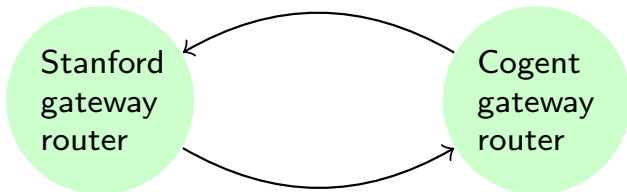
Algorithms: Design and Analysis, Part II

Introduction

Motivating Application: Distributed Shortest-Path Routing

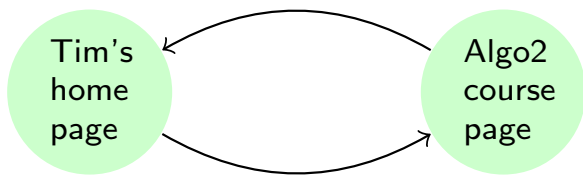
Graphs and the Internet

Claim: The Internet is a graph [vertices = end hosts + routers, directed edges = direct physical or wireless connections].



Other graphs related to the Internet:

Web graph. [vertices = web pages, edges = hyperlinks].



Social networks. [vertices = people, edges = friend/follow relationships].

Internet Routing

Suppose: Stanford gateway router needs to send data to the Cornell gateway router (over multiple hops).

Question: Which Stanford→Cornell route to use?

Obvious idea: How about the shortest? (e.g. fewest # of hops).

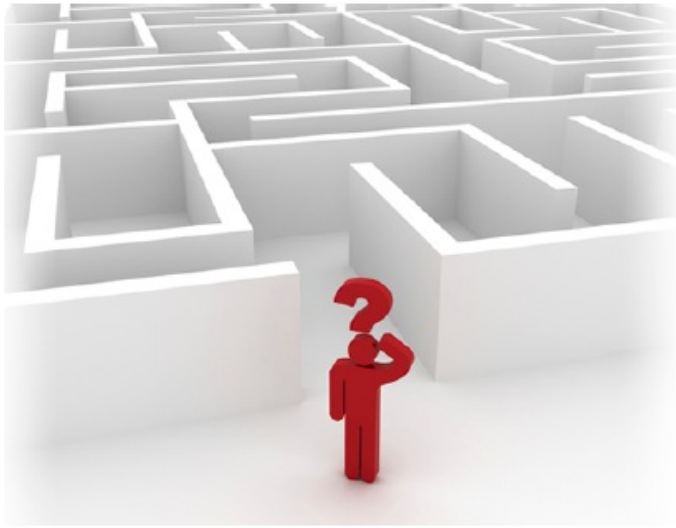
⇒ Need a shortest-path algorithm.

Recall from Part I: Dijkstra's algorithm does this (with nonnegative edge lengths).

Issue: Stanford gateway router would need to know entire Internet!

⇒ Need a shortest-path algorithm that uses only *local* computation.

Solution: the *Bellman-Ford* algorithm (bonus: also handles negative edge costs).



Algorithms: Design and Analysis, Part II

Introduction

Motivating Application: Sequence Alignment

Motivation

Sequence alignment: Fundamental problem in computational genomics.

Input: Two strings over the alphabet {A,C,G,T}. [Portions of one or more genomes]

Problem: Figure out how similar the two strings are.

Example:

A G G G C T

A G G C A

Example applications:

- Extrapolate function of genome substrings.
- Similar genomes can reflect proximity in evolutionary tree.

Measuring Similarity

Question: What does similar mean?

Intuition: AGGGCT, AGGCA are similar because they can be "nicely aligned".

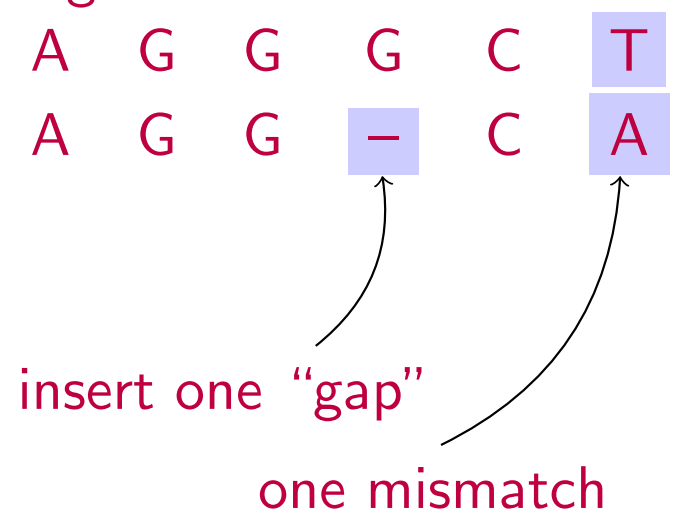
Idea: Measure similarity via quality of "best" alignment.

Assumption: Have experimentally determined *penalties* for gaps and the possible matches.

Example:

A G G G C T
A G G C A

Alignment:



Problem Statement

Input: 2 strings over {A,C,G,T}.

- Penalty $pen_{gap} \geq 0$ for each gap.
- Penalty $pen_{AT} \geq 0$ for mismatching A and T.
- etc.

Output: Alignment of the strings that minimizes the total penalty

$\Rightarrow$ Called the Needleman-Wunsch score [1970].

Small NW score $\approx$ Similar Strings

Algorithms are Fundamental

Note: This measure of genome similarity would be useless without an efficient algorithm to find the best alignment.

Brute-force search: Try all possible alignments, remember the best one.

Question: Suppose each string has length 500. Roughly how many possible alignments are there?

A) # of students in this class $\approx 10^4 - 10^5$

B) # of people on earth $\approx 10^9 - 10^{10}$

C) # of atoms in known universe $\approx 10^{80}$

D) More than any of the above $\geq 2^{500} \geq 10^{125}$

Point: Need a fast, clever algorithm.

Solution: Straightforward dynamic programming.

Introduction

About The Course

Design and Analysis
of Algorithms I

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
- Primitives for reasoning about graphs
- Use and implementation of data structures

Course Topics

- Vocabulary for design and analysis of algorithms
  - E.g., “Big-Oh” notation
  - “sweet spot” for high-level reasoning about algorithms

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
  - Will apply to: Integer multiplication, sorting, matrix multiplication, closest pair
  - General analysis methods ("Master Method/Theorem")

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
  - Will apply to: QuickSort, primality testing, graph partitioning, hashing.

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
- Primitives for reasoning about graphs
  - Connectivity information, shortest paths, structure of information and social networks.

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
- Primitives for reasoning about graphs
- Use and implementation of data structures
  - Heaps, balanced binary search trees, hashing and some variants (e.g., bloom filters)

Topics in Sequel Course

- Greedy algorithm design paradigm

Topics in Sequel Course

- Greedy algorithm design paradigm
- Dynamic programming algorithm design paradigm

Tim Roughgarden

Topics in Sequel Course

- Greedy algorithm design paradigm
- Dynamic programming algorithm design paradigm
- NP-complete problems and what to do about them

Topics in Sequel Course

- Greedy algorithm design paradigm
- Dynamic programming algorithm design paradigm
- NP-complete problems and what to do about them
- Fast heuristics with provable guarantees
- Fast exact algorithms for special cases
- Exact algorithms that beat brute-force search

Skills You'll Learn

- Become a better programmer

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills
- Start "thinking algorithmically"

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills
- Start "thinking algorithmically"
- Literacy with computer science's "greatest hits"

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills
- Start "thinking algorithmically"
- Literacy with computer science's "greatest hits"
- Ace your technical interviews

Who Are You?

- It doesn't really matter. (It's a free course, after all.)

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.
- Doesn't matter which language(s) you know.
  - But you should be capable of translating high-level algorithm descriptions into working programs in *some* programming language.

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.
- Doesn't matter which language(s) you know.
- Some (perhaps rusty) mathematical experience.
  - Basic discrete math, proofs by induction, etc.

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.
- Doesn't matter which language(s) you know.
- Some (perhaps rusty) mathematical experience.
  - Basic discrete math, proofs by induction, etc.

- *Excellent free reference: "*Mathematics for Computer Science", by Eric Lehman and Tom Leighton. (Easy to find on the Web.)

Supporting Materials

- All (annotated) slides available from course site.

Supporting Materials

- All (annotated) slides available from course site.
- No required textbook. A few of the many good ones:
  - Kleinberg/Tardos, *Algorithm Design*, 2005.
  - Dasgupta/Papadimitriou/Vazirani, *Algorithms*, 2006.
  - Cormen/Leiserson/Rivest/Stein, *Introduction to Algorithms*, 2009 (3rd edition).
  - Mehlhorn/Sanders, *Data Structures and Algorithms: The Basic Toolbox*, 2008.

Biggest influence on instructor

Freely available online

Supporting Materials

- All (annotated) slides available from course site.
- No required textbook. A few of the many good ones:
  - Kleinberg/Tardos, *Algorithm Design*, 2005.
  - Dasgupta/Papadimitriou/Vazirani, *Algorithms*, 2006.
  - Cormen/Leiserson/Rivest/Stein, *Introduction to Algorithms*, 2009 (3rd edition).
  - Mehlhorn/Sanders, *Data Structures and Algorithms: The Basic Toolbox*, 2008.
- No specific development environment required.
  - But you should be able to write and execute programs.

Assessment

- No grades per se. (Details on a certificate of accomplishment TBA.)
- Weekly homeworks.
  - Test understand of material
  - Synchronize students, greatly helps discussion forum
  - Intellectual challenge

Assessment

- No grades per se. (Details on a certificate of accomplishment TBA.)
- Weekly homeworks.
- Assessment tools currently just a “1.0” technology.
  - We’ll do our best!
- Will sometimes propose harder “challenge problems”
  - Will not be graded; discuss solutions via course forum

Design and Analysis
of Algorithms I

Introduction

Why Study Algorithms?

Why Study Algorithms?

- important for all other branches of computer science

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
  - "Everyone knows Moore's Law – a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years….in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed."
    - Excerpt from *Report to the President and Congress: Designing a Digital Future,* December 2010 (page 71).

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel "lens" on processes outside of computer science and technology
  - quantum mechanics, economic markets, evolution

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel "lens" on processes outside of computer science and technology
- challenging (i.e., good for the brain!)

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel "lens" on processes outside of computer science and technology
- challenging (i.e., good for the brain!)
- fun

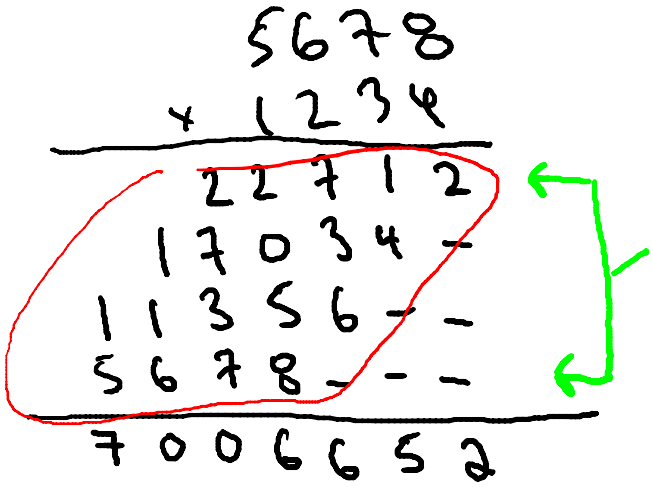
Integer Multiplication

Input : 2 n-digit numbers x and y

Output : product x*y

“Primitive Operation” - add or multiply 2 single-digit numbers

The Grade-School Algorithm



Roughly n operations per row up to a constant

# of operations overall ~ constant* n^2

The Algorithm Designer's Mantra

"Perhaps the most important principle for the good algorithm designer is to refuse to be content."

-Aho, Hopcroft, and Ullman, *The Design and Analysis of Computer Algorithms*, 1974

CAN WE DO BETTER ?
[than the "obvious" method]

A Recursive Algorithm

Write $x = 10^{n/2}a + b$ and $y = 10^{n/2}c + d$

Where a,b,c,d are n/2-digit numbers.

[example: a=56, b=78, c=12, d=34]

Then $x.y = (10^{n/2}a + b)(10^{n/2}c + d)$

$$= (10^n ac + 10^{n/2}(ad + bc) + bd \qquad (*)$$

Idea : recursively compute ac, ad, bc, bd, then compute (*) in the obvious way

Simple Base Case Omitted

Karatsuba Multiplication

$$x.y = (10^n ac + 10^{n/2}(ad + bc) + bd$$

1. Recursively compute ac
2. Recursively compute bd
3. Recursively compute (a+b)(c+d) = ac+bd+ad+bc

Gauss' Trick : (3) – (1) – (2) = ad + bc

Upshot : Only need 3 recursive multiplications (and some additions)

Q : which is the fastest algorithm ?

Design and Analysis
of Algorithms I

Introduction

Guiding Principles

Guiding Principle #1

"worst – case analysis" : our running time bound holds for every input of length n.
-Particularly appropriate for "general-purpose" routines

As Opposed to
--"average-case" analysis
--benchmarks

REQUIRES DOMAIN KNOWLEDGE

BONUS : worst case usually easier to analyze.

Guiding Principle #2

Won't pay much attention to constant factors, lower-order terms

<u>Justifications</u>

1. Way easier
2. Constants depend on architecture / compiler / programmer anyways
3. Lose very little predictive power (as we'll see)

Guiding Principle #3

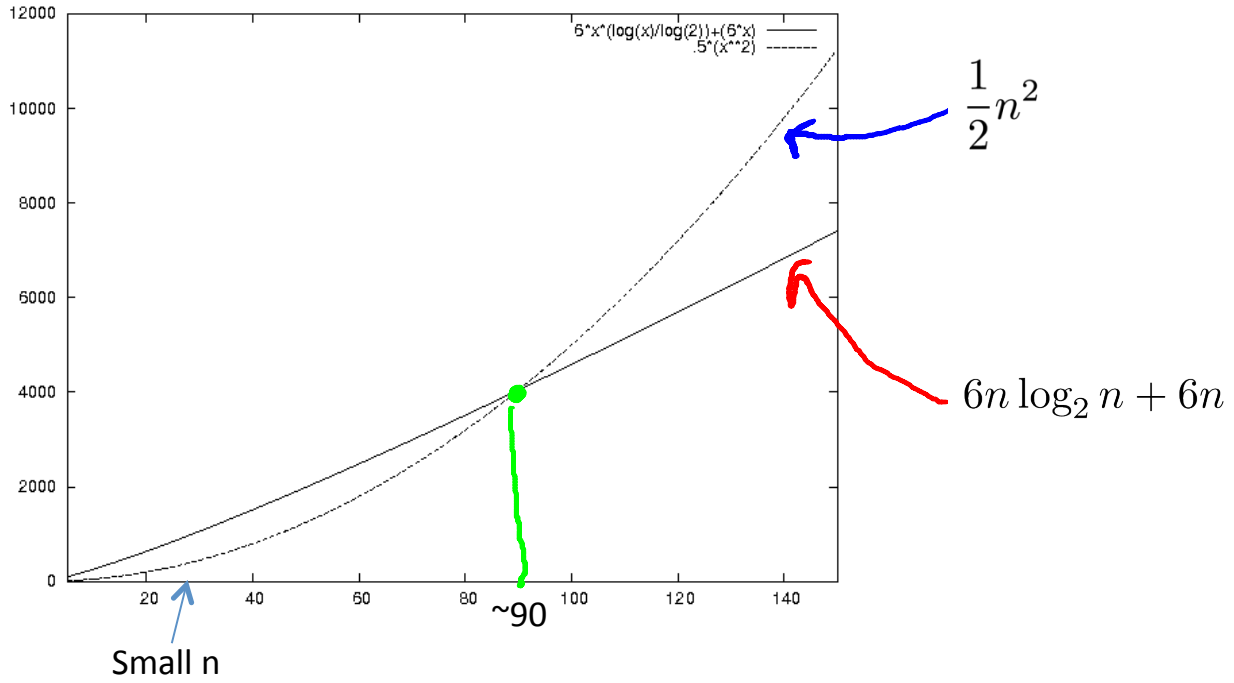
Asymptotic Analysis : focus on running time for large input sizes n

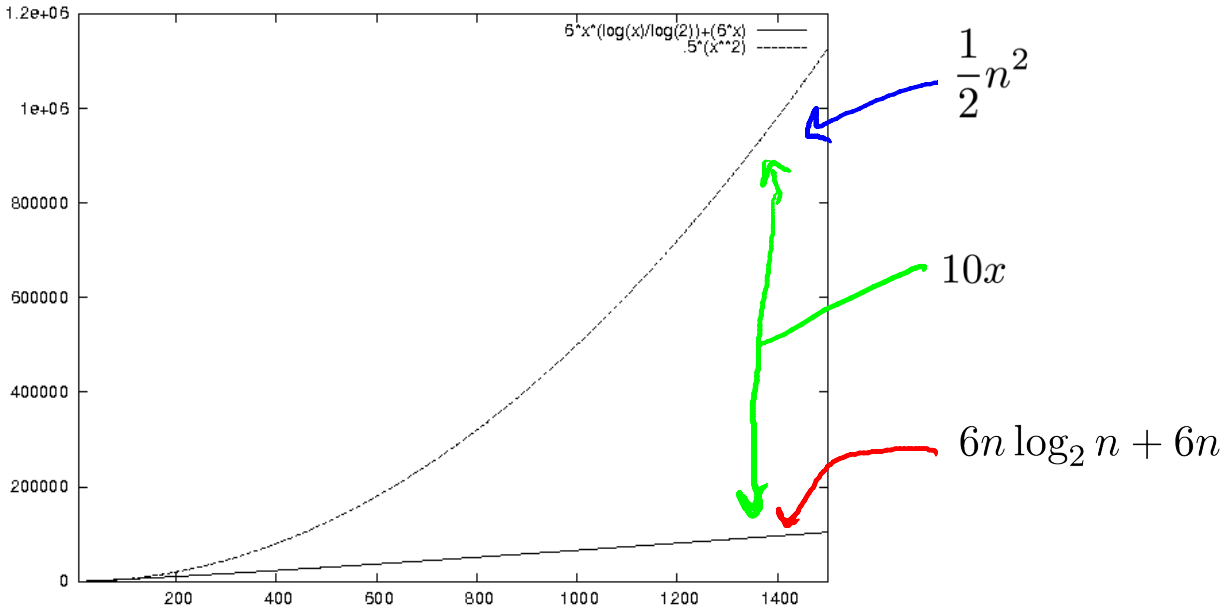
Eg : $6n \log_2 + 6n$ "better than" $\frac{1}{2}n^2$

MERGE SORT

INSERTION SORT

Justification: Only big problems are interesting!


6*x*(log(x)/log(2))+(6*x)
.5*(x**2)
12000
10000
8000
6000
4000
2000
0
20
40
60
80
100
120
140
~90
Small n
$\frac{1}{2}n^2$
$6n \log_2 n + 6n$



Tim Roughgarden

What Is a "Fast" Algorithm?

This Course : adopt these three biases as guiding principles

fast algorithm $\approx$ worst-case running time grows slowly with input size

Usually : want as close to linear (O(n)) as possible

Asymptotic Analysis

Big-Oh: Definition

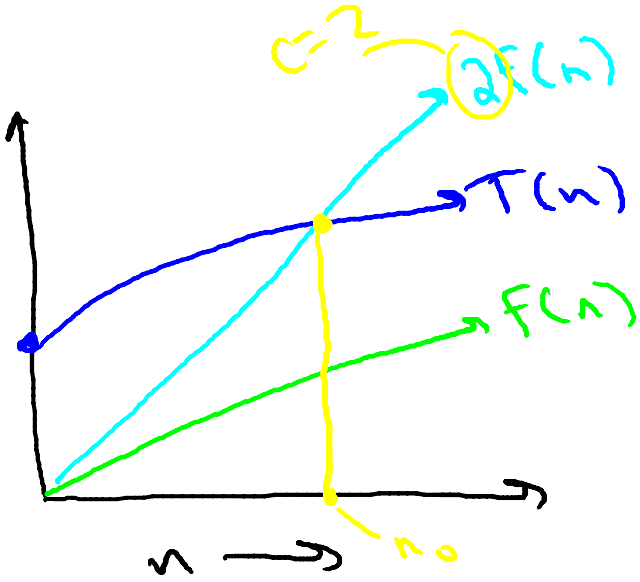
Design and Analysis of Algorithms I

Big-Oh: English Definition

Let T(n) = function on n = 1,2,3,…
[usually, the worst-case running time of an algorithm]

Q : When is T(n) = O(f(n)) ?
A : if eventually (for all sufficiently large n), T(n) is bounded above by a constant multiple of f(n)

Big-Oh: Formal Definition



Picture $T(n) = O(f(n))$

Formal Definition : T(n) = O(f(n)) if and only if there exist constants $c, n_0 > 0$ such that

$$T(n) \leq c \cdot f(n)$$

For all $n \geq n_0$

Warning : c, n_0 cannot depend on n

Design and Analysis
of Algorithms I

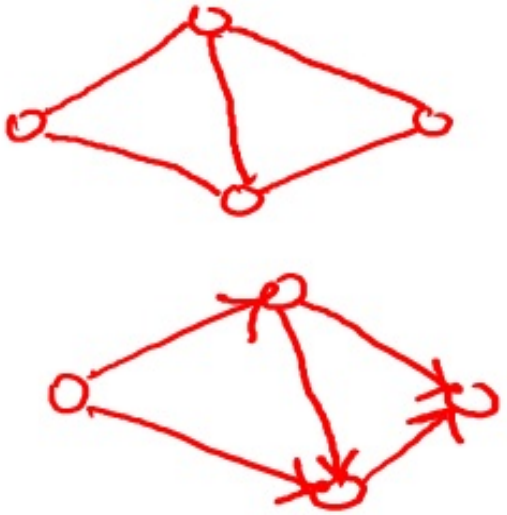
Graph Algorithms

Representing Graphs

Graphs

Two ingredients

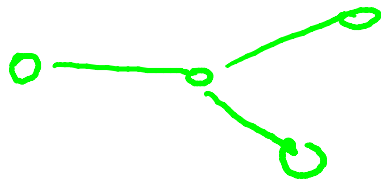
- Vertices aka nodes (V)
- Edges (E) = pairs of vertices
  - can be undirected [unordered pair] or directed [ordered pair] (aka arcs)

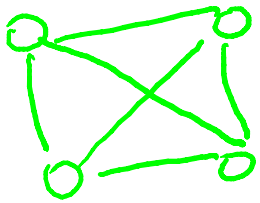


Examples: road networks, the Web, social networks, precedence constraints, etc.

Consider an undirected graph that has n vertices, no parallel edges, and is connected (i.e., “in one piece”). What is the minimum and maximum number of edges that the graph could have, respectively ?

- ○ $n-1$ and $n(n-1)/2$
- ○ $n-1$ and n^2
- ○ n and 2^n
- ○ n and n^n





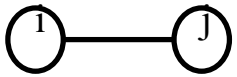
Sparse vs. Dense Graphs

Let n = # of vertices, m = # of edges.

In most (but not all) applications, m is $\Omega(n)$ and $O(n^2)$

- in a "sparse" graph, m is or is close to $O(n)$
- in a "dense" graph, m is closer to $\theta(n^2)$

The Adjacency Matrix

Represent G by a n x n 0-1 matrix A where
$A_{ij} = 1 \Leftrightarrow$ G has an i-j edge 

<u>Variants</u>

- A_{ij} = # of i-j edges (if parallel edges)
- A_{ij} = weight of i-j edge (if any)
- $A_{ij} = \begin{cases} +1 \text{ if } \circ\!\!\rightarrow\!\!\circ \\ -1 \text{ if } \circ\!\!\leftarrow\!\!\circ \end{cases}$

Ter
ver

How much space does an adjacency matrix require, as a function of the number n of vertices and the number m of edges?

- $\theta(n)$
- $\theta(m)$
- $\theta(m+n)$
- $\theta(n^2)$

Adjacency Lists

Ingredients

- array (or list) of vertices
- array (or list) of edges
- each edge points to its endpoints
- each vertex points to edges incident on it

Ter
ver

How much space does an adjacency list representation require, as a function of the number n of vertices and the number m of edges?

- $\theta(n)$
- $\theta(m)$
- $\theta(m+n)$
- $\theta(n^2)$

Adjacency Lists

Ingredients

- array (or list) of vertices
- array (or list) of edges
- each edge points to its endpoints
- each vertex points to edges incident on it

Space

$\theta(n)$

$\theta(m)$

$\theta(m)$

$\theta(m)$

$\theta(m+n)$

$[or \ \theta(max\{m,n\})]$

Question: which is better?

Answer: depends on graph density and operations needed.

This course: focus on adjacency lists.

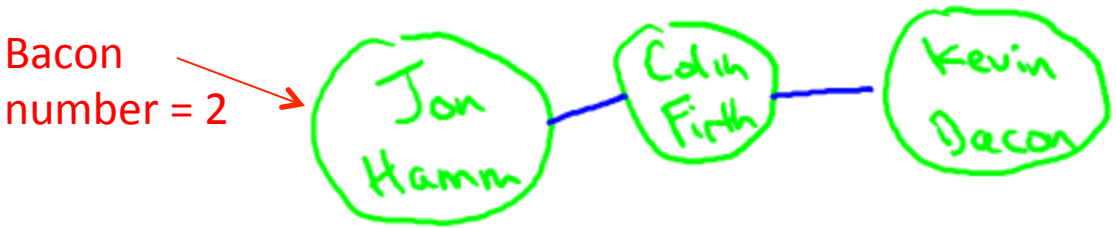
Graph Primitives

Introduction to Graph Search

Design and Analysis of Algorithms I

A Few Motivations

1. Check if a network is connected (can get to anywhere from anywhere else)



2. Driving directions
3. Formulate a plan [e.g., how to fill in a Sudoku puzzle]
   -- nodes = a partially completed puzzle -- arcs = filling in one new sequence
4. Compute the "pieces" (or "components") of a graph
   -- clustering, structure of the Web graph, etc.

Generic Graph Search

Goals : 1) find everything findable from a given start vertex

2) don't explore anything twice

Goal:
O(m+n) time

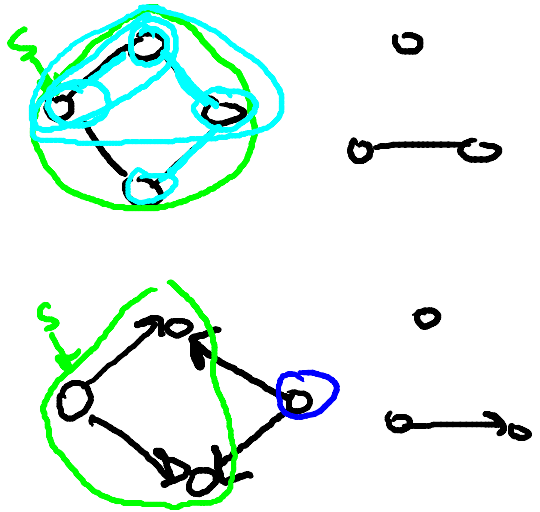
Generic Algorithm (given graph G, vertex s)

-- initially s explored, all other vertices unexplored

-- while possible : (if none, halt)

-- choose an edge (u,v) with u explored and v unexplored

-- mark v explored



Generic Graph Search (con'd)

Claim : at end of the algorithm, v explored <==> G has a path from s to v
(G undirected or directed)

Proof : (=>) easy induction on number of iterations (you check)
(<=) By contradiction. Suppose G has a path P from s to v:

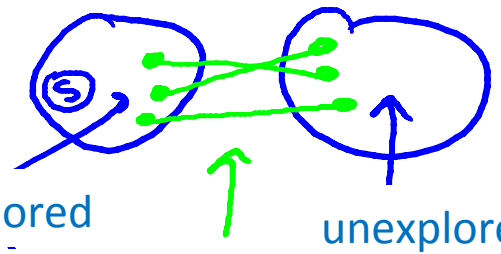
But v unexplored at end of the algorithm. Then there exists an edge (u,x) in P with u explored and x unexplored.
But then algorithm would not have terminated, contradiction.

Q.E.D.

Tim Roughgarden

BFS vs. DFS

Note : how to choose among the possibly many "frontier" edges ?



Breadth-First Search (BFS)

-- explored nodes in "layers"

-- can compute shortest paths

-- can compute connected components of an undirected graph

O(m+n) time using a queue (FIFO)

Depth-First Search (DFS)

-- explore aggressively like a maze, backtrack only when necessary

-- compute topological ordering of a directed acyclic graph

-- compute connected components in directed graphs

O(m+n) time using a stack (LIFO) (or via recursion)

Graph Primitives

Dijkstra's Algorithm: The Basics

Design and Analysis of Algorithms I

Single-Source Shortest Paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

- each edge has non negative length l_e
- source vertex s

Output: for each $v \in V$, compute

L(v) := length of a shortest s-v path in G

Length of path
= sum of edge lengths

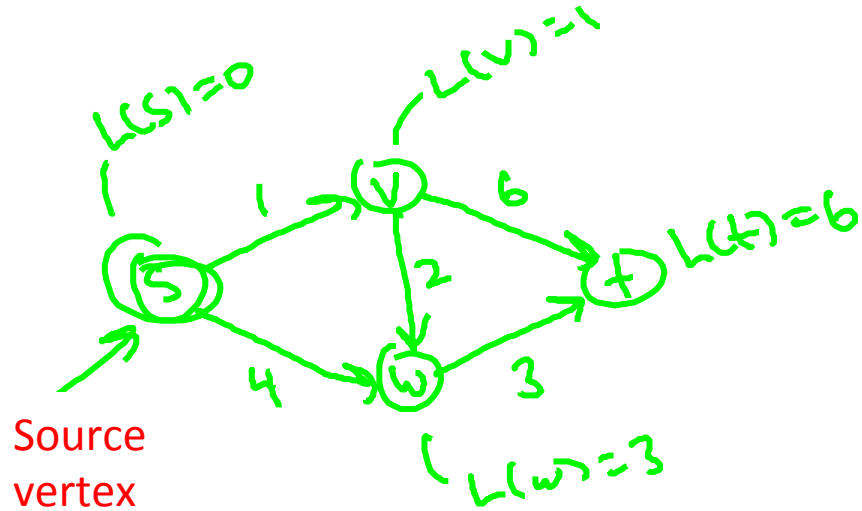
Path length = 6

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $le \geq 0 \ \forall e \in E$

One of the following is the list of shortest-path distances for the nodes s,v,w,t, respectively. Which is it?

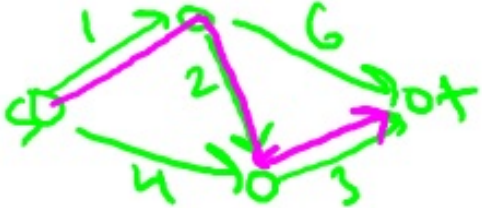
- 0,1,2,3
- 0,1,4,7
- 0,1,4,6
- 0,1,3,6



Why Another Shortest-Path Algorithm?

Question: doesn't BFS already compute shortest paths in linear time?

Answer: yes, IF $l_e = 1$ for every edge e.



Question: why not just replace each edge e by directed path of l_e unit length edges:

Answer: blows up graph too much

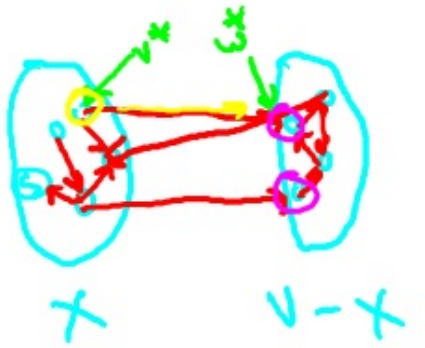
Solution: Dijkstra's shortest path algorithm.

Dijkstra's Algorithm

This array only to help explanation!

Initialize:
- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

Main Loop
- while X ‡ V:



-need to grow x by one node

Main Loop cont'd:
- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

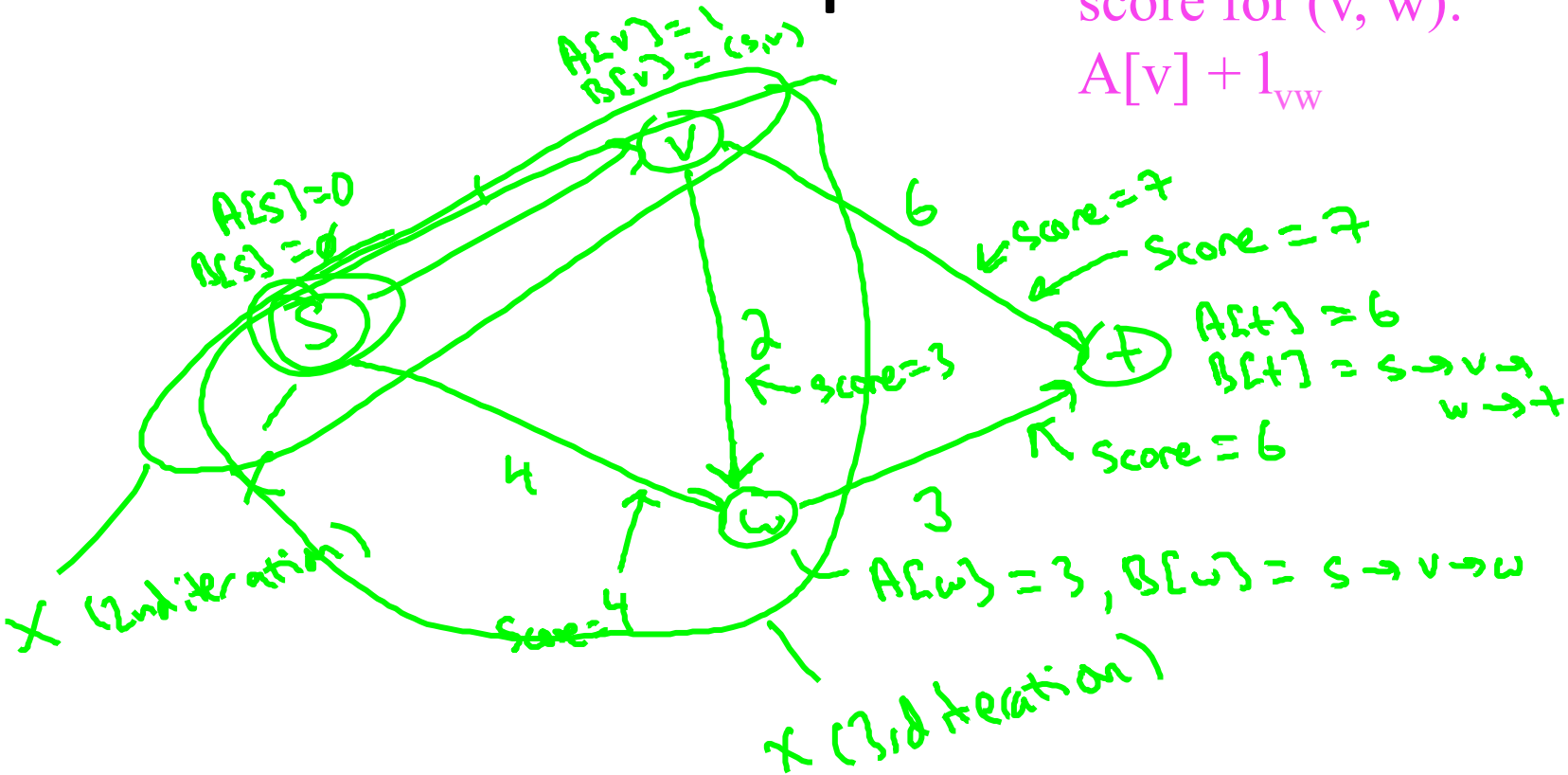
[call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

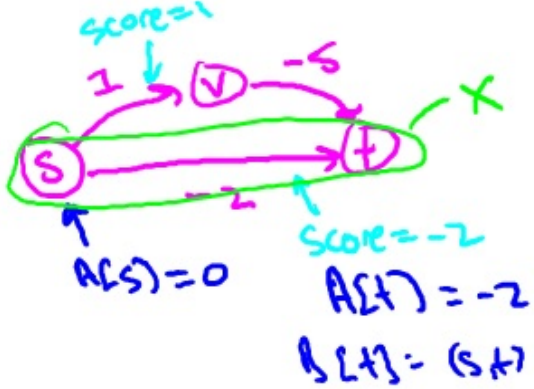
Example

Dijkstra's greedy score for (v, w): $A[v] + l_{vw}$



Non-Example

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)



Problem: doesn't preserve shortest paths !

Also: Dijkstra's algorithm incorrect on this graph !
(computes shortest s-t distance to be -2 rather than -4)

Data Structures

Introduction

Design and Analysis
of Algorithms I

Data Structures

Point : organize data so that it can be accessed quickly and usefully.

Examples : lists, stacks, queues, heaps, search trees, hashtables, bloom filters, union-find, etc.

Why so Many ? : different data structures support different sets of operations => suitable for different types of tasks.

Rule of Thumb : choose the “minimal” data structure that supports all the operations that you need.

Tim Roughgarden

Taking It To The Next Level

Level 0 - “what’s a data structure ?”

Level 1 - cocktail party-level literacy

Level 2 - “this problem calls out for a heap”

Level 3 - “I only use data structures that I create myself”

Tim Roughgarden

Data Structures

Heaps and Their Applications

Design and Analysis of Algorithms I

Heap: Supported Operations

- A container for objects that have keys
- Employer records, network edges, events, etc.

Insert: add a new object to a heap.

Running time : O(log(n))

Equally well, EXTRACT MAX

Extract-Min: remove an object in heap with a minimum key value. [ties broken arbitrarily]

Running time : O(log n) [n = # of objects in heap]

Also : HEAPIFY (n batched Inserts in O(n) time), DELETE(O(log(n)) time)

Application: Sorting

Canonical use of heap : fast way to do repeated minimum computations.

Example : SelectionSort ~$\theta(n)$ linear scans, $\theta(n^2)$ runtime on array of length n

Heap Sort : 1.) insert all n array elements into a heap
2.) Extract-Min to pluck out elements in sorted order

Running Time = 2n heap operations = O(nlog(n)) time.
=> optimal for a "comparison-based" sorting algorithm!

Application: Event Manager

"Priority Queue" – synonym for a heap.

Example : simulation (e.g., for a video game)

-Objects = event records [Action/update to occur at given time in the future]

- Key = time event scheduled to occur

- Extract-Min => yields the next scheduled event

Application: Median Maintenence

I give you : a sequence x1,…,xn of numbers, one-by-one.

You tell me : at each time step i, the median of {x1,….,xi}.

Constraint : use O(log(i)) time at each step i.

Solution : maintain heaps H_{Low} : supports Extract Max
H_{High} : supports Extract Min

Key Idea : maintain invariant that ~ i/2 smallest (largest) elements in H_{Low} (H_{High})

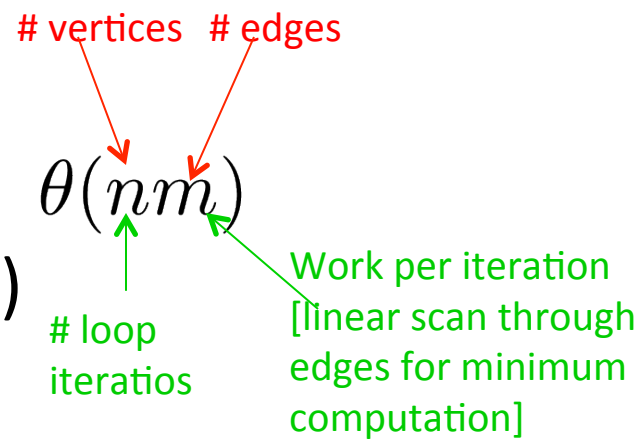
You Check : 1.) can maintain invariant with O(log(i)) work
2.) given invariant, can compute median in O(log(i)) work

Application: Speeding Up Dijkstra

Dijkstra's Shortest-Path Algorithm

-Naïve implementation => runtime = $\theta(nm)$

- with heaps => runtime = O(m log(n))



Greedy Algorithms

Introduction

Algorithms: Design and Analysis, Part II

Algorithm Design Paradigms

Algorithm Design: No single "silver bullet" for solving problems.

Design Paradigms:

- Divide & conquer (see Part I)
- Randomized algorithms (touched in Part I)
- Greedy algorithms (next)
- Dynamic programming (later in Part II)

Greedy Algorithms

"Definition": Iteratively make "myopic" decisions, hope everything works out at the end.

Example: Dijkstra's shortest path algorithm (from Part I)
- Processed each destination once, irrevocably.

Contrast with Divide & Conquer

1. Easy to propose multiple greedy algorithms for many problems.

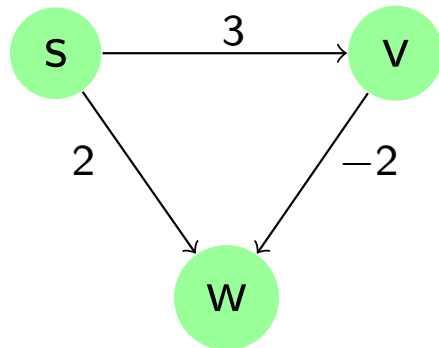
2. Easy running time analysis.
   (Contrast with Master method etc.)

3. Hard to establish correctness.
   (Contrast with straightforward inductive correctness proofs.)

DANGER: Most greedy algorithms are NOT correct. (Even if your intuition says otherwise!)

In(correctness)

Example: Dijkstra's algorithm with negative edge lengths. What does the algorithm compute as the length of a shortest s-w path, and what is the correct answer?



A) 2 and 2 C) 1 and 2

B) 2 and 0 D) 2 and 1

Proofs of Correctness

Method 1: Induction. ("greedy stays ahead")

Example: Correctness proof for Dijkstra's algorithm. (See Part I.)

Method 2: "Exchange argument".

Example: Coming right up!

Method 3: Whatever works!

Greedy Algorithms

Application: Optimal Caching

Algorithms: Design and Analysis, Part II

The Caching Problem

Small fast memory (the cache).

Big slow memory.

Process sequence of "page requests".

On a "fault" (that is, a cache miss), need to evict something from cache to make room – but what?

Example

Cache: | ~~a~~ | ~~b~~ | c | d |

e f

Request sequence: c d e f a b

$\Rightarrow$ 4 page faults
- 2 were inevitable (e & f)
- 2 consequences of poor eviction choices (should have evicted c & d instead of a & b)

The Optimal Caching Algorithm

Theorem: [Bélády 1960s] The "furthest-in-future" algorithm is optimal (i.e., minimizes the number of cache misses).

Why useful?

1. Serves as guideline for practical algorithms (e.g., Least Recently Used (LRU) should do well provided data exhibits locality of reference).
2. Serves as idealized benchmark for caching algorithms.

Proof: Tricky exchange argument. Open question: Find a simple proof!

Greedy Algorithms

A Scheduling Application: Problem Definition

Algorithms: Design and Analysis, Part II

A Scheduling Problem

Setup:

- One shared resource (e.g., a processor).
- Many "jobs" to do (e.g., processes).

Question: In what order should we sequence the jobs?

Assume: Each job has a:

- weight w_j ("priority")
- length l_J

Completion Times

Definition: The completion time C_j of job j = Sum of job lengths up to and including j.

Example: 3 jobs, $l_1 = 1, l_2 = 2, l_3 = 3$.

Schedule:

Question: What is C_1, C_2, C_3?

A) $1, 2, 3$

B) $3, 5, 6$

C) $1, 3, 6$

D) $1, 4, 6$

The Objective Function

Goal: Minimize the weighted sum of completion times: $\min \sum_{j=1}^{n} w_j C_j$.

Back to example: If $w_1 = 3, w_2 = 2, w_3 = 1$, this sum is $3 \cdot 1 + 2 \cdot 3 + 1 \cdot 6 = 15$.

Greedy Algorithms

A Scheduling Application: The Algorithm

Algorithms: Design and Analysis, Part II

Intuition for Algorithm

Recall: Want to $\min \sum_{j=1}^{n} w_j$.

Goal: Devise correct greedy algorithm.

Question:

1. With equal lengths, schedule larger or smaller-weight jobs earlier?
2. With equal weights, schedule shorter or longer jobs earlier?

A) Larger/shorter

C) Larger/longer

B) Smaller/shorter

D) Smaller/longer

Resolving Conflicting Advice

Question: What if $w_i > w_j$ but $l_i > l_j$?

Idea: Assign "scores" to jobs that are:

- inscreasing in weight
- decreasing in length

Guess (1): Order jobs by decreasing value of $w_j - l_j$.

Guess (2): Order w_j / l_j.

Breaking a Greedy Algorithm

To distinguish (1) & (2): Find example where the two algorithms produce different outputs. (At least one will be incorrect.)
Example:

$l_1 = 5, w_1 = 3$ (longer ratio)

$l_1 = 2, w_1 = 1$ (larger difference)

Question: What is the sum of weighted completion times of algorithms (1) & (2) respectively?

A) 22 and 23 C) 17 and 17

B) 23 and 22 D) 17 and 11

Alg#1: | #2 | #1 | $\rightarrow 1 \cdot 2 + 3 \cdot 7 = 23$

Alg#2: | #1 | #2 | $\rightarrow 3 \cdot 5 + 1 \cdot 7 = 22$

The Story So Far

So: Alg#1 not (always) correct.

Claim: Alg#2 (order by decreasing ratio w_j/l_j's) is always correct.
[not obvious! - proof coming up next]

Running time: $O(n \log n)$. [just need to sort]

Greedy Algorithms

A Scheduling Application: Correctness Proof Part I

Algorithms: Design and Analysis, Part II

Correctness Claim

Claim: Algorithm #2 (order jobs according to decreasing ratios w_j/l_j) is always correct.

Proof: By an Exchange Argument.

Plan: Fix arbitrary input of n jobs. Will proceed by contradiction.
Let σ = greedy schedule, σ^* = optimal schedule. (With σ^* better than σ.)
Will produce schedule even better than σ^*, contradicting purported optimality of σ^*.

Correctness Proof

Assume: All w_j/l_j's distinct.

Assume: [Just by renaming jobs] $w_1/l_1 > w_2/l_2 > \ldots > w_n/l_n$.

Thus: Greedy schedule σ is just $1, 2, 3, \ldots, n$.

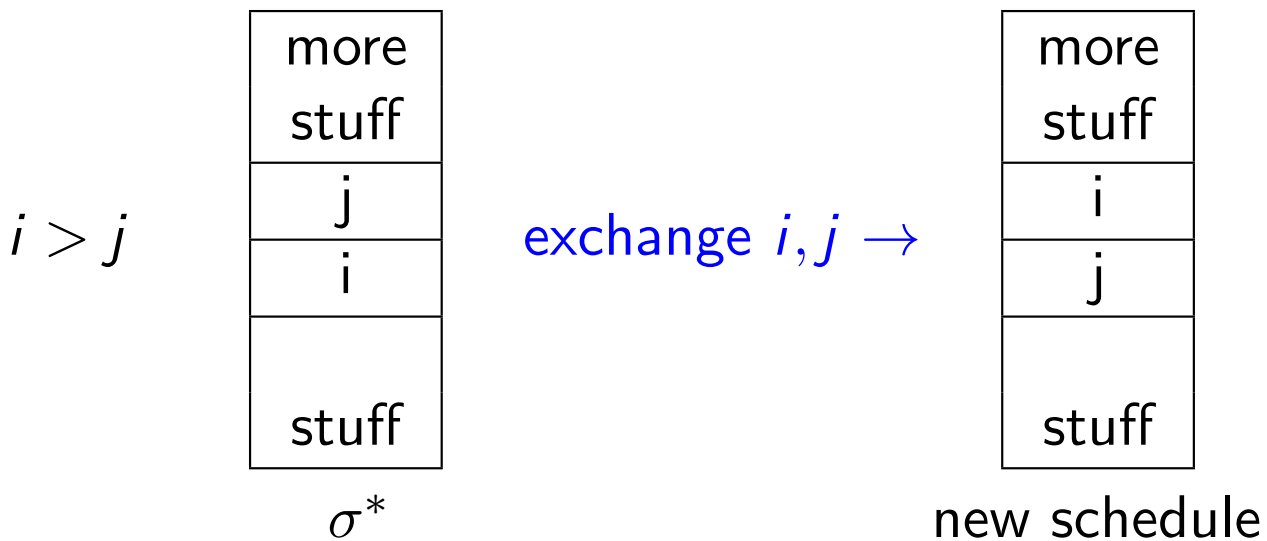
Thus: If optimal schedule $\sigma^* \neq \sigma$, then there are consecutive jobs i, j with $i > j$.
[Only schedule where indices always go up is $1, 2, 3, \ldots, n$]

Correctness Proof (con'd)

So far:

1. $w_1/l_1 > w_2/l_2 > \ldots > w_n/l_n$
2. In optimal σ^*, $\exists$ consecutive jobs i, j with $i > j$.

Thought experiment: Suppose we exchange order of $i\&j$ in σ^* (leaving other jobs unchanged):



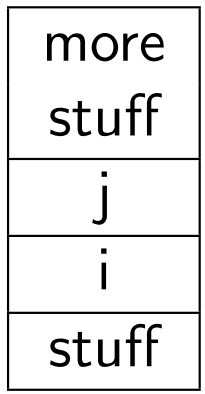
Greedy Algorithms

A Scheduling Application: Correctness Proof Part II

Algorithms: Design and Analysis, Part II

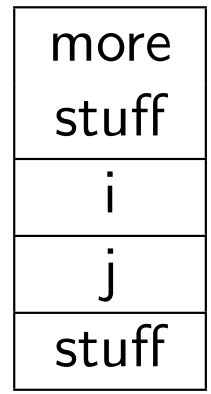
Cost-Benefit Analysis, Part I

$i > j$



σ^*

exchange $i, j \rightarrow$



new schedule

Question: What is the effect of this exchange on the completion time of (1) a job k other than i or j, (2) the job i, (3) the job j?

A) Not enough info/goes up/goes down

B) Not enough info/goes down/goes up

C) Unaffected/ goes up / goes down

D) Unaffected/goes down/goes up

by l_j

by l_i

Cost-Benefit Analysis, Part II

Upshot:

1. Cost of exchange $w_i l_j$. [C_i goes up by l_j]
2. Benefit of exchange is $w_j l_i$. [C_j goes down by l_i]

Note: $i > j \Rightarrow w_i/l_i < w_j/l_j \Rightarrow w_i l_j < w_j l_i \Rightarrow COST < BENEFIT$
$\Rightarrow$ Swap improves σ^*, contradicts optimality of σ^*.

QED!

Greedy Algorithms

A Scheduling Application: Handling Ties

Algorithms: Design and Analysis, Part II

Correctness Claim

Claim: Algorithm #2 (order jobs in nonincreasing order of ratio w_j/l_j) is always correct. [Even with ties]

New Proof Plan: Fix arbitrary input of n jobs. Let σ = greedy schedule, let σ^* = any other schedule.

Will show σ at least as good as $\sigma^* \Rightarrow$ Implies that greedy schedule is optimal.

Correctness Proof

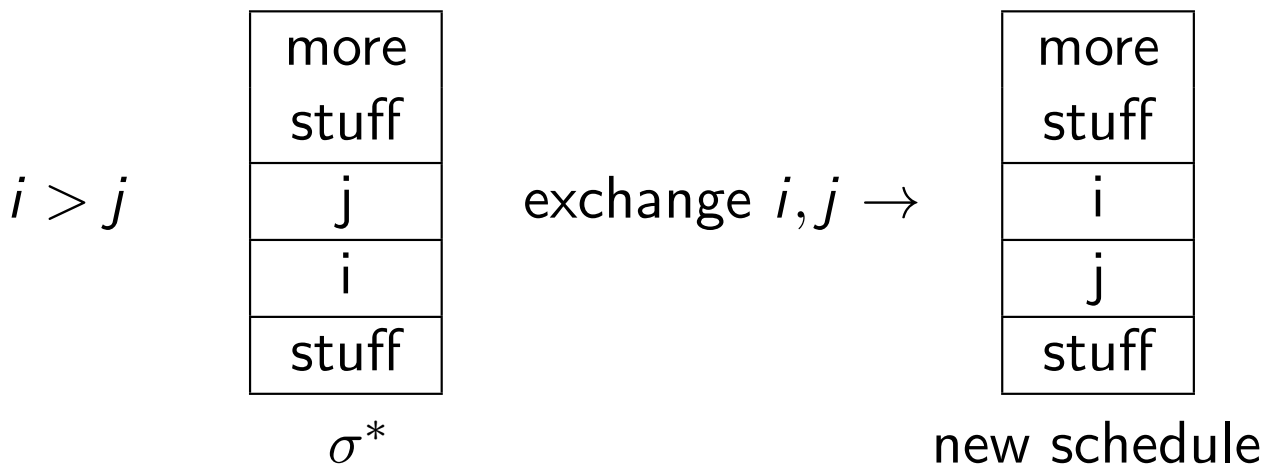
Assume: [Just by renaming jobs] Greedy schedule σ is just $1, 2, 3, \ldots, n$ (and so $w_1/l_1 \geq w_2/l_2 \geq \ldots \geq w_n/l_n$).

Consider arbitrary schedule σ^*. If $\sigma^* = \sigma$, done.

Else recall $\exists$ consecutive jobs i, j in σ^* with $i > j$. (From last time)

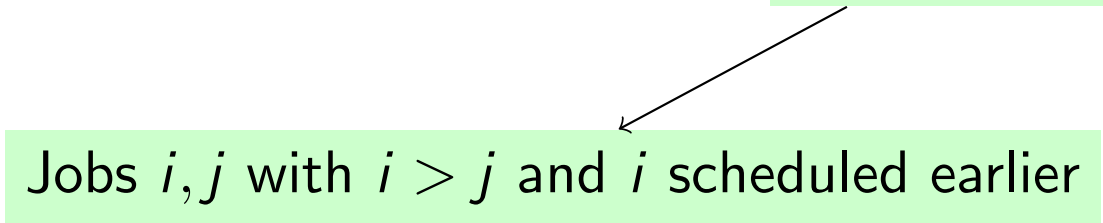
Note: $i > j \Rightarrow w_i/l_i \leq w_j/l_j \Rightarrow w_i l_j \leq w_j l_i$.

Recall: Exchanging $i\&j$ in σ^* has net benefit of $w_j l_i - w_i l_j \geq 0$.



Correctness Proof

Upshot: Exchanging an "adjacent inversion" like i, j only makes σ^* better, and it decreases the number of inverted pairs.



$\Rightarrow$ After at most $\binom{n}{2}$ such exchanges, can transform σ^* into σ.

$\Rightarrow$ σ at least as good as σ^*.

$\Rightarrow$ Greedy is optimal.

QED!

Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Problem Definition

Overview

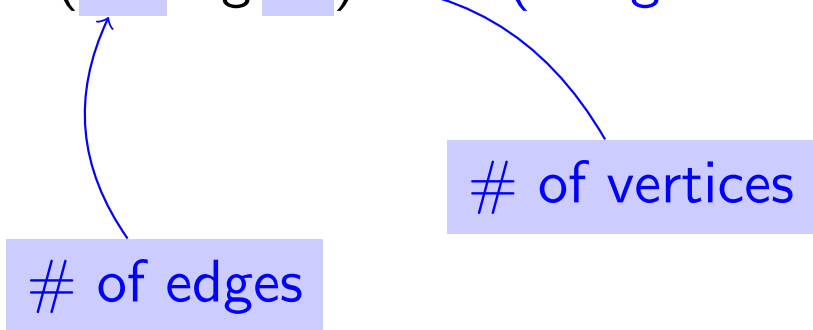
Informal Goal: Connect a bunch of points together as cheaply as possible.

Applications: Clustering (more later), networking.

Blazingly Fast Greedy Algorithms:

- Prim's Algorithm [1957; also Dijkstra 1959, Jarnik 1930]
- Kruskal's algorithm [1956]

$\Rightarrow O(m \log n)$ time (using suitable data structures)



Problem Definition

vertices edges

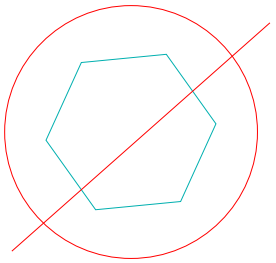
Input: Undirected graph $G = (V, E)$ and a cost c_e for each edge $e \in E$.

- Assume adjacency list representation (see Part I for details)
- OK if edge costs are negative

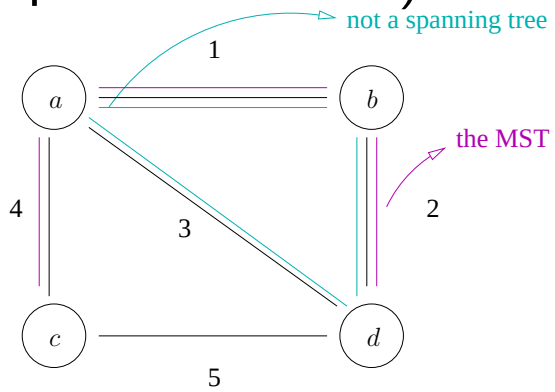
Output: minimum cost tree $T \subseteq E$ that spans all vertices.

i.e., sum of edge costs

I.e.: (1) T has no cycles, (2) the subgraph (V, T) is connected (i.e., contains path between each pair of vertices).



(disallowed)



Standing Assumptions

Assumption #1: Input graph G is connected.

- Else no spanning trees.
- Easy to check in preprocessing (e.g., depth-first search).

Assumption #2: Edge costs are distinct.

- Prim + Kruskal remain correct with ties (which can be broken arbitrarily).
- Correctness proof a bit more annoying (will skip).

Minimum Spanning Trees

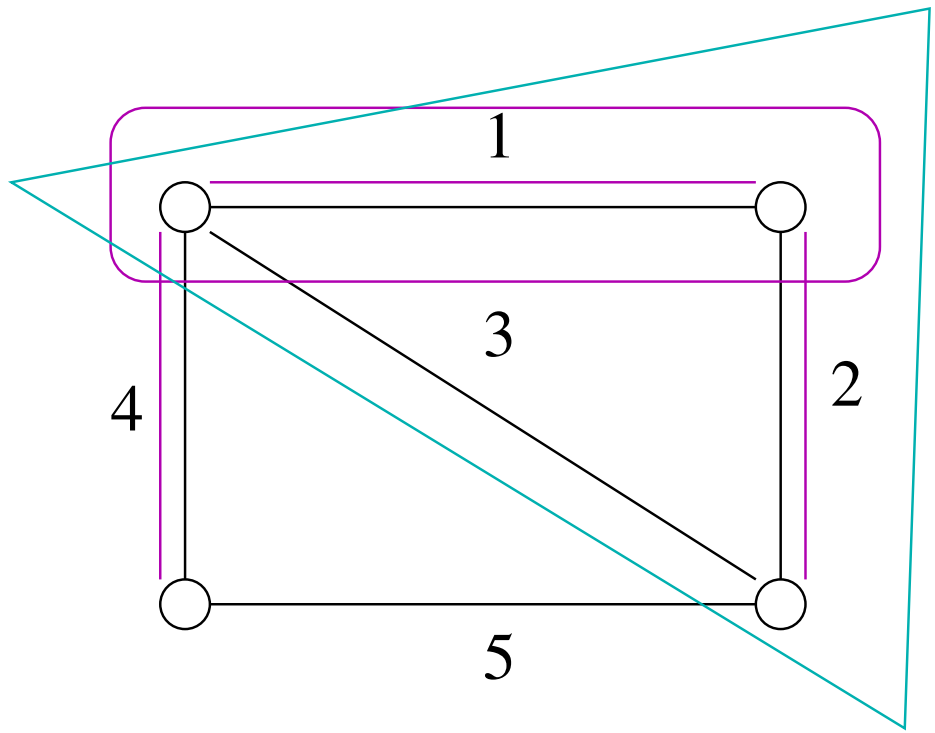
Algorithms: Design and Analysis, Part II

Prim's MST Algorithm

Example

[Purple edges = minimum spanning tree]

(Compare to Dijkstra's shortest-path algorithm)



Prim's MST Algorithm

- Initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: $X =$ vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T
  - Add v to X.

While loop: Increase # of spanned vertices in cheapest way possible.

Correctness of Prim's Algorithm

Theorem: Prim's algorithm always computes an MST.

Part I: Computes a spanning tree T^*.
[Will use basic properties of graphs and spanning trees] (Useful also in Kruskal's MST algorithm)

Part II: T^* is an MST.
[Will use the "Cut Property"] (Useful also in Kruskal's MST algorithm)

Later: Fast $[O(m \log n)]$ implementation using heaps.

Minimum Spanning Trees

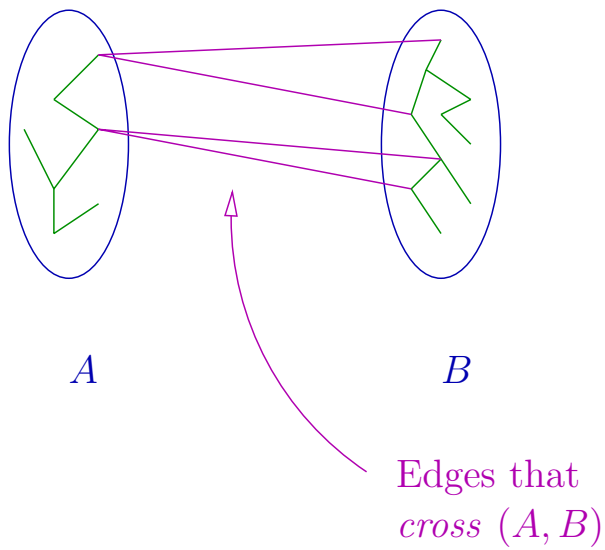
Algorithms: Design and Analysis, Part II

Correctness of Prim's Algorithm (Part I)

Cuts

Claim: Prim's algorithm outputs a spanning tree.

Definition: A cut of a graph $G = (V, E)$ is a partition of V into 2 non-empty sets.



Quiz on Cuts

Question: Roughly how many cuts does a graph with n vertices have?

A) n

B) n^2

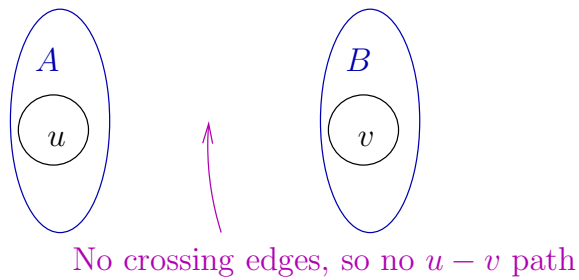
C) 2^n (for each vertex, choose whether in A or in B)

D) n^n

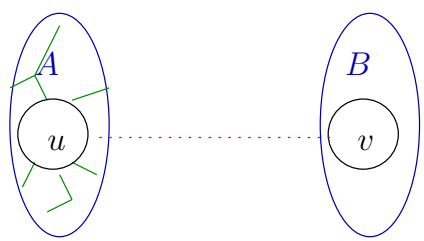
Empty Cut Lemma

Empty Cut Lemma: A graph is not connected $\iff$ $\exists$ cut (A, B) with no crossing edges.

Proof: ($\Leftarrow$) Assume the RHS. Pick any $u \in A$ and $v \in B$. Since no edges cross (A, B) there is no u, v path in G. $\Rightarrow$ G not connected.



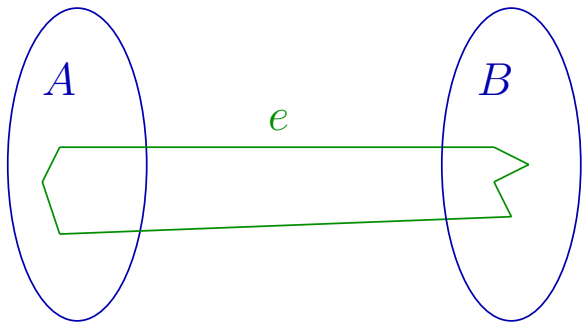
($\Rightarrow$) Assume the LHS. Suppose G has no $u - v$ path. Define
$A = \{$Vertices reachable from u in $G\}$ (u's connected component)
$B = \{$All other vertices$\}$ (all other connected components)
Note: No edges cross cut (A, B) (otherwise A would be bigger!)



QED!

Two Easy Facts

Double-Crossing Lemma: Suppose the cycle $C \subseteq E$ has an edge crossing the cut (A, B): then so does some other edge of C.

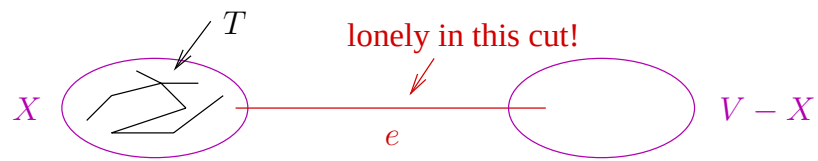


Lonely Cut Corollary: If e is the only edge crossing some cut (A, B), then it is not in any cycle. [If it were in a cycle, some other edge would have to cross the cut!]

Proof of Part I

Claim: Prim's algorithm outputs a spanning tree.
[Not claiming MST yet]

Proof: (1) Algorithm maintains invariant that T spans X
[straightforward induction - you check]



(2) Can't get stuck with $X \neq V$
[otherwise the cut $(X, V - X)$ must be empty; by Empty Cut Lemma input graph G is disconnected]

(3) No cycles ever get created in T. Why? Consider any iteration, with current sets X and T. Suppose e gets added.
Key point: e is the first edge crossing $(X, V - X)$ that gets added to $T \Rightarrow$ its addition can't create a cycle in T (by Lonely Cut Corollary). QED!

Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Correctness of Prim's Algorithm (Part II)

Correctness of Prim's Algorithm

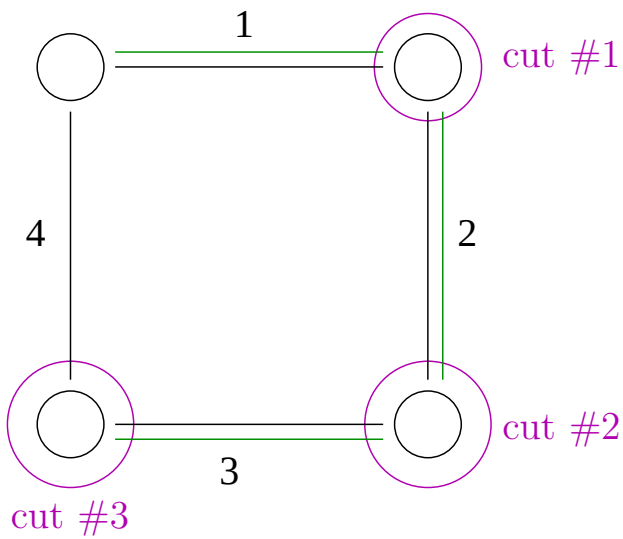
Theorem: Prim's algorithm always outputs a minimum-cost spanning tree.

Key Question: When is it "safe" to include an edge in the tree-so-far?

The Cut Property

CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.

Turns out MST is unique if edge costs are distinct



Cut Property Implies Correctness

Claim: Cut Property $\Rightarrow$ Prim's algorithm is correct.

Proof: By previous video, Prim's algorithm outputs a spanning tree T^*.

Key point: Every edge $e \in T^*$ is explicitly justified by the Cut Property.

$\Rightarrow$ T^* is a subset of the MST

$\Rightarrow$ Since T^* is already a spanning tree, it must be the MST

QED!

Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Proof of the Cut Property

The Cut Property

Assumption: Distinct edge costs.

CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.

Proof Plan

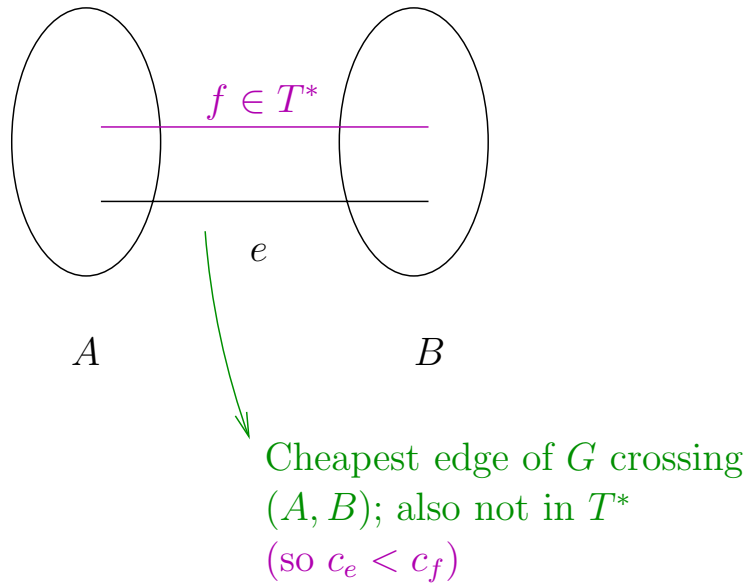
Will argue by contradiction, using an exchange argument.
[Compare to scheduling application]

Suppose there is an edge e that is the cheapest one crossing a cut (A, B), yet e is not in the MST T^*.

Idea: Exchange e with another edge in T^* to make it even cheaper (contradiction).

Question: Which edge to exchange e with?

Attempted Exchange



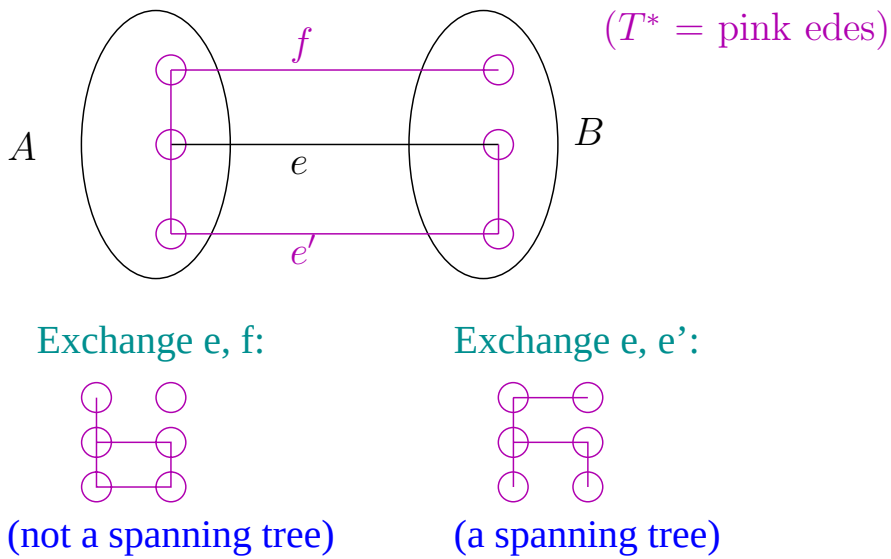
Note: Since T^* is connected, must construct an edge $f (\neq e)$ crossing (A, B).

Idea: Exchange e and f to get a spanning tree cheaper than T^* (contradiction).

Exchanging Edges

Question: Let T^* be a spanning tree of G, $e \notin T^*$, $f \in T^*$. Is $T^* \cup \{e\} - \{f\}$ a spanning tree of G?

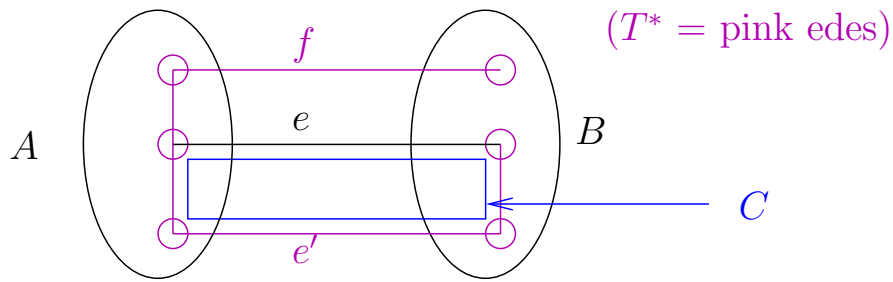
A) Yes always
B) No never
C) If e is the cheapest edge crossing some cut, then yes
D) Maybe, maybe not (depending on the choice of e and f)



Smart Exchanges

Hope: Can always find suitable edge e' so that exchange yields bona fide spanning tree of G.

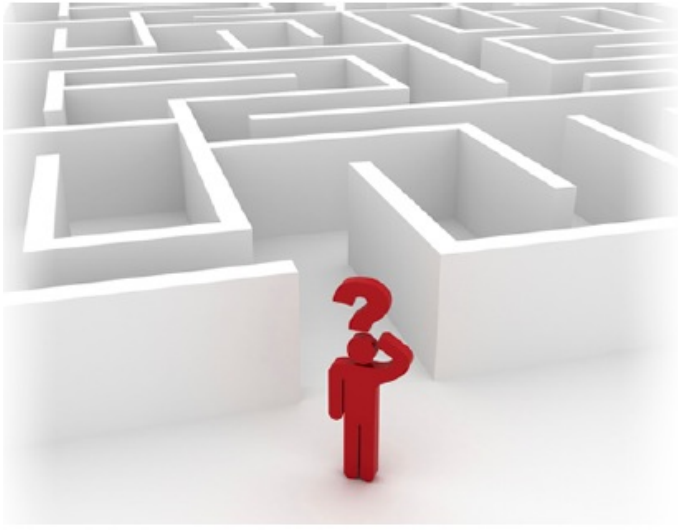
How? Let C = cycle created by adding e to T^*.



By the Double-Crossing Lemma: Some other edge e' of C [with $e' \neq e$ and $e' \in T^*$] crosses (A, B).

You check: $T = T^* \cup \{e\} - \{e'\}$ is also a spanning tree.

Since $c_e < c_{e'}$, T cheaper than purported MST T^*, contradiction.



Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Fast Implementation of Prim's Algorithm

Running Time of Prim's Algorithm

- Initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T, add v to X.

Running time of straightforward implementation:
- $O(n)$ iterations [where $n = \#$ of vertices]
- $O(m)$ time per iteration [where $m = \#$ of edges]

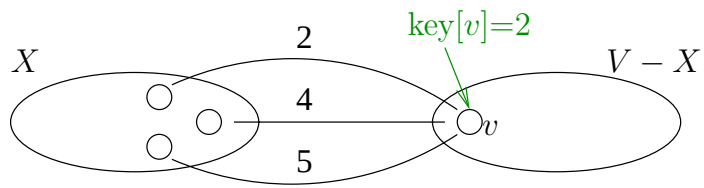
$\Rightarrow O(mn)$ time

BUT CAN WE DO BETTER?

Prim's Algorithm with Heaps

[Compare to fast implementation of Dijkstra's algorithm]
Invariant #1: Elements in heap = vertices of $V - X$.

Invariant #2: For $v \in V - X$, key[v] = cheapest edge (u, v) with $i \in X$ (or $+\infty$ if no such edges exist).



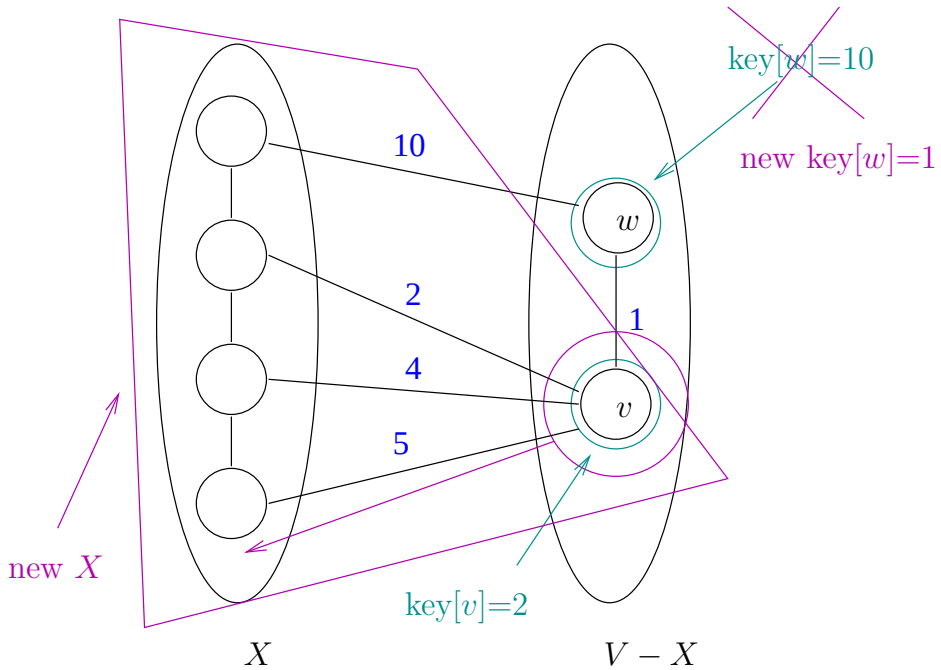
Check: Can initialize heap with $O(\ m + n \log n\ ) = O(m \log n)$ preprocessing.

To compare keys | $n - 1$ Inserts | $m \geq n - 1$ since G connected

Note: Given invariants, Extract-Min yields next vertex $v \notin X$ and edge (u, v) crossing $(X, V - X)$ to add to X and T, respectively.

Quiz: Issue with Invariant #2

Question: What is: (i) current value of key[v] (ii) current value of key[w] (iii) value of key[w] after one more iteration of Prim's algorithm?

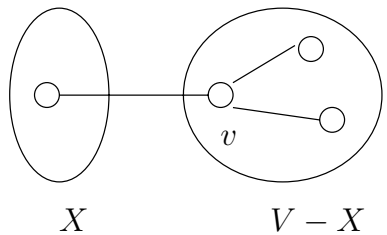


A) 11, 10, 4 C) 2, 10, 1

B) 2, 10, 10 D) 2, 10, 2

Maintaining Invariant #2

Issue: Might need to recompute some keys to maintain Invariant #2 after each Extract-Min.



Pseudocode: When v added to X:

- For each edge $(v, w) \in E$:
  - If $w \in V - X \rightarrow$ The only whose key might have changed
    (Update key if needed:)
    - Delete w from heap
    - Recompute key[w]:=min{key[w],c_{vw}}
    - Re-Insert into heap

Subtle point/exercise:

Think through book-keeping needed to pull this off

Running Time with Heaps

- Dominated by time required for heap operations

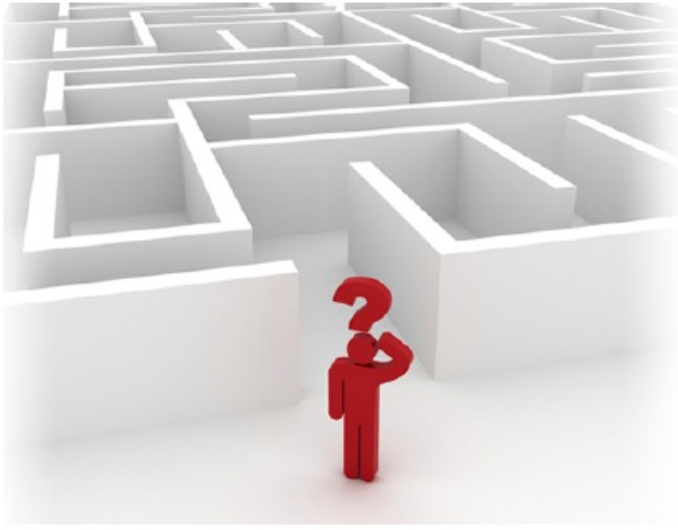
- $(n-1)$ Inserts during preprocessing

- $(n-1)$ Extract-Mins (one per iteration of while loop)

- Each edge (v, w) triggers one Delete/Insert combo
  [When its first endpoint is sucked into X]

$\Rightarrow$ $O(m)$ heap operations [Recall $m \geq n-1$ since G connected]

$\Rightarrow$ $O(m \log n)$ time [As fast as sorting!]



Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Fast Implementation of Prim's Algorithm

Running Time of Prim's Algorithm

- Initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T, add v to X.

Running time of straightforward implementation:
- $O(n)$ iterations [where $n = \#$ of vertices]
- $O(m)$ time per iteration [where $m = \#$ of edges]

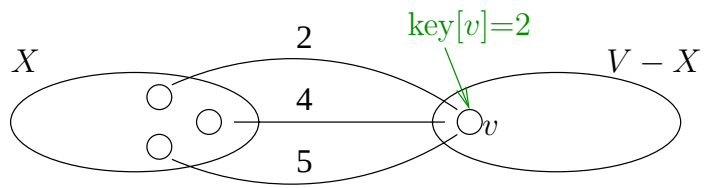
$\Rightarrow O(mn)$ time

BUT CAN WE DO BETTER?

Prim's Algorithm with Heaps

[Compare to fast implementation of Dijkstra's algorithm]
Invariant #1: Elements in heap = vertices of $V - X$.

Invariant #2: For $v \in V - X$, key[v] = cheapest edge (u, v) with $i \in X$ (or $+\infty$ if no such edges exist).



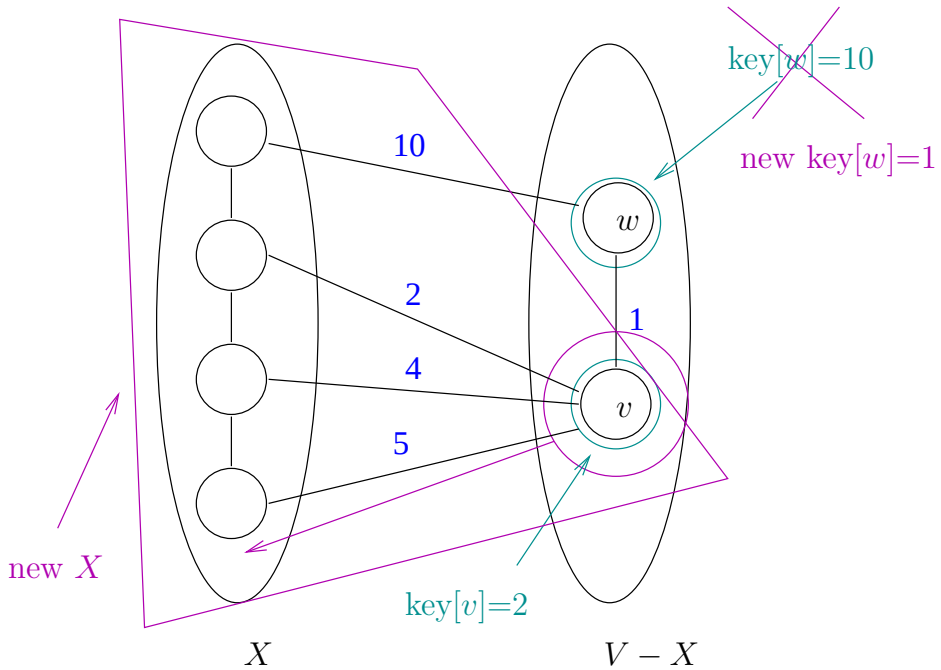
Check: Can initialize heap with $O(\ m + n \log n\ ) = O(m \log n)$ preprocessing.

To compare keys | $n - 1$ Inserts | $m \geq n - 1$ since G connected

Note: Given invariants, Extract-Min yields next vertex $v \notin X$ and edge (u, v) crossing $(X, V - X)$ to add to X and T, respectively.

Quiz: Issue with Invariant #2

Question: What is: (i) current value of key[v] (ii) current value of key[w] (iii) value of key[w] after one more iteration of Prim's algorithm?

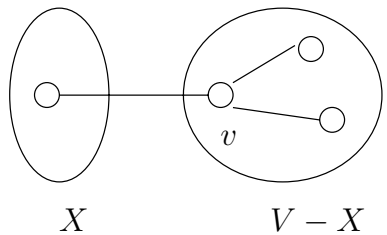


A) 11, 10, 4 C) 2, 10, 1

B) 2, 10, 10 D) 2, 10, 2

Maintaining Invariant #2

Issue: Might need to recompute some keys to maintain Invariant #2 after each Extract-Min.



Pseudocode: When v added to X:

- For each edge $(v, w) \in E$:
  - If $w \in V - X \rightarrow$ The only whose key might have changed
    (Update key if needed:)
    - Delete w from heap
    - Recompute key[w]:=min{key[w],c_{vw}}
    - Re-Insert into heap

Subtle point/exercise:

Think through book-keeping needed to pull this off

Running Time with Heaps

- Dominated by time required for heap operations

- $(n-1)$ Inserts during preprocessing

- $(n-1)$ Extract-Mins (one per iteration of while loop)

- Each edge (v, w) triggers one Delete/Insert combo
  [When its first endpoint is sucked into X]

$\Rightarrow$ $O(m)$ heap operations [Recall $m \geq n-1$ since G connected]

$\Rightarrow$ $O(m \log n)$ time [As fast as sorting!]