{n}{2}\right) + \Theta(n^1).$$

Here $x = \log_2 3 \approx 1.585$, $y = 1$ so the algorithm is in $O(n^{1.59})$.

Matrix Multiplication

Problem

Matrix Multiplication

Instance: *Two n by n matrices, A and B.*

Question: *Compute the n by n matrix product $C = AB$.*

The naive algorithm for **Matrix Multiplication** has complexity $\Theta(n^3)$.

Matrix Multiplication: Problem Decomposition

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

If A, B are n by n matrices, then $a, b, ..., h, r, s, t, u$ are $\frac{n}{2}$ by $\frac{n}{2}$ matrices, where

$$r = a\,e + b\,g \qquad s = a\,f + b\,h$$
$$t = c\,e + d\,g \qquad u = c\,f + d\,h$$

We require 8 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

Efficient D&C Matrix Multiplication

Define

$$P_1 = a(f-h) \qquad P_2 = (a+b)h$$
$$P_3 = (c+d)e \qquad P_4 = d(g-e)$$
$$P_5 = (a+d)(e+h) \qquad P_6 = (b-d)(g+h)$$
$$P_7 = (a-c)(e+f).$$

Then, compute

$$r = P_5 + P_4 - P_2 + P_6 \qquad s = P_1 + P_2$$
$$t = P_3 + P_4 \qquad u = P_5 + P_1 - P_3 - P_7.$$

We now require only 7 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

Complexity of Efficient D&C Matrix Multiplication

Can we use the **Master Theorem** to find the $\Theta(n)$ for the two algorithms for **Matrix Multiplication**?

For the naive algorithm the recurrence is

$$T(n) = 8\,T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Here $x = \log_2 8 = 3$, $y = 2$ so the algorithm is in $\Theta(n^3)$.

For the Divide-and-Conquer approach the recurrence is

$$T(n) = 7\,T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Here $x = \log_2 7 \approx 2.807$, $y = 2$ so the algorithm is in $O(n^{2.81})$.

Selection

Problem

Selection

Instance: *An array $A[1], \ldots, A[n]$ of distinct integer values, and an integer k, where $1 \leq k \leq n$.*

Find: *The kth smallest integer in the array A.*

The problem **Median** is the special case of **Selection** where $k = \lceil \frac{n}{2} \rceil$.

QuickSelect

Suppose we choose a **pivot** element y in the array A, and we **restructure** A so that all elements less than y precede y in A, and all elements greater than y occur after y in A. (This is exactly what is done in *Quicksort*, and it takes **linear time**.)

Suppose that $A[posn] = y$ after restructuring. Let A_L be the subarray $A[1], \ldots, A[posn-1]$ and let A_R be the subarray (of size $n - posn$) $A[posn+1], \ldots, A[n]$.

Then the kth smallest element of A is

$$\begin{cases} y & \text{if } k = posn \\ \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\ \text{the } (k - posn)\text{th smallest element of } A_R & \text{if } k > posn. \end{cases}$$

We make (at most) one recursive call at each level of the recursion.

Average-case Analysis of QuickSelect

We say that a pivot is **good** if $posn$ is in the middle half of A.

The probability that a pivot is good is $1/2$.

On average, after **two iterations**, we will encounter a good pivot.

If a pivot is good, then $|A_L| \leq 3n/4$ and $|A_R| \leq 3n/4$.

With an **expected** linear amount of work, the size of the subproblem is reduced by at least 25%.

It follows that the average-case complexity of the *QuickSelect* is linear.

Achieving $O(n)$ Worst-Case Complexity: A Strategy for Choosing the Pivot

We choose the pivot to be a certain **median-of-medians**:

step 1 Given $n \geq 15$, write $n = 10r + 5 + \theta$, where $r \geq 1$ and $0 \leq \theta \leq 9$ (guaranteeing an odd number of groups of five).

step 2 Divide A into $2r + 1$ disjoint subarrays of 5 elements. Denote these subarrays by $B_1, \ldots, B_{2r+1}$.

step 3 For $1 \leq i \leq 2r + 1$, find the median of B_i (nonrecursively), and denote it by m_i.

step 4 Define M to be the array consisting of elements $m_1, \ldots, m_{2r+1}$.

step 5 Find the median y of the array M (recursively).

step 6 Use the element y as the pivot for A.

Median-of-medians-QuickSelect

Algorithm: *Mom-QuickSelect*(k, n, A)

1. **if** $n \leq 14$ **then** sort A and **return** $(A[k])$
2. write $n = 10r + 5 + \theta$, where $0 \leq \theta \leq 9$
3. construct $B_1, \ldots, B_{2r+1}$ (subarrays of A, each of size 5)
4. find medians $m_1, \ldots, m_{2r+1}$ (non-recursively)
5. $M \leftarrow [m_1, \ldots, m_{2r+1}]$
6. $y \leftarrow$ *Mom-QuickSelect*$(r + 1, 2r + 1, M)$
7. $(A_L, A_R, posn) \leftarrow$ *Restructure*(A, y)
8. **if** $k = posn$ **then return** (y)
9. **else if** $k < posn$ **then return** (*Mom-QuickSelect*$(k, posn - 1, A_L)$)
10. **else return** (*Mom-QuickSelect*$(k - posn, n - posn, A_R)$)

Worst-case Analysis of Mom-QuickSelect

When the pivot is the median-of-medians, we have that $|A_L| \leq \lfloor \frac{7n+12}{10} \rfloor$ and $A_R \leq \lfloor \frac{7n+12}{10} \rfloor$.

The *Mom-QuickSelect* algorithm requires **two recursive calls**.

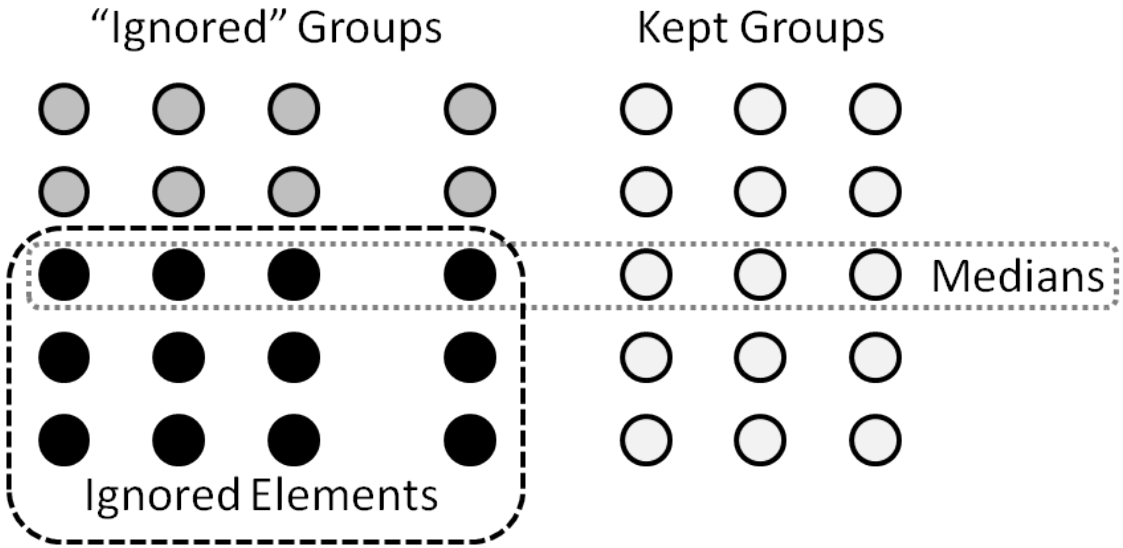
The worst-case complexity $T(n)$ of this algorithm satisfies the following recurrence:

$$T(n) \leq \begin{cases} T\left(\lfloor \frac{n}{5} \rfloor\right) + T\left(\lfloor \frac{7n+12}{10} \rfloor\right) + \Theta(n) & \text{if } n \geq 15 \\ \Theta(1) & \text{if } n \leq 14. \end{cases}$$

It can be shown that $T(n)$ is $O(n)$.

ref: http://www.cs.cmu.edu/~avrim/451f09/lectures/lect0903.pdf

Worst-case Analysis of Mom-QuickSelect



There are $\frac{n}{5} = g$ groups, where g is always an odd number.

$\lceil \frac{1}{2} g \rceil$ groups contain elements that we ignore, i.e. the black and grey dots.

And we ignore 3 element in each of those groups, i.e. the black dots.

The Recurrence Relation for Mom-QuickSelect

n	input size
$n-9$	the most we'll need to subtract off in order to guarantee an odd number of groups of 5
$\frac{1}{5}(n-9)$	the number of groups of 5
$\frac{1}{2}(\frac{1}{5}(n-9)+1)$	the number of "ignored" groups adding one is the same as taking the ceiling
$\frac{1}{10}(n-4)$	simplify
$\frac{3}{10}(n-4)$	the number of "ignored" elements
$\frac{3n-12}{10}$	simplify
$n-\frac{3n-12}{10}$	the number of elements to be kept
$\frac{7n+12}{10}$	simplify, final answer

Recurrence Relation for Mom-QuickSelect

How would you solve the following recurrence relation?

$$T(n) \leq \begin{cases} T\left(\left\lfloor \frac{1}{5}n \right\rfloor\right) + T\left(\left\lfloor \frac{7}{10}n \right\rfloor\right) + \Theta(n) & \text{if } n \geq 15 \\ \Theta(1) & \text{if } n \leq 14. \end{cases}$$

We process $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10}$ of the input in linear time, i.e. $\Theta(n)$, and discard $\frac{n}{10}$ of the input in each iteration. The total amount of work done is the sum of work done at each iteration, i.e.

$$n \sum_{i=1} \left(\frac{9}{10}\right)^i \in \Theta(n)$$

so the algorithm is linear.

Closest Pair - Algorithms Ch 33.4

Problem

Closest Pair

Instance: *a multiset Q of n points in the Euclidean plane,*

$$Q = \{Q[1], \ldots, Q[n]\}.$$

Find: *Two points $Q[i] = (x, y), Q[j] = (x', y')$ such that the Euclidean distance*

$$\sqrt{(x' - x)^2 + (y' - y)^2}$$

is minimized.

As per the course text we will allow Q to contain coincident points, i.e. two points with the same location.

Closest Pair: Problem Decomposition

Suppose we presort the points in Q with respect to their x-coordinates (this takes time $\Theta(n \log n)$). Call this sorted array Q_x.

Then we can easily find the vertical line that partitions the set of points Q_x into two sets of size $n/2$: this line has equation $x = Q_x[m].x$, where $m = n/2$.

The set Q_x is global with respect to the recursive procedure *ClosestPair1*.

At any given point in the recursion, we are examining a subarray $(Q[\ell], \ldots, Q[r])$, and $m = \lfloor (\ell + r)/2 \rfloor$.

We call *ClosestPair1*$(1, n)$ to solve the given problem instance.

Closest Pair: Solution 1

Algorithm: *ClosestPair1*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else
$$\begin{cases} m \leftarrow \lfloor (\ell + r)/2 \rfloor \\ \delta_L \leftarrow \textit{ClosestPair1}(\ell, m) \\ \delta_R \leftarrow \textit{ClosestPair1}(m+1, r) \\ \delta \leftarrow \min\{\delta_L, \delta_R\} \\ R \leftarrow \textit{SelectCandidates}(\ell, r, \delta, Q_x[m].x) \\ R \leftarrow \textit{SortY}(R) \\ \delta \leftarrow \textit{CheckStrip}(R, \delta) \end{cases}$$

return (δ)

Find the smallest distance, δ, on the left half and on the right half, by calling *ClosestPair1* recursively.

Check to see if there are any points on either side of the boundary that are closer than δ, using *SelectCandidates*, *SortY*, and *CheckStrip*.

Selecting Candidates from the Vertical Strip

Algorithm: *SelectCandidates*$(\ell, r, \delta, xmid)$

$j \leftarrow 0$

for $i \leftarrow \ell$ **to** r

do { **if** $|Q[i].x - xmid| \leq \delta$

then { $j \leftarrow j + 1$

$R[j] \leftarrow Q[i]$

return (R)

If a point $Q[i]$ is within δ of the boundary line, $xmid$, then add that point to R, the list of points near the boundary.

Checking the Vertical Strip

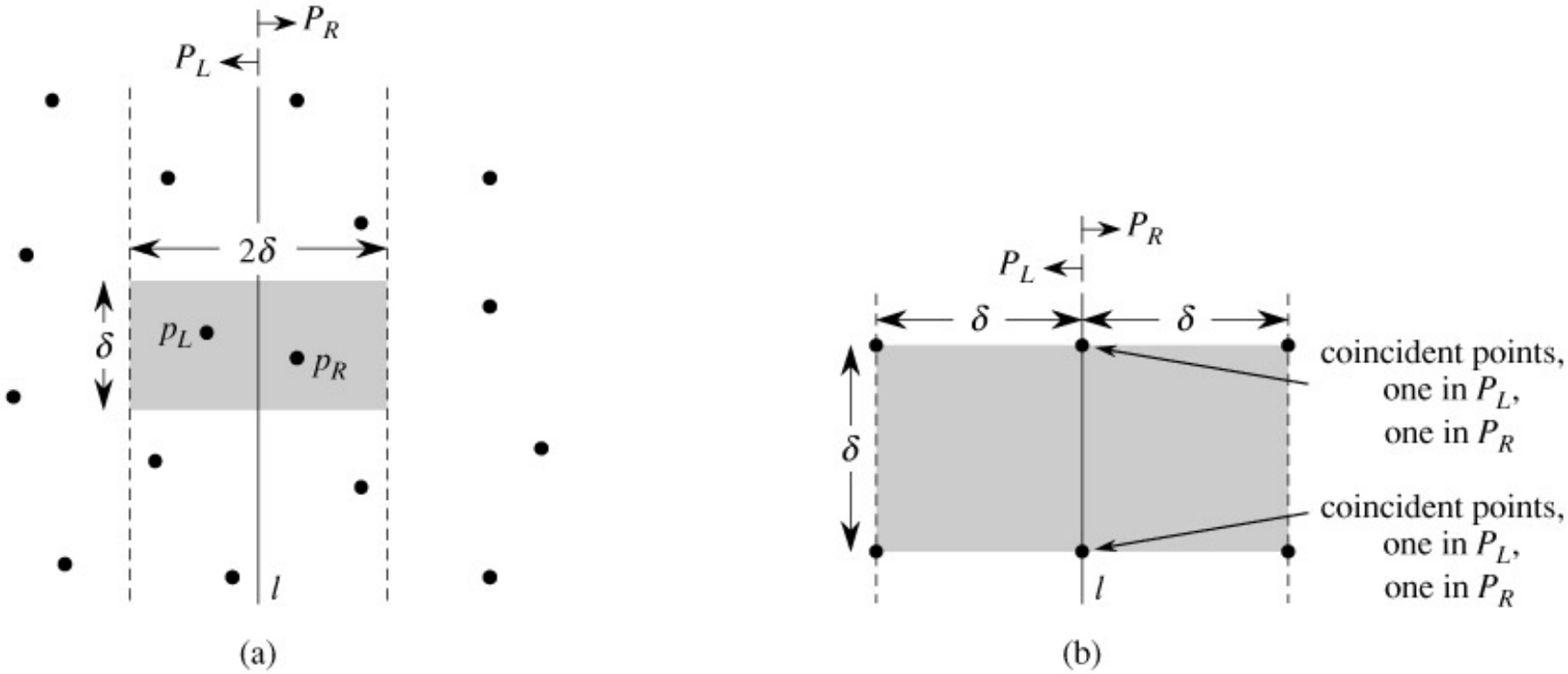
Algorithm: *CheckStrip*(R, δ)

$t \leftarrow$ *size*(R)
$\delta' \leftarrow \delta$
for $j \leftarrow 1$ **to** $t - 1$
do
  for $k \leftarrow j + 1$ **to** $\min\{t, j + 7\}$
  do
    $x \leftarrow R[j].x$
    $x' \leftarrow R[k].x$
    $y \leftarrow R[j].y$
    $y' \leftarrow R[k].y$
    $\delta' \leftarrow \min\left\{\delta', \sqrt{(x' - x)^2 + (y' - y)^2}\right\}$
return (δ')

No need to check every point, just the ones that could be within δ.

How many points can be packed in the strip and still all be δ away from each other?

Checking the Vertical Strip



If you allow coincident points, you need to check at most the next 7 points.

What if you did not allow coincident points?

image source: course text, Figure 33.11

Closest Pair: Solution 2

Algorithm: *ClosestPair2*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else
- $m \leftarrow \lfloor (\ell + r)/2 \rfloor$
- $\delta_L \leftarrow$ *ClosestPair2*(ℓ, m)
- comment: $Q[\ell], \ldots, Q[m]$ is sorted WRT y-coordinates
- $\delta_R \leftarrow$ *ClosestPair2*$(m + 1, r)$
- comment: $Q[m + 1], \ldots, Q[r]$ is sorted WRT y-coordinates
- $\delta \leftarrow \min\{\delta_L, \delta_R\}$
- *Merge*(ℓ, m, r)
- $R \leftarrow$ *SelectCandidates*$(\ell, r, \delta, Q[m].x)$
- $\delta \leftarrow$ *CheckStrip*(R, δ)

return (δ)

By keeping the array Q (not Q_x) sorted by y-coordinate as you call *ClosestPair2* and *Merge* you eliminate having to sort it each time.

Guess-and-Verify (Ch 4.3)

- What do I do if none of the standard techniques seem to work?
- Answer: **Guess-and-Verify**.
- **Guess:** If I'm trying to characterize the asymptotic behaviour of a recurrence relation, I could code up the recurrence relation or I could just take a guess based on the fact that it looks similar to something I've seen before.
- **Verify:** Typically you would use induction.

Remember: definitions are your friend!

In the definition of $O(g(n))$, you must find *some* c and *some* n_0. Take advantage of these choices.

Recall:

$f(n) \in O(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0$.

Guess-and-Verify (Ch 4.3)

- Find $O(n)$ for

$$T(n) = \begin{cases} 2T\left(\lfloor n/2 \rfloor\right) + n & \text{if } n > 2, \\ O(1) & \text{if } n = 2. \end{cases}$$

- **Guess:** This is in $O(n \lg(n))$.
- **Verify:** Base case $n = 2$ is easy. For the inductive step, assume $T(m) \leq cm \lg(m)$ for all $m < n$ and for some $c > 0$.

$$\begin{aligned} T(n) &\leq 2\left(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)\right) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg(n) - cn \lg(2) + n \\ &= cn \lg(n) - cn + n \\ &\leq cn \lg(n), \text{ for } c > 1. \end{aligned}$$

Guess-and-Verify (Ch 4.3)

- Find $O(n)$ for

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 & \text{if } n > 1, \\ O(1) & \text{if } n = 1. \end{cases}$$

- **Guess:** This is in $O(n)$.
- **Verify:** Base case $n = 1$ is easy. For the inductive step, assume $T(m) \leq cm$ for all $m < n$ and for some $c > 0$.

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \\ &= O(n), \text{ that would be wrong!} \end{aligned}$$

Guess-and-Verify (Ch 4.3)

- Find $O(n)$ for

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 & \text{if } n > 1, \\ O(1) & \text{if } n = 1. \end{cases}$$

- **Guess:** This is in $O(n)$.
- **Verify:** Base case $n = 1$ is easy. For the inductive step, assume $T(m) \leq cm - 1$ for all $m < n$ and for some $c > 0$.

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - 1) + (c\lceil n/2 \rceil - 1) + 1 \\ &= cn - 2 + 1 \\ &= cn - 1 \end{aligned}$$

Note: subtracting 1 from cm means that now the choice of cm must be bigger.

Recall the Master Theorem

The Master Theorem provides a formula for the solution of many recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

The following is a simplified version of the Master Theorem:

Theorem

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^y). \tag{2}$$

Denote $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x. \end{cases}$$

Proof of the Master Theorem (simplified version)

Suppose that $a \geq 1$ and $b \geq 2$ are integers and

$$T(n) = a\,T\left(\frac{n}{b}\right) + c\,n^y, \qquad T(1) = d.$$

Let $n = b^j$.

size of subproblem	# nodes	cost/node	total cost
$n = b^j$	1	$c\,n^y$	$c\,n^y$
$n/b = b^{j-1}$	a	$c\,(n/b)^y$	$c\,a\,(n/b)^y$
$n/b^2 = b^{j-2}$	a^2	$c\,(n/b^2)^y$	$c\,a^2\,(n/b^2)^y$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n/b^{j-1} = b$	a^{j-1}	$c\,(n/b^{j-1})^y$	$c\,a^{j-1}\,(n/b^{j-1})^y$
$n/b^j = 1$	a^j	d	$d\,a^j$

Computing $T(n)$

Summing the costs of all levels of the recursion tree, we have that

$$T(n) = d\,a^j + c\,n^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i.$$

Recall that $b^x = a$ and $n = b^j$. Hence $a^j = (b^x)^j = (b^j)^x = n^x$.

The formula for $T(n)$ is a **geometric sequence** with ratio $r = a/b^y = b^{x-y}$:

$$T(n) = d\,n^x + c\,n^y \sum_{i=0}^{j-1} r^i.$$

There are **three cases**, depending on whether $r > 1$, $r = 1$ or $r < 1$.

Complexity of $T(n)$

case	r	y, x	complexity of $T(n)$
heavy leaves	$r > 1$	$y < x$	$T(n) \in \Theta(n^x)$
balanced	$r = 1$	$y = x$	$T(n) \in \Theta(n^x \log n)$
heavy top	$r < 1$	$y > x$	$T(n) \in \Theta(n^y)$

heavy leaves means that cost of the recursion tree is dominated by the cost of the leaf nodes.

balanced means that costs of the levels of the recursion tree stay constant (except for the last level)

heavy top means that cost of the recursion tree is dominated by the cost of the root node.

Master Theorem (modified general version)

Theorem

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n).$$

Denote $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } f(n) \in O(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^x \log n) & \text{if } f(n) \in \Theta(n^x) \\ \Theta(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is an increasing function of } n \\ & \text{for some } \epsilon > 0. \end{cases}$$

Topic 3: Greedy Algorithms

This topic describes the **Greedy** design strategy and investigates the proofs of optimality for **Greedy** Algorithms.

Readings for this topic from *Introduction to Algorithms, 3rd ed.*

- Sections 16.1, Greedy Algorithms

Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

Problem Instance: **Input** for the specified problem.

Problem Constraints: **Requirements** that must be satisfied by any feasible solution.

Feasible Solution: For any problem instance I, *feasible*(I) is the set of all outputs (i.e., solutions) for the instance I that satisfy the given constraints.

Objective Function: A function f : *feasible*$(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of f as being a **profit** or a **cost** function.

Optimal Solution: A feasible solution $X \in$ *feasible*(I) such that the profit $f(X)$ is maximized (or the cost $f(X)$ is minimized).

Example: Closest Pair

Closest Pair: Find the closest pair of points in a set of points.

Problem Instance: A list of n distinct points in the Euclidean plane, $P = \{P[0], P[1], \ldots, P[n-1]\}$.

Problem Constraints: Two points which must be distinct.

Feasible Solution: $P[0]$ and $P[1]$.

Objective Function: The Euclidean distance between the two points.

Optimal Solution: To two points with the smallest Euclidean distance between them.

Example: Travelling Salesman Problem (TSP)

Travelling Salesman Problem: Given a list of cities and a table of distances between the cities, find the shortest cycle that passes through all the cities.

Problem Instance: A list of cities plus a table that contains all the distances between them.

Problem Constraints: Must visit each city at least once and return to the city you started in.

Feasible Solution: An ordered list of cities to visit.

Objective Function: The total distance travelled.

Optimal Solution: The route that minimizes the total distance travelled.

The Greedy Method

partial solutions

Given a problem instance I, it should be possible to write a feasible solution X as a tuple $[x_1, x_2, \ldots, x_n]$ for some integer n, where $x_i \in \mathcal{X}$ for all i. A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a **partial solution** if no constraints are violated.
Note: it may be the case that a partial solution cannot be extended to a feasible solution.

TSP Example

If the list of cities is {Cambridge, Guelph, Hamilton, Kitchener, Stratford, Waterloo} then a **partial solution** might be {Waterloo, Kitchener, Cambridge}.

The Greedy Method (cont.)

choice set

For a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, we define the **choice set**

$$\mathit{choice}(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$

TSP Example

The **choice set** would be the remaining cities, Guelph, Hamilton, Stratford.

local evaluation criterion

For any $y \in \mathcal{X}$, $g(y)$ is a **local evaluation criterion** that measures the cost or profit of including y in a (partial) solution.

TSP Example

The **local evaluation criterion** would be the distance from the last city on our list, Cambridge, to the next distinct city we are considering.

The Greedy Method (cont.)

extension

Given a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, choose $y \in \mathit{choice}(X)$ so that $g(y)$ is as small (or large) as possible. Update X to be the $(i+1)$-tuple $[x_1, \ldots, x_i, y]$.

TSP Example

The **extension** of our partial solution {Waterloo, Kitchener, Cambridge} would be {Waterloo, Kitchener, Cambridge, Guelph}.

greedy algorithm

Starting with the "empty" partial solution, repeatedly extend it until a feasible solution X is constructed. This feasible solution may or may not be optimal.

TSP Example

Start with say Waterloo and repeatedly add the city that is the closest to the last city on our list until all cities have been added. Then add Waterloo to the end of the list.

Example of the Greedy Method

- For example: **making change**.
- Given any amount less than \$5, make change using the minimum number of coins.
- Mathematicly, given an integer $0 \leq x < 5$ express $x = 200t + 100l + 25q + 10d + 5n + p$ such that the total number of coins, $t + l + q + d + n + p$, is minimized.
- The greedy algorithm: choose as many toonies as possible, then loonies, then quarters, dimes, etc.
- The greedy approach will always produce the optimal solution.
- The is not true for other coin systems. Say three coins (12, 5, 1).
- To make change for 15, the greedy approach is $(1 * 12) + (3 * 1)$ but $3 * 5$ is optimal.
- Moral: the greedy algorithm does not always give the best solution.

Features of the Greedy Method

Greedy algorithms do no **looking ahead** and no **backtracking**.

Greedy algorithms can usually be implemented efficiently. Often they consist of a **preprocessing step** based on the function g, followed by a **single pass** through the data.

In a greedy algorithm, only **one feasible solution** is constructed.

The execution of a greedy algorithm is based on **local criteria** (i.e., the values of the function g).

Correctness: For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!

Interval Selection

Problem

Interval Selection

Instance: *A set* $\mathcal{A} = \{A_1, \ldots, A_n\}$ *of* **intervals**.
For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, *where* s_i *is the* **start time** *of interval* A_i *and* f_i *is the* **finish time** *of* A_i.
Feasible solution: *A subset* $\mathcal{B} \subseteq \mathcal{A}$ *of* **pairwise disjoint intervals**.
Find: *A feasible solution of maximum size (i.e., one that maximizes* $|\mathcal{B}|$*).*

These types of problems usually occur as part of job scheduling, see course text chapter 16.1.

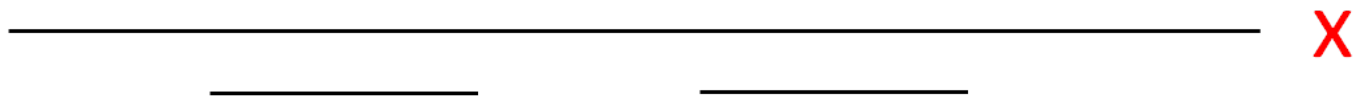
Possible Greedy Strategies for Interval Selection

1. Choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is s_i).
2. Choose the interval of **minimum duration** that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).
3. Choose the interval with the **minimum number of overlaps** with the remaining intevals. (i.e., the local evaluation criterion is count of the number of intersections between one of the remaining intervals with the rest).
4. Choose the **earliest finishing** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is f_i).

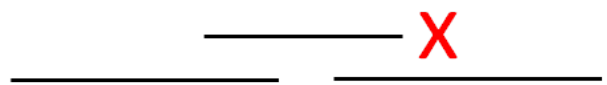
Does one of these strategies yield a correct greedy algorithm?

Some Greedy Approaches that Fail

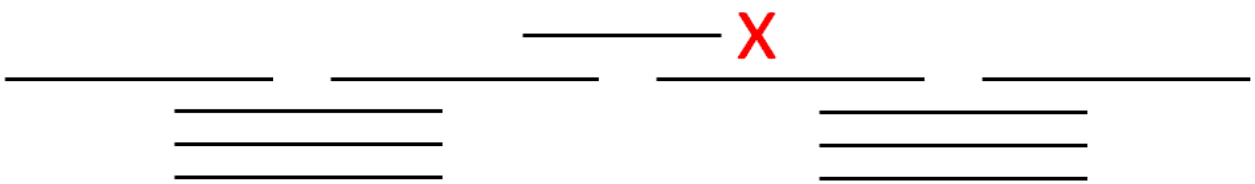
Choose the earliest starting interval ...



Choose the interval of minimum duration...



Choose the interval with the minimum number of overlaps ...



A Greedy Algorithm for Interval Selection

Algorithm: *GreedyIntervalSelection*$(\mathcal{A})$

rename the intervals, by sorting if necessary, so that $f_1 \leq \cdots \leq f_n$

$\mathcal{B} \leftarrow \{A_1\}$

$prev \leftarrow 1$

comment: $prev$ is the index of the last selected interval

for $i \leftarrow 2$ **to** n

do { **if** $s_i \geq f_{prev}$ **then** { $\mathcal{B} \leftarrow \mathcal{B} \cup \{A_i\}$; $prev \leftarrow i$

return $(\mathcal{B})$

Summary: The input is $\mathcal{A}$ and the output is $\mathcal{B}$. The algorithm sorts the intervals by finishing time, f_i and greedily picks the next interval, s_i, so that it does not overlap with the finishing time of the previous interval f_{prev}.

Is the Greedy Algorithm Optimal?

- Proof by contradiction.
- Let the result of the greedy algorithm be $G = \{g_1, g_2, \ldots, g_n\}$.
- Of all the possible schedules that beat the greedy approach, pick the one $B = \{b_1, b_2, \ldots, b_m\}$ that **first** *disagrees with the greedy approach the furthest along* the schedule, say at d, i.e. $g_d \neq b_d$.
- Since the greedy algorithm chooses the earliest finish time then $f(g_d) \leq f(b_d)$.
- Since b_{d+1} is scheduled after b_d, then $s(b_{d+1}) \geq f(b_d) \geq f(g_d)$
- Create a new schedule $B' = \{g_1, g_2, \ldots, g_d, b_{d+1}, b_{d+2}, \ldots, b_m\}$.
- This new schedule, B', does not disagree with G until **at least** interval $d+1$, which contradicts the fact that B was the schedule that first disagreed with the G the furthest along in the schedule. But B disagrees with G at d.
- If both agree up to the the selection at $\min(n, m)$, then clearly if one cannot fit in another interval neither can the other, so $n = m$.

Interval Colouring

Problem

Interval Colouring

Instance: *A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of* **intervals**.
For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where s_i is the **start time** *of interval A_i and f_i is the* **finish time** *of A_i.*

Feasible solution: *A* **c-colouring** *is a mapping $col : \mathcal{A} \to \{1, \ldots, c\}$ that assigns each interval a* **colour** *such that two intervals receiving the same colour are always disjoint.*

Find: *A c-colouring of $\mathcal{A}$ with the minimum number of colours.*

Again think of job scheduling. You have a list of jobs to be done and you are asking how many machines (or processors) you need to complete all the jobs without delay.

Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.

At a given point in time, suppose we have coloured the first $i < n$ intervals using nc colours.

We will colour the $(i+1)$st interval with the **any permissible colour**. If it cannot be coloured using any of the existing nc colours, then we introduce a **new colour** and nc is increased by 1.

Question: In **what order** should we consider the intervals?

Possibilities: start time, finishing time, shortest duration
Answer: start time

A Greedy Algorithm for Interval Colouring

Algorithm: *GreedyIntervalColouring*$(\mathcal{A})$

rename the intervals, by sorting if necessary, so that $s_1 \leq \cdots \leq s_n$
$nc \leftarrow 1$
$colour[1] \leftarrow 1$
$finish[1] \leftarrow f_1$
for $i \leftarrow 2$ **to** n

do
- $available \leftarrow$ **false**
- $c \leftarrow 1$
- **while** $c \leq nc$ **and** (**not** $available$)
  - **do**
    - **if** $finish[c] \leq s_i$ **then**
      - $colour[i] \leftarrow c$
      - $finish[c] \leftarrow f_i$
      - $available \leftarrow$ **true**
    - **else** $c \leftarrow c + 1$
- **if not** $available$ **then**
  - $nc \leftarrow nc + 1$
  - $colour[i] \leftarrow nc$
  - $finish[nc] \leftarrow f_i$

return $(nc, colour)$

Parameters and Structure

Basic Idea:

- First sort the interval by start time and loop through each one.
- For each interval (the outer "do loop") find a colour that is not currently being used by an overlapping partition (inner "do loop").
- Check if a new colour is needed ("**if not** available").

Parameters

- nc - number of colours used so far
- colour[i]=c: the i^{th} interval has colour c
- finish[c]=f_i: we will finish using colour c at time f_i
- available: true if we can reuse an existing colour
- c: colour we are currently considering

Comments and Questions

In the algorithm on the previous two slides, at any point in time, $finish[c]$ denotes the finishing time of the **last interval** that has received colour c. Therefore, a new interval A_i can be assigned colour c if $s_i \geq finish[c]$.

The complexity of the algorithm is $O(n \times nc)$.

If it turns out that $nc \in \Omega(n)$, then the best we can say is that the complexity is $O(n^2)$.

What **inefficiencies** exist in this algorithm?

Finding which tasks have finished, i.e. which colors are now available. Currently we are looping through all the colours used so far and checking each one.

Comments and Questions

What **data structure** would allow a more efficient algorithm to be designed?

Something like a heap which would allow you to find and remove the element with the earliest finishing time quickly.

What would be the **asymptotic complexity** of an algorithm making use of an appropriate data structure?

$O(n \lg nc)$

The Stable Marriage Problem

Problem

Stable Marriage

Instance: *A set of n men, say $M = [m_1, \ldots, m_n]$, and a set of n women, $W = [w_1, \ldots, w_n]$.*

Each man m_i has a preference ranking of the n women, and each woman w_i has a preference ranking of the n men: $pref(m_i, j) = w_k$ if w_k is the j-th favourite woman of man m_i; and $pref(w_i, j) = m_k$ if m_k is the j-th favourite man of woman w_i.

Find: *A matching of the n men with the n women such that there does not exist a couple (m_i, w_j) who are not engaged to each other, but prefer each other to their existing matches. A matching with this this property is called a stable matching.*

Overview of the Gale-Shapley Algorithm

Men (suitors) propose to women (reviewers).

If a woman accepts a proposal, then the couple is **engaged**.

An unmatched woman **must accept** a proposal.

If an engaged woman receives a proposal from a man whom she prefers to her current match, the she **cancels** her existing engagement and she becomes engaged to the new proposer; her previous match is no longer engaged.

If an engaged woman receives a proposal from a man, but she prefers her current match, then the proposal is **rejected**.

Engaged women never become unengaged.

A man might make a number of proposals (up to n); the order of the proposals is determined by the man's preference list.

Gale-Shapley Algorithm

Algorithm: *Gale-Shapley*$(M, W, pref)$

$Match \leftarrow \emptyset$

while there exists an unengaged man m_i

do
- let w_j be the next woman in m_i's preference list
- **if** w_j is not engaged
  - **then** $Match \leftarrow Match \cup \{m_i, w_j\}$
  - **else**
    - suppose $\{m_k, w_j\} \in Match$
    - **if** w_j prefers m_i to m_k
      - **then**
        - $Match \leftarrow Match \backslash \{m_k, w_j\} \cup \{m_i, w_j\}$
        - comment: m_k is now unengaged

return $(Match)$

D. Gale and L. S. Shapley won the 2012 Nobel Prize in Economics for their work on this subject.

Optimal for Suitors not Reviewers

The solution may be optimal for the suitors, the males, but not the reviewers, the females.

Suppose there are three males (A,B,C) and three females (X,Y,Z).

Suppose they have the following preferences

- A: YXZ B: ZYX C: XZY
- X: BAC Y: CBA Z: ACB

The algorithm would result in the pairings (AY, BZ, CX)

Suitors get their first choice and reviewers their last choice.

Suitors ask reviewers in rank order; whereas, reviews can only pick from those who have asked her.

Questions

How do we prove that the *Gale-Shapley* algorithm always **terminates**?

- *A woman cannot reject an offer if she is unengaged.*
- *Since there are equal number of men and women, if a man is available, so is a woman.*
- *As some point the man will ask the woman. She will be unengaged and accept.*

How many **iterations** does this algorithm require in the worst case?

- *In each iteration at least one man will ask the next woman on his list.*
- *If she says no, he will now consider the next woman on his list.*
- *If she say yes, the man she was previously engage to will now consider the next woman on his list.*
- *There are n men and their lists are n long so there are $O(n^2)$ iterations.*

Questions

How do we prove that this algorithm is **correct**, i.e., that it finds a stable matching?

- Let Alice and Bob prefer each other to their current partners.
- If Bob prefers Alice, he must have proposed to her before he proposed to his current fiancee.
- The fact that Alice said no to Bob means either 1) she was with someone she preferred over Bob and rejected his offer or 2) accepted Bob and then someone better came along and she dropped Bob. Either way Alice prefers her current partner over Bob.

Questions

Is there an efficient way to **identify** an unengaged man at any point in the algorithm? What **data structure** would be helpful in doing this?

- *A first in first out (FIFO) queue or possibly a last in first out (LIFO) stack.*

What can we say about the **complexity** of the algorithm?

- *Assume it is $O(1)$ for remove a pairing from the Match list: i.e an array that stores who, if anyone, a woman is engaged to.*
- *Assume it is $O(1)$ to find which man a woman prefers, i.e. an array, $Ranking[i]$, that stores the ranking of the i^{th} man.*
- *Then asymtotic worse case running time is $O(n^2)$.*

Knapsack Problems

Problem

Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An* n*-tuple* $X = [x_1, \ldots, x_n]$ *where* $\sum_{i=1}^{n} w_i x_i \leq M$.

In the **0-1 Knapsack** *problem (often denoted just as* **Knapsack***), we require that* $x_i \in \{0, 1\}$, $1 \leq i \leq n$.

In the **Rational Knapsack** *problem, we require that* $x_i \in \mathbb{Q}$ *and* $0 \leq x_i \leq 1$, $1 \leq i \leq n$.

Find: *A feasible solution* X *that maximizes* $\sum_{i=1}^{n} p_i x_i$.

Possible Greedy Strategies for Knapsack Problems

Does one of these strategies yield a correct greedy algorithm for the **0-1 Knapsack** or **Rational Knapsack** problem?

1. Consider the items in decreasing order of **profit** (i.e., the local evaluation criterion is p_i).

   Example: *capacity is 6 and you have three items, in terms (weight, profit): (6,6), (3,5), (3,5). The first option has the highest profit, but the combination of the last two yields a higher total profit.*

2. Consider the items in increasing order of **weight** (i.e., the local evaluation criterion is w_i).

   Example: *capacity is 6 and you have three items, in terms (weight, profit): (3,3), (3,3), (6,9). The first two have the lowest weight, but the selecting the last one yields a higher profit.*

3. Consider the items in decreasing order of **profit divided by weight** (i.e., the local evaluation criterion is p_i/w_i). **Correct!**

A Greedy Algorithm for Rational Knapsack

Algorithm: *GreedyRationalKnapsack*$(P, W : array; M : integer)$

rename the items, sorting if necessary, so that $p_1/w_1 \geq \cdots \geq p_n/w_n$

$X \leftarrow [0, \ldots, 0]$

$i \leftarrow 1$

$CurW \leftarrow 0$

while $(CurW < M)$ **and** $(i \leq n)$

do
- **if** $CurW + w_i \leq M$
  - **then**
    - $x_i \leftarrow 1$
    - $CurW \leftarrow CurW + w_i$
  - **else**
    - $x_i \leftarrow (M - CurW)/w_i$
    - $CurW := M$

return (X)

Is the Greedy Approach Optimal?

How would you go about proving that the greedy approach is optimal?

This proof is extremely similar to the one proving that the greedy approach to the interval select problem is optimal.

- Prove by contradiction.
- Let the result of the greedy algorithm be $G = \{g_1, g_2, \ldots, g_n\}$.
- Of all the possible ordered list of choices that beat the greedy approach, pick the one $B = \{b_1, b_2, \ldots, b_m\}$ that **first** *disagrees with the greedy approach the furthest along* the ordered list of items, say at d, i.e. $g_d \neq b_d$.
- Since the greedy algorithm chooses the item with the highest profit per weight then the profit due to g_d is higher than or equal to the profit due to b_d.

Is the Greedy Approach Optimal?

How would you go about proving that the greedy approach is optimal?

Prove by contradiction continued...

- Use this fact to create a new ordered list, B', that does not differ from G until **at least** item $d+1$, that is $B' = \{g_1, g_2, \ldots, g_d, b_{d+1}, b_{d+2}, \ldots, b_m\}$.
- This new list, B', contradicts the fact that B first disagreed with the greedy approach the furtherest along the order list, which was at d.
- If both G and B agree up to the item $\min(n, m)$, then clearly if one cannot fit in all or part of an item then neither can the other, so $n = m$.

Topic 4: Graph Algorithms

This topic describes various **Graph** algorithms and investigates the proofs that these algorithms work.

Readings for this topic from *Introduction to Algorithms, 3rd ed.*

- Chapter 22, Elementary Graph Algorithms
- Chapter 23, Minimum Spanning Trees
- Section 24.3, Dijkstra's algorithm
- Section 25.2, The Floyd-Warshall algorithm

Graphs and Digraphs

A **graph** is a pair $G = (V, E)$. V is a set whose elements are called **vertices** and E is a set whose elements are called **edges**. Each edge joins two distinct vertices. An edge can be represented as a **set** of two vertices, e.g., $\{u, v\}$, where $u \neq v$. We may also write this edge as uv or vu.

We often denote the number of vertices by n and the number of edges by m. Clearly $m \leq \binom{n}{2}$.

A **directed graph** or **digraph** is also a pair $G = (V, E)$. The elements of E are called **directed edges** or **arcs** in a digraph. Each arc joins two vertices, and an arc can be represented as an **ordered pair**, e.g., (u, v). The arc (u, v) is directed from u (the **tail**) to v (the **head**), and we allow $u = v$.

If we denote the number of vertices by n and the number of arcs by m, then $m \leq n^2$.

Data Structures for Graphs: Adjacency Matrices

There are two main data structures to represent graphs: an **adjacency matrix** and a set of **adjacency lists**.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The **adjacency matrix** of G is an n by n matrix $A = (a_{u,v})$, which is indexed by V, such that

$$a_{u,v} = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

There are exactly $2m$ entries of A equal to 1, i.e. 2 for each edge.

If G is a digraph, then

$$a_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For a digraph, there are exactly m entries of A equal to 1.

Data Structures for Graphs: Adjacency Lists

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.

An **adjacency list representation** of G consists of n linked lists.

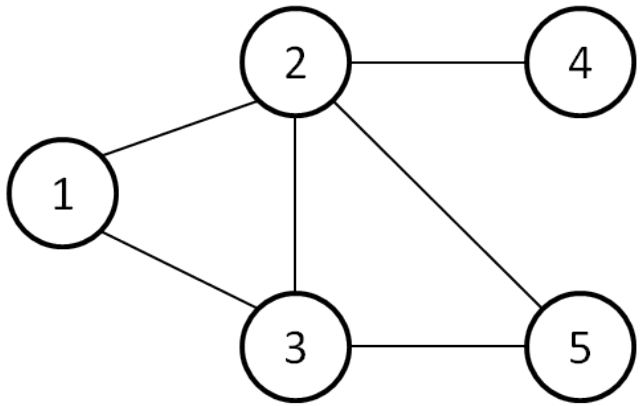
For every $u \in V$, there is a linked list (called an **adjacency list**) which is named $Adj[u]$.

For every $v \in V$ such that $uv \in E$, there is a node in $Adj[u]$ labelled v. (This definition is used for both directed and undirected graphs.)

In an undirected graph, every edge uv corresponds to nodes in two adjacency lists: there is a node v in $Adj[u]$ and a node u in $Adj[v]$.

In a directed graph, every edge corresponds to a node in only one adjacency list.

Data Structures for Graphs: Examples



Adjacency Matrix

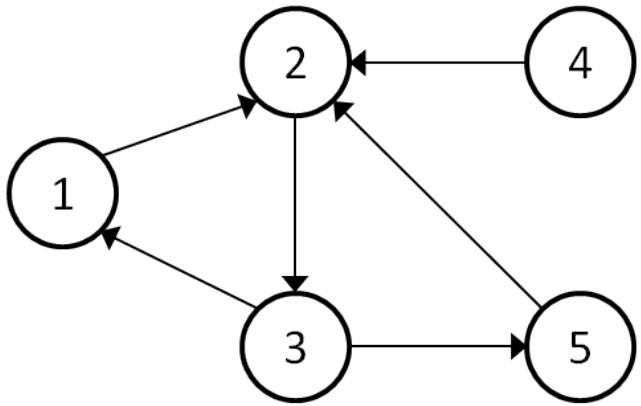
$$\begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{vmatrix}$$

Adjacency List

```
1: 2, 3
2: 1, 3, 4, 5
3: 1, 2, 5
4: 2
5: 2, 3
```

For (undirected) graphs, each edge corresponds to two entries in the adjacency matrix or adjacency lists.

Data Structures for Digraphs: Examples



Adjacency Matrix

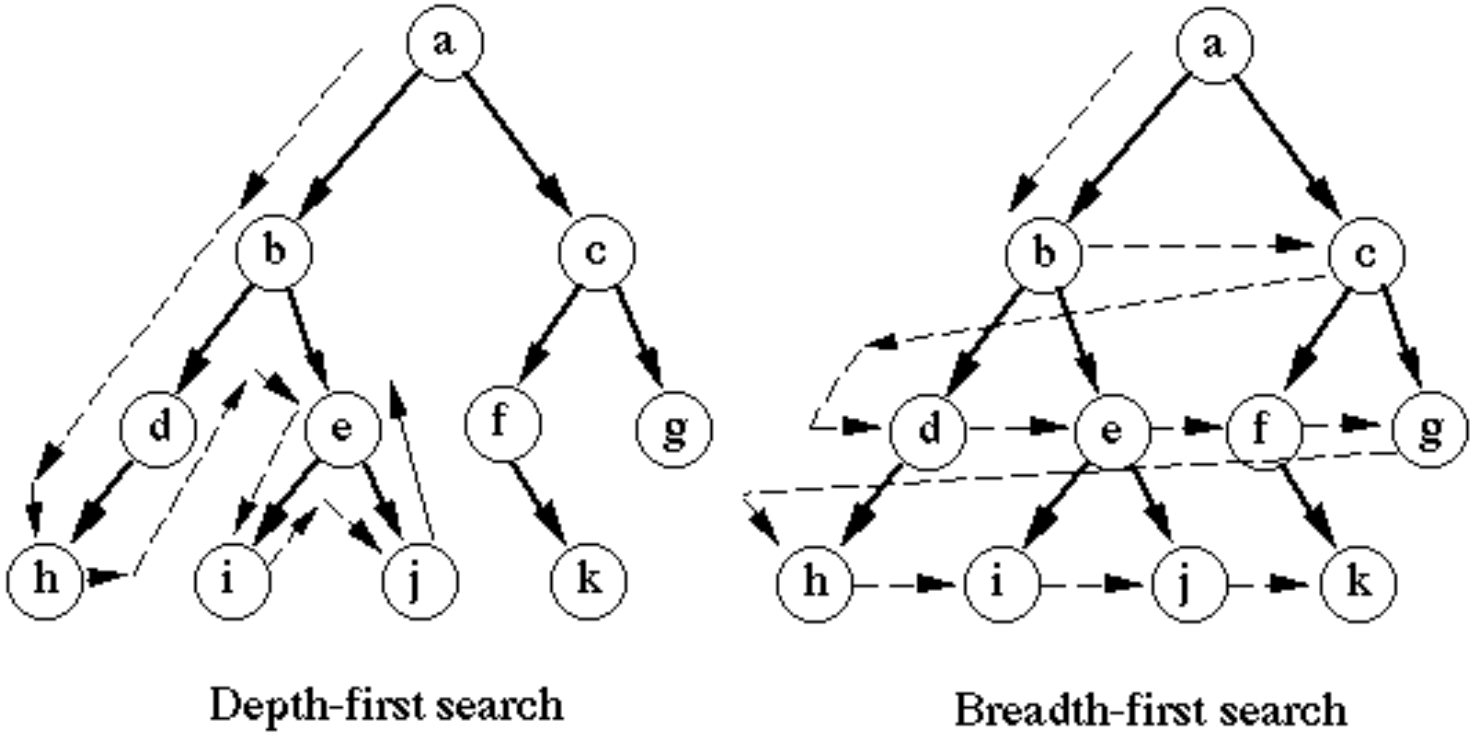
$$\begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

Adjacency List

```
1: 2
2: 3
3: 1, 5
4: 2
5: 2
```

For digraphs, each directed edge corresponds to one entry in the adjacency matrix or adjacency lists.

Depth-first Search vs. Breadth-first Search



The two major types of searches in a graph are:

- Depth-first Search (DFS) - if a node a has children, search any children of the first child, bm before consider a's other child(ren) c.
- Breadth-first Search (BFS) - if a node a has children, search all of its children, b and c, before consider grandchildren, d, e, f, and g.

Figure from: http://www.cse.unsw.edu.au/~billw/Justsearch.html

Breadth-first Search of an Undirected Graph

A **breadth-first search** (BFS) of an undirected graph begins at a specified vertex s.

The search "spreads out" from s, proceeding in **layers**.

First, all the neighbours of s are **explored**.

Next, the neighbours of those neighbours are explored.

This process continues until all vertices have been explored.

A **queue** is used to keep track of the vertices to be explored.

The colours white, gray, and black are used to track the state of each vertex.

Tracking the State Using Colours

A vertex is

- white if it is **undiscovered**,
- gray if it is **being processed**, i.e. it has been discovered and its adjacent vertices are in the process of being discovered.
- black if it **completely processed**, i.e. all its adjacent vertices have been discovered.

During the algorithm a vertex will progress from white to gray to black.

The gray vertices represent the vertices that are in the queue, i.e. that the algorithm is currently processing.

If G is **connected**, then eventually every vertex is coloured black.

When all the vertices have been coloured black the algorithm is done.

Breadth-first Search (BFS)

Algorithm: *BFS*(G, s)

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \text{gray}$

InitializeQueue(Q)

Enqueue(Q, s)

while $Q \neq \emptyset$

do $\begin{cases} u \leftarrow \textit{Dequeue}(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \text{white} \\ \quad \textbf{then} \begin{cases} colour[v] = \text{gray} \\ \pi[v] \leftarrow u \\ \textit{Enqueue}(Q, v) \end{cases} \end{cases} \\ colour[u] \leftarrow \text{black} \end{cases}$

Overview of Breadth-first Search

The *BFS*(G, s) algorithm takes two parameters: a graph G, and a starting vertex s. *BFS* does the following.

1. To initialize *BFS*, each vertex in G is coloured white (in the first **for each** loop).
2. Also to initialize *BFS*, the vertex s is coloured grey and added to the queue.
3. In the main **while** loop, each vertex u in the queue is removed and its neighbours are checked to see if they are white. If a neighbour is white, colour it grey and add it to the queue.
4. When you are finished processing all the neighbours of u, colour u black.

For many applications it is useful to keep track of the predecessor of each vertex. The predecessor of v is stored in the array $\pi[v]$.

π and Breadth-first Search Tree

$\pi[v]$ is the predecessor of v

Every vertex $v \neq s$ has a unique predecessor $\pi[v]$ in the BFS tree.

$\pi[v]$ is used to traverse the graph efficiently (as a tree).

The BFS tree consists of all the tree edges.

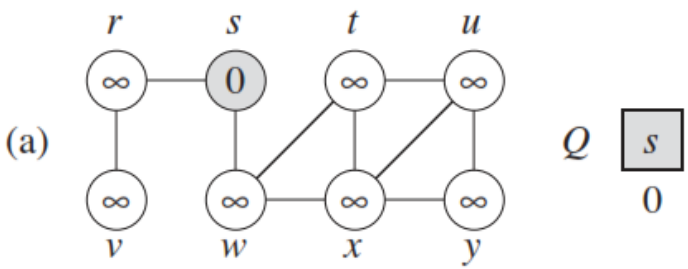
In a tree, if an edge contects a vertex to

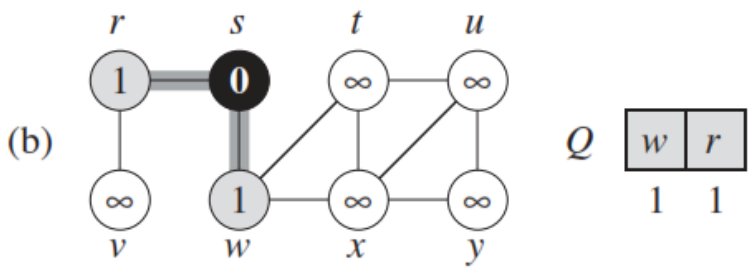
- an ancestor it is called a **back edge**.
- a descendant it is called a **forward edge**.
- a vertex that is neither an ancestor nor a descendant, it is called a **cross edge**.

When we explore an edge $\{u, v\}$ starting from u:

- if v is white, then uv is a **tree edge** and $\pi[v] = u$ is the **predecessor** of v in the **BFS tree**
- otherwise, uv is a **cross edge**.

Example of Breadth-first Search





(a) Start at s.

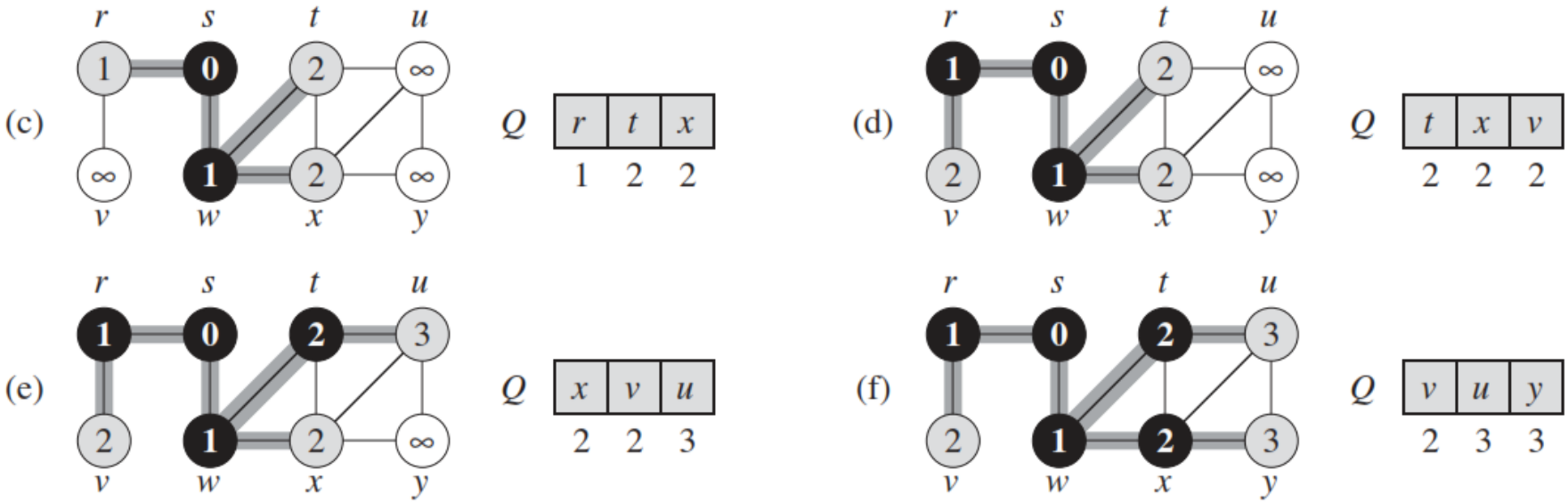
Put s in the queue, Q, and colour it grey.

(b) Put s's white neighbours, $\{w, r\}$, into Q and colour them grey.

We are finished processing s, so colour it black.

Diagram from course text, Figure 22.3.

Example of Breadth-first Search



(c) Put w's white neighbours, $\{t, x\}$, into Q. We are finished with w.

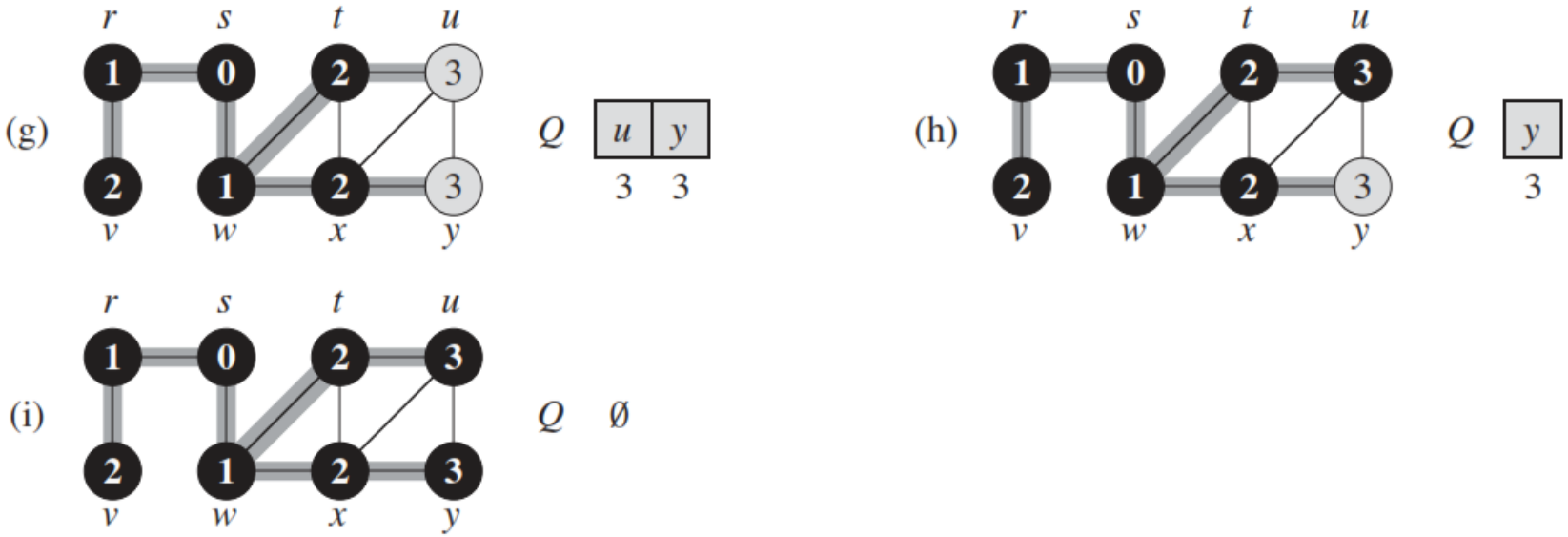
(d) Put r's white neighbour, $\{v\}$, into Q. We are finished with r.

(e) Put t's white neighbour, $\{u\}$, into Q. We are finished with t.

(f) Put x's white neighbour, $\{y\}$, into Q. We are finished with x.

Diagram from course text, Figure 22.3.

Example of Breadth-first Search



(g) Put v's white neighbours (none) into Q. We are finished with v.

(h) Put u's white neighbours (none) into Q. We are finished with u.

(i) Put y's white neighbours (none) into Q. We are finished with y.

Q is now empty, so the algorithm terminates.

Diagram from course text, Figure 22.3.

Distances in Breadth-first Search

Here we will use $\pi[i]$ to calculate the shortest distance between the "start" of the graph and each other vertex.

Modify the previous BFS algorithm by tracking distance.

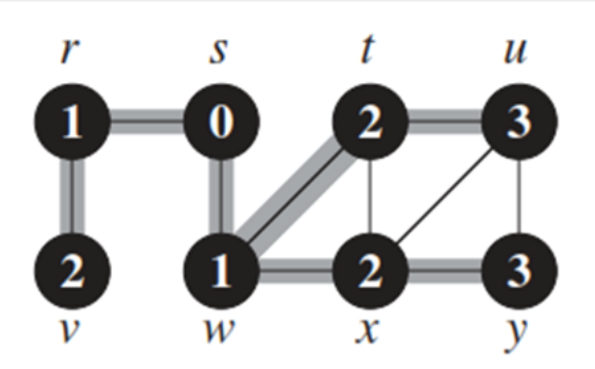
If $\{u, v\}$ is **any edge**, then $|dist[u] - dist[v]| \leq 1$.

If uv is a **tree edge**, then $dist[v] = dist[u] + 1$.

$dist[u]$ is the length of the **shortest path** from s to u.

$dist[u]$ is also called the **distance** from s to u.

Shortest Path



Distance from s

r:	1	v:	2
s:	0	w:	1
t:	2	x:	2
u:	3	y:	2

The example of Breadth-first search on the previous slides was also keeping track of the **distance** or **shortest path** from s.

When processing a vertex u, each time *BFS* encounters a white neighbour v, then the distance to $v = 1 +$ the distance to u.

Tracking the distance information only requires a simple modification to the *BFS* algorithm, i.e. adding and updating the $dist[v]$ array.

Diagram from course text, Figure 22.3.

Shortest Paths via Breadth-first Search

Algorithm: $BFS(G, s)$

for each $v \in V(G)$ **do** $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$colour[s] \leftarrow \text{gray}$

$\boxed{dist[s] \leftarrow 0}$

$InitializeQueue(Q)$

$Enqueue(Q, s)$

while $Q \neq \emptyset$

do $\begin{cases} u \leftarrow Dequeue(Q) \\ \textbf{for each } v \in Adj[u] \\ \quad \textbf{do} \begin{cases} \textbf{if } colour[v] = \text{white} \quad \textbf{then} \begin{cases} colour[v] = \text{gray} \\ \pi[v] \leftarrow u \\ Enqueue(Q, v) \\ \boxed{dist[v] \leftarrow dist[u] + 1} \end{cases} \end{cases} \\ colour[u] \leftarrow \text{black} \end{cases}$

Bipartite Graphs and Breadth-first Search

A graph is **bipartite** if the vertex set can be partitioned as $V = X \cup Y$, in such a way that all edges have one endpoint in X and one endpoint in Y.

A **cycle** is a path that starts and end at the same vertex.

An **odd cycle** is a cycle that has an odd number of vertices.

Prove that a graph is bipartite if and only if it does not contain an odd cycle?

- Assume the graph is bipartite. Colour the vertices in X blue and the vertices in Y red.
- Any path will alternate between red and blue vertices.
- Any path that traverses an odd number of edges will start and stop on different colours. Any path that traverses an even number of edges will start and stop on the same colour.
- A cycle has to stop and start on the same colour, so it cannot traverse an odd number of edges.

Bipartite Graphs and Breadth-first Search

Prove that a graph is bipartite if and only if it does not contain an odd cycle? (continued)

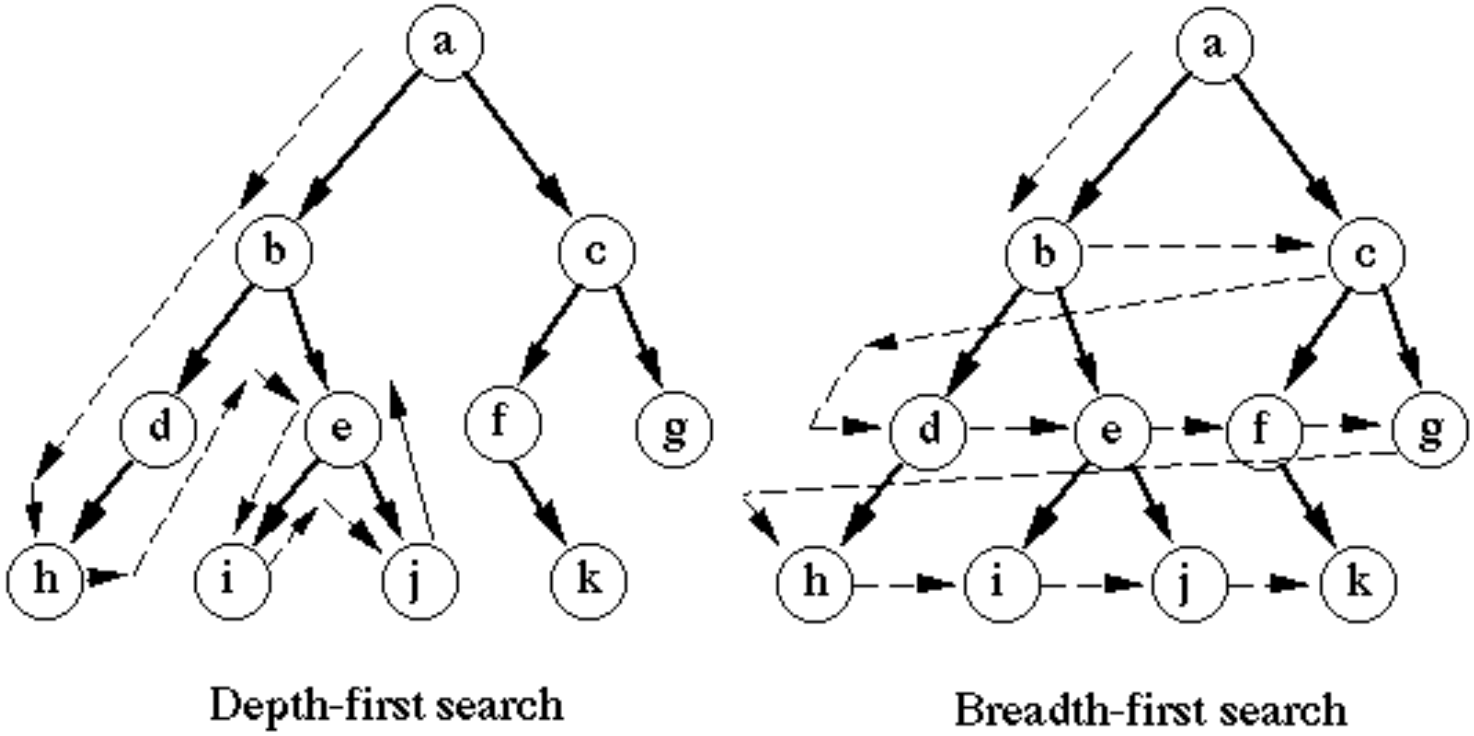
- For the converse, assume the graph does not contain any odd cycles.
- Pick a starting vertex, u.
- Let X contain any vertices that lie on a path that is an even number of edges from u and Y those vertices that lie on a path that is an odd number of edges from u.
- This assignment is consistent because if a vertex v had both a path with an even number of edges from u and an odd number of edges from u, then the two paths together would create an cycle with an odd number of edges, which is a contradiction.
- If the graph contains more than one component, repeat this procedure for each component.

Bipartite Graphs and Breadth-first Search

Design an algorithm based on *BFS* to test if a graph is bipartite.

- This is very similar to the 2nd part of the proof presented on the previous slide.
- If we encounter an edge $\{u, v\}$ with $dist[u] = dist[v]$, then G is not bipartite, whereas
- If no such edge is found, then define $X = \{u : dist[u] \text{ is even}\}$ and $Y = \{u : dist[u] \text{ is odd}\}$; then X, Y forms a bipartition.

Recall: Depth-first Search vs. Breadth-first Search



The two major types of searches in a graph are:

- Depth-first Search (DFS) - if a node a has children, search any children of the first child, bm before consider a's other child(ren) c.
- Breadth-first Search (BFS) - if a node a has children, search all of its children, b and c, before consider grandchildren, d, e, f, and g.

Figure from: http://www.cse.unsw.edu.au/~billw/Justsearch.html

Depth-first Search of a Directed Graph

A **depth-first search** uses a **stack** (or **recursion**) instead of a queue.

Replacing recursion by an explicit stack is called **unrolling a recursive function** and is done to improve run time.

We define predecessors and colour vertices as in BFS.

- A white vertex is **undiscovered**.
- A gray vertex is **being processed**.
- A black vertex has been **completely processed**.

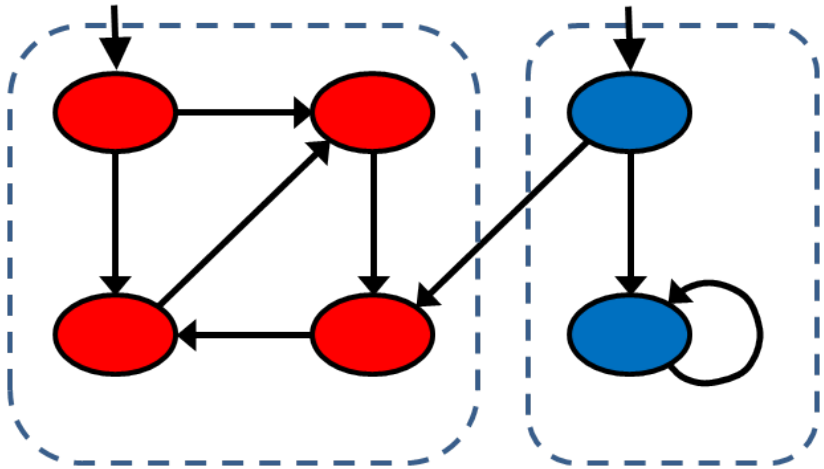
It is also useful to specify a **discovery time** $d[v]$ and a **finishing time** $f[v]$ for every vertex v.

We increment a **time counter** every time a value $d[v]$ or $f[v]$ is assigned.

$d[v]$ is when a vertex is first discover (i.e. it is coloured gray)

$f[v]$ is when we are finished with a vertex (i.e. it is coloured black)

Depth-first Search of a Directed Graph



We eventually visit all the vertices, and the algorithm constructs a **depth-first forest**.

A **forest** is just a collection of trees.

In the graph above, if we start at the upper left hand vertex, the DFS would result in a forest of two trees, one tree representing the red vertices and the other the blue.

Would the result be different if we started in the upper blue vertex?

Depth-first Search

Algorithm: *DFS*(G)
for each $v \in V(G)$
do $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$
$time \leftarrow 0$
for each $v \in V(G)$
do $\begin{cases} \textbf{if } colour[v] = \text{white} \\ \quad \textbf{then } \textit{DFSvisit}(v) \end{cases}$

Initialize by colouring each vertex, v, white and setting its predecessor, $\pi[v]$, to null.

Next, call *DFSvisit* on each white (i.e. undiscovered) vertex in the graph G.

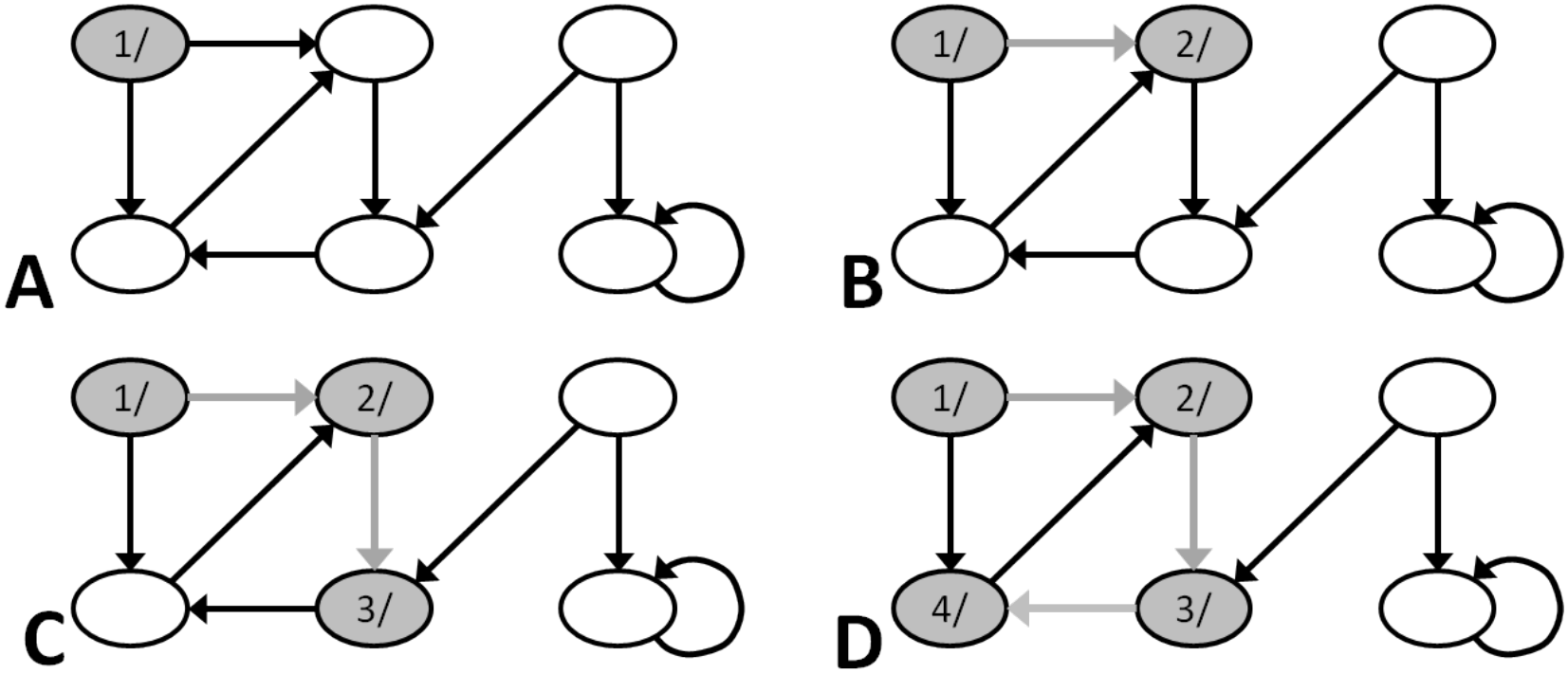
Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow$ gray
$time \leftarrow time + 1$
$d[v] \leftarrow time$
comment: $d[v]$ is the discovery time for vertex v
for each $w \in Adj[v]$
do { **if** $colour[w] =$ white
then { $\pi[w] \leftarrow v$
DFSvisit(w)
$colour[v] \leftarrow$ black
$time \leftarrow time + 1$
$f[v] \leftarrow time$
comment: $f[v]$ is the finishing time for vertex v

Note v was white before *DFSvisit* was called. Colour it grey then recursive process all its white children before colouring v black.

Depth-first Search



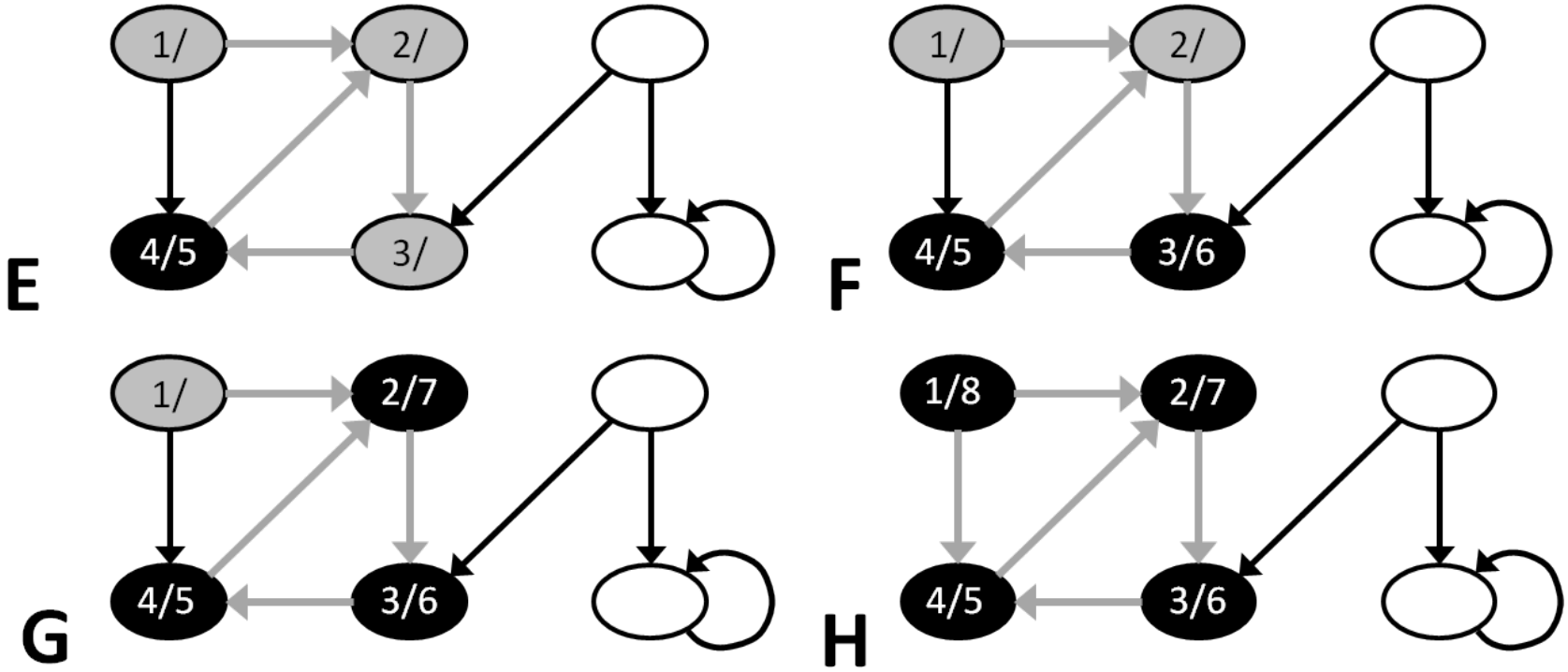
A: Start in upper left vertex, color it gray

B: Look at first descendant of 1/, color it gray

C: Look at first descendant of 2/, color it gray

D: Look at first descendant of 3/, color it gray

Depth-first Search



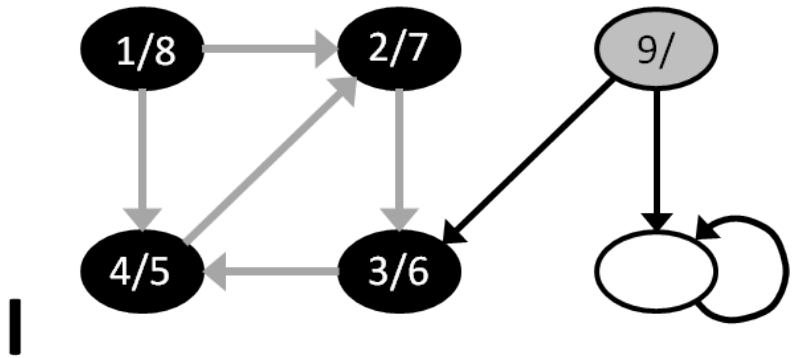
E: Descendant(s) of 4/5 are all gray, color 4/5 black

F: No more descendants of 3/6, color it black

G: No more descendants of 2/7, color it black

H: Descendant(s) of 1/8 are all colored, color 1/8 black

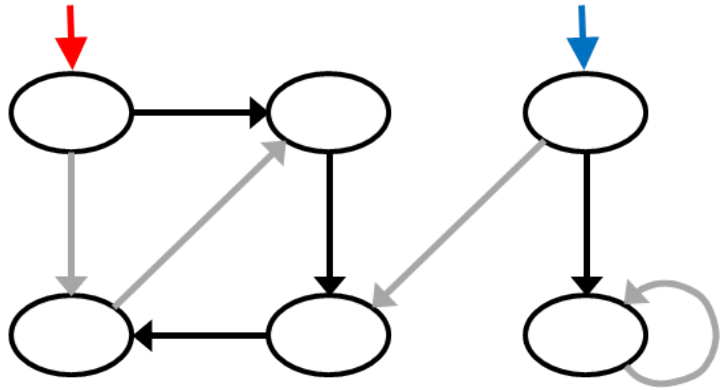
Depth-first Search



I: Begin process over again with vertex in upper right hand corner. This new vertex will be the root of a new tree.

Classification of Edges in Depth-first Search

- uv is a **tree edge** if $u = \pi[v]$
- uv is a **forward edge** if it is not a tree edge, and v is a descendant of u in a tree in the depth-first forest
- uv is a **back edge** if u is a descendant of v in a tree in the depth-first forest
- any other edge is a **cross edge**.



- If DFS was done on the graph above, starting at the red arrow, then the blue, where would any forward, back or cross edges occur?

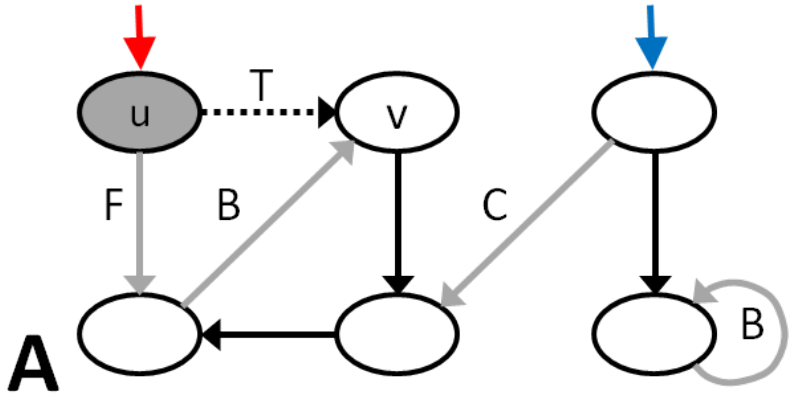
Properties of Edges in Depth-first Search

In the following table, we indicate the colour of a vertex v when an edge uv is discovered, and the relation between the start and finishing times of u and v, for each possible type of edge uv.

edge type	colour of v	discovery/finish times
tree	white	$d[u] < d[v] < f[v] < f[u]$
back	gray	$d[v] < d[u] < f[u] < f[v]$
forward	black	$d[u] < d[v] < f[v] < f[u]$
cross	black	$d[v] < f[v] < d[u] < f[u]$

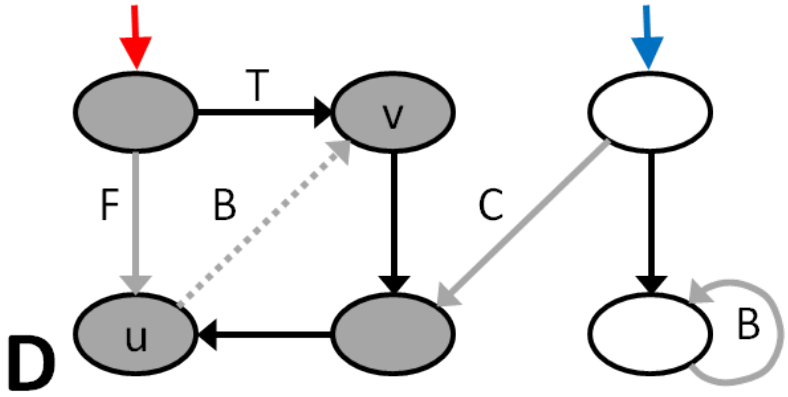
Observe that two intervals $(d[u], f[u])$ and $(d[v], f[v])$ never **overlap**. Two intervals are either **disjoint** or **nested**. This is sometimes called the parenthesis theorem.

Depth-first Search - tree edge



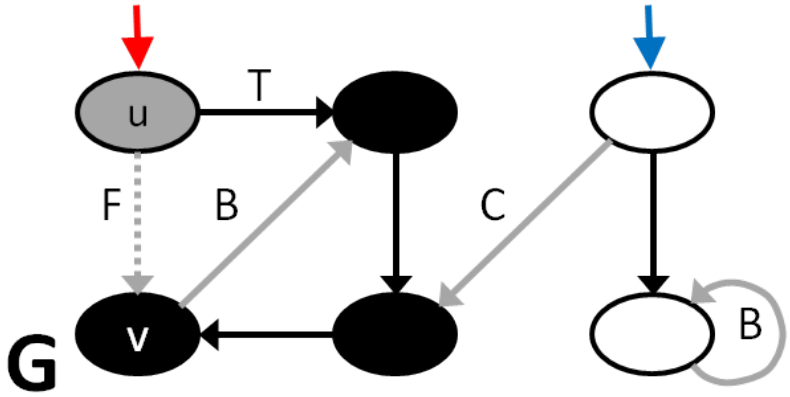
edge type	colour of v	discovery/finish times
tree	white	$d[u] < d[v] < f[v] < f[u]$

Depth-first Search - back edge



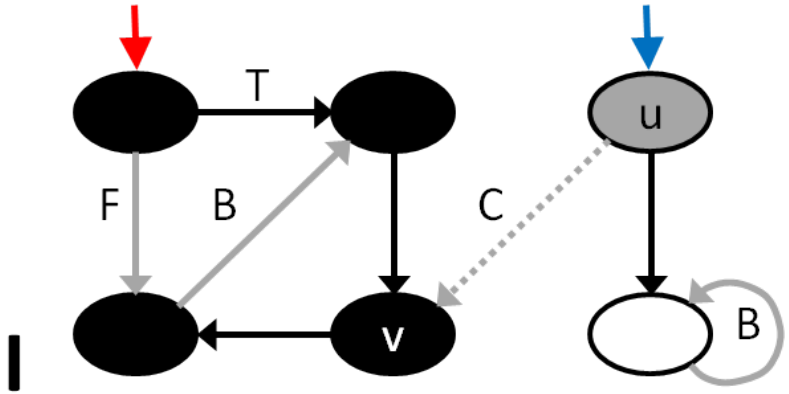
edge type	colour of v	discovery/finish times
back	gray	$d[v] < d[u] < f[u] < f[v]$

Depth-first Search - forward edge



edge type	colour of v	discovery/finish times
forward	black	$d[u] < d[v] < f[v] < f[u]$

Depth-first Search - cross edge



edge type	colour of v	discovery/finish times
cross	black	$d[v] < f[v] < d[u] < f[u]$

Midterm

This is the end of the material that will be covered on the midterm.

Topological Orderings and DAGs

A directed graph G is a **directed acyclic graph**, or **DAG**, if G contains no directed cycle.

A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in V such that $u < v$ whenever $uv \in E$.

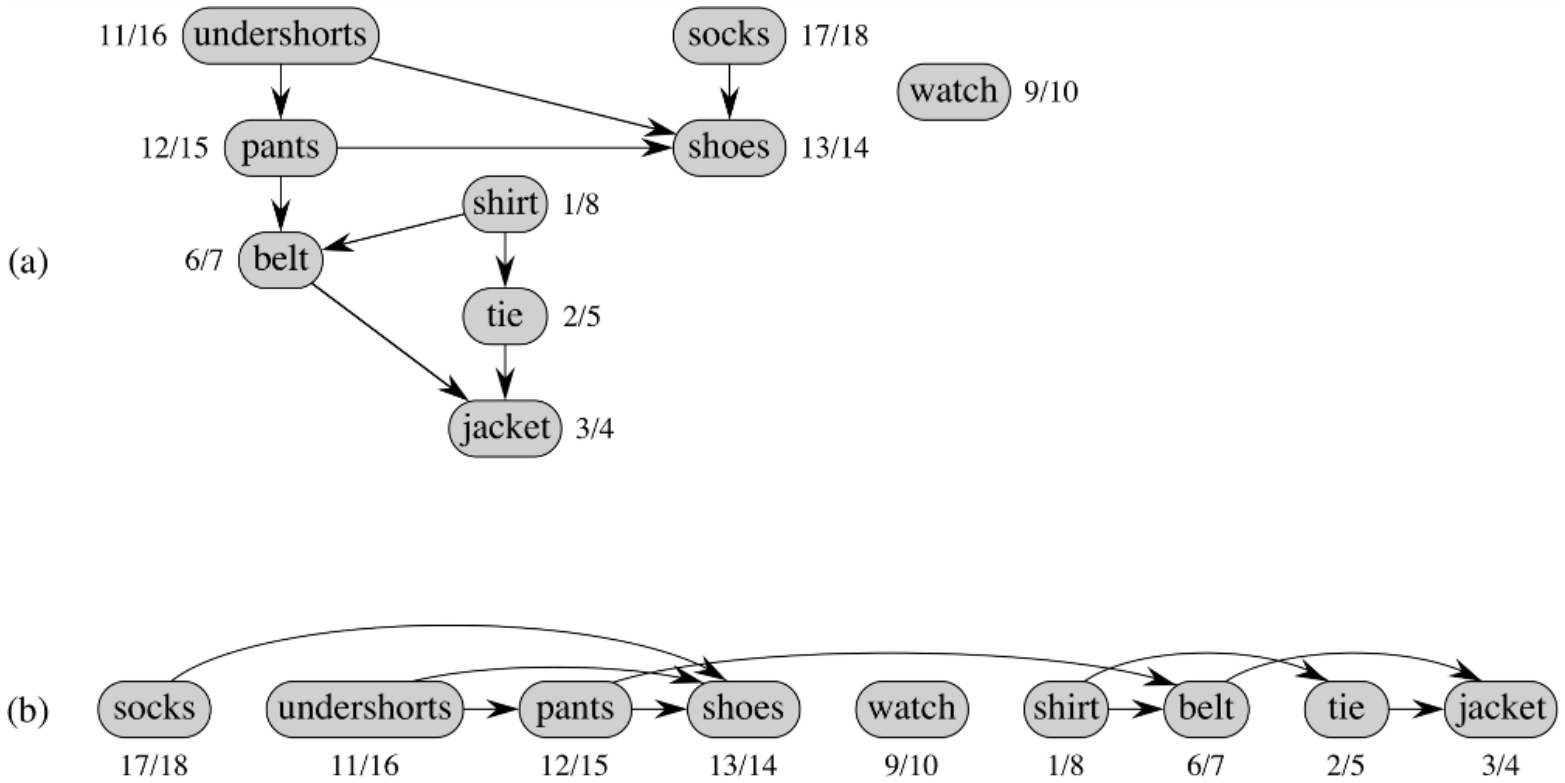
Are all trees DAGs?

Yes.

Are all DAGs trees?

No, for example there can be multiple roots in DAGs, but not in trees.

Topological Orderings and DAGs



a) Sample DAG and b) sample DAG that has been topologically ordered.

figure from course text, Figure 22-7

Topological Orderings and DAGs

Some useful facts:

A DAG contains a vertex of in-degree 0 (no edges coming into the vertex).

- *Pick a vertex at random. If it has in-degree 0 we are done.*
- *Otherwise go backwards on a path that leads to this vertex.*
- *Since there are no cycles (it is a DAG) you will eventually reach a vertex with in-degree 0.*

A directed graph G has a topological ordering if and only if it is a DAG.

- Assume the graph G has a topological sort.
- All the edges go in the same direction and none can return to the starting point.

Topological Orderings and DAGs

A directed graph G has a topological ordering if and only if it is a DAG (continued).

- Assume the graph G is a DAG.
- Proof by induction.
- Base case: true if $n = 1$, no cycles, hence no edges, so it can be topologically sorted.
- Inductive case: find a vertex v with no incoming arcs.
- Remove it from the G. By inductive hypothesis the graph G' has a topological ordering.
- Adding v to the front of G' will preserve the topological ordering since v has no incomining arcs.

Topological Orderings and DAGs

Some useful facts:

A directed graph G is a DAG if and only if a DFS of G has no back edges.

- *If uv is a back edge then the path from v to u followed by the arc uv forms a cycle.*
- *If the graph has a cycle consider the vertex, v, in the cycle with the largest discover time $d[v]$. It connects to a vertex with a smaller discovery time, hence it is a back edge.*

If uv is an edge in a DAG, then a DFS of G has $f[v] < f[u]$.

- *If v is white, BFS will finish with v before returning to u*
- *If v is gray, it is not a DAG, it has a back edge.*
- *If v is black, it means $f[v]$ has already occurred but $f[u]$ has yet to occur.*

Topological Orderings and DAGs

To compute a topological ordering of the graph G,

- Modify vanilla DFS by adding a flag DAG and a stack S.
- Find a white node v with no incoming edges and order it first (i.e. add it to the stack S).
- Using recursion, topologically order all of v's children.
- Append the children after v (i.e. add them to the stack S).
- Remove v from consideration (i.e. colour it black).
- Have a flag, DAG, that is false if the graph is not a DAG, (i.e. runs into a gray vertex, which means it has a back edge).

In the algorithm below why is being gray ignored in *DFS*, but an error in *DFSvisit*?

Topological Ordering via Depth-first Search

Algorithm: *DFS*(G)

InitializeStack(S)

$DAG \leftarrow true$

for each $v \in V(G)$

do $\begin{cases} colour[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

$time \leftarrow 0$

for each $v \in V(G)$

do $\begin{cases} \textbf{if } colour[v] = \text{white} \\ \quad \textbf{then } DFSvisit(v) \end{cases}$

if DAG **then return** (S) **else return** (DAG)

Topological Ordering via Depth-first Search (cont.)

Algorithm: *DFSvisit*(v)

$colour[v] \leftarrow \text{gray}$

$time \leftarrow time + 1$

$d[v] \leftarrow time$

comment: $d[v]$ is the discovery time for vertex v

for each $w \in Adj[v]$

do
- **if** $colour[w] = \text{white}$
  - **then**
    - $\pi[w] \leftarrow v$
    - *DFSvisit*(w)
- **if** $colour[w] = \text{gray}$ **then** $DAG \leftarrow false$

$colour[v] \leftarrow \text{black}$

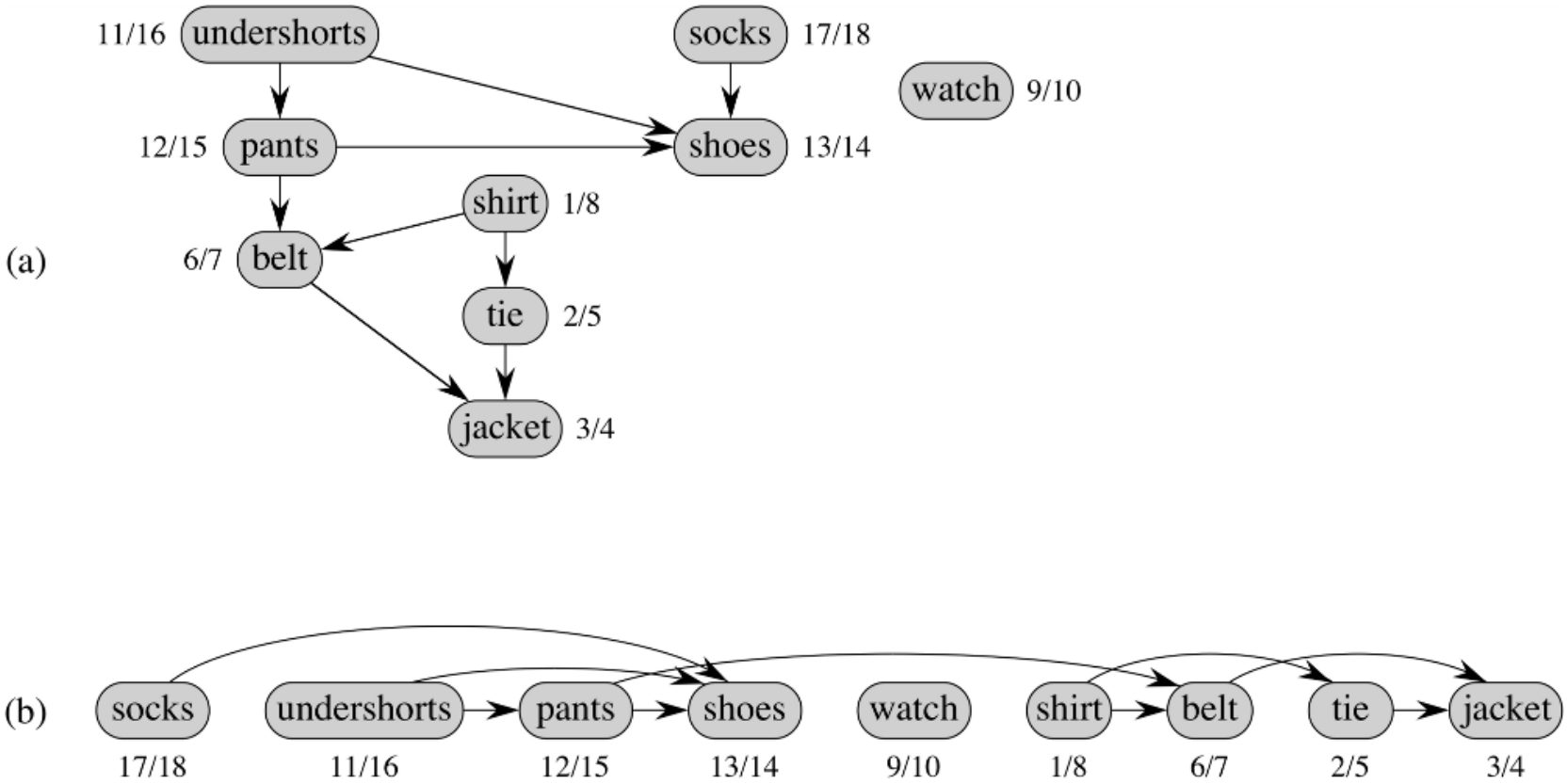
Push(S, v)

$time \leftarrow time + 1$

$f[v] \leftarrow time$

comment: $f[v]$ is the finishing time for vertex v

Topological Orderings and DAGs



Let us reconsider this slide in light of the topological ordering algorithm.

figure from course text, Figure 22-7

Topological Orderings and DAGs

Initially *DFS* calls *DFSVisit* on *shirt*(1).

DFSVisit follows the outgoing arc to *tie*(2) then *jacket*(3).

Jacket has no outgoing arcs, so its finishing time is calculated *jacket*(3/4), it is pushed onto the stack ($S =$ *jacket*) and *DFSVisit* returns to its predecessor, *tie*.

Tie has no more outgoing arcs, so it's finishing time is calculated, *tie*(2/5), it is pushed onto the stack ($S =$ *jacket, tie*) and *DFSVisit* returns to its predecessor, *shirt*.

Shirt does have another neighbour so the arc is followed to *belt*(6).

Belt has no white neighbours, so its finishing time is calculated *belt*(6/7), it is pushed onto the stack ($S =$ *jacket, tie, belt*) and *DFSVisit* returns to its predecessor, *shirt*.

Shirt has no more outgoing arcs, so its finishing time is calculated *shirt*(1/8), it is pushed onto the stack ($S =$ *jacket, tie, belt, shirt*) etc.

Strongly Connected Components of a Digraph G

For two vertices x and y of G, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from x to y **and** from y to x.

The relation $\sim$ is an **equivalence relation**.

The **strongly connected components** of G are the equivalence classes of vertices defined by the relation $\sim$.

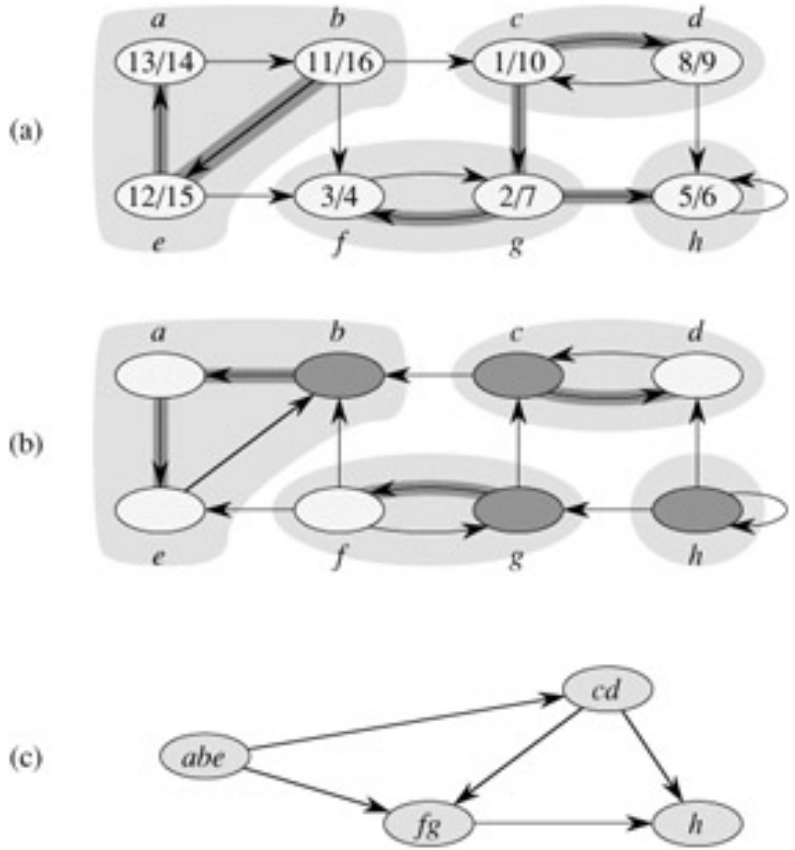
The **component graph** of G is a directed graph whose vertices are the strongly connected components of G. There is an arc from C_i to C_j if and only if there is an arc in G from some vertex of C_i to some vertex of C_j.

For a strongly connected component C, define $f[C] = \max\{f[v] : v \in C\}$ and $d[C] = \min\{d[v] : v \in C\}$.

Some interesting/useful facts:

- The component graph of G is a DAG.
- If C_i, C_j are strongly connected components, and there is an arc from C_i to C_j in the component graph, then $f[C_i] > f[C_j]$.

Strongly Connected Components of a Digraph



a) a directed graph G, b) the transpose graph $H = G^T$, c) the acyclic component graph G^{SCC}

image taken from course text Fig 22.9.

An Algorithm to Find the Strongly Connected Components

step 1 Perform a depth-first search of G, recording the finishing times $f[v]$ for all vertices v.

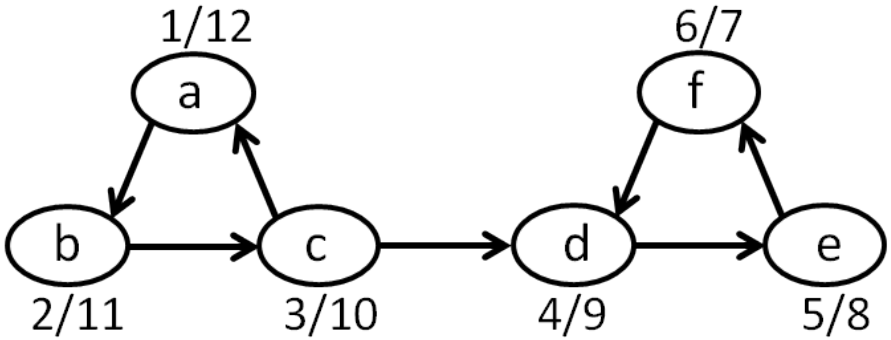
step 2 Construct a directed graph H from G by **reversing** the direction of all edges in G.

step 3 Perform a depth-first search of H, considering the vertices in **decreasing** order of the values $f[v]$ computed in step 1.

step 4 The strongly connected components of G are the trees in the depth-first forest constructed in step 3.

Why do we reverse the direction of the vertices and start at the highest value of $f[v]$?

Strongly Connected Components of a Digraph

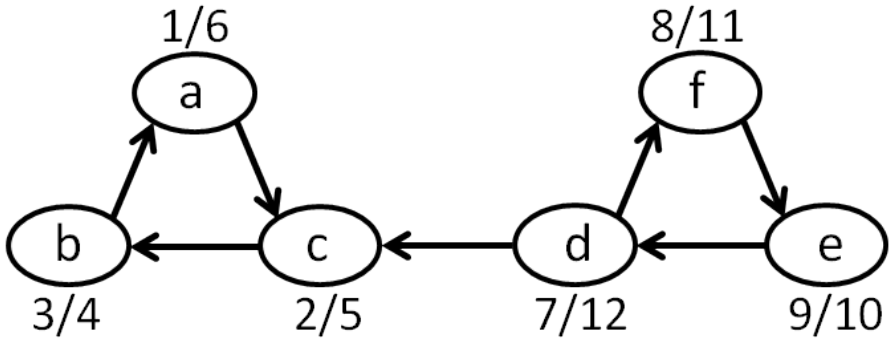


This graph consists of two strongly connected components: $\{a, b, c\}$ and $\{d, e, f\}$

The discovery and finishing times are for a DFS starting at a.

Because $\{d, e, f\}$ are **reachable** from a, their finishing times $\{7, 8, 9\}$ are **less than** the finishing time of a, i.e. 12.

Strongly Connected Components of a Digraph



This graph has the same vertices as before, but every edge has been reversed.

The discovery and finishing times are for a DFS starting at a and then d.

Because $\{d, e, f\}$ are **not reachable** from a, their finishing times $\{10, 11, 12\}$ are **greater than** the finishing time of a, i.e. 6.

Reversing the edges does not prevent a DFS starting at a from reaching vertices in its own strongly connected component, i.e. $\{b, c\}$, but it does prevent from reaching vertices in another strongly connected component, i.e. $\{d, e, f\}$.

Depth-first Search of H

Assume that $f[v_{i_1}] > f[v_{i_2}] > \cdots > f[v_{i_n}]$.

Algorithm: *DFS*(H)

for $j \leftarrow 1$ **to** n
  do $colour[v_{i_j}] \leftarrow$ white

$\boxed{scc \leftarrow 0}$

for $j \leftarrow 1$ **to** n
  do **if** $colour[v_{i_j}] =$ white
    then $\boxed{scc \leftarrow scc + 1}$
      DFSvisit$(H, v_{i_j}, \boxed{scc})$

return $(comp)$

Comment: scc is the name given to each strongly connected component. It is incremented by one each time a new strongly connected component is found.

DFSvisit for H

Algorithm: *DFSvisit*$(H, v, \boxed{scc})$

$colour[v] \leftarrow \text{gray}$

$\boxed{comp[v] \leftarrow scc}$

for each $w \in Adj[v]$

do **if** $colour[w] = \text{white}$
then *DFSvisit*$(H, w, \boxed{scc})$

$colour[v] \leftarrow \text{black}$

Comment: $comp$ is a global array that stores the strongly connected component of each vertex v.

An Algorithm to Find the Strongly Connected Components

- scc - the "name" we give to each strongly connect component, actually the integers $1, 2, 3, \ldots$
- $comp$ - a global array that gives the "name" or scc label of each vertex, $comp[v]$ is the name of the component that vertex v is in.
- *DFS* - colour all vertices white then call *DFSvisit* on all white, i.e. unvisited, vertices.
- *DFSvisit* - these are all vertices in the same strongly connected component, so colour them all black and label their component number, $comp[v]$, with the current value of scc.

Minimum Spanning Trees

A **spanning tree** in a connected, undirected graph $G = (V, E)$ is a subgraph T that is a tree which contains every vertex of V.

T is a spanning tree of G if and only if T is an acyclic subgraph of G that has $n - 1$ edges (where $n = |V|$).

Problem

Minimum Spanning Tree

Instance: *A connected, undirected graph $G = (V, E)$ and a* **weight function** $w : E \to \mathbb{R}$.

Find: *A spanning tree T of G such that*

$$\sum_{e \in T} w(e)$$

is minimized (this is called a **minimum spanning tree**, *or* **MST**).

A General Greedy Algorithm to Find an MST

Algorithm: *GreedyMST*(G, w)

$A \leftarrow \emptyset$

while $|A| < n - 1$

do { find smallest *candidate* edge, e_j
if $A \cup \{e_j\}$ does not contain a cycle
then $A \leftarrow A \cup \{e_j\}$

return (A)

- A is a growing set of edges that will eventually become the MST.
- A is a forest during *Kruskal*'s algorithm.
- A is a single tree during *Prim*'s algorithm.

Kruskal's Algorithm

Sort edges by weight, i.e $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, where $m = |E|$).

Algorithm: *Kruskal*(G, w)

$A \leftarrow \emptyset$

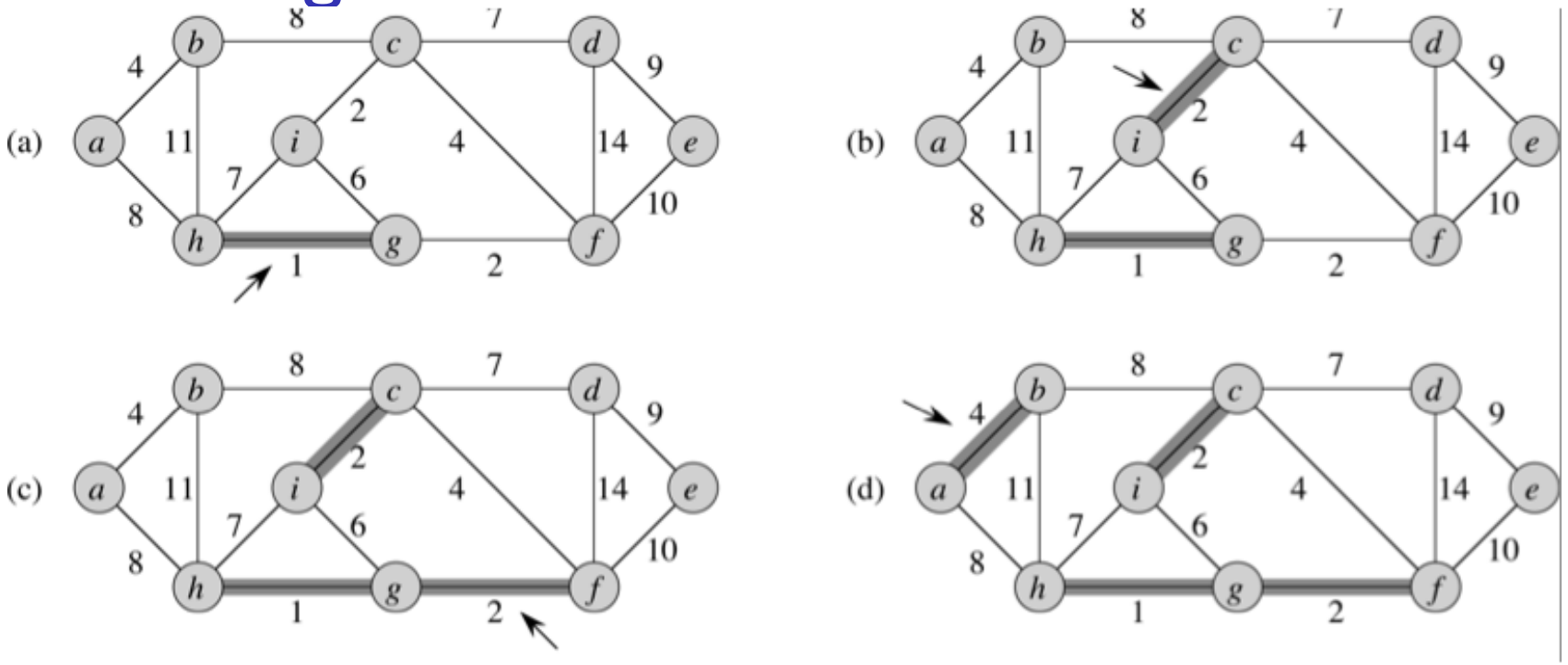
for $j \leftarrow 1$ **to** m

do **if** $A \cup \{e_j\}$ does not contain a cycle
then $A \leftarrow A \cup \{e_j\}$

return (A)

Need a data structure to check if $A \cup \{e_j\}$ contains a cycle. Represent each tree by a set of vertices (connected by that tree).

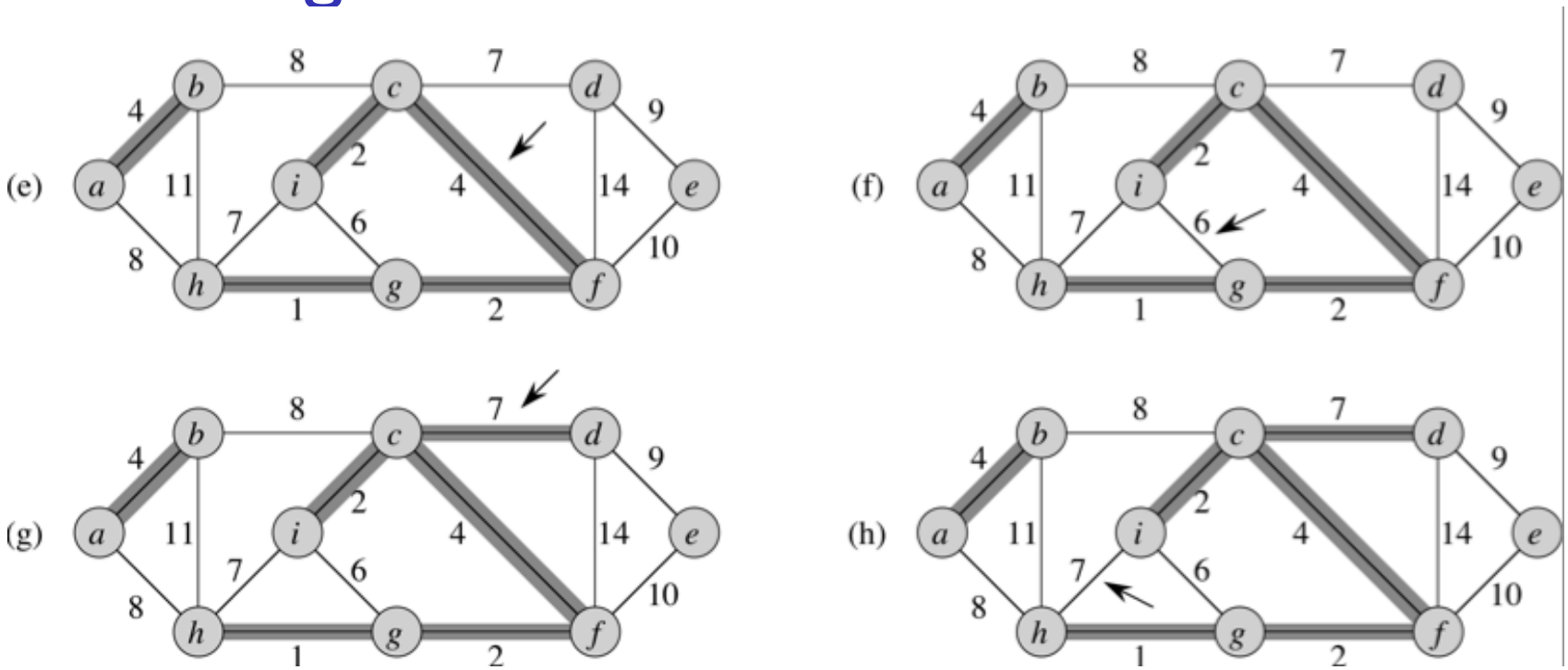
Kruskal's Algorithm



- (a) Sets: {a}, {b}, {c}, {d}, {e}, {f}, {g, h}, {i}
- (b) Sets: {a}, {b}, {c, i}, {d}, {e}, {f}, {g, h}
- (c) Sets: {a}, {b}, {c, i}, {d}, {e}, {f, g, h}
- (d) Sets: {a, b}, {c, i}, {d}, {e}, {f, g, h}

Image taken from course text, Fig 23.4.

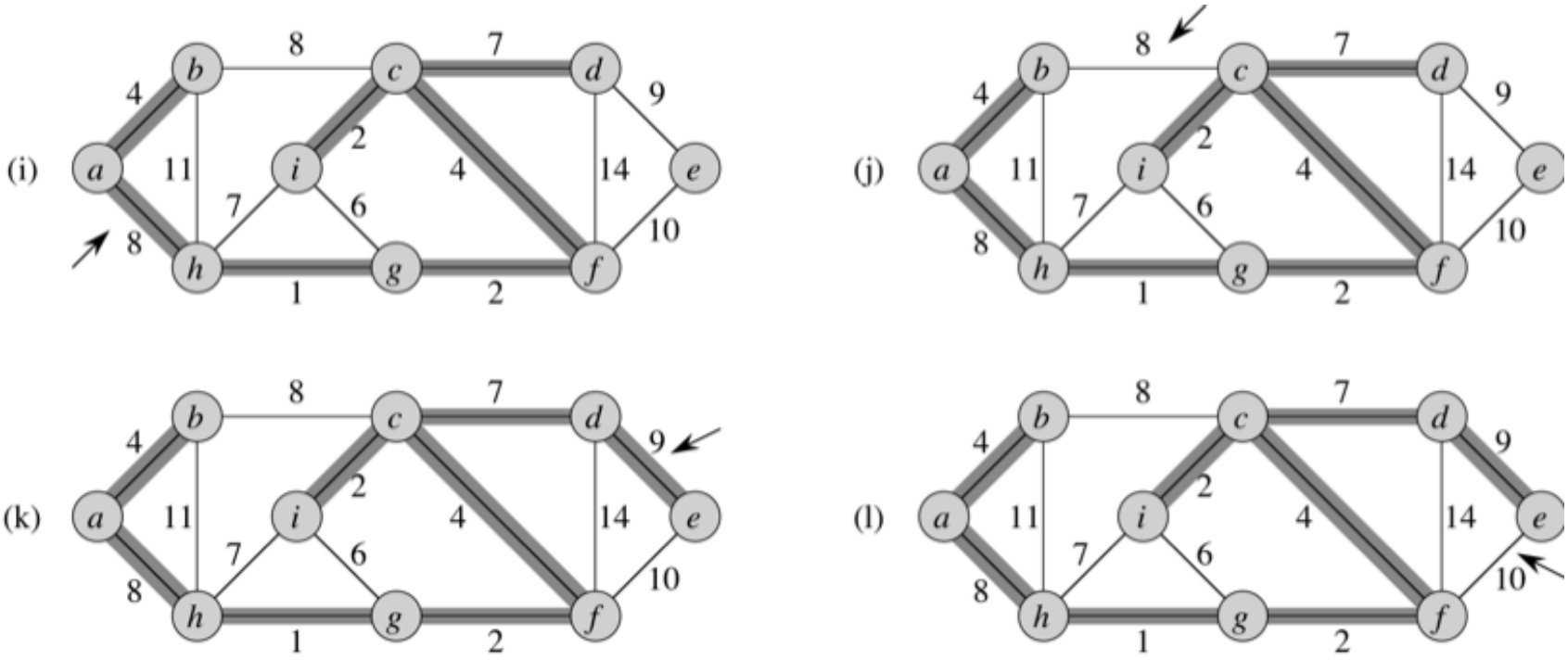
Kruskal's Algorithm



- (e) Sets: {a, b}, {d}, {e}, {c, f, g, h, i}
- (f) edge ig ignored
- (g) Sets: {a, b}, {e}, {c, d, f, g, h, i}
- (h) edge hi ignored

Image taken from course text, Fig 23.4.

Kruskal's Algorithm



- (i) Sets: {e}, {a, b, c, d, f, g, h, i}
- (j) edge bc ignored
- (k) Sets: {a, b, c, d, e, f, g, h, i}

Image taken from course text, Fig 23.4.

Kruskal's Algorithm

What sort of data structure would be useful for Kruskal's algorithm?

Use a Union-Find data structure for disjoint sets (a CS466 topic).

What is the running time for Kruskal's algorithm?

Sorting the edges takes $\Theta(m \lg m) = \Theta(m \lg n)$ *where* m *is the number of edges and* n *the number of vertices. The Find part of Union-Find is* $O(\log n)$ *and it is called* m *times, so overall it* $\Theta(m \lg n)$

Is Kruskal's algorithm optimal? (Hint: proof by contradiction).

- Let $K = \{k_1, k_2, \ldots k_n\}$ be the edges selected by Kruskal's algorithm and let $S = \{s_1, s_2, \ldots s_n\}$ be the edges of a spanning tree that has lower cost than Kruskal's algorithm, sorted from least to most expensive edge.
- If there is more than one spanning tree that beats Kruskal's algorithm pick the one that **first** differs the furtherest along the sorted list, say $s_d \neq k_d$, etc.

Kruskal's Algorithm

- Add edge k_d to the tree S. This creates a cycle.
- There must be at least one edge in this cycle, say s_i that is not in K, since K has no cycles.
- Remove s_i. Say edge s_i connected vertex a to vertex b. You can still get from a to b through the edge k_d since adding it created a cycle that included s_i.
- The cost of this new tree is cost(S) + cost(k_d) - cost(s_i).
- Since the edges lists are sorted by cost and agree for the first $d-1$ edges, cost(s_i) is greater than or equal to cost(k_d).
- This new tree s_i replaced by k_d has cost less than or equal to S but disagrees with K at a value greater than d contradicting the assumption that S was the minimal spanning tree that first disagreed the furtherest along the sorted edge list.

Prim's Algorithm (idea)

A is the answer that we are constructing, i.e. a set of edges.

We initially choose an arbitrary vertex u_0 to start and define $A = \{e\}$, where e is the **minimum weight** edge incident with u_0. In the example on the next page $u_0 = a$ and $e = \{a, b\}$.

A is always a **single tree**, and at each step we select the minimum weight edge, v, that joins a vertex in A to a vertex not in A.

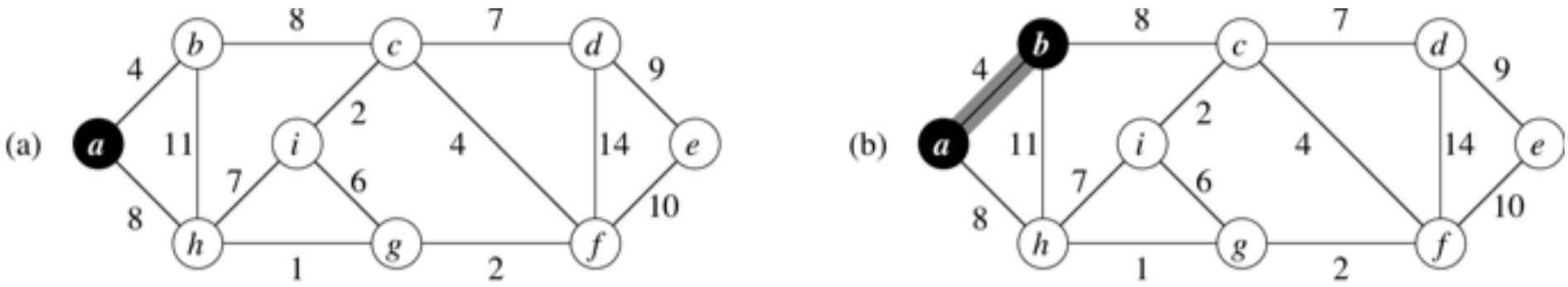
For a vertex $v \notin VA$ define (the set of vertices in the tree A) define

$$\begin{aligned} N[v] &= \text{a minimum weight edge } \{u, v\} \text{ such that } u \in VA \\ W[v] &= w(N[v], v). \end{aligned}$$

Assume $w(u, v) = \infty$ if $\{u, v\}$ does not have one of its vertices in A.

Over time, we adjust the weights of edges that are outside A but adjacent to a vertex in A.

Prim's Algorithm

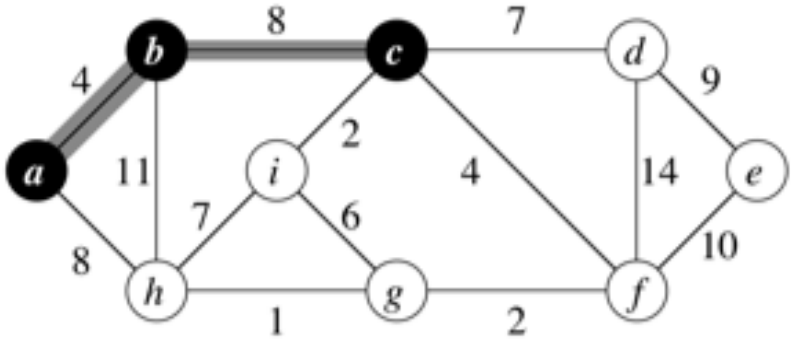


- Initial starting point is vertex a, i.e. VA $= \{ a \}$.
- Update the weight of all edges leaving VA.
- (a) W[b]=4, W[h]=8, rest weigh ∞
- The minimun edge length is 4, add b to VA, i.e VA $= \{a, b\}$.
- Update the weight of all edges leaving VA.
- (b) W[c]=8, W[h]=8, rest weigh ∞

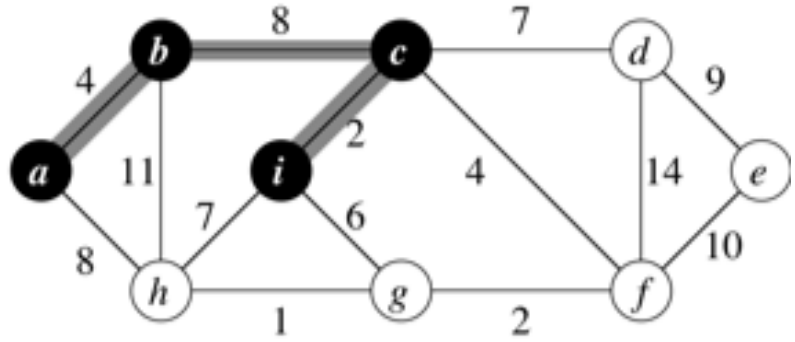
Image on this and next two pages taken from course text, Fig 23.5.

Prim's Algorithm

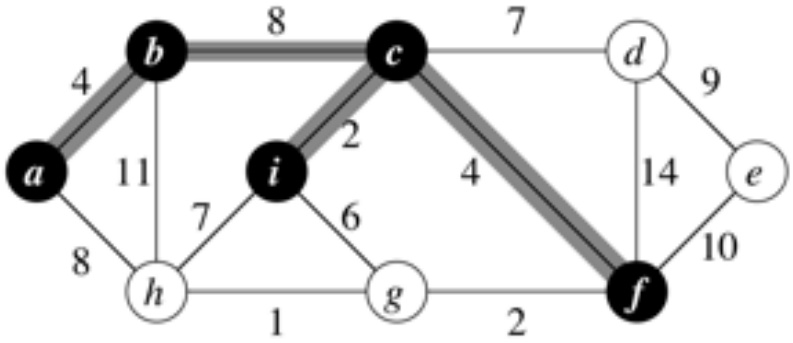
(c)



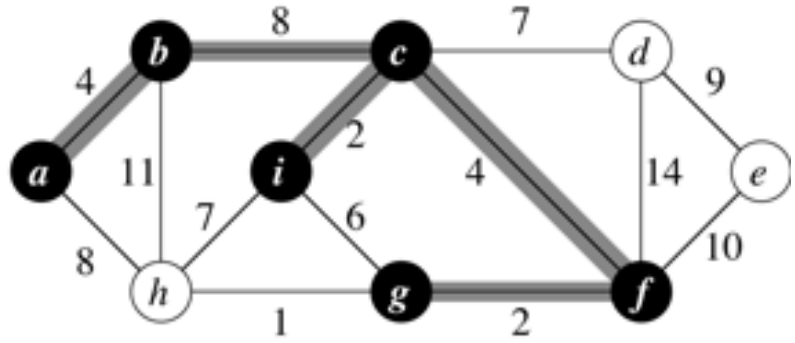
(d)



(e)



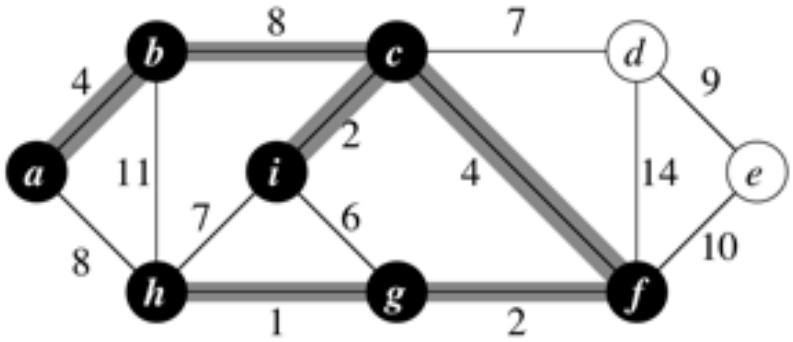
(f)



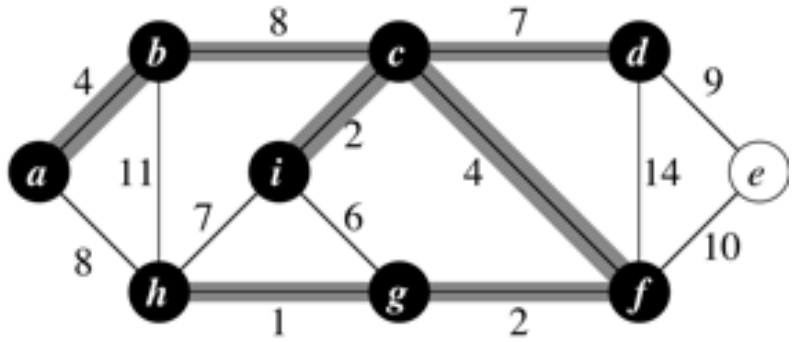
- (c) W[i]= 2, W[f]= 4, W[d]= 7, W[h]=8, W[e] = W[g] = ∞
- (d) W[f]= 4, W[g]=6, W[d]= 7, W[h]=7, W[e] = ∞
- (e) W[g]=2, W[d]= 7, W[h]=7, W[e]=10,
- (f) W[h]=1, W[d]= 7, W[e]=10,

Prim's Algorithm

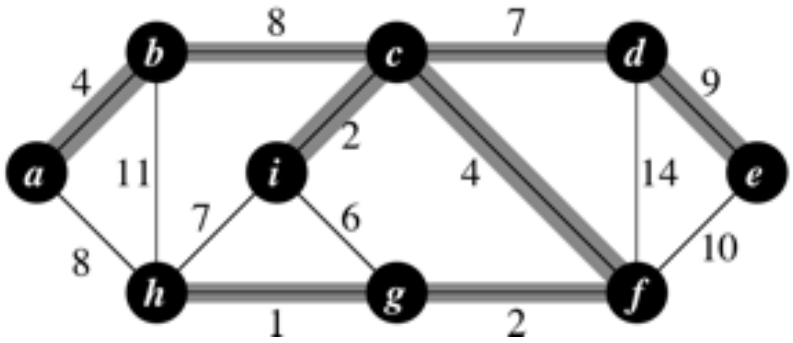
(g)



(h)



(i)



- (g) W[d]= 7, W[e]=10,
- (h) W[e]=9
- (i) done

Prim's Algorithm

Algorithm: *Prim*(G, w)

$A \leftarrow \emptyset$

$VA \leftarrow \{u_0\}$, where u_0 is arbitrary

for all $v \in V \backslash \{u_0\}$

do $\begin{cases} W[v] \leftarrow w(u_0, v) \\ N[v] \leftarrow u_0 \end{cases}$

while $|A| < n - 1$

do $\begin{cases} \text{choose } v \in V \backslash VA \text{ such that } W[v] \text{ is minimized} \\ VA \leftarrow VA \cup \{v\} \\ u \leftarrow N[v] \\ A \leftarrow A \cup \{uv\} \\ \textbf{for all } v' \in V \backslash VA \\ \quad \textbf{do} \begin{cases} \textbf{if } w(v, v') < W[v'] \\ \quad \textbf{then} \begin{cases} W[v'] \leftarrow w(v, v') \\ N[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$

return (A)

Prim's Algorithm - Variables

A - the answer that we are constructing, i.e. a set of edges.

VA - the vertices that are in the tree so far. VA is expanding throughout the algorithm.

$V \backslash VA$ - the vertices that are not in VA so far. $V \backslash VA$ is shrinking throughout the algorithm.

$V \backslash \{u_0\}$ - the vertices not yet in VA, at the beginning.

$W[v]$ - the cost of connecting v to A, i.e. the weight of the edge that connects v to some vertex in A.

$N[v]$ - the vertex that connects v to A with the lowest weight.

Prim's Algorithm

What sort of data structure would be useful for Prim's algorithm?

Priority Queue (i.e. a heap).

What sort running time would Prim's algorithm have?

The *Insert*, *Decrease-Key*, and *Delete-Key* operations are all $O(\log n)$ where n is the number of elements in the Priority Queue (i.e. heap).

How would you go about proving Prim's algorithm?

Very similar to that of Kruskal's algorithm. Could do it by induction. To go from case $n-1$ to n use the same idea that adding the greedy edges is just as inexpensive as the optimal solution's choice.

Single Source Shortest Paths

Problem

Single Source Shortest Paths

Instance: *A directed graph $G = (V, E)$, a non-negative* **weight function** $w : E \to \mathbb{R}^+ \cup \{0\}$, *and a* **source vertex** $u_0 \in V$.

Find: *For every vertex $v \in V$, a directed path P from u_0 to v such that*

$$w(P) = \sum_{e \in P} w(e)$$

is minimized.

The term **shortest path** really means **minimum weight path**.

We are asked to find n different shortest paths, one for each vertex $v \in V$.

If all edges have weight 1, we can just use *BFS* to solve this problem.

Dijkstra's Algorithm (Main Ideas)

S is a subset of vertices such that the shortest paths from u_0 to all vertices in S are known; initially, $S = \{u_0\}$. One of these vertices may be coloured grey (our current location) and the rest will be coloured black.

The vertices in $V \backslash S$ will be coloured white.

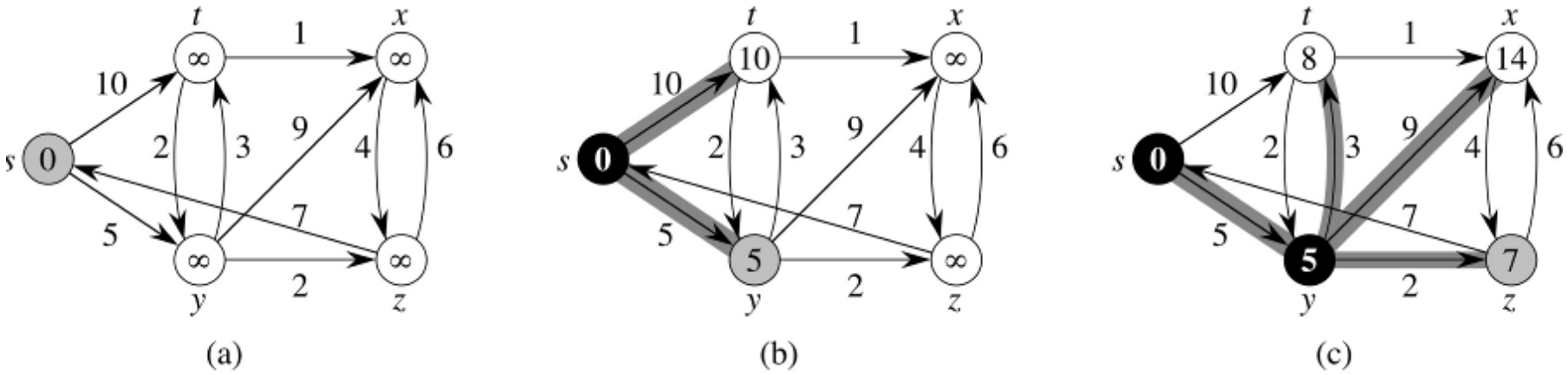
For all vertices $v \in S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v, and all vertices on P_v are in the set S.

For all vertices $v \notin S$, $D[v]$ is the weight of the shortest path P_v from u_0 to v in which all interior vertices are in S.

For $v \neq u_0$, $\pi[v]$ is the **predecessor** of v on the path P_v.

At each stage of the algorithm, we choose $v \in V \backslash S$ so that $D[v]$ is minimized, and we add v to S (outer bottom `do loop`). Then we update the vertices adjacent to v. If going through v creates a shorter path, then D and π are updated appropriately (inner bottom `do loop`).

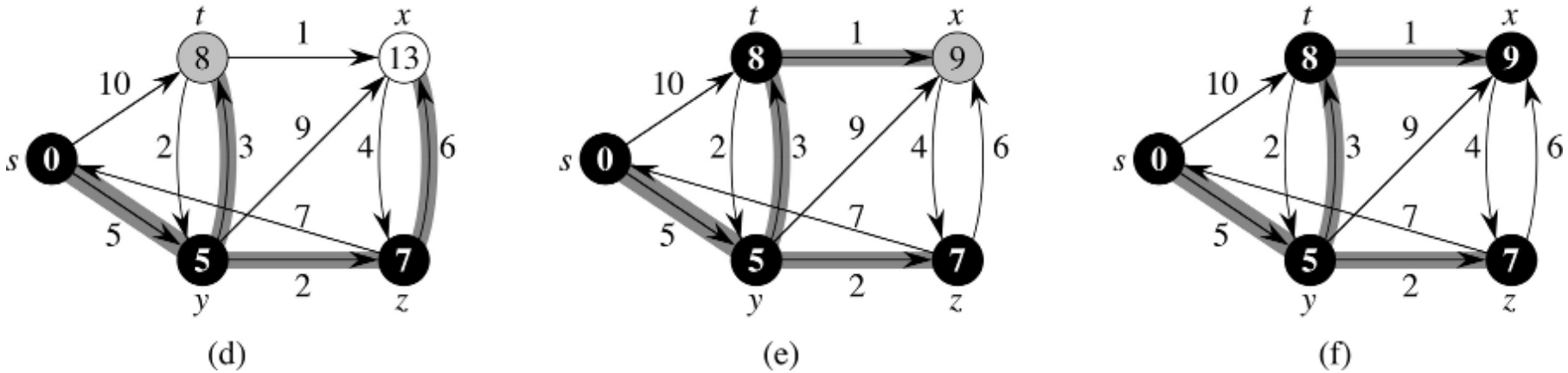
Dijkstra's Algorithm



- (a) Pick min, i.e. s,
- (b) set D[y]=0+5=5, D[t]=0+10=10, pick min, i.e. y
- (c) set D[t]= 5+3=8, D[x]=5+9=14, D[z]=5+2=7, pick min, i.e. z

Image taken from course text, Fig 24.6.

Dijkstra's Algorithm



- (d) set D[x]=7+6=13, keep D[s]=0, Pick min, i.e. t,
- (e) set D[x]=8+1=9, keep D[y]=5, pick min, i.e. x
- (f) all nodes black, done

Image taken from course text, Fig 24.6.

Dijkstra's Algorithm

Algorithm: *Dijkstra*(G, w, u_0)

$S \leftarrow \{u_0\}$

$D[u_0] \leftarrow 0$

for all $v \in V \backslash \{u_0\}$

do $\begin{cases} D[v] \leftarrow w(u_0, v) \\ \pi[v] \leftarrow u_0 \end{cases}$

while $|S| < n$

do $\begin{cases} \text{choose } v \in V \backslash S \text{ such that } D[v] \text{ is minimized} \\ S \leftarrow S \cup \{v\} \\ \textbf{for all } v' \in V \backslash S \\ \quad \textbf{do} \begin{cases} \textbf{if } D[v] + w(v, v') < D[v'] \\ \quad \textbf{then} \begin{cases} D[v'] \leftarrow D[v] + w(v, v') \\ \pi[v'] \leftarrow v \end{cases} \end{cases} \end{cases}$

return (D, π)

Finding the Shortest Paths

Algorithm: *FindPath*(u_0, π, v)

$path \leftarrow v$

$u \leftarrow v$

while $u \neq u_0$

do $\begin{cases} u \leftarrow \pi[u] \\ path \leftarrow u \parallel path \end{cases}$

return $(path)$

Finding the path from u_0 to v with the predecessor array $\pi[i]$

- start from the end, i.e. v,
- add the current vertex to the front of a FIFO queue, $path$
- go to its predecessor
- repeat until you are at the start of the path, u_0

Topic 5: Dynamic Programming Algorithms

This topic investigates Dynamic Programming Algorithms.

Readings for this topic from *Introduction to Algorithms, 3rd ed.*

- Section 15.3, Element of dynamic programming
- Chapter 15.4, Longest Common Subsequence
- Section 25.2, All-Pairs Shortest Paths

Computing Fibonacci Numbers Inefficiently

Recall the Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

Algorithm: *BadFib*(n)

if $n = 0$ **then** $f \leftarrow 0$

else if $n = 1$ **then** $f \leftarrow 1$

else
$$\begin{cases} f_1 \leftarrow \textit{BadFib}(n-1) \\ f_2 \leftarrow \textit{BadFib}(n-2) \\ f \leftarrow f_1 + f_2 \end{cases}$$

return (f);

Computing Fibonacci Numbers More Efficiently

Algorithm: *BetterFib*(n)
$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
  do $f[i] \leftarrow f[i-1] + f[i-2]$
return $(f[n])$

Why is this approach more efficient then the approach on the previous slide?

Answer: because the asymptotic time complexity of BetterFib is in $\Theta(n)$ *whereas BabFib is in* $\Theta(2^n)$.

Dynamic Programming Design Strategy

BadFib

- uses recursion
- stores data implicitly in system stack
- each subcase is recalculated many times
- top-down approach: start at n and work its way down to the base cases, $0, 1$, via recursion.

BetterFib

- uses a **for** loop
- store data explicitly in a table
- each subcase is calculated only once
- bottom-up approach: start at the base cases, $0, 1$, and work its way up to n by fill up the table.

Designing Dynamic Programming Algorithms for Optimization Problems

Optimal Structure

Examine the structure of an optimal solution to a problem instance I, and determine if an optimal solution for I can be expressed in terms of optimal solutions to certain **subproblems** of I.

E.g. If you know $\mathit{Fib}(n-1)$ and $\mathit{Fib}(n-2)$, you can easily calculate $I = \mathit{Fib}(n)$.

Define Subproblems

Define a set of subproblems $\mathcal{S}(I)$ of the instance I, the solution of which enables the optimal solution of I to be computed. I will be the last or largest instance in the set $\mathcal{S}(I)$.

E.g. $\mathcal{S}(I) = \{\mathit{Fib}(1), \mathit{Fib}(2), \ldots, \mathit{Fib}(n-1)\}$.

Designing Dynamic Programming Algorithms (cont.)

Recurrence Relation

Derive a **recurrence relation** on the optimal solutions to the instances in $\mathcal{S}(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $\mathcal{S}(I)$ and/or base cases.
E.g. $\mathit{Fib}(n) = \mathit{Fib}(n-1) + \mathit{Fib}(n-2)$.

Compute Optimal Solutions

Compute the optimal solutions to all the instances in $\mathcal{S}(I)$ using the recurrence relation in a **bottom-up** fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to I.

0-1 Knapsack

Problem

Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An* n*-tuple* $X = [x_1, \ldots, x_n]$, *where* $x_i \in \{0, 1\}$ *for* $1 \leq i \leq n$, *and*

$$\sum_{i=1}^{n} w_i x_i \leq M.$$

Find: *A feasible solution* X *that maximizes*

$$\sum_{i=1}^{n} p_i x_i.$$

0-1 Knapsack

The Dynamic Programming Table $P[i, m]$

- Table Entry: the entry $P[i, m]$ gives the optimal (largest) *profit* for all possible combinations of items $1, 2, \ldots, i$ whose total weight is less than or equal to m.
- Items: first consider item $\{1\}$, then items $\{1, 2\}$, then items $\{1, 2, 3\}$, etc. These represent the 1^{st}, 2^{nd}, and 3^{rd} *rows* of the table.
- Weight Capacity: track optimal answers for various weigh levels, m, lower than or equal to our knapsack capacity, M. Consider items that add up to weight 1, then weight 2, then weight 3, etc. These represent the 1^{st}, 2^{nd}, and 3^{rd} *columns* of the table.

0-1 Knapsack

Use the n x M table, $P[i, m]$, when deciding to add item i with profit p_i and weight w_i. Ask which of the two options results in a higher profit?

- Keep our previous choice for this weight level m, which did not include item i, i.e. $P[i, m] = P[i-1, m]$.
- Add item i with weight w_i and profit p_i. Then $P[i, m] = P[i-1, m-w_i] + p_i$, i.e. take the profit associated with weight level $m - w_i$ and add p_i.

In either case you are answering a question about whether or not it is better to include the i^{th} element, i.e. filling out the i^{th} row of the table, in terms of the answers you have previously stored in the $i - 1^{th}$ row of the table (i.e. concerning items $1, 2, \ldots i-1$).

0-1 Knapsack

Filling in the table $P[i, m]$

If you are on row i, representing item i, there are two ways of framing the question. Suppose item i has weight w_i and profit p_i.

1. Is $P[i-1, m-w_i] + p_i > P[i-1, m]$?
2. Is $P[i-1, m] + p_i > P[i-1, m+w_i]$?

What is the difference between the two methods?

Ans: the first method is for calculating the entry P[i, m] and the second method, entry P[i, m+w_i].

- In both cases you answer the question about row i in terms of row $i-1$: Is it more profitable to take the answer from the previous row (i.e. do not include item i) or include item i and add its weight w_i?
- To be able to answer this question we must track the profit for all previous items $\{1, 2, \ldots, i-1\}$ and all previous weight levels $\{1, 2, \ldots, m-1\}$.

A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: *0-1Knapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M)$

for $m \leftarrow 0$ **to** M

do { **if** $m \geq w_1$
  then $P[1, m] \leftarrow p_1$
  else $P[1, m] \leftarrow 0$

for $i \leftarrow 2$ **to** n

do { **for** $m \leftarrow 0$ **to** M

do { **if** $m < w_i$
  then $P[i, m] \leftarrow P[i-1, m]$
  do { **else** $P[i, m] \leftarrow \max\{P[i-1, m-w_i] + p_i, P[i-1, m]\}$

return $(P[n, M])$;

A Dynamic Programming Algorithm for 0-1 Knapsack

Summary of the Algorithm

- The first **for ... do** loop fills in the first row of the table. It includes item 1 when the value of the current weight level (m) is high enough (i.e. when $m \geq w_1$).
- The second, nested pair of **for ... do** loops fills in the rest of the table. In each case, when the current weight level is high enough (i.e. when $m \geq w_i$), it maximizes profit ($P[i, m]$) by choosing between including item i ($P[i-1, m-w_i] + p_i$) or not including it ($P[i-1, m]$) and sticking with previous answer.

Dynamic Programming Example

- In the first **for..do** loop, the first row of the table $P[i, m]$ is filled out.
- Item 1, with profit $p_1 = 1$ and weight $w_1 = 5$, is selected when the weight level, m, is greater than or equal to 5.

Input

item i	1	2	3	4
profit p_i	1	4	3	5
weight w_i	5	4	6	3

capacity M	9

Table P[i,m]

m =	1	2	3	4	5	6	7	8	9
i=1	0	0	0	0	1	1	1	1	1
i=2									
i=3									
i=4									

Dynamic Programming Example

- In the second, nested pair of **for..do** loops, the remaining rows of the table $P[i, m]$ are filled out.
- When $i = 2$ the second row is being filled out. The decision being made is: When does selecting Item 2 (either to replace Item 1 or in addition to Item 1) increase the profit?
- Item 2, with profit $p_2 = 4$ and weight $w_2 = 4$, replaces Item 1 when $m = 4$.
- Item 1 is chosen in addition to Item 2 when $m = 9$.

Input

item i	1	2	3	4
profit p_i	1	4	3	5
weight w_i	5	4	6	3

capacity M	9

Table P[i,m]

m =	1	2	3	4	5	6	7	8	9
i=1	0	0	0	0	1	1	1	1	1
i=2	0	0	0	4	4	4	4	4	5
i=3									
i=4									

Dynamic Programming Example

- In row $i = 3$ where Item 3 is under consideration, since Item 3 weighs more and has less profit than Item 2 it never replaces Item 2. The combination of Item 2 and 3 exceeds the weight capacity of the knapsack, so item 3 is never chosen at all and Row 3 remains the same as Row 2.
- In row $i = 4$ where Item 4 is under consideration. Since it weighs less and has more profit, its replaces Item 2 starting when $m = 3$.
- When $m \geq 7$ the combination of Item 2 and Item 4 fills out the rest of the table.

Input

item i	1	2	3	4
profit p_i	1	4	3	5
weight w_i	5	4	6	3

capacity M	9

Table P[i,m]

m =	1	2	3	4	5	6	7	8	9
i=1	0	0	0	0	1	1	1	1	1
i=2	0	0	0	4	4	4	4	4	5
i=3	0	0	0	4	4	4	4	4	5
i=4	0	0	5	5	5	5	9	9	9

Dynamic Programming Example

- The items that comprise the solution are not stored in the table in part because they can be read from it.
- Since $P[4,9] \neq P[3,9]$ (green oval) you know that Item 4 was selected here. Its weight is 3 so look at $P[3,9-3]$ which is $P[3,6]$.
- Since $P[3,6] = P[2,6]$ (green oval) you know that Item 3 was not selected, so look at $P[2,6]$.
- Since $P[2,6] \neq P[1,6]$ you know that Item 2 was selected here. Its weight is 4 so look at $P[1,6-4]$ which is $P[1,2]$.

Input

item i	1	2	3	4
profit p_i	1	4	3	5
weight w_i	5	4	6	3

capacity M	9

Table P[i,m]

m =	1	2	3	4	5	6	7	8	9
i=1	0	0	0	0	1	1	1	1	1
i=2	0	0	0	4	4	4	4	4	5
i=3	0	0	0	4	4	4	4	4	5
i=4	0	0	5	5	5	5	9	9	9

0-1 Knapsack: Retrieving the Selected Items X

Working back from the last entry in the table, set $x[i] = 1$ if the i^{th} item was selected, $x[i] = 0$ otherwise.

Algorithm: *0-1Knapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M, P)$

$m \leftarrow M$

$p \leftarrow P[n, M]$

for $i \leftarrow n$ **downto** 2

do
- **if** $p = P[i-1, m]$
  - **then** $x_i \leftarrow 0$
  - **else**
    - $x_i \leftarrow 1$
    - $p \leftarrow p - p_i$
    - $m \leftarrow m - w_i$

if $p = 0$

then $x_1 \leftarrow 0$

else $x_1 \leftarrow 1$

return (X);

0-1 Knapsack Optimal Structure and Subproblems

Optimal Structure

If I know the optimal answers for filling the knapsack with items $\{1, 2, \ldots i-1\}$, I can use those answers to make a decision about whether or not to include item i.

Define Subproblems

The subproblems $\mathcal{S}(I)$ are the table entries $P[i, m]$ where $i \leq n$ and $m \leq M$.

Recurrence Relation

The recurrence relation is:

$$P[i, m] = \max\{p_i + P[i-1, m-w_i],\ \ P[i-1, m]\}$$

Compute Optimal Solutions

Fill in the table $P[i, m]$ where $i \leq n$ and $m \leq M$.

0-1 Knapsack - Summary

The key to understanding this algorithm is to knowing *exactly* what $P[i, m], P[i-1, m]$ and $P[i-1, m-w_i]$ represent.

$P[i, m]$: represents the maximum profit possible by selecting from items $\{1, 2, \ldots i\}$ to fill the knapsack up to level m.

$P[i-1, m]$: represents the maximum profit possible by selecting from items $\{1, 2, \ldots i-1\}$ to fill the knapsack to level m.

The question the algorithm asks at each iteration: Is the profit higher if I include item i or if I skip it?

If I include item i: it takes up weight w_i in the knapsack, then I fill the rest, $m - w_i$, by selecting from items $1, 2, \ldots i-1$.

The total profit equals the profit from including item i plus the profit from filling up the rest of the knapsack using items from $\{1, 2, \ldots i-1\}$, i.e. $p_i + P[i-1, m-w_i]$.

0-1 Knapsack - Summary

If I do not include item i: then I fill the knapsack to level m by selecting from items $\{1, 2, \ldots i-1\}$. Here, total profit is $P[i-1, m]$.

To *maximize* profits: I take the *maximum* of these two options:

$$P[i, m] = \max\{p_i + P[i-1, m-w_i],\ \ P[i-1, m]\}$$

When making a decision about $P[i, m]$: I'm comparing the entry above and to the left by w_i columns, $P[i-1, m-w_i]$, to the entry just above it, $P[i-1, m]$.

Tracking which items I picked:

I picked item i when $P[i, m] = P[i-1, m-w_i] + p_i$.

I did not pick item i when $P[i, m] = P[i-1, m]$.

Longest Common Subsequence - Definitions

Given a sequence of characters: bacboambpuabtbberb

- "computer" is a subsequence of bacboambpuabtabera
- "computer" is not a substring of bacboambpuabtbberb

A sequence of length m has 2^m possible subsequences.

Given two sequences

- Sequence 1: bacboambpuabtabera
- Sequence 2: xcomyxyxpyxuyxtyerx

"comp" is a common subsequence of 1 and 2

"computer" is the longest common subsequence (LCS) of 1 and 2

Longest Common Subsequence

Problem

Longest Common Subsequence

Instance: *Two sequences* $X = (x_1, \ldots, x_m)$ *and* $Y = (y_1, \ldots, y_n)$ *over some finite alphabet* Γ.

Find: *A maximum length sequence* Z *that is a subsequence of both* X *and* Y.

$Z = (z_1, \ldots, z_k)$ is a **subsequence** of X if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, Z is a subsequence of Y if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$.

Longest Common Subsequence

Unpacking the Definition

$Z = (z_1, \ldots, z_k)$ is a **subsequence** of X if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

For example...

```
X = x c o m y x y x p y x
```

```
Z = c o m p
```

Z is a subsequence of X because there exists indices $I = \{2, 3, 4, 9\}$ such that $z_j = x_{i_j}$ for $1 \leq j \leq 4$

When $j = 1$, then z_j is the first element of Z, namely c.

And, i_j is the first element of I, namely 2.

Finally, element x_{i_j} is x_2 which is the 2nd element of X which is also c.

Hence $z_j = x_{i_j}$ for $j = 1$.

Longest Common Subsequence - Substructure

If "compute" is the longest common subsequence of 1 and 2

- Sequence 1: bacboambpuabtabe
- Sequence 2: xcomyxyxpyxuyxtye

then "computer" is the longest common subsequence (LCS) of 1 and 2

- Sequence 1: bacboambpuabtabera
- Sequence 2: xcomyxyxpyxuyxtyerx

The LCS has optimal structure.

We can use the LCS of the prefixes (the beginning of the sequence) to help find the LCS of the entire sequences.

Longest Common Subsequence

Using the example in the course text, section 15.4, find the LCS of $X =$ ABCBDAB and $Y =$ BDCABA and.

- Find all the occurrences of $x_1 =$ A in Y
- Then all the occurrences of $x_2 =$ B (and A..B) in Y
- Then all the occurrences of $x_3 =$ C (and B..C, A..C, A..B..C) in Y
- etc.

Fill in a table of counts $c[i, j]$ tracking the maximum matches you have found between the first i characters of X and the first j characters of Y:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i, j-1],\ \ c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Here x_i means the i^{th} character in X.

Unpacking the Recursion

$c[i, j] = 0$ if $i = 0$ or $j = 0$

E.g. initialize the number of matches to 0.

$c[i, j] = c[i - 1, j - 1] + 1$ if $i, j > 0$ and $x_i = y_j$

E.g. if there is a match between the i^{th} character of X and the j^{th} character of Y then increment the number of matches by one.

$c[i, j] = \max\{c[i, j - 1], c[i - 1, j]\}$ if $i, j > 0$ and $x_i \neq y_j$

E.g. if the characters do not match, decide if you are better off continuing with your previous best answer $c[i - 1, j]$ (which started at an earlier character) or continuing on with your current starting place $c[i, j - 1]$.

Longest Common Subsequence - Example

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

- First row and column of the table, $c[i,j]$ is 0.
- Second row, find all the occurrences of $x_1 =$ A inBDCABA

	Y->	B	D	C	A	B	A
X	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0						
C	0						

Longest Common Subsequence - Example

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

- Third row, find all the occurrences of $x_2 =$ B in BDCABA
- You are also finding all occurrences of A..B (i.e. $x_1..x_2$) in Y

	Y->	B	D	C	A	B	A
X	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0						

Computing the Length of the LCS of X and Y

Algorithm: *LCS1*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m
  do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
  do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  do $\begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \quad \textbf{do } \begin{cases} \textbf{if } x_i = y_j \quad \textbf{then } c[i, j] \leftarrow c[i-1, j-1] + 1 \\ \quad \textbf{else } c[i, j] \leftarrow \max\{c[i, j-1], c[i-1, j]\} \end{cases} \end{cases}$
return $(c[m, n])$;

Next, we need a way to track if the choice came from the left (L or $\leftarrow$), up above (U or $\uparrow$), or upper left diagonal (D or $\nwarrow$). Use a table $\pi[i, j]$ to track this information.

Finding the LCS of X and Y

Algorithm: *LCS2*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m **do** $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ **to** n **do** $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ **to** m

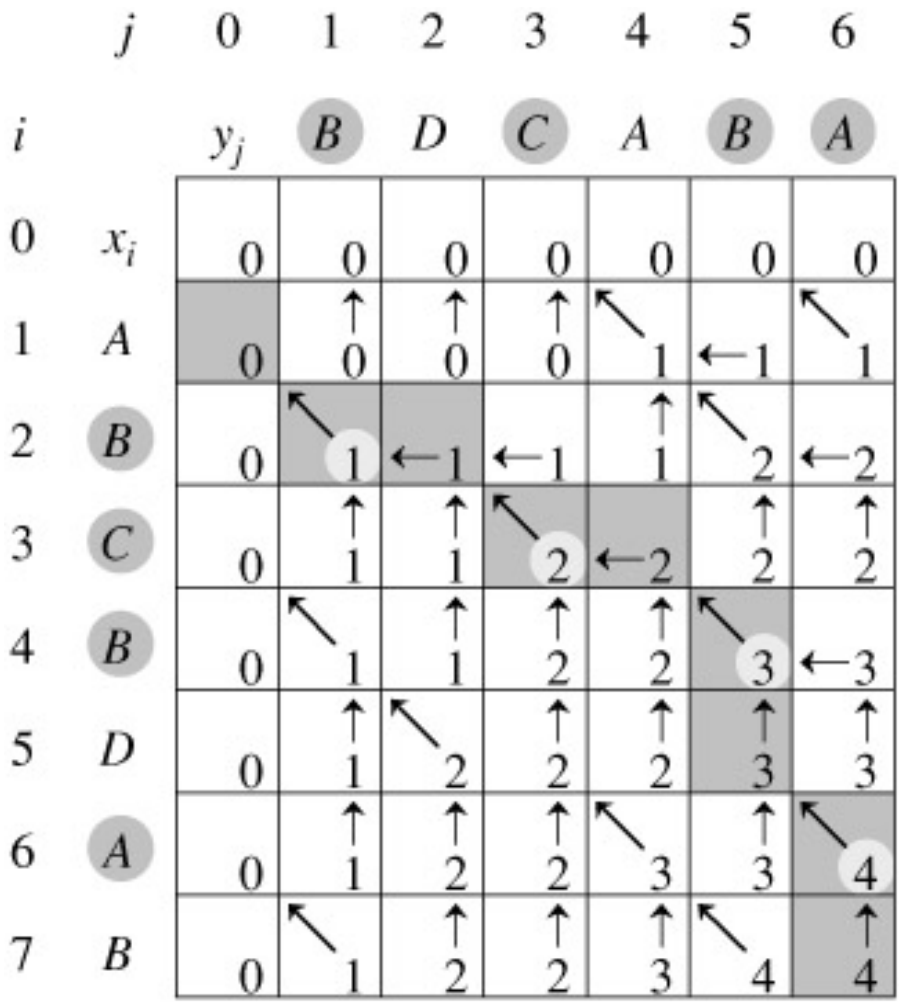
do
- **for** $j \leftarrow 1$ **to** n
  - **do**
    - **if** $x_i = y_j$
      - **then** $c[i, j] \leftarrow c[i-1, j-1] + 1$; $\pi[i, j] \leftarrow \mathrm{D}$
    - **else if** $c[i, j-1] > c[i-1, j]$
      - **then** $c[i, j] \leftarrow c[i, j-1]$; $\pi[i, j] \leftarrow \mathrm{L}$
    - **else** $c[i, j] \leftarrow c[i-1, j]$; $\pi[i, j] \leftarrow \mathrm{U}$

return (c, π);

This expands the previous slide by adding a table π which tracks if the choice came from the left (L), up above (U), or diagonal (D).

Identifying the Longest Common Subsequence

Filling up the whole table we have the following, Fig 15.8, from the course text.



Finding the LCS

Algorithm: *FindLCS*(c, π, X)

$seq \leftarrow ()$
$i \leftarrow m$
$j \leftarrow n$
while $\min\{i, j\} > 0$
do
- **if** $\pi[i, j] = \mathrm{D}$
  - **then**
    - $seq \leftarrow x_i \parallel seq$
    - $i \leftarrow i - 1$
    - $j \leftarrow j - 1$
- **else if** $\pi[i, j] = \mathrm{L}$ **then** $j \leftarrow j - 1$
- **else** $i \leftarrow i - 1$

return (seq)

Start at the bottom right corner, working up and to the left. Each time you encounter a D it means the corresponding character is part of the LCS.

All-Pairs Shortest Paths

Problem

All-Pairs Shortest Paths

Instance: *A directed graph $G = (V, E)$, and a* **weight matrix** *W, where $W[i, j]$ denotes the weight of edge ij, for all $i, j \in V$, $i \neq j$.*

Find: *For all pairs of vertices $u, v \in V$, $u \neq v$, a directed path P from u to v such that*

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in G.

All-Pairs Shortest Paths - Observations

- Compare with Dijkstra's Algorithm (Topic 4d) which works with the shorts paths from a *single* vertex to all other vertices.
- Here it is a collection of the shortest paths from any vertex u in a graph to any other vertex v.
- All pairs shortest paths exhibits optimal structure: if v is on the shortest path from u to w then the part of the path from u to v must be the shortest possible from u to v and the path from v to w must also be the shortest possible.

All-Pairs Shortest Paths - Overview

- W - a table of the edges weights between any pair of vertices. Assume an entry could be ∞ if there is no edge between the two vertices.
- m - the number of edges in the path, i.e. we start by considering all paths with two edges in them, then three edges in them, and so on.
- $L_m[i, j]$ - the length (sum of the weights) of all paths with m or less edges in them from vertex i to j.
- i, j - the start and the end of the path
- k - a possible intermediate vertex on the path from i to j

As the number of edges, m, in a path goes from 2 to $n - 1$, the algorithm asks if for paths from i to j is it cheaper to take the path from i to k and then the edge from (k, j)?

First Solution

Algorithm: *SlowAllPairsShortestPath*(W)

$L_1 \leftarrow W$

for $m \leftarrow 2$ **to** $n-1$ comment: path lengths

do **for** $i \leftarrow 1$ **to** n comment: starting points

do **for** $j \leftarrow 1$ **to** n comment: end points

do $\ell \leftarrow \infty$

for $k \leftarrow 1$ **to** n comment: mid points

do $\ell \leftarrow \min\{\ell, L_{m-1}[i,k] + W[k,j]\}$

$L_m[i,j] \leftarrow \ell$

return (L_{n-1})

Four nested $O(n)$ loops for a total of $O(n^4)$. Can it be sped up?

Second Solution

Algorithm: *FasterAllPairsShortestPath*(W)

$L_1 \leftarrow W$
$m \leftarrow 2$
while $m < n - 1$ comment: path length
do
  for $i \leftarrow 1$ **to** n comment: starting point
    do
      for $j \leftarrow 1$ **to** n comment: end point
        do
          $\ell \leftarrow \infty$
          for $k \leftarrow 1$ **to** n comment: midpoint
            do $\ell \leftarrow \min\{\ell, L_{m/2}[i,k] + L_{m/2}[k,j]\}$
          $L_m[i,j] \leftarrow \ell$
  $m \leftarrow 2m$
return (L_m)

Double the number of edges considered in each iteration and let vertex k be between i and j (rather than adjacent to j). It is now $O(n^3 \lg n)$.

Third Solution

Algorithm: *FloydWarshall*(W)

$D_0 \leftarrow W$

for $m \leftarrow 1$ **to** n comment: path length

do { **for** $i \leftarrow 1$ **to** n comment: starting point

do { **for** $j \leftarrow 1$ **to** n **do** comment: end point

$D_m[i,j] \leftarrow \min\{D_{m-1}[i,j], D_{m-1}[i,m] + D_{m-1}[m,j]\}$

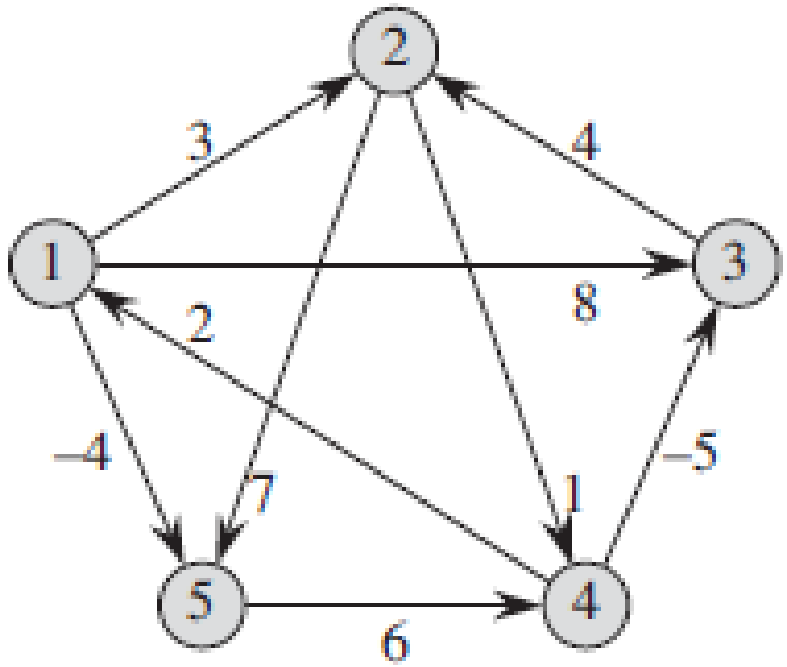
return (D_n)

Combine the idea of considering increasing path lengths, m, with the idea of going through an intermediate vertex k.

At each iteration $m = 1$ to n, check to see if the path from i to j passing through m is cheaper than our previous path which only passed through vertices 1 to $m-1$ as intermediate nodes. The algorithm is now in $O(n^3)$.

Note, the course text has a worked example in Figure 25.4 on page 696.

Floyd-Warshall Example



Input to Floyd-Warshall algorithm.

figure from course text, Figure 25.1

Floyd-Warshall Example: D_0

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

The input is a table W that stores the weights of each arc from the graph presented in the previous slide.

E.g. in the graph the weight of the edge v_1v_2 is 3 and the entry d_{12} is 3.

For an edge v_iv_i the entry is 0 and if there is no edge v_iv_j then the entry d_{ij} is ∞.

The first line of the algorithm is to set $D_0 \leftarrow W$, that is D_0 is the cost to go *directly* from one vertex to another.

figure from course text, Figure 25.4

Floyd-Warshall Example: D_1

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

When m=1 we are asking: Is it less expensive to *go directly* from v_i to v_j or to *go through the intermediate* vertex v_1.

That is: $D_1[i,j] = \min\{D_0[i,j],\ \ D_0[i,1] + D_0[1,j]\}$

It is cheaper to go from v_4 to v_2 through v_1 (2+3=5) rather than to try to go directly (∞).

It is cheaper to go from v_4 to v_5 through v_1 (2-4= -2) rather than to try to go directly (∞).

figure from course text, Figure 25.4

Floyd-Warshall Example: D_2

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

When m=2 we are asking: Is it less expensive to *go directly* from v_i to v_j or to *go through the intermediate* vertices $\{v_1, v_2\}$.

That is: $D_2[i,j] = \min\{D_1[i,j],\ \ D_1[i,2] + D_1[2,j]\}$

It is cheaper to go from v_1 to v_4 through v_2 (3+1=4) rather than to try to go directly (∞).

It is cheaper to go from v_3 to v_4 through v_2 (4+1=5) rather than to try to go directly (∞).

It is cheaper to go from v_3 to v_5 through v_1 (4+7=11) rather than to try to go directly (∞).

Floyd-Warshall: The Recurrence

Going through vertex v_2 is included explicitly in the recurrence i.e.
$D_2[i,j] = \min\{D_1[i,j],\ \ D_1[i,2] + D_1[2,j]\}$

Key Point: Going through vertex v_1 is included implicitly in the recurrence because table D_1 allows for the fact that you may (or may not) use v_1 as an intermediate vertex.

So the term $D_1[i,j]$ includes consideration of the paths $v_i \rightarrow v_j$ and $v_i \rightarrow v_1 \rightarrow v_j$.

And the term $D_1[i,2]$ includes consideration of the paths $v_i \rightarrow v_2$ and $v_i \rightarrow v_1 \rightarrow v_2$.

And the term $D_1[2,j]$ includes consideration of the paths $v_2 \rightarrow v_j$ and $v_2 \rightarrow v_1 \rightarrow v_j$.

So essentially you are taking the minimum of all combinations of paths that start at v_i and end at v_j that may or may not include any of $\{v_1, v_2\}$ in any order.

Floyd-Warshall: The Predecessor Table

We can recreate the path from v_i to v_j by keeping track of which choice we made in the $\min$ function.

$$D_2[i,j] = \min\{D_1[i,j],\ \ D_1[i,2] + D_1[2,j]\}$$

The table Π^k_{ij} will keep track of the predecessor to v_j in the path from v_i to v_j when m had the value k.

If we took the first choice, i.e. $D_2[i,j] = D_1[i,j]$, then the predecessor of v_j in D_2 is the same as the predecessor of v_j in the path v_i to v_j in D_1 and so $\Pi^2_{ij} = \Pi^1_{ij}$

If we took the second choice, i.e. $D_2[i,j] = D_1[i,2] + D_1[2,j]$, then the predecessor of v_j in D_2 is the same as the predecessor of v_j in the path v_2 to v_j in D_1 and so $\Pi^2_{ij} = \Pi^1_{2j}$.

Floyd-Warshall: The Predecessor Table Π^0

$$\Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

The base case for the recurrence is as follows:

- For $\pi_{ii} = \text{NIL}$.
- If the graph contains the edge $v_i v_j$ then $\pi_{ij}^0 = i$.
- If the graph does not contain edge $v_i v_j$ then $\pi_{ij}^0 = \text{NIL}$.

figure from course text, Figure 25.4

Floyd-Warshall: The Predecessor Table Π^1

$$\Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

In the case $D_1[4,2] = \min\{D_0[4,2], \;\; D_0[4,1] + D_0[1,2]\}$

- The cost of going directly from v_4 to v_2 is ∞, i.e. there is no edge.
- The cost of going $v_4 \to v_1 \to v_2$ is $2 + 3 = 5$.
- Since we took the second option, the predecessor of v_2 on the path from v_4 to v_2 is v_1 so $\Pi^1_{42} = 1$ (whereas Π^0_{42} equaled NIL).

To read off the path from $v4$ to $v2$ start at the end an work backwards.

- $\Pi^1_{42} = 1$ so the predecessor of v_2 is v_1.
- $\Pi^1_{41} = 4$ so the predecessor of v_1 is v_4.

figure from course text, Figure 25.4

Topic 6 - Intractability

This section investigates the concept of intractable problems.

Readings for this topic from *Introduction to Algorithms, 3rd ed.*

- Chapter 34, NP-Completeness

intractable (*adjective*): not easily governed, managed, or directed.

In common usage, an **intractable problem** is a problem that cannot be easily managed or solved.

In computer science, an **intractable problem** is a problem that cannot be solved with a polynomial time algorithm.

Types of Problems - Informal Version

Currently you are familiar with two classes of problems.

1. Those for which polynomial time solutions exist. Eg. practically every algorithm you've seen so far. We will characterize these more formally with the **Complexity Class P**.
2. **Undecidable**: Those for which no single algorithm can give a correct answer for all possible inputs.. E.g. Alan Turing's Halting Problem (1936) and Alonzo Church's Equivalence of λ-Calculus expressions (1936).

But there are other classes of problems.

3. **Complexity Class NP**: Those whose yes solution can be verified in polynomial time, e.g. Does x have any non-trivial factors?
4. **Complexity Class NPC**: Those problems in NP that if they could be solved in polynomial time, then all problems in NP could be solved in polynomial time.

Decision Problems

Problem Instance: Input for the specified problem. E.g. adjacency lists for a graph G.

Decision Problem: Given a problem instance I, answer a certain question "yes" or "no." E.g. does the graph G contain a cycle?

Problem Solution: Correct answer ("yes" or "no") for the specified problem instance. I is a **yes-instance** if the correct answer for the instance I is "yes". I is a **no-instance** if the correct answer for the instance I is "no".

Size of a problem instance: $\mathit{Size}(I)$ is the number of bits required to specify (or encode) the instance I. E.g. how big are our adjacency lists?

Constant difference between input encoded as binary vs. decimal.
Exponential difference between input encoded as binary vs. unary.

The Complexity Class P

Algorithm Solving a Decision Problem: An algorithm A is said to **solve** a decision problem Π provided that A finds the correct answer ("yes" or "no") for every instance I of Π in finite time.

Polynomial-time Algorithm: An algorithm A for a decision problem Π is said to be a **polynomial-time algorithm** provided that the complexity of A is $O(n^k)$, where k is a positive integer and $n = \text{Size}(I)$.

The Complexity Class P denotes the set of all decision problems that have polynomial-time algorithms solving them. We write $\Pi \in P$ if the decision problem Π is in the complexity class P.

Examples of Decision Problems: Cycles in Graphs

Problem

Cycle

Instance: *An undirected graph $G = (V, E)$.*

Question: *Does G contain a cycle?*

Problem

Hamiltonian Cycle

Instance: *An undirected graph $G = (V, E)$.*

Question: *Does G contain a Hamiltonian cycle?*

A **Hamiltonian cycle** is a cycle that passes through every vertex in V exactly once.

Finding a cycle is easy. Finding a Hamiltonian cycle is hard.

Examples of Decision Problems: Knapsack Problems

Problem

0-1 Knapsack-Dec

Instance: *a list of* **profits**, $P = [p_1, \ldots, p_n]$; *a list of* **weights**, $W = [w_1, \ldots, w_n]$; *a* **capacity**, M; *and a* **target profit**, T.

Question: *Is there an* n*-tuple* $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ *such that* $\sum w_i x_i \leq M$ *and* $\sum p_i x_i \geq T$*?*

Problem

Rational Knapsack-Dec

Instance: *a list of* **profits**, $P = [p_1, \ldots, p_n]$; *a list of* **weights**, $W = [w_1, \ldots, w_n]$; *a* **capacity**, M; *and a* **target profit**, T.

Question: *Is there an* n*-tuple* $[x_1, x_2, \ldots, x_n] \in [0,1]^n$ *such that* $\sum w_i x_i \leq M$ *and* $\sum p_i x_i \geq T$*?*

Solving the Rational Knapsack problem is easy. Solving the 0-1 Knapsack problem is hard.

Certificates

Certificate: Informally, a certificate for a yes-instance I is some "extra information" C which makes it easy to **verify** that I is a yes-instance.

E.g. if you are trying to find one of the factors of an integer, n, then one of the factors of n would be a certificate.

Certificate Verification Algorithm: Suppose that *Ver* is an algorithm that verifies certificates for yes-instances. Then *Ver*(I, C) outputs "yes" if I is a yes-instance and C is a valid certificate for I. If *Ver*(I, C) outputs "no", then either I is a no-instance, or I is a yes-instance and C is an invalid certificate.

E.g. checking if f_1 divides into n without remainder would be verifying that f_1 is a factor of n. *Ver*$(35, 6) =$ "no" and *Ver*$(35, 5) =$ "yes."

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm *Ver* is a polynomial-time certificate verification algorithm if the complexity of *Ver* is $O(n^k)$, where k is a positive integer and $n =$ *Size*(I).

The Complexity Class NP

Certificate Verification Algorithm for a Decision Problem: A certificate verification algorithm *Ver* is said to **solve** a decision problem Π provided that

- **for every** yes-instance I, **there exists** a certificate C such that *Ver*(I, C) outputs "yes".
- **for every** no-instance I and **for every** certificate C, *Ver*(I, C) outputs "no".

The Complexity Class NP denotes the set of all decision problems that have polynomial-time certificate verification algorithms solving them. We write $\Pi \in NP$ if the decision problem Π is in the complexity class *NP*.

Informally, the complexity class *NP* is the set of all decision problems where we can verify that an answer is correct in polynomial time in the size of the problem input.

Certificate Verification Algorithm for Hamiltonian Cycle

A certificate consists of an n-tuple, $X = [x_1, \ldots, x_n]$, that might be a hamiltonian cycle for a given graph $G = (V, E)$ (where $n = |V|$).

Algorithm: *Hamiltonian Cycle Certificate Verification*(G, X)

$flag \leftarrow$ **true**
$Used \leftarrow \{x_1\}$
$j \leftarrow 2$
while $(j \leq n)$ **and** $flag$
do
- $flag \leftarrow (x_j \notin Used)$ **and** $(\{x_{j-1}, x_j\} \in E)$
- **if** $(j = n)$ **then** $flag \leftarrow flag$ **and** $(\{x_n, x_1\} \in E)$
- $Used \leftarrow Used \cup \{x_j\}$

return $(flag)$

Use $flag$ to verify (1) that each pair of vertices along the cycle, (x_{j-1}, x_j), is a valid edge *and* (2) that each vertex, x_j, has only been visited once.

The Complexity Class ExpTime

An algorithm is an **exponential-time** algorithm if its running time is $O(2^{p(n)})$, where $p(n)$ is a polynomial in n and $n = \mathit{Size}(I)$.

Is there anything "bigger," i.e. that grows faster, than $O(2^{p(n)})$? Ans: n^n

The Complexity Class ExpTime denotes the set of all decision problems that have exponential-time algorithms solving them. We write $\Pi \in \mathit{ExpTime}$ if the decision problem Π is in the complexity class *ExpTime*.

Theorem

$P \subseteq NP \subseteq \mathit{ExpTime}$.

- $P \subseteq NP$: ignore the certificate and compute the answer
- $NP \subseteq \mathit{ExpTime}$: generate every possible certificate and check to see if any return a *yes* answer.

Polynomial-time Reductions

For a decision problem Π, let $\mathcal{I}(\Pi)$ denote the set of all instances of Π. Let $\mathcal{I}_{\text{yes}}(\Pi)$ and $\mathcal{I}_{\text{no}}(\Pi)$ denote the set of all yes-instances and no-instances (respectively) of Π.

Suppose that Π_1 and Π_2 are decision problems. We say that there is a **polynomial-time reduction** from Π_1 to Π_2 (denoted $\Pi_1 \leq_P \Pi_2$) if there exists a function $f : \mathcal{I}(\Pi_1) \to \mathcal{I}(\Pi_2)$ such that the following properties are satisfied:

- $f(I)$ is computable in polynomial time (as a function of *size*(I), where $I \in \mathcal{I}(\Pi_1)$)
- if $I \in \mathcal{I}_{\text{yes}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{yes}}(\Pi_2)$
- if $I \in \mathcal{I}_{\text{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{no}}(\Pi_2)$

Informally, this means that after you reformat the input a bit (i.e. call $f(I)$) you can use the algorithm/function that solves Π_2 to solve Π_1. item The reverse is not necessarily true. An algorithm that solves Π_1 may not be able to solve Π_2.

Two Graph Theory Problems

Problem

Clique-Dec

Instance: *An undirected graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.*

Question: *Does G contain a clique of size $\geq k$? (A **clique** is a subset of vertices $W \subseteq V$ such that $uv \in E$ for all $u, v \in W$, $u \neq v$.)*

Problem

Vertex Cover-Dec

Instance: *An undirected graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.*

Question: *Does G contain a vertex cover of size $\leq k$? (A **vertex cover** is a subset of vertices $W \subseteq V$ such that $\{u, v\} \cap W \neq \emptyset$ for all edges $uv \in E$.)*

Clique-Dec $\leq_P$ Vertex-Cover-Dec

Informally, A clique is a complete subgraph (every vertex connected to every other vertex) within a graph. A vertex cover is a subset of vertices, W, such that each edge in the graph has exactly one endpoint in W.

Suppose that $I = (G, k)$ is an instance of **Clique-Dec**, where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ and $1 \leq k \leq n$.

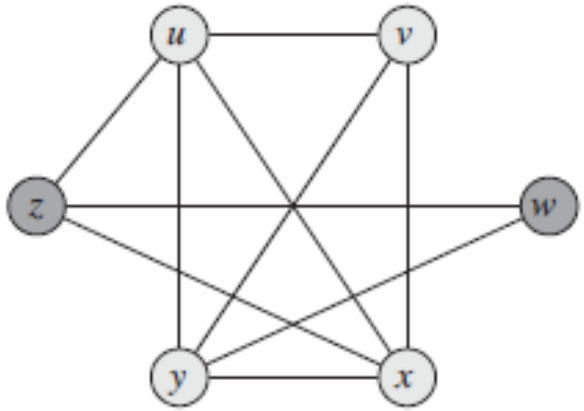
Construct an instance $f(I) = (G^c, \ell)$ of **Vertex Cover-Dec**, where $G^c = (V, E^c)$, $\ell = n - k$ and

$$v_i v_j \in E^c \Leftrightarrow v_i v_j \notin E.$$

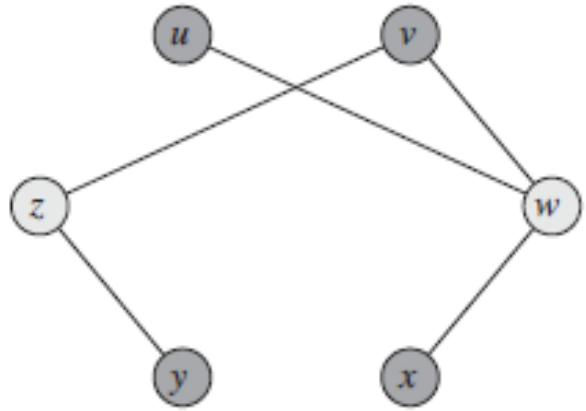
G^c is called the **complement** of G, because every edge of G is a non-edge of G^c and every non-edge of G is an edge of G^c.

Informally, $f(I)$ is changing the format of the input, I, (by creating the complement of G) and then calling a pre-existing function (Vertex-Cover-Dec) to solve the problem.

Clique vs. Vertex Cover



(a)



(b)

- In (a), $W = \{u, v, x, y\}$ is a clique of size four, i.e. every vertex in W **is connected** to every other vertex in W.
- In (b), $W^c = \{w, z\}$ is a vertex cover of size two, i.e. every edge **has at least one endpoint** in W^c.
- Note that (b) is the complement of (a): if $v_i v_j$ **is** an edge in one graph then it **is not** an edge in the other.

figure from course text, Figure 34.15

Clique-Dec $\leq_P$ Vertex-Cover-Dec

Assume $G(V, E)$ has a clique of size k, then $G^c(V, E^c)$ has a vertex cover of size $n - k$ where n is the number of vertices.

- If W is a clique of size k in G, it suffices to show that $W^c = V \setminus W$ is a vertex cover in G^c.
- Let uv be an edge in E^c.
- Hence, uv is not an edge in E.
- Hence, a least one of $\{u, v\}$ is not in W, since W is a clique.
- Hence, at least one of $\{u, v\}$ is in W^c.
- Hence W^c is a vertex cover for G^c.

Clique-Dec $\leq_P$ Vertex-Cover-Dec

Conversely, assume $G^c(V, E^c)$ has a vertex cover of size $n - k$ then $G(V, E)$ has a clique of size k.

- If W^c is a vertex cover of size $n - k$ in G^c and let $W = V \setminus W^c$.
- Let $\{u, v\}$ be two vertices such that $u \notin W^c$ and $v \notin W^c$. There are k such vertices.
- Then uv is not an edge in G^c since W^c is a vertex cover of G^c.
- Then uv is an edge in G.
- There are k vertices in W.
- Hence, G has a clique of size k.

Also note that the f is a polynomial time function. It is just taking the complement of the adjacency lists/matrix.

Hence Clique-Dec $\leq_P$ Vertex-Cover-Dec.

Properties of Polynomial-time Reductions

Suppose that $\Pi_1, \Pi_2, \ldots$ are decision problems.

Theorem

If $\Pi_1 \leq_P \Pi_2$ and $\Pi_2 \in P$, then $\Pi_1 \in P$.

Approach: Map an instance of Π_1 onto Π_2 and then use the subroutine that solves Π_2.

Theorem

$\Pi_1 \leq_P \Pi_2$ and $\Pi_2 \leq_P \Pi_3$, then $\Pi_1 \leq_P \Pi_3$.

Approach: Suppose f_{12} maps an instance of Π_1 onto Π_2 and bf_{23} maps an instance of Π_2 onto Π_3 then consider $f_{23}(f_{12}(I))$.

The Complexity Class NPC

The complexity class *NPC* denotes the set of all decision problems Π that satisfy the following two properties:

- $\Pi \in NP$
- For all $\Pi' \in NP$, $\Pi' \leq_P \Pi$.

NPC is an abbreviation for **NP-complete**.

Theorem

If $P \cap NPC \neq \emptyset$, *then* $P = NP$.

Approach: if Π *is a problem in the intersection, reduce other problems in NP to* Π *and solve* Π *in poly time.*

Theorem

If $P = NP$, *then* $P = NP = NPC$.

Approach: Use the fact that $NPC \subseteq NP$.

Key Concepts

Understand these key concepts and you should be able to handle the types of intractability problems we deal with in CS 341. Note: These definitions are all *informal*.

- *P* - the set of problems that can be *solved* in polynomial time.
- *NP* - the set of problems whose solutions can be *verifed* in polynomial time.
- *Ver*(I, Cert) - the function that *verfies* if Cert is a valid solution to the problem instance I. It is used to show that problems are in *NP*.
- *NPC* - the subset of problems within *NP* that seems to be the hardest to solve.
- $\Pi_k \leq_P \Pi_u$ - problem Π_k polynomially reduces to problem Π_u, i.e. an algorithm that solves Π_u can be used as a subroutine to solve Π_k after Π_k's input has been suitably reformatted by $f()$. This relationship is expressed as "I is a yes-instance of $\Pi_k \iff f(I)$ is a yes-instance of Π_u." It is used to show that Π_u is in *NPC*.

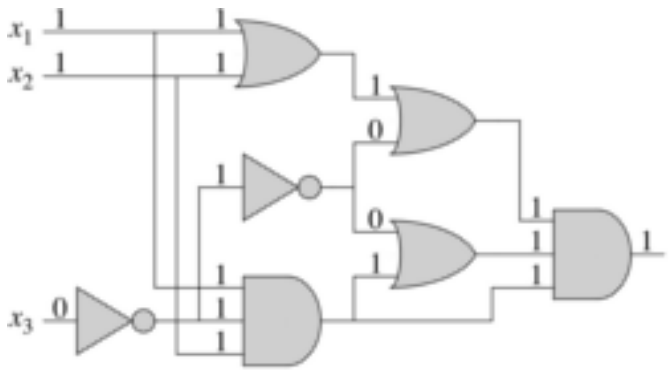
Satisfiability Problems

Problem

Satisfiability *(a.k.a.* **Circuit-Sat***)*

Instance: *A Boolean circuit* $\mathcal{C}$ *having* n *Boolean inputs,* $x_1, \ldots, x_n$*, and one Boolean output.* $\mathcal{C}$ *contains* and*,* or *and* not *gates.*

Question: *Is there a* **truth assignment** $t : \{x_1, \ldots, x_n\} \in \{0,1\}^n$ *such that* $\mathcal{C}$ *outputs a 1?*



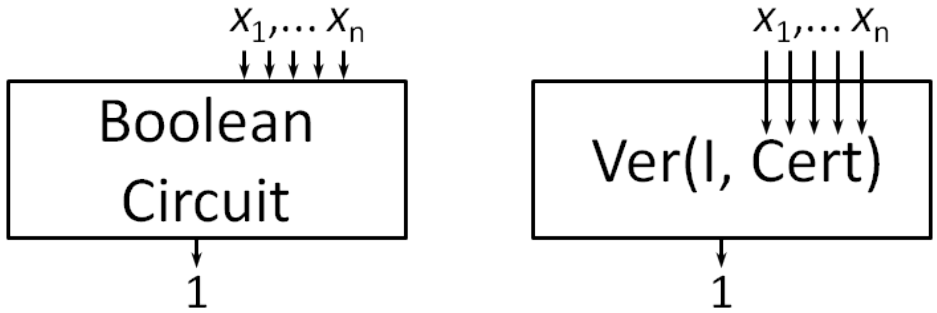
In the example above (from the course text Fig 34.8a) an assignment of $\{x_1, x_2, x_3\} = \{1, 1, 0\}$ results in the output 1 (or true). This circuit is satisfiable with the input $\{1, 1, 0\}$.

Satisfiability Problems

Problem

Satisfiability *(a.k.a.* **Circuit-Sat***)*

Instance: *A Boolean circuit $\mathcal{C}$ having n Boolean inputs, $x_1, \ldots, x_n$, and one Boolean output. $\mathcal{C}$ contains* and, or *and* not *gates.*



- In the example on the left, the **Circuit-Sat** asks "For a given Boolean circuit, find an input, $x_1, \ldots x_n$ that results in the output 1."
- In the example on the right, if the Boolean circuit implements a verification algorithm for an *NP* problem, then **Circuit-Sat** asks "For a given instance I, find a valid certificate, $x_1, \ldots x_n$."

Cook's Theorem - Example: Hamiltonian Cycle

Theorem

Satisfiability $\in$ *NPC*.

Example: reduce $\Pi =$ **Hamiltonian Cycle** in *NP* to **Satisfiability**:

- Since **Hamiltonian Cycle** is in *NP*, there is a polynomial-time certificate verification algorithm, *Ver*(I, Cert), where I is an instance of a graph, $G(V, E)$ (say as adjacency lists), and Cert is a certificate (say the Hamiltonian cycle $\{h_1, h_2, \ldots, h_n\}$).
- Convert the algorithm *Ver* to a Boolean circuit $\mathcal{C}$ that performs the same computation, i.e. checks to see if the input $\{h_1, h_2, \ldots, h_n\}$ is indeed a Hamiltonian cycle in $G(V, E)$.
- Fix (i.e., hardwire) the part if *Ver*(I, Cert) that takes the graph $G(V, E)$ as input.

Cook's Theorem - Example Continued

Example: reduce **Hamiltonian Cycle** in *NP* to **Satisfiability**.

- Now, the only inputs to this circuit, $\mathcal{C}$, is a potential certificate, Cert.
- The only output to $\mathcal{C}$ indicates whether the certificate, Cert, is valid or not.
- The only inputs that satisfy the circuit would be a Hamiltonian cycle for the graph $G(V, E)$, i.e. a valid certificate for this instance (problem).
- Key Point: If you could satisfy this circuit (find an input that gives an output 1) $\iff$ you could find a Hamiltonian cycle for the graph $G(V, E)$.
- There is nothing special about the Hamiltonian cycle version of the input. This same type of circuit could be constructed for any problem in *NP*, i.e. any problem whose certificate can be verified in polynomial time.

Cook's Theorem

Cook's Theorem proves that at least one NP-complete problem exists:

Theorem

Satisfiability $\in$ *NPC*.

How to reduce an arbitrary problem Π in *NP* to **Satisfiability**:

1. Suppose Π is in *NP*. Then there is a polynomial-time certificate verification algorithm for Π, say *Ver*(I, Cert), where I is an instance of Π and Cert is a certificate.
2. Convert the algorithm *Ver* to a Boolean circuit $\mathcal{C}$ that performs the same computation.
3. Given an instance I of Π, fix (i.e., hardwire) the Boolean inputs to $\mathcal{C}$ that correspond to the given instance I. The resulting circuit is defined to be $f(I)$, the encoding of I as a circuit satisfiability problem.
4. You can satisfy the circuit $\mathcal{C}$ $\iff$ you can find a certificate for the instance I.

Proving Problems NP-complete

Now, given a known NP-complete problem, say Π_k, other problems in *NP* can be proven to be NP-complete via polynomial reductions **from** Π_k, as stated in the following theorem:

Theorem

Suppose that the following conditions are satisfied:

- $\Pi_k \in$ *NPC, i.e. it is a known NPC problem,*
- $\Pi_k \leq_P \Pi_u$, *i.e.* Π_u *can be used to solve the NPC problem,* Π_k
- $\Pi_u \in$ *NP, verifying* Π_u*'s certificate is in P.*

Then $\Pi_u \in$ *NPC.*

Note: you reduce the *known* hard problem, Π_k, to the yet *unproven* hard problem, Π_u.

Proving Problems NP-complete

In order to prove a problem Π_u is NP-complete, show each the following.

1. **Show that Π_u is in NP**. That is, show that given an instance I and certificate, Cert, then *Ver*(I, Cert) will correctly verify the certificate in polynomial time.
2. **Select a known NPC problem, Π_k, to reduce from,** i.e. you reduce from a known NPC problem to the one you are uncertain about, $\Pi_k \leq_P \Pi_u$
3. **Create an algorithm f that solves Π_k using Π_u as a subroutine**.
4. **Prove the correctness of your algorithm.** That is, (a) f maps yes-instances of Π_k onto yes-instances of Π_u and (b) f maps no-instances of Π_k onto no-instances of Π_u.
5. **Show your algorithm f has polynomial time complexity.**

Here Π_k is the problem *known* to be in NPC and Π_u is the problem you are *uncertain* about.

Satisfiability Problems

Problem

CNF-Satisfiability

Instance: *A Boolean formula F in n Boolean variables $x_1, \ldots, x_n$, such that F is the* **conjunction** *(logical "and" or "$\wedge$") of m* **clauses**, *where each clause is the* **disjunction** *(logical "or" or "$\vee$") of literals. (A* **literal** *is a Boolean variable or its negation.)*

Question: *Is there a truth assignment such that F evaluates to* true*?*

E.g. given a formula in Conjuctive Normal Form (CNF) say

$$(x_1 \vee \bar{x_4}) \wedge (x_2 \vee x_5 \vee \bar{x_7}) \wedge (x_1 \vee \bar{x_3})$$

is there an assignment that satisfies the formula?

Satisfiability $\leq_P$ CNF-Satisfiability

Suppose that $\mathcal{C}$ is an instance of **SAT**, where $\mathcal{C}$ has inputs $X = \{x_1, \ldots, x_n\}$.

Denote the gates in $\mathcal{C}$ by $G_1, \ldots, G_m$, where G_m is the output gate.

We will construct $f(I)$, which will be an instance of **CNF-SAT**.

The instance $f(I)$ has Boolean variables $X' = X \cup \mathcal{G}$, where $\mathcal{G} = \{g_1, \ldots, g_m\}$.

The Boolean variable $g_i \in \mathcal{G}$ will "correspond" to the gate G_i in I.

Now, we can construct the clauses in $f(I)$:

- Suppose G_i is an or gate, say "x or y". We create this clauses in $f(I)$:

$$(g_i \vee \overline{x}) \wedge (g_i \vee \overline{y}) \wedge (\overline{g_i} \vee x \vee y).$$

  I.e. if x and y are false, the first two disjuctions ("*or*" terms) are true and so $\overline{g_i}$ must be true to make the last disjunction true.

Satisfiability $\leq_P$ CNF-Satisfiability (cont.)

- IF G_i is an and gate, say "x and y". We create the following clause in $f(I)$:

$$(\overline{g_i} \vee x) \wedge (\overline{g_i} \vee y) \wedge (g_i \vee \overline{x} \vee \overline{y}).$$

  If either x or y is false (last disjunction) then so is g_i (from the first two disjunctions).
- If G_i is a not gate, say "not x". We create the following in $f(I)$:

$$(g_i \vee x) \wedge (\overline{g_i} \vee \overline{x}).$$

- If G_i is a true gate (i.e., it is hard-wired to output the value T). We create one clause in $f(I)$:

$$(g_i).$$

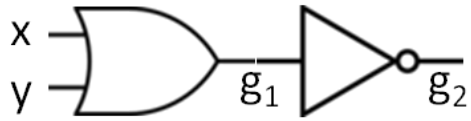
- If G_i is a false gate (i.e., it is hard-wired to output the value F). We create one clause in $f(I)$:

$$(\overline{g_i}).$$

- Finally, we construct one additional clause in $f(I)$, namely (g_m).

Satisfiability $\leq_P$ CNF-Satisfiability (cont.)

Consider two gates: *or* followed by *not*.



These gates would be implemented as

$$(g_1 \vee \overline{x}) \wedge (g_1 \vee \overline{y}) \wedge (\overline{g_1} \vee x \vee y) \wedge (g_2 \vee g_1) \wedge (\overline{g_2} \vee \overline{g_1}) \wedge (g_2)$$

and a satisfying solution would be $x = 0, y = 0, g_1 = 0, g_2 = 1$.

Q: Why not just implement the *or* gate as $(x \vee y)$ rather than as the more complicated $(g_1 \vee \overline{x}) \wedge (g_1 \vee \overline{y}) \wedge (\overline{g_1} \vee x \vee y)$?

A: There could be no possible solution if you expressed the *or* gate directly as $(x \vee y) \wedge \ldots$ Since each clause must be true for the whole formula to be true, then $x = 0, y = 0$ could not possibly be part of any satisfying solution, no matter what else you put after the first clause.

Prove CNF-Satisfiability $\in$ NPC

- $\Pi_1 \in NPC$, i.e. it is a known NPC problem.

  Satisfiability is in NPC via Cook's Theorem.

- $\Pi_1 \leq_P \Pi_2$, i.e. Π_2 can be used to solve the NPC problem, Π_1.

  In the previous two slides we showed how **CNF-Satisfiability** can be used to solve **Satisfiability**. This relationship is an *iff* one. That is, I is a yes-instance of **Satisfiability** $\iff$ $f(I)$ is a yes-instance of **CNF-Satisfiability**.

- $\Pi_2 \in NP$, verifying Π_2's certificate is in P.

  Given the value of each Boolean variable, the formula can be verified in O(n) where n is the literals in the entire formula.

Therefore, **CNF-Satisfiability** $\in NPC$.

Prove 3-CNF-Satisfiability $\in$ NPC

Problem

3-CNF-Satisfiability

Instance: *A Boolean formula F in n Boolean variables, such that F is the conjunction of m clauses, where each clause is the disjunction of exactly* **three** *distinct literals.*

Question: *Is there a truth assignment such that F evaluates to* true*?*

For example,

$$(x_1 \vee \bar{x_2} \vee \bar{x_4}) \wedge (x_2 \vee x_3 \vee x_r) \wedge (x_1 \vee x_2 \vee \bar{x_3}).$$

Each clause has *exactly three distinct* literals connected by "*or*" also written as "$\vee$".

CNF-Satisfiability $\leq_P$ 3-CNF-Satisfiability

Suppose that $(X, \mathcal{C})$ is an instance of **CNF-SAT**, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{C} = \{C_1, \ldots, C_m\}$. For each C_j, do the following:

case 1 If $|C_j| = 1$, say $C_j = \{z\}$, construct four clauses

$$\{z, a, b\}, \{z, a, \overline{b}\}, \{z, \overline{a}, b\}, \{z, \overline{a}, \overline{b}\}.$$

case 2 If $|C_j| = 2$, say $C_j = \{z_1, z_2\}$, construct two clauses

$$\{z_1, z_2, c\}, \{z_1, z_2, \overline{c}\}.$$

case 3 If $|C_j| = 3$, then leave C_j unchanged.

case 4 If $|C_j| \geq 4$, say $C_j = \{z_1, z_2, \ldots, z_k\}$, then construct $k - 2$ new clauses

$$\{z_1, z_2, d_1\}, \{\overline{d_1}, z_3, d_2\}, \{\overline{d_2}, z_4, d_3\}, \ldots,$$
$$\{\overline{d_{k-4}}, z_{k-2}, d_{k-3}\}, \{\overline{d_{k-3}}, z_{k-1}, z_k\}.$$

Prove 3-CNF-Satisfiability $\in$ NPC

- $\Pi_1 \in NPC$, i.e. it is a known *NPC* problem.

  CNF-Satisfiability is in *NPC* via the previous proof.

- $\Pi_1 \leq_P \Pi_2$, i.e. Π_2 can be used to solve the *NPC* problem, Π_1.

  In the previous slides we showed how **3-CNF-Satisfiability** can be used to solve **CNF-Satisfiability**.This relationship is an *iff* one. That is, I is a yes-instance of **CNF-Satisfiability** $\iff$ $f(I)$ is a yes-instance of **3-CNF-Satisfiability**.

- $\Pi_2 \in NP$, verifying Π_2's certificate is in *P*.

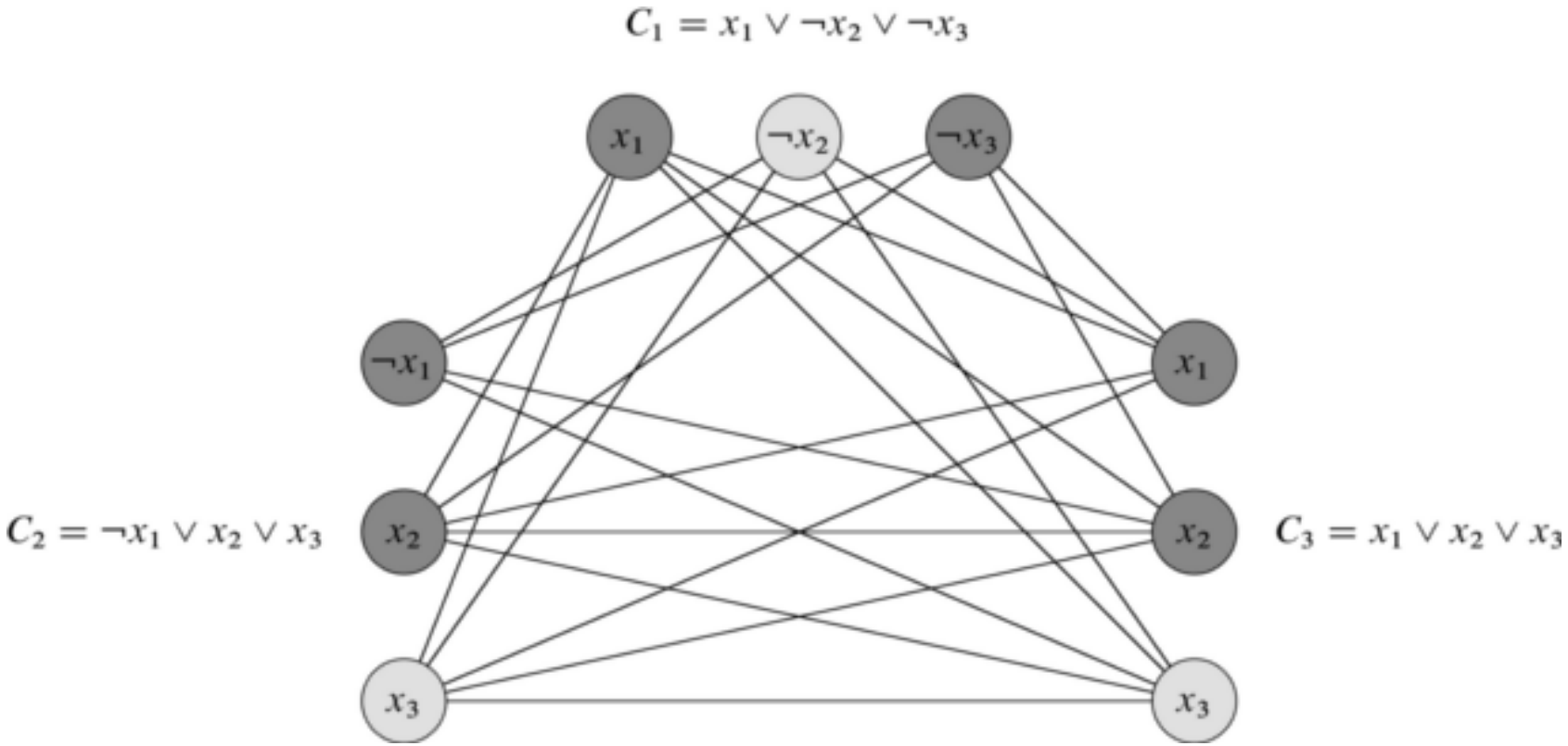
  Each Satisfiability clause of size n can be rewritten in a $O(n)$ 3-CNF-Satisfiability clauses, with each rewrite taking $O(1)$ time.

Therefore, **3-CNF-Satisfiability** $\in$ *NPC*.

Prove Clique $\in$ NPC: Basic Idea

Task: Prove **Clique** $\in$ *NPC* by the reduction **3-CNF-SAT** $\leq_P$ **Clique**.

Idea: Two vertices are connected by an edge if they represent literals that are in different clauses and are logically consistent. I.e. x is logically consistent with everything but $\overline{x}$.



Prove Clique $\in$ NPC: The Reduction

More formally, let I be the instance of **3-CNF-SAT** consisting of m clauses, $C_1, \ldots, C_m$ with each clause, C_i, consisting of three distinct literals $(z_1^i \vee z_2^i \vee z_3^i)$.

Define $f(I) = (G, k)$, where $G = (V, E)$ according to the following rules. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ add three vertices to the graph v_1^r, v_2^r, and v_3^r. Include an edge between two vertices, v_i^r and v_j^s, if

- v_i^r and v_j^s are in different triples (i.e. $r \neq s$) and
- the literals are consistent (i.e. l_i^r is not the negation of l_j^s).

There is a clique of size $m \iff$ there is an a solution that satisfies the 3-CNF formula.

Prove Clique $\in$ NPC

- **3-CNF-SAT** $\in$ *NPC*, proved a few slides back.
- **3-CNF-SAT** $\leq_P$ **Clique**, i.e. **Clique** can be used to solve the *NPC* problem, **3-CNF-SAT**. This relationship is an *iff* one. That is, I is a yes-instance of **3-CNF-Satisfiability** $\iff$ $f(I)$ is a yes-instance of **Clique**.
- **Clique** $\in$ *NP*, verifying **Clique**'s certificate is in *P*. For each pair of different vertices in the clique, say u, v check to see if uv is a valid edge in the graph. This verification can be done in $O(|V|^3)$.

Therefore **Clique** $\in$ *NPC*.

Subset Sum

Problem

Subset Sum

Instance: *A list of* **sizes** $S = \{s_1, \ldots, s_n\}$*; and a* **target sum**, t*. These are all positive integers.*

Question: *Does there exist a subset* $J \subseteq \{1, \ldots, n\}$ *such that* $\sum_{i \in J} s_i = t$*?*

For example, let $S = \{1, 3, 5, 17, 23, 35, 41\}$ and let the target sum $t = 59$. A solution would be $\{1, 17, 41\}$ because $1 + 17 + 41 = 59$.

Clearly this problem is in *NP*. The certificate would the be elements of S that sum up to t and the verification algorithm would merely added up these values and verify that their sum is t.

We will reduce a problem known to be in *NPC*, **3-CNF-Satisfiability**, to **Subset Sum**, i.e. **3-CNF-Sat** $\leq_P$ **Subset Sum**.

3-CNF-Sat $\leq_P$ Subset Sum

For example, convert the following 3-CNF-Sat formula into a subset sum.

$$(x_1 \vee \bar{x_2} \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

		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

3-CNF-Sat $\leq_P$ Subset Sum

Literals

The first six row corresponds to possible truth values $x_1, \bar{x_1}, x_2, \bar{x_2}, x_3, \bar{x_3}$.

The first three columns corresponds to the constraint on these values.

Since the sum of each of the first three columns must be 1, then only one of the two options x_i or $\bar{x_i}$ can be true. So the first three columns are enforcing the constraint that either x_1 or $\bar{x_1}$ is "true."

Clauses

The last four columns correspond to the clauses. The bottom half of the columns, $(s_1, \ldots s_4')$, sums to three. So at least one row from the top six, $(v_1, \ldots v_3')$, must be included in the sum. This arrangement enforces the constraint that each clause has to have at least one true literal.

A clause can have one, two or three true literals so rows $(s_1, \ldots s_4')$ are added to allow for any of those three possibilities and still have the column sum up to four.

3-CNF-Sat $\leq_P$ Subset Sum

Let I be the instance of **3-CNF-SAT** consisting of n variables, $x_1, \ldots, x_n$, and m clauses, $C_1, \ldots, C_m$.

We construct a $2(n+m)$ by $n+m$ integer-valued matrix M.

The first $2n$ rows of M are indexed by the $2n$ literals $x_1, \bar{x_1}, \ldots, x_n, \bar{x_n}$ and the last $2m$ rows are named $s_1, s'_1, \ldots, s_m, s'_m$.

The columns are named $x_1, \ldots, x_n, C_1, \ldots, C_m$.

Define M as follows:

$$
\begin{array}{ll}
M[v_i, x_j] = 1 & \text{if } v_i = x_j \text{ or } v_i = \bar{x_j} \\
M[v_i, C_j] = 1 & \text{if } v_i \in C_j \\
M[s_j, C_j] = 1 & \text{for } j = 1, \ldots, m \\
M[s'_j, C_j] = 2 & \text{for } j = 1, \ldots, m \\
M[i, j] = 0 & \text{otherwise.}
\end{array}
$$

3-CNF-Sat $\leq_P$ Subset Sum (cont.)

Each row i of M is interpreted as an $(n+m)$-digit (base-10) integer, s_i.

The target sum is $t = \underbrace{11\cdots1}_{n}\underbrace{44\cdots4}_{m}$.

Define $f(I)$ to be the instance of **Subset Sum** consisting of sizes $[s_1, \ldots, s_{2m+2n}]$ and target sum t.

Prove TSP-Dec $\in$ NPC

Problem

TSP-Decision

Instance: *A graph G, edge weights $w : E \to \mathbb{Z}^+$, and a target T.*

Question: *Does there exist a tour H in G with $w(H) \leq T$?*

Select a known problem in *NPC* that will be reduced to **TSP-Dec**.

- The problem **Hamiltonian Cycle** or **HAM** is a known *NPC* problem.

Show that **HAM** $\leq_P$ **TSP-Dec**, i.e. that **TSP-D** can be used to solve the known *NPC* problem, **HAM**.

- Suppose our input instance, I, for the **HAM** problem is a graph G.
- We want to find a function $f(G)$ that will be an input to **TSP-Dec**, i.e. f will create a new graph G_{TSP} and a threshold T that will act as input to the algorithm that solves **TSP-Dec**.

Prove TSP-Dec $\in$ NPC

Show that **HAM** $\leq_P$ **TSP-D** cont.

- Create a complete graph G_{TSP} that has the same vertices as G but with the following two properites, the weight of an edge uv in G_{TSP} is
  - 0 if uv is an edge in G,
  - 1 otherwise.
- The graph G has a hamiltonian cycle $\iff$ the graph G_{TSP} has a tour of length 0.

Check that **TSP-Dec** $\in$ *NP*, i.e. verifying **TSP-Dec**'s certificate is in *P*.

- Let the certificate be the list of edges in the TSP tour. Clearly it takes polynomial time to look up the weight of each edge and check that their sum is less that or equal to the target sum T and check that each city (vertex) has been visited only once.

Therefore **TSP-Dec** $\in$ *NPC*.

Subset Sum $\leq_P$ 0-1 Knapsack

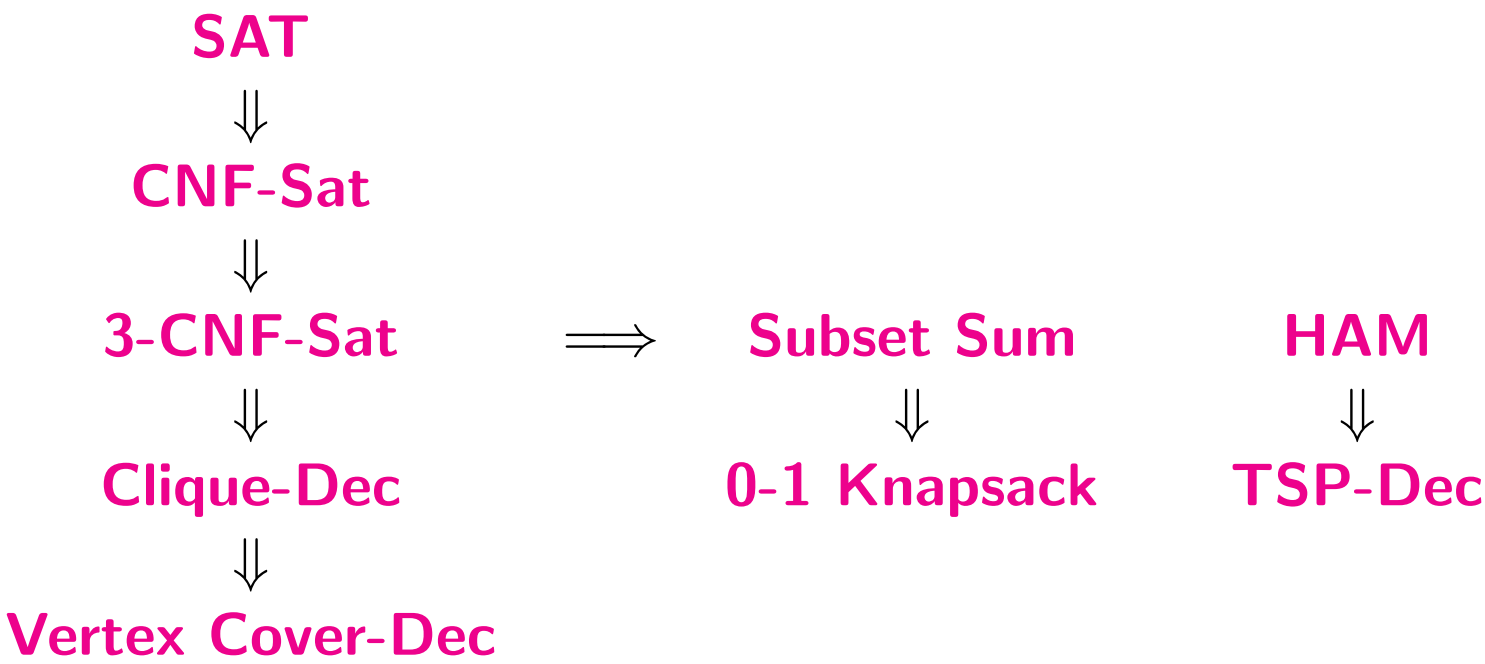
Let I be an instance of **Subset Sum** consisting of sizes $[s_1, \ldots, s_n]$ and target sum t.

Define

$$
\begin{aligned}
p_i &= s_i,\ 1 \leq i \leq n \\
w_i &= s_i,\ 1 \leq i \leq n \\
M &= t \\
T &= t.
\end{aligned}
$$

Then define $f(I)$ to be the instance of **0-1 Knapsack** consisting of profits $[p_1, \ldots, p_n]$, weights $[w_1, \ldots, w_n]$, capacity M and target profit T.

Reductions among NP-complete Problems



In the above diagram, arrows denote polynomial reductions. The seven that you are allowed to use on assignments and the final are: CNF-Sat, 3-CNF-Sat, Clique, Vertex Cover, Hamiltonian Cycle (HAM), Travelling Salesperson Problem–Decision (TSP-Dec), and Subset Sum.

Standard NP-complete Problems for CS341

Problem

CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables $x_1, \ldots, x_n$, such that F is the* **conjunction** *(logical "and" or "$\wedge$") of m* **clauses**, *where each clause is the* **disjunction** *(logical "or" or "$\vee$") of literals. (A* **literal** *is a boolean variable or its negation.)*

Question: *Is there a truth assignment such that F evaluates to* true*?*

Problem

3-CNF-Satisfiability

Instance: *A boolean formula F in n boolean variables, such that F is the conjunction of m clauses, where each clause is the disjunction of exactly* **three** *literals.*

Question: *Is there a truth assignment such that F evaluates to* true*?*

Standard NP-complete Problems for CS341

Problem

Clique-Dec

Instance: *An undirected graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.*

Question: *Does G contain a clique of size $\geq k$? (A **clique** is a subset of vertices $W \subseteq V$ such that $uv \in E$ for all $u, v \in W$, $u \neq v$.)*

Problem

Vertex Cover-Dec

Instance: *An undirected graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.*

Question: *Does G contain a vertex cover of size $\leq k$? (A **vertex cover** is a subset of vertices $W \subseteq V$ such that $\{u, v\} \cap W \neq \emptyset$ for all edges $uv \in E$.)*

Standard NP-complete Problems for CS341

Problem

Subset Sum

Instance: *A list of* **sizes** $S = \{s_1, \ldots, s_n\}$*; and a* **target sum**, t*. These are all positive integers.*

Question: *Does there exist a subset* $J \subseteq S$ *such that* $\sum_{s_i \in J} s_i = t$*?*

Problem

Hamiltonian Cycle

Instance: *An undirected graph* $G = (V, E)$*.*

Question: *Does* G *contain a Hamiltonian cycle?*

Problem

TSP-Decision

Instance: *A graph* G*, edge weights* $w : E \to \mathbb{Z}^+$*, and a target* T*.*

Question: *Does there exist a tour* H *in* G *with* $w(H) \leq T$*?*

Polynomial-time Turing Reductions

So far we have been characterizing decision problems, but what about optimization problems or actually finding the answer, e.g. the actual tour of cities for the optimal Travelling Salesman tour.

1. **Optimizing**

   Do a binary search to find the optimal value c, the cost of the tour, by bounding the interval from above with a large value, e.g. the sum of the $|V| + 1$ largest edge lengths.

2. **Creating a Certificate**

   Once you found the optimal value for c start setting edge lengths to infinity. If changing an edge length increase the cost of the tour, then that edge is on the tour.

Travelling Salesperson Problems

Problem

TSP-Optimization

Instance: *A graph G and edge weights $w : E \to \mathbb{Z}^+$.*

Find: *A hamiltonian cycle H in G such that $w(H) = \sum_{e \in H} w(e)$ is minimized.*

Problem

TSP-Decision

Instance: *A graph G, edge weights $w : E \to \mathbb{Z}^+$, and a target T.*

Question: *Does there exist a hamiltonian cycle H in G with $w(H) \leq T$?*

Clearly if solving **TSP-Decision** was in P one could solve **TSP-Optimization** in polynomial time.

It could require calling **TSP-Decision** up to a polynomial number of time.

TSP-Optimization $\leq_P^T$ TSP-Decision

Algorithm: *TSP-Optimization*(G, w)
external *TSP-Dec-Solver*
$hi \leftarrow \sum_{e \in E} w(e)$
$lo \leftarrow 0$
if not *TSP-Dec-Solver*(G, w, hi) **then return** (∞)
while $hi > lo$
do $\begin{cases} mid \leftarrow \lfloor \frac{hi+lo}{2} \rfloor \\ \textbf{if } \textit{TSP-Dec-Solver}(G, w, mid) \textbf{ then } hi \leftarrow mid \\ \quad \textbf{else } lo \leftarrow mid + 1 \end{cases}$
return (hi)

Here you are using the algorithm that solves the decision version of the TSP, *TSP-Dec-Solver*, to solve the optimization version, *TSP-Optimization* by performing a binary search for the lowest tour length.

Polynomial-time Turing Reductions

Previously, we were transforming a instance I for one decision problem Π_k into an instance $f(I)$ for another decision problem Π_u.

Now, we are calling a subroutine that solves an NPC problem up to a polynomial number of times.

Suppose Π_1 and Π_2 are problems (not necessarily decision problems). A (hypothetical) algorithm $\mathbf{A_2}$ to solve Π_2 is called an **oracle** for Π_2.

Suppose that $\mathbf{A}$ is an algorithm that solves Π_1, assuming the existence of an oracle $\mathbf{A_2}$ for Π_2. ($\mathbf{A_2}$ is used as a subroutine within the algorithm $\mathbf{A}$.)

We say that $\mathbf{A}$ is a **Turing reduction** from Π_1 to Π_2, denoted $\Pi_1 \leq^T \Pi_2$.

A Turing reduction $\mathbf{A}$ is a **polynomial-time Turing reduction** if the running time of $\mathbf{A}$ is polynomial, under the assumption that the oracle $\mathbf{A_2}$ has **unit cost** running time.

We denote a polynomial-time Turing reduction as $\Pi_1 \leq_P^T \Pi_2$.

NP-hard Problems

Suppose Π is a problem (not necessarily a decision problem).

Π is **NP-hard** if there exists a problem $\Pi' \in NPC$ such that $\Pi' \leq_P^T \Pi$.

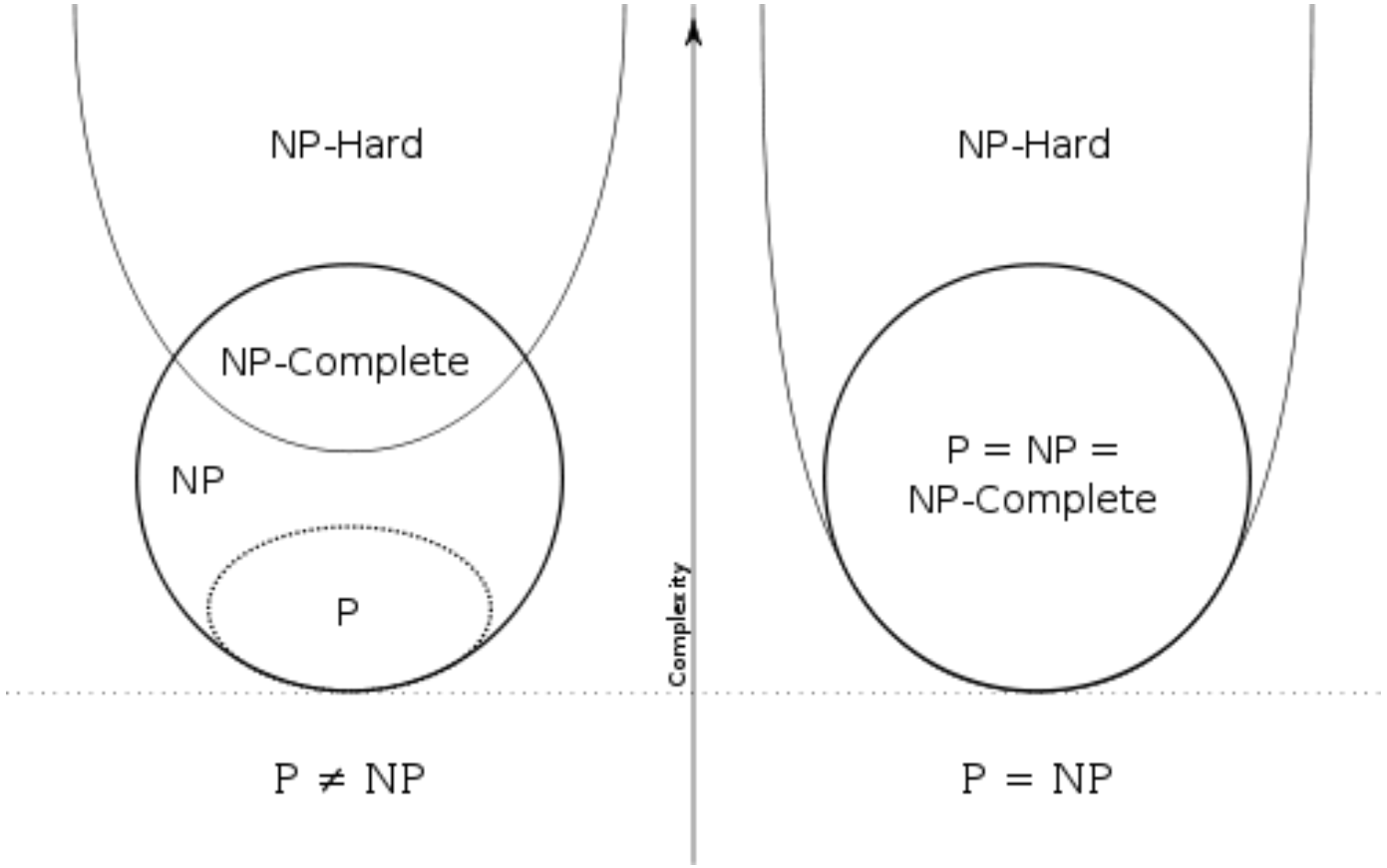
If Π is an optimization or optimal value problem and the related decision problem is NP-complete, then Π is NP-hard.

This is because there are trivial poly-time Turing reductions from the decision problem to the optimal value and optimization versions of the problem.

Theorem

If Π is NP-hard and Π is solvable in polynomial time, then $P = NP$.

NP-hard Problems



NP-Hard problems include: NP-complete problems (these are decision problems), optimization versions of NP-complete problems, and other problems such as the Halting problem.

Image source http://en.wikipedia.org/wiki/Np-hard.

Conclusion

In this course you have hopefully ...

- enhanced your ability to analyze programs with various bounds $O, \Theta, \Omega, o, \omega$.
- learned a number of different programming paradigms: greedy, divide-and-conquer, graph algorithms, and dynamic programming.
- learned to recognize problems that are difficult if not impossible to solve effeciently,
- through class brainstorming seen how to approach a difficult problem by taking an educated guess, identifying where it went wrong (if it did) and then refining your approach.

Stay tuned to Piazza for a *pinned* posting covering what is on the final.

Thank-you for your attention and participation in the class.