OLD CS 473: Fundamental Algorithms

May 5, 2015

Contents

Chapter 1

Administrivia, Introduction, Graph basics and DFS

OLD CS 473: Fundamental Algorithms, Spring 2015
January 20, 2015

1.0.0.1 The word "algorithm" comes from...

Muhammad ibn Musa al-Khwarizmi
780-850 AD
The word "algebra" is taken from the title of one of his books.

1.1 Administrivia

1.1.0.2 Online resources

(A) **Webpage:** `http://courses.engr.illinois.edu/cs473/sp2015/`
General information, homeworks, etc.
(B) **Moodle:** `https://learn.illinois.edu/course/view.php?id=10239`
Quizzes, solutions to homeworks.
(C) **Online questions/announcements:** Piazza
`http://piazza.com/illinois/spring2015/cs473/home`
Online discussions, etc.

1.1.0.3 Textbooks

(A) **Prerequisites:** CS 173 (discrete math), CS 225 (data structures) and CS 373 (theory of computation)
(B) **Recommended books:**
   (A) Algorithms by Dasgupta, Papadimitriou & Vazirani.
   Available online for free!
   (B) Algorithm Design by Kleinberg & Tardos
(C) **Lecture notes:** Available on the web-page after every class.

(D) **Additional References**
   (A) Previous class notes of Jeff Erickson, and the instructor.
   (B) Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein.
   (C) Computers and Intractability: Garey and Johnson.

1.1.0.4 Prerequisites

(A) **Asymptotic notation:** $O(), \Omega(), o()$.
(B) **Discrete Structures**: sets, functions, relations, equivalence classes, partial orders, trees, graphs
(C) **Logic**: predicate logic, boolean algebra
(D) **Proofs**: **by induction**, by contradiction
(E) **Basic sums and recurrences**: sum of a geometric series, unrolling of recurrences, basic calculus
(F) **Data Structures**: arrays, multi-dimensional arrays, linked lists, trees, balanced search trees, heaps
(G) **Abstract Data Types**: lists, stacks, queues, dictionaries, priority queues
(H) **Algorithms:** sorting (merge, quick, insertion), pre/post/in order traversal of trees, depth/breadth first search of trees (maybe graphs)
(I) **Basic analysis of algorithms**: loops and nested loops, deriving recurrences from a recursive program
(J) **Concepts from Theory of Computation**: languages, automata, Turing machine, undecidability, non-determinism
(K) **Programming**: in some general purpose language
(L) **Elementary Discrete Probability**: event, random variable, independence
(M) **Mathematical maturity**

1.1.0.5 Homeworks

(A) One quiz every week: Due by midnight on Sunday.
(B) One homework every week: Assigned on Tuesday and due the following Monday at noon.
(C) Submit in homework box in the basement.
(D) Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).
   (A) Short quiz-style questions are to be answered individually on *Moodle*.
(E) Groups can be changed a *few* times only
(F) Unlike previous years no *oral* homework this semester due to large enrollment.

1.1.0.6 More on Homeworks

(A) No extensions or late homeworks accepted.
(B) To compensate, the homework with the least score will be dropped in calculating the homework average.
(C) **Important:** Read homework faq/instructions on website.

1.1.0.7 Advice

(A) Attend lectures, please ask plenty of questions.
(B) Clickers...
(C) Attend discussion sessions.
(D) Don't skip homework and don't copy homework solutions.
(E) Study regularly and keep up with the course.
(F) Ask for help promptly. Make use of office hours.

1.1.0.8 Homeworks

(A) HW 0 is posted on the class website. Quiz 0 available
(B) Quiz 0 due by Sunday January 25 midnight
   HW 0 due on Monday, January 26 at noon.

(C) HW 0 to be submitted individually.

1.2 Course Goals and Overview

1.2.0.9 Topics

(A) Some fundamental algorithms
(B) Broadly applicable techniques in algorithm design
  (A) Understanding problem structure
  (B) Brute force enumeration and backtrack search
  (C) Reductions
  (D) Recursion
    (A) Divide and Conquer
    (B) Dynamic Programming
  (E) Greedy methods
  (F) Network Flows and Linear/Integer Programming (optional)
(C) Analysis techniques
  (A) Correctness of algorithms via induction and other methods
  (B) Recurrences
  (C) Amortization and elementary potential functions
(D) Polynomial-time Reductions, NP-Completeness, Heuristics

1.2.0.10 Goals

(A) Algorithmic thinking
(B) Learn/remember some basic tricks, algorithms, problems, ideas
(C) Understand/appreciate limits of computation (intractability)
(D) Appreciate the importance of algorithms in computer science and beyond (engineering, mathematics, natural sciences, social sciences, ...)
(E) Have fun!!!

1.3 Some Algorithmic Problems in the Real World

1.3.0.11 Shortest Paths

1.3.0.12 Shortest Paths - Paris to Berlin

1.3.0.13 Digital Information: Compression and Coding

Compression: reduce size for storage and transmission
Coding: add redundancy to protect against errors in storage and transmission

Efficient algorithms for compression/coding and decompressing/decoding part of most modern gadgets (computers, phones, music/video players ...)

1.3.1 Search and Indexing

1.3.1.1 String Matching and Link Analysis

(A) Web search: Google, Yahoo!, Microsoft, Ask, ...
(B) Text search: Text editors (Emacs, Word, Browsers, ...)
(C) Regular expression search: grep, egrep, emacs, Perl, Awk, compilers

1.3.1.2 Public-Key Cryptography

Foundation of Electronic Commerce

RSA Crypto-system: generate key $n = pq$ where p, q are *primes*
Primality: Given a number N, check if N is a prime or composite.

Factoring: Given a composite number N, find a non-trivial factor

1.3.1.3 Programming: Parsing and Debugging

[godavari: /temp/test] chekuri % gcc main.c

Parsing: Is main.c a syntactically valid C program?
Debugging: Will main.c go into an infinite loop on some input?

Easier problem ??? Will main.c halt on the specific input 10?

1.3.1.4 Optimization

Find the cheapest of most profitable way to do things
(A) Airline schedules - AA, Delta, ...
(B) Vehicle routing - trucking and transportation (UPS, FedEx, Union Pacific, ...)
(C) Network Design - AT&T, Sprint, Level3 ...
Linear and Integer programming problems

1.4 Algorithm Design

1.4.0.5 Important Ingredients in Algorithm Design

(A) What is the problem (really)?
  (A) What is the input? How is it represented?
  (B) What is the output?
(B) What is the model of computation? What basic operations are allowed?
(C) Algorithm design
(D) Analysis of correctness, running time, space etc.
(E) Algorithmic engineering: evaluating and understanding of algorithm's performance in practice, performance tweaks, comparison with other algorithms etc. (Not covered in this course)

1.5 Primality Testing

1.5.0.6 Primality testing

Problem Given an integer $N > 0$, is N a prime?

```
SimpleAlgorithm:
    for i = 2 to ⌊√N⌋ do
        if i divides N then
            return ''COMPOSITE''
    return ''PRIME''
```

Correctness? If N is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$
Running time? $O(\sqrt{N})$ divisions? Sub-linear in input size! **Wrong!**

1.5.1 Primality testing

1.5.1.1 ...Polynomial means... in input size

How many bits to represent N in binary? $\lceil \log N \rceil$ bits.

Simple Algorithm takes $\sqrt{N} = 2^{(\log N)/2}$ time.

Exponential in the input size $n = \log N$.

(A) Modern cryptography: binary numbers with 128, 256, 512 bits.
(B) Simple Algorithm will take 2^{64}, 2^{128}, 2^{256} steps!
(C) Fastest computer today about 3 petaFlops/sec: 3×2^{50} floating point ops/sec.

Lesson: Pay attention to representation size in analyzing efficiency of algorithms. Especially in *number* problems.

1.5.1.2 Efficient algorithms

So, is there an *efficient/good/effective* algorithm for primality?

Question: What does efficiency mean?

In this class *efficiency* is broadly equated to *polynomial time*.

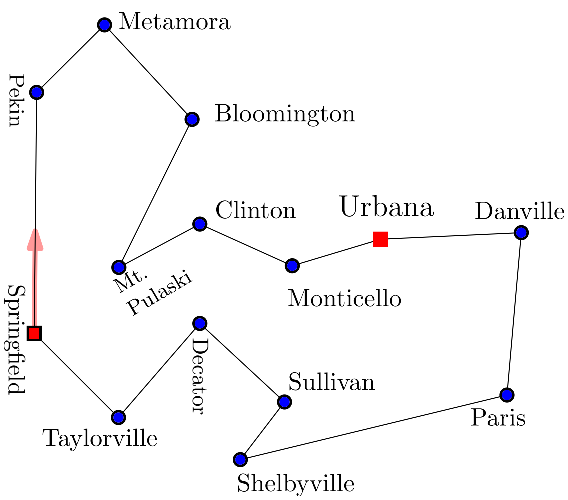
$O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where n is size of the input.

Why? Is n^{100} really efficient/practical? Etc.

Short answer: polynomial time is a robust, mathematically sound way to define efficiency. Has been useful for several decades.

1.5.2 TSP problem

1.5.2.1 Lincoln's tour



(A) Circuit court - ride through counties staying a few days in each town.
(B) Lincoln was a lawyer traveling with the Eighth Judicial Circuit.
(C) Picture: travel during 1850.
    (A) Very close to optimal tour.
    (B) Might have been optimal at the time..

1.5.3 Solving TSP by a Computer

1.5.3.1 Is it hard?

(A) n = number of cities.
(B) n^2: size of input.
(C) Number of possible solutions is

$$n * (n-1) * (n-2) * ... * 2 * 1 = n!.$$

(D) $n!$ grows very quickly as n grows.
$n = 10$: $n! \approx 3628800$
$n = 50$: $n! \approx 3 * 10^{64}$
$n = 100$: $n! \approx 9 * 10^{157}$

1.5.4 Solving TSP by a Computer

1.5.4.1 Fastest computer...

(A) Fastest super computer can do (roughly)

$$2.5 * 10^{15}$$

operations a second.
(B) Assume: computer checks $2.5 * 10^{15}$ solutions every second, then...
  (A) $n = 20 \implies$ 2 hours.
  (B) $n = 25 \implies$ 200 years.
  (C) $n = 37 \implies 2 * 10^{20}$ years!!!

1.5.5 What is a good algorithm?

1.5.5.1 Running time...

Input size	n^2 ops	n^3 ops	n^4 ops	$n!$ ops
5	0 secs	0 secs	0 secs	0 secs
20	0 secs	0 secs	0 secs	16 mins
30	0 secs	0 secs	0 secs	$3 \cdot 10^9$ years
100	0 secs	0 secs	0 secs	never
8000	0 secs	0 secs	1 secs	never
16000	0 secs	0 secs	26 secs	never
32000	0 secs	0 secs	6 mins	never
64000	0 secs	0 secs	111 mins	never
200,000	0 secs	3 secs	7 days	never
2,000,000	0 secs	53 mins	202.943 years	never
10^8	4 secs	12.6839 years	10^9 years	never
10^9	6 mins	12683.9 years	10^{13} years	never

1.5.6 What is a good algorithm?

1.5.6.1 Running time...

"No, Thursday's out. How about never—is never good for you?"

1.5.7 Primality

1.5.7.1 Primes is in P!

Theorem 1.5.1 (Agrawal-Kayal-Saxena'02). *There is a polynomial time algorithm for primality.*

First polynomial time algorithm for testing primality. Running time is $O(\log^{12} N)$ further improved to about $O(\log^6 N)$ by others. In terms of input size $n = \log N$, time is $O(n^6)$.

Breakthrough announced in August 2002. Three days later announced in New York Times. Only 9 pages!

Neeraj Kayal and Nitin Saxena were undergraduates at IIT-Kanpur!

1.5.7.2 What about before 2002?

Primality testing a key part of cryptography. What was the algorithm being used before 2002?

Miller-Rabin *randomized* algorithm:

(A) runs in polynomial time: $O(\log^3 N)$ time
(B) if N is prime correctly says "yes".
(C) if N is composite it says "yes" with probability at most $1/2^{100}$ (can be reduced further at the expense of more running time).

Based on Fermat's little theorem and some basic number theory.

1.5.8 Factoring

1.5.8.1 Factoring

(A) Modern public-key cryptography based on RSA (Rivest-Shamir-Adelman) system.
(B) Relies on the difficulty of factoring a composite number into its prime factors.
(C) There is a polynomial time algorithm that decides whether a given number N is prime or not (hence composite or not) but no known polynomial time algorithm to factor a given number.

Lesson Intractability can be useful!

1.5.8.2 Digression: decision, search and optimization

Three variants of problems.

(A) **Decision problem**: answer is yes or no.
Example: Given integer N, is it a composite number?
(B) **Search problem**: answer is a feasible solution if it exists.
Example: Given integer N, if N is composite output *a* non-trivial factor p of N.
(C) **Optimization problem**: answer is the *best* feasible solution (if one exists).
Example: Given integer N, if N is composite output the *smallest* non-trivial factor p of N.

For a given underlying problem:

$$\text{Optimization} \geq \text{Search} \geq \text{Decision}$$

1.5.8.3 Quantum Computing

Theorem 1.5.2 (Shor'1994). *There is a polynomial time algorithm for factoring on a* quantum *computer.*

RSA and current commercial cryptographic systems can be broken if a quantum computer can be built!

Lesson Pay attention to the model of computation.

1.5.8.4 Problems and Algorithms

Many many different problems.

(A) Adding two numbers: efficient and simple algorithm
(B) Sorting: efficient and not too difficult to design algorithm
(C) Primality testing: simple and basic problem, took a long time to find efficient algorithm
(D) Factoring: no efficient algorithm known.
(E) Halting problem: important problem in practice, undecidable!

1.6 Multiplication

1.6.0.5 Multiplying Numbers

Problem Given two n-digit numbers x and y, compute their product.

Grade School Multiplication Compute “partial product” by multiplying each digit of y with x and adding the partial products.

$$\begin{array}{r} 3141 \\ \times 2718 \\ \hline 25128 \\ 3141 \\ 21987 \\ 6282 \\ \hline 8537238 \end{array}$$

1.6.0.6 Time analysis of grade school multiplication

(A) Each partial product: $\Theta(n)$ time
(B) Number of partial products: $\leq n$
(C) Adding partial products: n additions each $\Theta(n)$ (Why?)
(D) Total time: $\Theta(n^2)$
(E) Is there a faster way?

1.6.0.7 Fast Multiplication

Best known algorithm: $O(n \log n \cdot 2^{O(\log^* n)})$ time [Furer 2008]
Previous best time: $O(n \log n \log \log n)$ [Schonhage-Strassen 1971]

Conjecture: there exists and $O(n \log n)$ time algorithm

We don’t fully understand multiplication!
Computation and algorithm design is non-trivial!

1.6.0.8 Course Approach

Algorithm design requires a mix of skill, experience, mathematical background/maturity and ingenuity.

Approach in this class and many others:
(A) Improve skills by showing various tools in the abstract and with concrete examples
(B) Improve experience by giving **many** problems to solve
(C) Motivate and inspire
(D) Creativity: you are on your own!

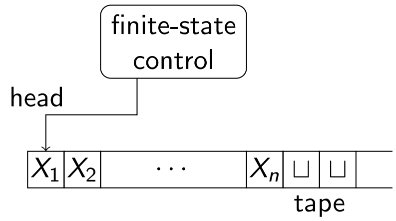
1.7 Model of Computation

1.7.0.9 What model of computation do we use?

Turing Machine?

1.7.0.10 Turing Machines: Recap

(A) Infinite tape
(B) Finite state control
(C) Input at beginning of tape
(D) Special tape letter "blank" $\sqcup$
(E) Head can move only one cell to left or right



1.7.0.11 Turing Machines

(A) Basic unit of data is a bit (or a single character from a finite alphabet)
(B) Algorithm is the finite control
(C) Time is number of steps/head moves

Pros and Cons:

(A) theoretically sound, robust and simple model that underpins computational complexity.
(B) polynomial time equivalent to any reasonable "real" computer: Church-Turing thesis
(C) too low-level and cumbersome, does not model actual computers for many realistic settings

1.7.0.12 "Real" Computers vs Turing Machines

How do "real" computers differ from TMs?

(A) random access to memory
(B) pointers
(C) arithmetic operations (addition, subtraction, multiplication, division) in constant time

How do they do it?

(A) basic data type is a word: currently 64 bits
(B) arithmetic on words are basic instructions of computer
(C) memory requirements assumed to be $\leq 2^{64}$ which allows for pointers and indirect addressing as well as random access

1.7.0.13 Unit-Cost RAM Model

Informal description:

(A) Basic data type is an integer/floating point number
(B) Numbers in input fit in a word
(C) Arithmetic/comparison operations on words take constant time
(D) Arrays allow random access (constant time to access $A[i]$)
(E) Pointer based data structures via storing addresses in a word

1.7.0.14 Example

Sorting: input is an array of n numbers

(A) input size is n (ignore the bits in each number),
(B) comparing two numbers takes $O(1)$ time,
(C) random access to array elements,

(D) addition of indices takes constant time,
(E) basic arithmetic operations take constant time,
(F) reading/writing one word from/to memory takes constant time.

We will usually not allow (or be careful about allowing):

(A) bitwise operations (and, or, xor, shift, etc).
(B) floor function.
(C) limit word size (usually assume unbounded word size).

1.7.0.15 Caveats of RAM Model

Unit-Cost RAM model is applicable in wide variety of settings in practice. However it is not a proper model in several important situations so one has to be careful.

(A) For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two n-digit numbers, primality etc.
(B) Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by 2^k where k is word length.
(C) Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very big numbers from initial numbers.

1.7.0.16 Models used in class

In this course:

(A) Assume unit-cost RAM by default.
(B) We will explicitly point out where unit-cost RAM is not applicable for the problem at hand.

1.8 Graph Basics

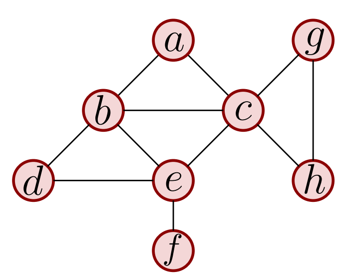
1.8.0.17 Why Graphs?

(A) Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc etc.
(B) Fundamental objects in Computer Science, Optimization, Combinatorics
(C) Many important and useful optimization problems are graph problems
(D) Graph theory: elegant, fun and deep mathematics

1.8.0.18 Graph

Definition 1.8.1. *An undirected (simple) graph $G = (V, E)$ is a 2-tuple:*

(A) V is a set of vertices (also referred to as nodes/points)
(B) E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.



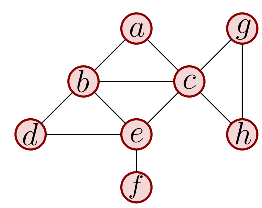
Example 1.8.2. *In figure,* $G = (V, E)$ *where* $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ *and*
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

1.8.0.19 Notation and Convention

Notation An edge in an undirected graphs is an *unordered* pair of nodes and hence it is a set. Conventionally we use (u, v) for $\{u, v\}$ when it is clear from the context that the graph is undirected.

(A) u and v are the **end points** of an edge $\{u, v\}$
(B) **Multi-graphs** allow
  (A) *loops* which are edges with the same node appearing as both end points
  (B) *multi-edges*: different edges between same pairs of nodes
(C) In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.

1.8.0.20 Graph Representation I



	a	b	c	d	e	f	g	h
a	0	1	1	0	0	0	0	0
b	1	0	1	1	1	0	0	0
c	1	1	0	0	1	0	1	1
d	0	1	0	0	1	0	0	0
e	0	1	1	1	0	1	0	0
f	0	0	0	0	1	0	0	0
g	0	0	1	0	0	0	0	1
h	0	0	1	0	0	0	1	0

Adjacency Matrix Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ adjacency matrix A where

(A) $A[i, j] = A[j, i] = 1$ if $\{i, j\} \in E$ and $A[i, j] = A[j, i] = 0$ if $\{i, j\} \notin E$.
(B) Advantage: can check if $\{i, j\} \in E$ in $O(1)$ time
(C) Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$

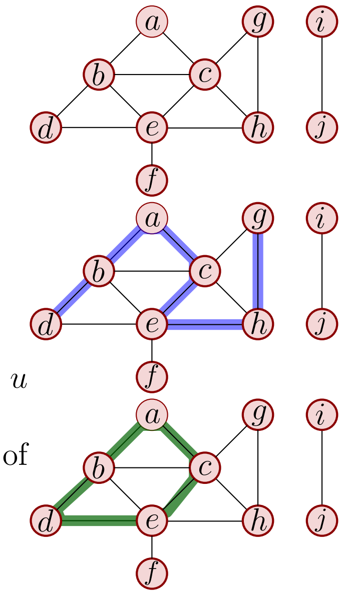
1.8.0.21 Graph Representation II

Adjacency Lists Represent $G = (V, E)$ with n vertices and m edges using adjacency lists:

(A) For each $u \in V$, $\mathrm{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u. Sometimes $\mathrm{Adj}(u)$ is the list of edges incident to u.
(B) Advantage: space is $O(m + n)$
(C) Disadvantage: cannot "easily" determine in $O(1)$ time whether $\{i, j\} \in E$
  (A) By sorting each list, one can achieve $O(\log n)$ time
  (B) By hashing "appropriately", one can achieve $O(1)$ time

Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

1.8.0.22 Connectivity



Given a graph $G = (V, E)$:

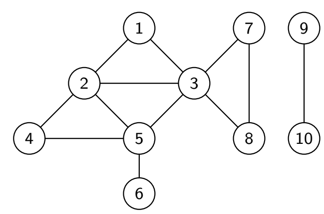
(A) ***path***: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$.
For $i = 1, \ldots, k-1$: $v_i v_{i+1} \in E$
length of path $= k - 1$.
The path is from v_1 to v_k

(B) ***cycle***: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$
$\forall i \quad v_i v_{i+1} \in E$ and $\{v_1, v_k\} \in E$.

(C) A vertex u is **connected** to v if there is a path from u to v.

(D) The ***connected component*** of u, $\mathrm{con}(u)$, is the set of all vertices connected to u.

1.8.0.23 Connectivity contd



Define a relation C on $V \times V$ as uCv if u is connected to v

(A) In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.

(B) Graph is **connected** if only one connected component.

1.8.0.24 Connectivity Problems

Algorithmic Problems

(A) Given graph G and nodes u and v, is u *connected* to v?

(B) Given G and node u, find all nodes that are connected to u.

(C) Find all connected components of G.

Can be accomplished in $O(m + n)$ time using **BFS** or **DFS**.

1.8.0.25 Basic Graph Search

Given $G = (V, E)$ and vertex $u \in V$:

```
Explore(u):
    Initialize S = {u}
    while there is an edge (x, y) with x ∈ S and y ∉ S do
        add y to S
```

Proposition 1.8.3. Explore(u) *terminates with* $S = con(u)$.

Running time: depends on implementation
(A) Breadth First Search (**BFS**): use **queue** data structure
(B) Depth First Search (**DFS**): use **stack** data structure
(C) Review CS 225 material!

1.9 DFS

1.9.1 DFS

1.9.1.1 Depth First Search

DFS: versatile graph exploration strategy. Hopcroft and Tarjan demonstrated the power of **DFS** to understand graph structure. **DFS** can be used to obtain linear time ($O(m + n)$) time algorithms for
(A) Finding cut-edges and cut-vertices of undirected graphs.
(B) Finding strong connected components of directed graphs.
(C) Linear time algorithm for testing whether a graph is planar.

1.9.1.2 DFS in Undirected Graphs

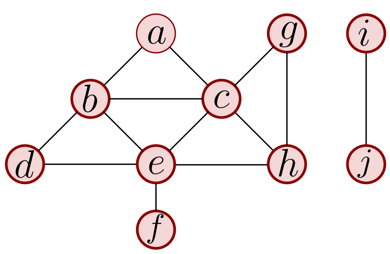
Recursive version.

```
DFS(G)
    Mark all nodes u as unvisited
    while there is an unvisited node u do
        DFS(u)
```

```
DFS(u)
    Mark u as visited
    for each edge (u,v) in Adj(u) do
        if v is not marked
            DFS(v)
```

Implemented using a global array `Mark` for all recursive calls.

1.9.1.3 Example



1.9.1.4 DFS Tree/Forest

```
DFS(G)
    Mark all nodes as unvisited
    T is set to ∅
    while ∃ unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    for uv in Ajd(u) do
        if v is not marked
            add uv to T
            DFS(v)
```

Edges classified into two types: $uv \in E$ is a

(A) **tree edge:** belongs to T

(B) **non-tree edge:** does not belong to T

1.9.1.5 Properties of DFS tree

Proposition 1.9.1. *(A) T is a forest*

(B) connected components of T are same as those of G*.*

(C) If $uv \in E$ is a non-tree edge then, in T, either:

(A) u is an ancestor of v, or

(B) v is an ancestor of u.

Question: Why are there no *cross-edges*?

1.9.1.6 DFS with Visit Times

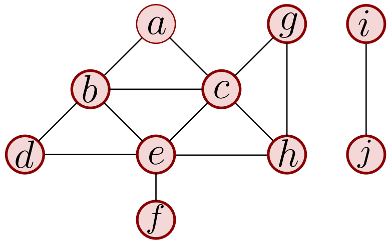
Keep track of when nodes are visited.

```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
    T is set to ∅
    time = 0
    while ∃unvisited u do
        DFS(u)
    Output T
```

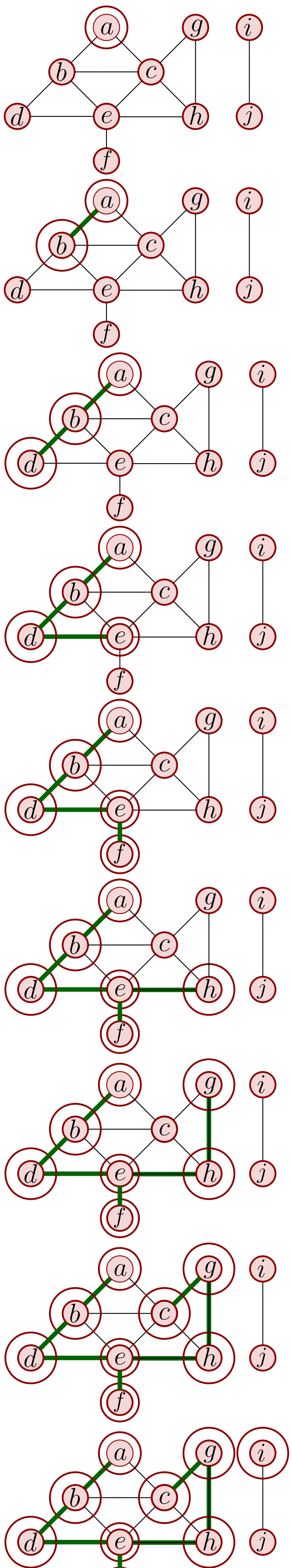
```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each uv in Out(u) do
        if v is not marked then
            add edge uv to T
            DFS(v)
    post(u) = ++time
```

1.9.1.7 Scratch space

1.9.1.8 Example: DFS with visit times



1.9.1.9 Example



a: [1, 16]
b: [2, 15]
d: [3, 14]
e: [4, 13]
f: [5, 6]
h: [7, 12]
g: [8, 11]
c: [9, 10]
i: [17, 20]
j: [18,19]

1.9.1.10 pre **and** post **numbers**

Node u is ***active*** in time interval $[\mathrm{pre}(u), \mathrm{post}(u)]$

Proposition 1.9.2. *For any two nodes u and v, the two intervals $[\mathrm{pre}(u), \mathrm{post}(u)]$ and $[\mathrm{pre}(v), \mathrm{post}(v)]$ are either disjoint or one is contained in the other.*

Proof: (A) Assume without loss of generality that $\mathrm{pre}(u) < \mathrm{pre}(v)$. Then v visited after u.

(B) If **DFS**(v) invoked before **DFS**(u) finished, $\mathrm{post}(u) > \mathrm{post}(v)$.

(C) If **DFS**(v) invoked after **DFS**(u) finished, $\mathrm{pre}(v) > \mathrm{post}(u)$. ∎

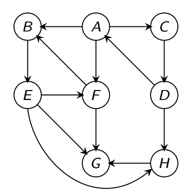
pre and post numbers useful in several applications of **DFS**- soon!

1.10 Directed Graphs and Decomposition

1.11 Introduction

1.11.0.11 Directed Graphs

Definition 1.11.1. *A directed graph $G = (V, E)$ consists of*
(A) set of vertices/nodes V and
(B) a set of edges/arcs $E \subseteq V \times V$.



(A) An edge is an *ordered* pair of vertices.
(B) Directed edge written as (u, v) or $(u \to v)$.
(C) $(u \to v)$ is different from $(v \to u)$.

1.11.0.12 Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:
(A) Road networks with one-way streets.
(B) Web-link graph: vertices are web-pages. Edge from page p to page p' if p has a link to p'. Web graphs used by Google with PageRank algorithm to rank pages.
(C) Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
(D) Program Analysis: functions/procedures are vertices and there is an edge from x to y if x calls y.

1.11.0.13 Representation

Graph $G = (V, E)$ with n vertices and m edges:

(A) ***Adjacency Matrix***: $n \times n$ *asymmetric* matrix A. $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ if $(u, v) \notin E$. $A[u, v]$ is not same as $A[v, u]$.
(B) ***Adjacency Lists***: for each node u, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from u.

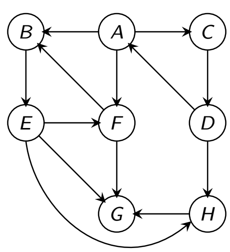
Default representation is adjacency lists.

1.11.0.14 Directed Connectivity

Given a graph $G = (V, E)$:
(A) A ***(directed) path*** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k-1$. The length of the path is $k-1$ and the path is from v_1 to v_k
(B) A ***cycle*** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \le i \le k-1$ and $(v_k, v_1) \in E$.
(C) A vertex u can **reach** v if there is a path from u to v. Alternatively v can be reached from u
(D) Let **rch**(u) be the set of all vertices reachable from u.

1.11.0.15 Connectivity contd

Asymmetricity: A can reach B but B cannot reach A



Questions:
(A) Is there a notion of connected components?
(B) How do we understand connectivity in directed graphs?

1.11.0.16 Connectivity and Strong Connected Components

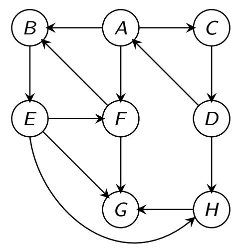
Definition 1.11.2. *Given a directed graph* G*, u is* **strongly connected** *to v if u can reach v* and *v can reach u. In other words $v \in rch(u)$ and $u \in rch(v)$.*

(A) Define relation C where uCv if u is (strongly) connected to v.

Proposition 1.11.3. *C is an equivalence relation $\implies$ reflexive, symmetric and transitive.*

(B) Equivalence classes of C: ***strong connected components*** G.
(C) They *partition* the vertices of G.
SCC(u): strongly connected component containing u.

1.11.0.17 Strongly Connected Components: Example



1.11.0.18 Problems on Directed Graph Connectivity

(A) Given G and nodes u and v, can u reach v?
(B) Given G and u, compute $\text{rch}(u)$.
(C) Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.
(D) Find the strongly connected component containing node u, that is $\text{SCC}(u)$.
(E) Is G strongly connected (a single strong component)?
(F) Compute *all* strongly connected components of G.

First four problems can be solve in $O(n+m)$ time by adapting **BFS**/**DFS** to directed graphs. The last one requires a clever **DFS** based algorithm.

1.12 DFS in Directed Graphs

1.12.0.19 DFS in Directed Graphs

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    while there is an unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each edge (u,v) in Out(u) do
        if v is not marked
            add edge (u,v) to T
            DFS(v)
    post(u) = ++time
```

1.12.0.20 DFS Properties

Generalizing ideas from undirected graphs:
(A) $DFS(u)$ outputs a directed out-tree T rooted at u
(B) A vertex v is in T if and only if $v \in \text{rch}(u)$
(C) For any two vertices x, y the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint are one is contained in the other.
(D) The running time of $DFS(u)$ is $O(k)$ where $k = \sum_{v \in \text{rch}(u)} |Adj(v)|$ plus the time to initialize the Mark array.

(E) **DFS**(G) takes $O(m+n)$ time. Edges in T form a disjoint collection of of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.

1.12.0.21 DFS Tree

Edges of G can be classified with respect to the **DFS** tree T as:

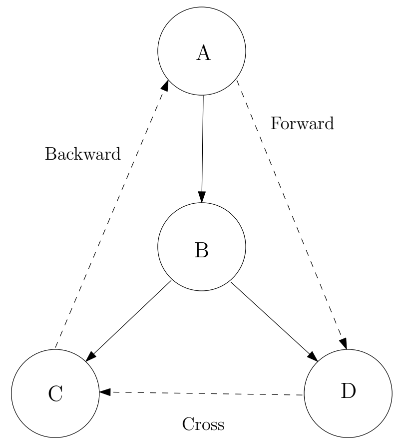
(A) ***Tree edges*** that belong to T

(B) A ***forward edge*** is a non-tree edges (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

(C) A ***backward edge*** is a non-tree edge (x, y) such that $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$.

(D) A ***cross edge*** is a non-tree edges (x, y) such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.

1.12.0.22 Types of Edges



1.12.0.23 Directed Graph Connectivity Problems

(A) Given G and nodes u and v, can u reach v?

(B) Given G and u, compute $\text{rch}(u)$.

(C) Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

(D) Find the strongly connected component containing node u, that is $\text{SCC}(u)$.

(E) Is G strongly connected (a single strong component)?

(F) Compute *all* strongly connected components of G.

1.13 Algorithms via DFS

1.13.0.24 Algorithms via DFS- I

(A) Given G and nodes u and v, can u reach v?

(B) Given G and u, compute $\text{rch}(u)$.

Use $DFS(G, u)$ to compute $\text{rch}(u)$ in $O(n+m)$ time.

1.13.0.25 Algorithms via DFS- II

(A) Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.

Definition 1.13.1 (Reverse graph.). *Given $G = (V, E)$, G^{rev} is the graph with edge directions reversed*
$G^{rev} = (V, E')$ where $E' = \{(y, x) \mid (x, y) \in E\}$

Compute $\text{rch}(u)$ in G^{rev}!

(A) **Correctness:** exercise

(B) **Running time:** $O(n+m)$ to obtain G^{rev} from G and $O(n+m)$ time to compute $\text{rch}(u)$ via **DFS**. If both $Out(v)$ and $In(v)$ are available at each v then no need to explicitly compute G^{rev}. Can do it $DFS(u)$ in G^{rev} implicitly.

1.13.0.26 Algorithms via DFS- III

$SC(G,u) = \{v \mid u \text{ is strongly connected to } v\}$

(A) Find the strongly connected component containing node u. That is, compute $\text{SCC}(G,u)$.
$\text{SCC}(G,u) = \text{rch}(G,u) \cap \text{rch}(G^{rev},u)$

Hence, $\text{SCC}(G,u)$ can be computed with two **DFS**es, one in G and the other in G^{rev}. Total $O(n+m)$ time.

1.13.0.27 Algorithms via DFS- IV

(A) Is G strongly connected?
Pick arbitrary vertex u. Check if $SC(G,u) = V$.

1.13.0.28 Algorithms via DFS- V

(A) Find *all* strongly connected components of G.

```
for each vertex u ∈ V do
    find SC(G,u)
```

Running time: $O(n(n+m))$.
Q: Can we do it in $O(n+m)$ time?

1.13.0.29 Reading and Homework 0

Chapters 1 from Dasgupta etal book, Chapters 1-3 from Kleinberg-Tardos book.

Proving algorithms correct - Jeff Erickson's notes (see link on website)

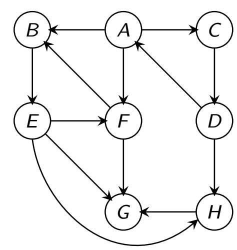
Chapter 2

DFS in Directed Graphs, Strong Connected Components, and DAGs

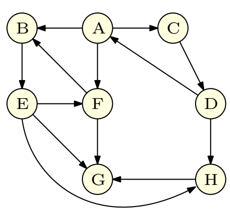
OLD CS 473: Fundamental Algorithms, Spring 2015
January 22, 2015

2.0.0.30 Strong Connected Components (SCCs)

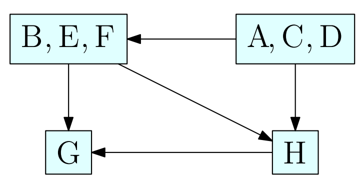
Algorithmic Problem Find all SCCs of a given directed graph. Previous lecture:
Saw an $O(n \cdot (n+m))$ time algorithm.
This lecture: $O(n+m)$ time algorithm.



2.0.0.31 Graph of SCCs



Graph G



G^{SCC}: Graph of SCCs

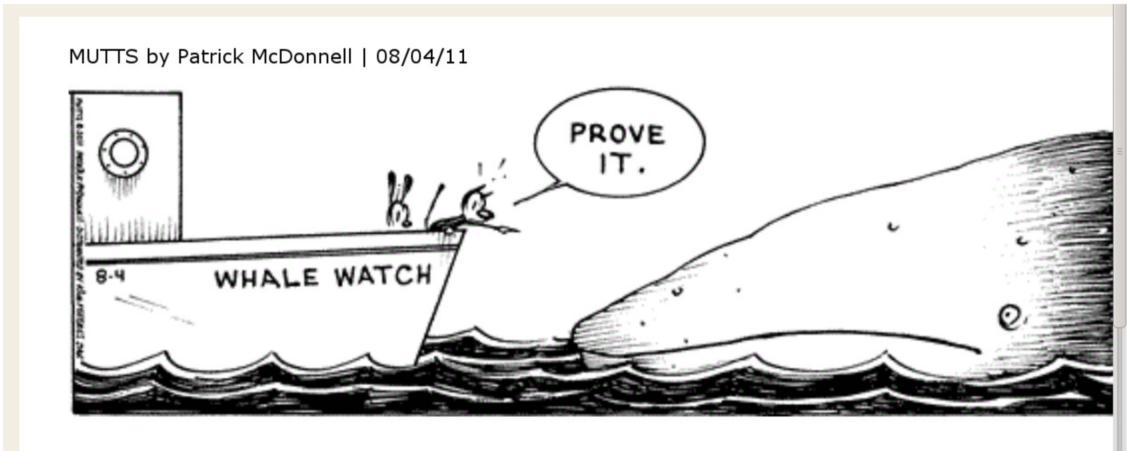
Meta-graph of SCCs Let $S_1, S_2, \ldots S_k$ be the strong connected components (i.e., SCCs) of G. The graph of SCCs is G^{SCC}

(A) Vertices are $S_1, S_2, \ldots S_k$
(B) There is an edge (S_i, S_j) if there is some $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge in G.

2.0.0.32 Reversal and SCCs

Proposition 2.0.2. *For any graph G, the graph of SCCs of G^{rev} is the same as the reversal of G^{SCC}.*

Proof: Exercise. ■



2.0.0.33 SCCs and DAGs

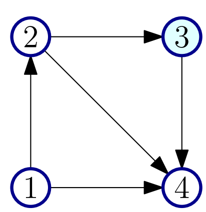
Proposition 2.0.3. *For any graph G, the graph $\mathsf{G}^{\mathrm{SCC}}$ has no directed cycle.*

Proof: If $\mathsf{G}^{\mathrm{SCC}}$ has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ should be in the same SCC in G. Formal details: exercise. ■

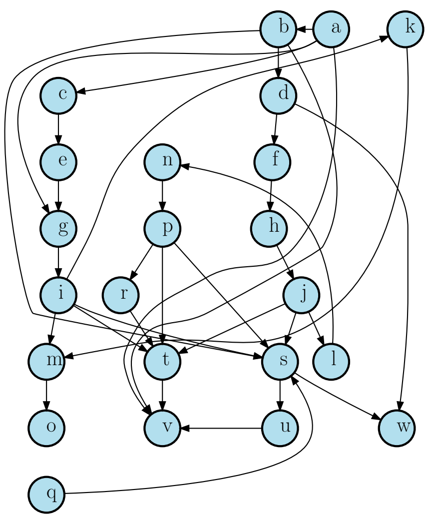
2.1 Directed Acyclic Graphs

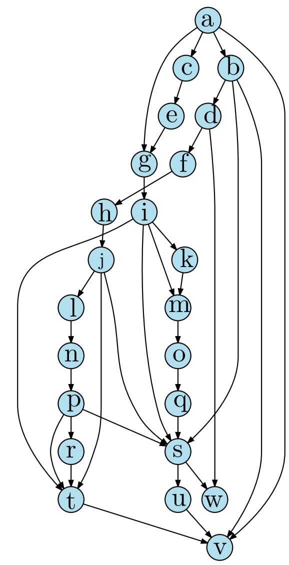
2.1.0.34 Directed Acyclic Graphs

Definition 2.1.1. *A directed graph G is a* **directed acyclic graph** *(DAG) if there is no directed cycle in G.*

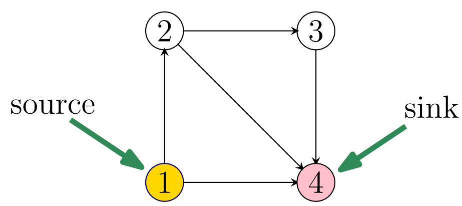


2.1.0.35 Is this a DAG?





2.1.0.36 Sources and Sinks

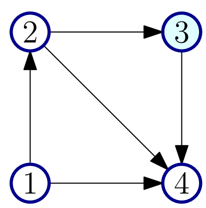


Definition 2.1.2. *(A) A vertex u is a* **source** *if it has no in-coming edges.*
(B) A vertex u is a **sink** *if it has no out-going edges.*

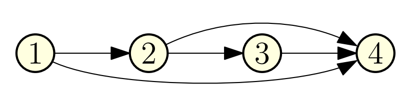
2.1.0.37 Simple DAG Properties

(A) Every DAG G has at least one source and at least one sink.
(B) If G is a DAG if and only if G^{rev} is a DAG.
(C) G is a DAG if and only each node is in its own strong connected component.
    Formal proofs: exercise.

2.1.0.38 Topological Ordering/Sorting



Graph G



Topological Ordering of G

Definition 2.1.3. *A* **topological ordering/topological sorting** *of $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.*

Informal equivalent definition: One can order the vertices of the graph along a line (say the x-axis) such that all edges are from left to right.

2.1.0.39 DAGs and Topological Sort

Lemma 2.1.4. *A directed graph* G *can be topologically ordered iff it is a* DAG*.*

Proof: $\implies$: Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle $C = u_1, u_2, \ldots, u_k, u_1$.

Then $u_1 \prec u_2 \prec \ldots \prec u_k \prec u_1$!

That is... $u_1 \prec u_1$.

A contradiction (to $\prec$ being an order).

Not possible to topologically order the vertices. ■

2.1.0.40 DAGs and Topological Sort

Lemma 2.1.5. *A directed graph* G *can be topologically ordered iff it is a* DAG*.*

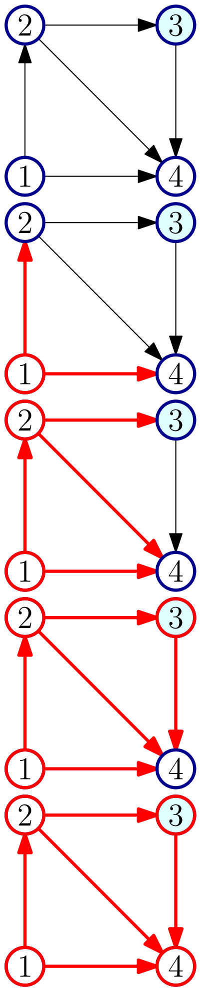
Proof:[Continued] $\Leftarrow$: Consider the following algorithm:

(A) Pick a source u, output it.
(B) Remove u and all edges out of u.
(C) Repeat until graph is empty.
(D) Exercise: prove this gives an ordering.

■

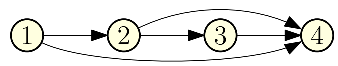
Exercise: show above algorithm can be implemented in $O(m + n)$ time.

2.1.0.41 Topological Sort: An Example



Output: 1 2 3 4

2.1.0.42 Topological Sort: Another Example



2.1.0.43 DAGs and Topological Sort

Note: A DAG G may have many different topological sorts.

Question: What is a DAG with the most number of distinct topological sorts for a given number n of vertices?

Question: What is a DAG with the least number of distinct topological sorts for a given number n of vertices?

Chapter 3

More on DFS in Directed Graphs, and Strong Connected Components, and DAGs

OLD CS 473: Fundamental Algorithms, Spring 2015
January 27, 2015

3.0.1 Using DFS...

3.0.1.1 ... to check for Acylicity and compute Topological Ordering

Question Given G, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

(A) Compute **DFS**(G)
(B) If there is a back edge then G is not a DAG.
(C) Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition 3.0.6. *G is a DAG iff there is no back-edge in* **DFS**(G)*.*

Proposition 3.0.7. *If G is a DAG and* $\mathrm{post}(v) > \mathrm{post}(u)$*, then* $(u \to v)$ *is not in G.*

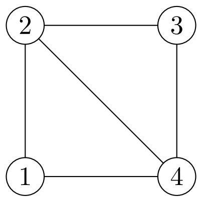
Proof: There are several possibilities:

(A) $[\mathrm{pre}(v), \mathrm{post}(v)]$ comes after $[\mathrm{pre}(u), \mathrm{post}(u)]$ and they are disjoint.
(B) But then, u was visited first by the **DFS**, if $(u, v) \in E(G)$ then **DFS** will visit v during the recursive call on u. But then, $\mathrm{post}(v) < \mathrm{post}(u)$. A contradiction.
(C) $[\mathrm{pre}(v), \mathrm{post}(v)] \subseteq [\mathrm{pre}(u), \mathrm{post}(u)]$: impossible as $\mathrm{post}(v) > \mathrm{post}(u)$.
(D) $[\mathrm{pre}(u), \mathrm{post}(u)] \subseteq [\mathrm{pre}(v), \mathrm{post}(v)]$. But then **DFS** visited v, and then visited u. Namely there is a path in G from v to u. But then if $(u, v) \in E(G)$ then there would be a cycle in G, and it would not be a DAG. Contradiction.

(E) No other possibility - since "lifetime" intervals of **DFS** are either disjoint or contained in each other.

∎

3.0.1.2 Example



3.0.1.3 Back edge and Cycles

Proposition 3.0.8. *G has a cycle iff there is a back-edge in* **DFS**(G).

Proof:

(A) If: (u, v) is a back edge $\implies$ there is a cycle C in G:
$C =$ path from v to u in **DFS** tree + edge $(u \to v)$.

(B) Only if: Suppose there is a cycle $C = v_1 \to v_2 \to \ldots \to v_k \to v_1$.
  (A) Let v_i be first node in C visited in **DFS**.
  (B) All other nodes in C are descendants of v_i since they are reachable from v_i.
  (C) Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge.

∎

3.0.1.4 Topological sorting of a DAG

Input: DAG G. With n vertices and m edges.
$O(n + m)$ algorithms for topological sorting

(A) Put source s of G as first in the order, remove s, and repeat.
(Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Question How to avoid sorting?
No need to sort - post numbering algorithm can output vertices...

3.0.1.5 DAGs and Partial Orders

Definition 3.0.9. *A* **partially ordered set** *is a set S along with a binary relation $\preceq$ such that $\preceq$ is*

1. **reflexive** *($a \preceq a$ for all $a \in V$),*

2. **anti-symmetric** *($a \preceq b$ and $a \neq b$ implies $b \not\preceq a$), and*

3. **transitive** *($a \preceq b$ and $b \preceq c$ implies $a \preceq c$).*

Example: For numbers in the plane define $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$.

Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

3.0.2 What's DAG but a sweet old fashioned notion

3.0.2.1 Who needs a DAG...

Example

(A) V: set of n products (say, n different types of tablets).
(B) Want to buy one of them, so you do market research...
(C) Online reviews compare only pairs of them.
...Not everything compared to everything.
(D) Given this partial information:
  (A) Decide what is the best product.
  (B) Decide what is the ordering of products from best to worst.
  (C) ...

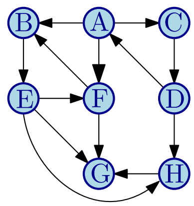
3.0.3 What DAGs got to do with it?

3.0.3.1 Or why we should care about DAGs

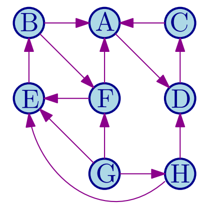
(A) DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
(B) Questions about DAGs:
  (A) Is a graph G a DAG?
  $\iff$
  Is the partial ordering information we have so far is consistent?
  (B) Compute a topological ordering of a DAG.
  $\iff$
  Find an a consistent ordering that agrees with our partial information.
  (C) Find comparisons to do so DAG has a unique topological sort.
  $\iff$
  Which elements to compare so that we have a consistent ordering of the items.

3.1 Linear time algorithm for finding all strong connected components of a directed graph

3.1.0.2 Reminder I: Graph G and its reverse graph G^{rev}



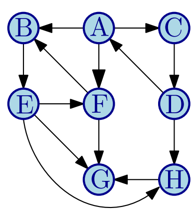
Graph G



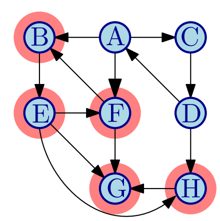
Reverse graph G^{rev}

3.1.1 Reminder II: Graph G a vertex F

3.1.1.1 .. and its reachable set $\text{rch}(\mathsf{G}, F)$



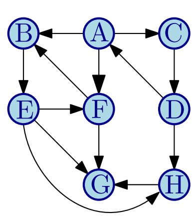
Graph G



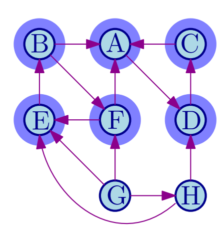
Reachable set of vertices from F

3.1.2 Reminder III: Graph G a vertex F

3.1.2.1 .. and the set of vertices that can reach it in G: $\text{rch}(\mathsf{G}^{\text{rev}}, F)$



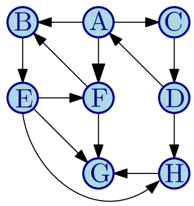
Graph G



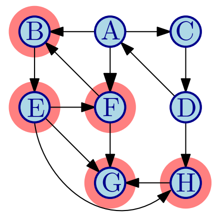
Set of vertices that can reach F, computed via **DFS** in the reverse graph G^{rev}.

3.1.3 Reminder IV: Graph G a vertex F and...

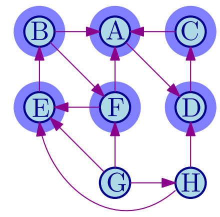
3.1.3.1 its strong connected component in G: $SCC(\mathsf{G}, F)$



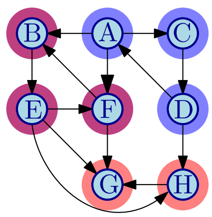
Graph G



$\mathbf{rch}(\mathsf{G}, F)$

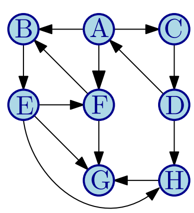


$\mathbf{rch}(\mathsf{G}^{\text{rev}}, F)$

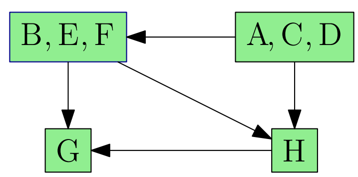


$$\text{SCC}(\mathsf{G}, F) = \mathbf{rch}(\mathsf{G}, F) \cap \mathbf{rch}(\mathsf{G}^{\text{rev}}, F)$$

3.1.3.2 Reminder II: Strong connected components (SCC)



Graph G



Graph of SCCs G^{SCC}

3.1.3.3 Finding all SCCs of a Directed Graph

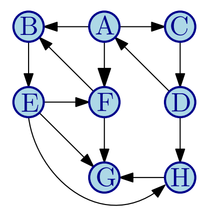
Problem Given a directed graph $G = (V, E)$, output *all* its strong connected components.

Straightforward algorithm:

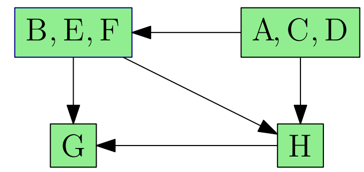
Mark all vertices in V as not visited.
for each vertex $u \in V$ not visited yet **do**
    find $\texttt{SCC}(G, u)$ the strong component of u:
        Compute $\mathbf{rch}(G, u)$ using $DFS(G, u)$
        Compute $\mathbf{rch}(G^{\text{rev}}, u)$ using $DFS(G^{\text{rev}}, u)$
        $\texttt{SCC}(G, u) \Leftarrow \mathbf{rch}(G, u) \cap \mathbf{rch}(G^{\text{rev}}, u)$
        $\forall u \in \texttt{SCC}(G, u)$: Mark u as visited.

Running time: $O(n(n+m))$ Is there an $O(n+m)$ time algorithm?

3.1.3.4 Structure of a Directed Graph



Graph G



Graph of SCCs $\mathsf{G}^{\mathrm{SCC}}$

Reminder $\mathsf{G}^{\mathrm{SCC}}$ is created by collapsing every strong connected component to a single vertex.

Proposition 3.1.1. *For a directed graph G, its meta-graph $\mathsf{G}^{\mathrm{SCC}}$ is a DAG.*

3.1.4 Linear-time Algorithm for SCCs: Ideas

3.1.4.1 Exploit structure of meta-graph...

Wishful Thinking Algorithm

(A) Let u be a vertex in a *sink* SCC of $\mathsf{G}^{\mathrm{SCC}}$
(B) Do **DFS**(u) to compute SCC(u)
(C) Remove SCC(u) and repeat

Justification

(A) **DFS**(u) only visits vertices (and edges) in SCC(u)
(B) ... since there are no edges coming out a sink!
(C) **DFS**(u) takes time proportional to size of SCC(u)
(D) Therefore, total time $O(n + m)$!

3.1.4.2 Big Challenge(s)

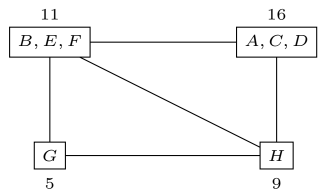
How do we find a vertex in a sink SCC of $\mathsf{G}^{\mathrm{SCC}}$?

Can we obtain an *implicit* topological sort of $\mathsf{G}^{\mathrm{SCC}}$ without computing $\mathsf{G}^{\mathrm{SCC}}$?

Answer: **DFS**(G) gives some information!

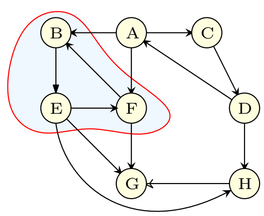
3.1.4.3 Post-visit times of SCCs

Definition 3.1.2. *Given G and a SCC S of G, define* $\mathrm{post}(S) = \max_{u \in S} \mathrm{post}(u)$ *where* post *numbers are with respect to some* **DFS**(G).

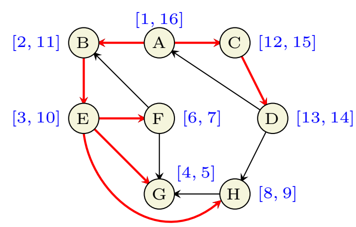


G^{SCC} with post times

3.1.4.4 An Example



Graph G



Graph with pre-post times for **DFS**(A); black edges in tree

3.1.5 Graph of strong connected components

3.1.5.1 ... and post-visit times

Proposition 3.1.3. *If S and S' are SCCs in G and (S, S') is an edge in G^{SCC} then* $\text{post}(S) > \text{post}(S')$.

Proof: Let u be first vertex in $S \cup S'$ that is visited.
(A) If $u \in S$ then all of S' will be explored before **DFS**(u) completes.
(B) If $u \in S'$ then all of S' will be explored before any of S.

■

A False Statement: If S and S' are SCCs in G and (S, S') is an edge in G^{SCC} then for *every* $u \in S$ and $u' \in S'$, $\text{post}(u) > \text{post}(u')$.

3.1.5.2 Topological ordering of the strong components

Corollary 3.1.4. *Ordering SCCs in decreasing order of* $\text{post}(S)$ *gives a topological ordering of G^{SCC}*

Recall: for a DAG, ordering nodes in decreasing post-visit order gives a topological sort.
So...
DFS(G) gives some information on topological ordering of G^{SCC}!

3.1.5.3 Finding Sources

Proposition 3.1.5. *The vertex u with the highest post visit time belongs to a source SCC in G^{SCC}*

Proof:¡2-¿
(A) post(SCC(u)) = post(u)
(B) Thus, post(SCC(u)) is highest and will be output first in topological ordering of G^{SCC}.

∎

3.1.5.4 Finding Sinks

Proposition 3.1.6. *The vertex u with highest post visit time in* **DFS**(G^{rev}) *belongs to a sink SCC of* G.

Proof:¡2-¿
(A) u belongs to source SCC of G^{rev}
(B) Since graph of SCCs of G^{rev} is the reverse of G^{SCC}, SCC(u) is sink SCC of G.

∎

3.1.6 Linear Time Algorithm

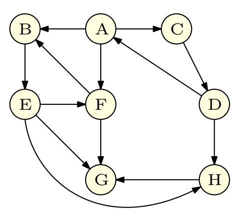
3.1.6.1 ...for computing the strong connected components in G

```
do DFS(G^rev) and sort vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Analysis Running time is $O(n + m)$. (Exercise)

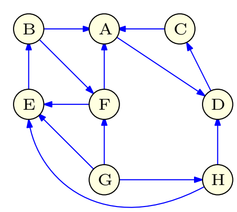
3.1.6.2 Linear Time Algorithm: An Example - Initial steps

Graph G:



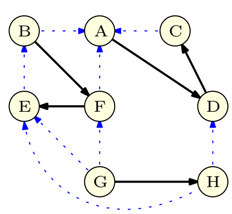
$\Longrightarrow$

Reverse graph G^{rev}:



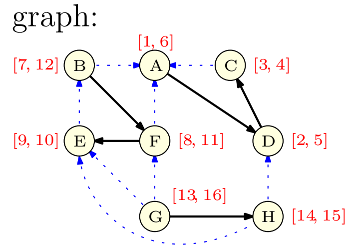
DFS of reverse graph:

$\Longrightarrow$



Pre/Post **DFS** numbering of reverse graph:

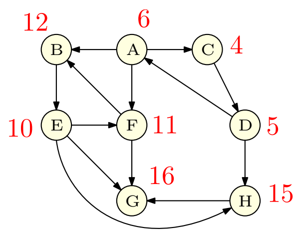
$\Longrightarrow$



3.1.7 Linear Time Algorithm: An Example

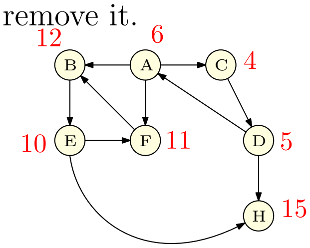
3.1.7.1 Removing connected components: 1

Original graph G with rev post numbers:



$\Longrightarrow$

Do **DFS** from vertex G remove it.

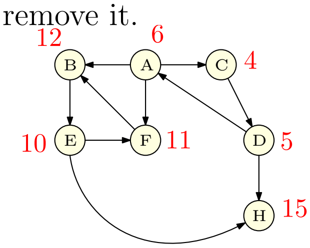


SCC computed:
$\{G\}$

3.1.8 Linear Time Algorithm: An Example

3.1.8.1 Removing connected components: 2

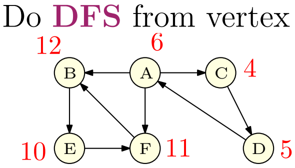
Do **DFS** from vertex G remove it.



SCC computed:
$\{G\}$

$\Longrightarrow$

Do **DFS** from vertex H, remove it.

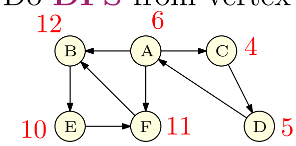


SCC computed:
$\{G\}, \{H\}$

3.1.9 Linear Time Algorithm: An Example

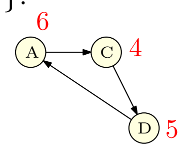
3.1.9.1 Removing connected components: 3

Do **DFS** from vertex H, remove it.



$\Longrightarrow$

Do **DFS** from vertex B
Remove visited vertices:
$\{F, B, E\}$.



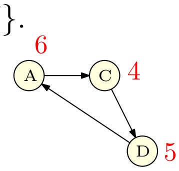
SCC computed:
$\{G\}, \{H\}$

SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

3.1.10 Linear Time Algorithm: An Example

3.1.10.1 Removing connected components: 4

Do **DFS** from vertex F
Remove visited vertices:
$\{F, B, E\}$.



SCC computed:
$\{G\}, \{H\}, \{F, B, E\}$

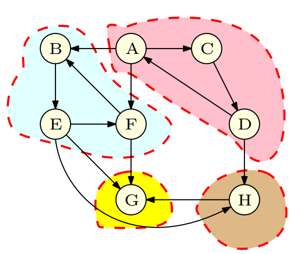
$\Longrightarrow$

Do **DFS** from vertex A
Remove visited vertices:
$\{A, C, D\}$.

SCC computed:
$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$

3.1.11 Linear Time Algorithm: An Example

3.1.11.1 Final result



SCC computed:
$\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}$

Which is the correct answer!

3.1.12 Obtaining the meta-graph...

3.1.12.1 Once the strong connected components are computed.

Exercise:

Given all the strong connected components of a directed graph $G = (V, E)$ show that the meta-graph $\mathsf{G}^{\mathrm{SCC}}$ can be obtained in $O(m + n)$ time.

3.1.12.2 Correctness: more details

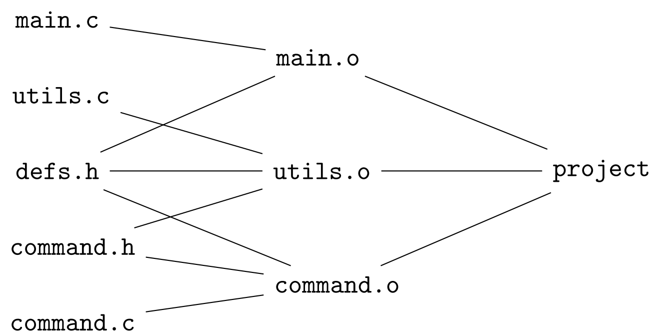
(A) let $S_1, S_2, \ldots, S_k$ be strong components in G
(B) Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
(C) consider **DFS**(G^{rev}) and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
(D) Assume without loss of generality that $post(u_k) > post(u_{k-1}) \geq \ldots \geq post(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
(E) u_k has highest post number and **DFS**(u_k) will explore all of S_k which is a sink component in G.
(F) After S_k is removed u_{k-1} has highest post number and **DFS**(u_{k-1}) will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

3.2 An Application to `make`

3.2.1 `make` utility

3.2.1.1 `make` Utility [Feldman]

(A) Unix utility for automatically building large software applications
(B) A makefile specifies
    (A) Object files to be created,
    (B) Source/object files to be used in creation, and
    (C) How to create them



3.2.1.2 An Example makefile

```
project:  main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o:  main.c defs.h
    cc -c main.c
utils.o:  utils.c defs.h command.h
    cc -c utils.c
command.o:  command.c defs.h command.h
    cc -c command.c
```

3.2.1.3 makefile as a Digraph

3.2.2 Computational Problems

3.2.2.1 Computational Problems for make

(A) Is the `makefile` reasonable?
(B) If it is reasonable, in what order should the object files be created?
(C) If it is not reasonable, provide helpful debugging information.
(D) If some file is modified, find the fewest compilations needed to make application consistent.

3.2.2.2 Algorithms for make

(A) Is the `makefile` reasonable? **Is G a DAG?**
(B) If it is reasonable, in what order should the object files be created? **Find a topological sort of a DAG.**
(C) If it is not reasonable, provide helpful debugging information. **Output a cycle. More generally, output all strong connected components.**
(D) If some file is modified, find the fewest compilations needed to make application consistent.
    (A) **Find all vertices reachable (using DFS/BFS) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.**

3.2.2.3 Take away Points

(A) Given a directed graph G, its SCCs and the associated acyclic meta-graph $\mathsf{G}^{\mathrm{SCC}}$ give a structural decomposition of G that should be kept in mind.
(B) There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
(C) DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).

3.3 Not for lecture - why do we have to use the reverse graph in computing the SCC?

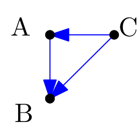
3.3.0.4 Finding a sink via post numbers in a DAG

Lemma 3.3.1. *Let G be a DAG, and consider the vertex u in G that minimizes* $\mathrm{post}(u)$*. Then u is a sink of G.*

Proof: The minimum $\mathrm{post}(\cdot)$ is assigned the first time **DFS** returns for its recursion. Let $\pi = v_1, v_2, \ldots, v_k = u$ be the sequence of vertices visited by the **DFS** at this point. Clearly, u (i.e., v_k), can not have an edge going into $v_1, \ldots, v_{k-1}$ since this would violates the assumption that there are no cycles. Similarly, u can not have an outgoing edge going into a vertex $z \in \mathsf{V}(\mathsf{G}) \setminus \{v_1, \ldots, v_k\}$, since the **DFS** would have continued into z, and u would not have been the first vertex to get assigned a post number. We conclude that u has no outgoing edges, and it is thus a sink. ■

3.3.0.5 Counterexample: Finding a source via min post numbers in a DAG

Counter example Let G be a DAG, and consider the vertex u in G that minimizes $\mathrm{post}(u)$ is a source. This is FALSE.



the **DFS** numbering might be:
A:[1,4]
B:[2,3]
C:[5,6] But clearly B is not a source.

3.3.0.6 Finding a source via post numbers in a DAG

Lemma 3.3.2. *Let G be a DAG, and consider the vertex u in G that maximizes* $\mathrm{post}(u)$*. Then u is a source of G.*

Proof: Exercise (And should already be in the slides.)

3.3.0.7 Meta graph computing the sink..

We proved:

Lemma 3.3.3. *Consider the graph G^{SCC}, with every CC $S \in V\big(G^{\mathrm{SCC}}\big)$ numbered by* $\mathrm{post}(S)$*. Then:*

$$\forall\, (S \to T) \in E(G^{\mathrm{SCC}}) \qquad \mathrm{post}(S) > \mathrm{post}(T).$$

(A) So, the SCC realizing $\min \mathrm{post}(S)$ is indeed a sink of $\mathsf{G}^{\mathrm{SCC}}$.
(B) But how to compute this? Not clear at all.

3.3.0.8 Meta graph computing a source is easy!

(A) The SCC realizing $\max \mathrm{post}(S)$ is a source of $\mathsf{G}^{\mathrm{SCC}}$.
(B) Furthermore, computing

$$\max_{S \in \mathsf{V}(\mathsf{G}^{\mathrm{SCC}})} \mathrm{post}(S) = \max_{S \in \mathsf{V}(\mathsf{G}^{\mathrm{SCC}})} \max_{v \in S} \mathrm{post}(v) = \max_{v \in \mathsf{V}(\mathsf{G})} \mathrm{post}(v).$$

is easy!
(C) So computing a source in the meta-graph is easy from the post numbering.
(D) But the algorithm needs a sink of the meta graph. Thus, we compute a vertex in the source SCC of the meta-graph of $(\mathsf{G}^{\mathrm{rev}})^{\mathrm{SCC}} = \big(\mathsf{G}^{\mathrm{SCC}}\big)^{\mathrm{rev}}$.

Chapter 4

Breadth First Search, Dijkstra's Algorithm for Shortest Paths

OLD CS 473: Fundamental Algorithms, Spring 2015
January 29, 2015

4.1 Breadth First Search

4.1.0.9 Breadth First Search (BFS)

Overview

(A) **BFS** is obtained from **BasicSearch** by processing edges using a ***queue*** data structure.

(B) It processes the vertices in the graph in the order of their shortest distance from the vertex s (the start vertex).

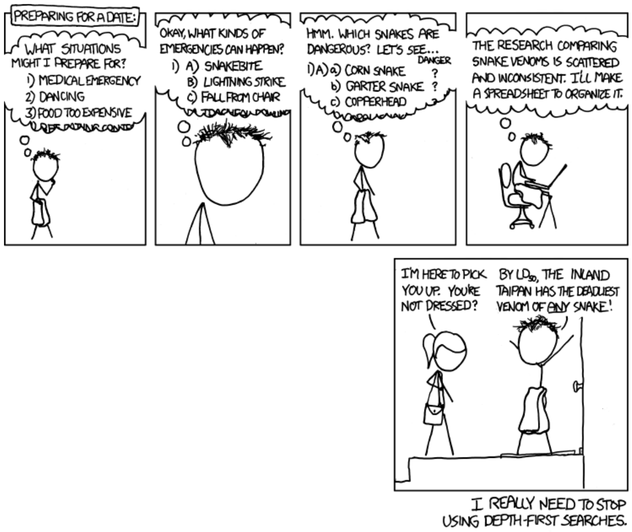
As such...

(A) **DFS** good for exploring graph structure
(B) **BFS** good for exploring *distances*

4.1.0.10 Queue Data Structure

Queues

queue: list of elements which supports the operations:
(A) **enqueue**: Adds an element to the end of the list
(B) **dequeue**: Removes an element from the front of the list
Elements are extracted in ***first-in first-out (FIFO)*** order, i.e., elements are picked in the order in which they were inserted.



4.1.0.11 BFS Algorithm

Given (undirected or directed) graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ and node $s \in \mathsf{V}$

```
BFS(s)
    Mark all vertices as unvisited
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u)
            if v is not visited then
                add edge (u, v) to T
                Mark v as visited and enq(v)
```

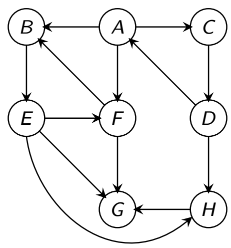
Proposition 4.1.1. **BFS**(s) *runs in* $O(n + m)$ *time.*

4.1.0.12 BFS: An Example in Undirected Graphs

1.	[1]	4.	[4,5,7,8]	7.	[8,6]
2.	[2,3]	5.	[5,7,8]	8.	[6]
3.	[3,4,5]	6.	[7,8,6]	9.	[]

BFS tree is the set of black edges.

4.1.0.13 BFS: An Example in Directed Graphs



4.1.0.14 BFS with Distance

```
BFS(s)
    Mark all vertices as unvisited and for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u) do
            if v is not visited do
                add edge (u, v) to T
                Mark v as visited, enq(v)
                and set dist(v) = dist(u) + 1
```

4.1.0.15 Properties of BFS: Undirected Graphs

Proposition 4.1.2. *The following properties hold upon* termination *of* **BFS***(s)*

(A) V(**BFS** *tree comp.*) = *set vertices in connected component* s.

(B) If $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v.

(C) $\forall u \in V$, $\mathrm{dist}(u) =$ *the length of shortest path from* s *to* u.

(D) If $u, v \in$ *connected component of* s, *and* $e = uv$ *is an edge of* G, *then either* $e \in$ **BFS** *tree, or* $|\mathrm{dist}(u) - \mathrm{dist}(v)| \leq 1$.

Proof: Exercise. ∎

4.1.0.16 Properties of BFS: <u>Directed</u> Graphs

Proposition 4.1.3. *The following properties hold upon termination of* $T \leftarrow$ **BFS**(s):

(A) For search tree T. $V(T) =$ *set of vertices reachable from* s

(B) If $\mathrm{dist}(u) < \mathrm{dist}(v)$ *then* u *is visited before* v

(C) $\forall u \in V(T)$: $\mathrm{dist}(u) =$ *length of shortest path from* s *to* u

(D) If u *is reachable from* s, $e = (u \to v) \in E(G)$.
Then either (i) e *is an edge in the search tree,*
or (ii) $\mathrm{dist}(v) - \mathrm{dist}(u) \leq 1$.
Not necessarily the case that $\mathrm{dist}(u) - \mathrm{dist}(v) \leq 1$.

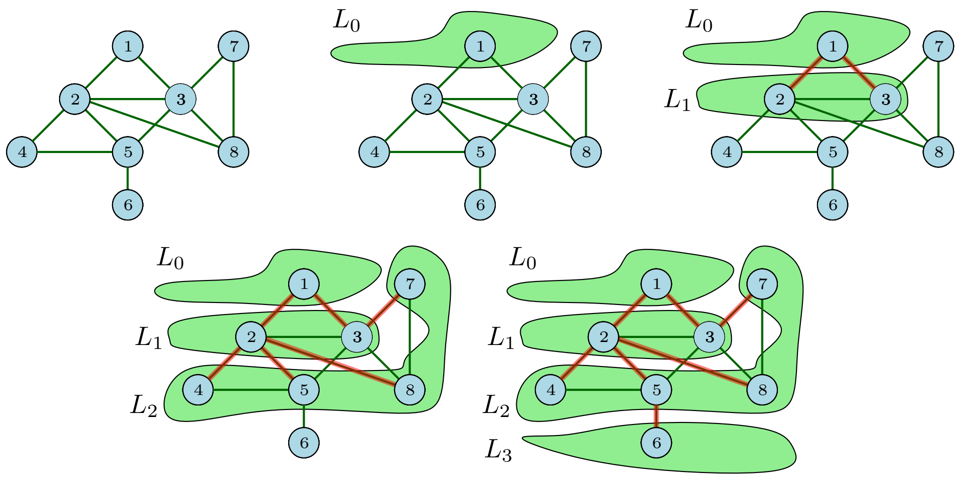
Proof: Exercise. ■

4.1.0.17 BFS with Layers

```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
                if v is not visited
                        mark v as visited
                        add (u, v) to tree T
                        add v to L_{i+1}
            i = i + 1
```

Running time: $O(n + m)$

4.1.0.18 Example



4.1.0.19 BFS with Layers: Properties

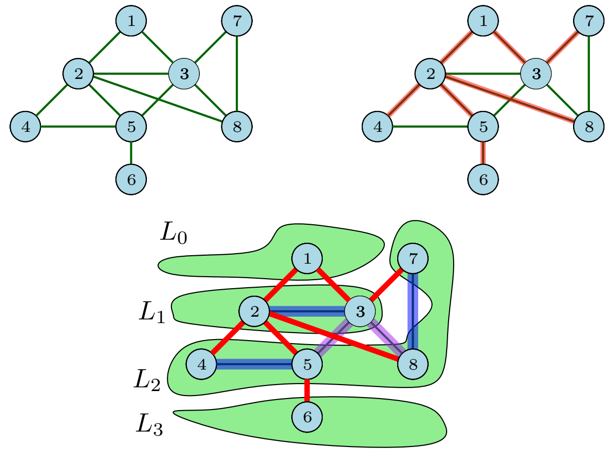
Proposition 4.1.4. *The following properties hold on termination of* **BFSLayers**(s)*.*

(A) **BFSLayers**(s) *outputs a* **BFS** *tree*

(B) L_i *is the set of vertices at distance exactly* i *from* s

(C) If G *is undirected, each edge* $e = uv$ *is one of three types:*

- *(A)* **tree** *edge between two* consecutive *layers*
- *(B) non-tree* **forward/backward** *edge between two consecutive layers*
- *(C) non-tree* **cross-edge** *with both* u, v *in same layer*
- *(D)* $\implies$ *Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.*



4.1.0.20 Example: Tree/cross/forward (backward) edges

4.1.1 BFS with Layers: Properties

4.1.1.1 For directed graphs

Proposition 4.1.5. *The following properties hold on termination of* **BFSLayers**(s)*, if* G *is directed.*

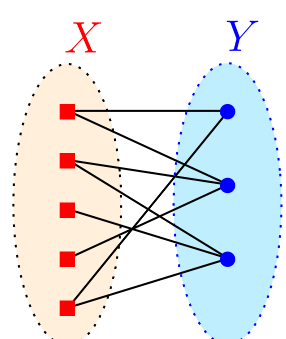
For each edge $e = (u \to v)$ *is one of four types:*

(A) a **tree** *edge between consecutive layers,* $u \in L_i, v \in L_{i+1}$ *for some* $i \geq 0$
(B) a non-tree **forward** *edge between consecutive layers*
(C) a non-tree **backward** *edge*
(D) a **cross-edge** *with both* u, v *in same layer*

4.2 Bipartite Graphs and an application of BFS

4.2.0.2 Bipartite Graphs

Definition 4.2.1 (Bipartite Graph). *Undirected graph* $G = (V, E)$ *is a* **bipartite graph** *if* V *can be partitioned into* X *and* Y *s.t. all edges in* E *are between* X *and* Y*.*



4.2.0.3 Bipartite Graph Characterization

Question When is a graph bipartite?

Proposition 4.2.2. *Every tree is a bipartite graph.*

Proof: Root tree T at some node r. Let L_i be all nodes at level i, that is, L_i is all nodes at distance i from root r. Now define X to be all nodes at even levels and Y to be all nodes at odd level. Only edges in T are between levels. ∎

Proposition 4.2.3. *An odd length cycle is not bipartite.*

4.2.0.4 Odd Cycles are not Bipartite

Proposition 4.2.4. *An odd length cycle is not bipartite.*

Proof: Let $C = u_1, u_2, \ldots, u_{2k+1}, u_1$ be an odd cycle. Suppose C is a bipartite graph and let X, Y be the partition. Without loss of generality $u_1 \in X$. Implies $u_2 \in Y$. Implies $u_3 \in X$. Inductively, $u_i \in X$ if i is odd $u_i \in Y$ if i is even. But $\{u_1, u_{2k+1}\}$ is an edge and both belong to X! ∎

4.2.0.5 Subgraphs

Definition 4.2.5. *Given a graph $G = (V, E)$ a* **subgraph** *of G is another graph $H = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$.*

Proposition 4.2.6. *If an undirected G is bipartite then any subgraph H of G is also bipartite.*

Proposition 4.2.7. *An undirected graph G is not bipartite if G has an odd cycle C as a subgraph.*

Proof: If G is bipartite then since C is a subgraph, C is also bipartite (by above proposition). However, C is not bipartite! ∎

4.2.0.6 Bipartite Graph Characterization

Theorem 4.2.8. *An undirected graph G is bipartite $\iff$ it has no odd length cycle as subgraph.*

Proof: **Only If:** G has an odd cycle implies G is not bipartite.
If: G has no odd length cycle. Assume without loss of generality that G is connected.
(A) Pick u arbitrarily and do **BFS**(u)
(B) $X = \cup_{i \text{ is even}} L_i$ and $Y = \cup_{i \text{ is odd}} L_i$
(C) **Claim:** X and Y is a valid partition if G has no odd length cycle.

∎

4.2.0.7 Proof of Claim

Claim 4.2.9. *In* **BFS**(u) *if* $a, b \in L_i$ *and* $ab \in \mathsf{E}(\mathsf{G})$ *then there is an odd length cycle containing* ab*.*

Proof: Let v be least common ancestor of a, b in **BFS** tree T.
v is in some level $j < i$ (could be u itself).
Path from $v \rightsquigarrow a$ in T is of length $j - i$.
Path from $v \rightsquigarrow b$ in T is of length $j - i$.
These two paths plus (a, b) forms an odd cycle of length $2(j - i) + 1$. ∎

4.2.0.8 Proof of Claim: Figure

4.2.0.9 Another tidbit

Corollary 4.2.10. *There is an* $O(n+m)$ *time algorithm to check if* G *is bipartite and output an odd cycle if it is not.*

4.3 Shortest Paths and Dijkstra's Algorithm

4.3.0.10 Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u \to v)$, $\ell(e) = \ell(u \to v)$ is its length.

(A) Given nodes s, t find shortest path from s to t.
(B) Given node s find shortest path from s to all other nodes.
(C) Find shortest paths for all pairs of nodes.
Many applications!

4.3.1 Single-Source Shortest Paths:

4.3.1.1 Non-Negative Edge Lengths

Single-Source Shortest Path Problems

(A) **Input**: A (undirected or directed) graph $G = (V, E)$ with **non-negative** edge lengths. For edge $e = (u \to v)$, $\ell(e) = \ell(u \to v)$ is its length.
(B) Given nodes s, t find shortest path from s to t.
(C) Given node s find shortest path from s to all other nodes.
(A) Restrict attention to directed graphs
(B) Undirected graph problem can be reduced to directed graph problem - how?
    (A) Given undirected graph G, create a new directed graph G' by replacing each edge $\{u, v\}$ in G by $(u \to v)$ and (v, u) in G'.
    (B) set $\ell(u \to v) = \ell(v, u) = \ell(\{u, v\})$
    (C) Exercise: show reduction works

4.3.1.2 Single-Source Shortest Paths via BFS

(A) **Special case:** All edge lengths are 1.
  (A) Run **BFS**(s) to get shortest path distances from s to all other nodes.
  (B) $O(m+n)$ time algorithm.
(B) **Special case:** Suppose $\ell(e)$ is an integer for all e?
  Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(e)-1$ dummy nodes on e.
(C) Let $L = \max_e \ell(e)$. New graph has $O(mL)$ edges and $O(mL+n)$ nodes. **BFS** takes $O(mL+n)$ time. Not efficient if L is large.

4.3.1.3 Towards an algorithm

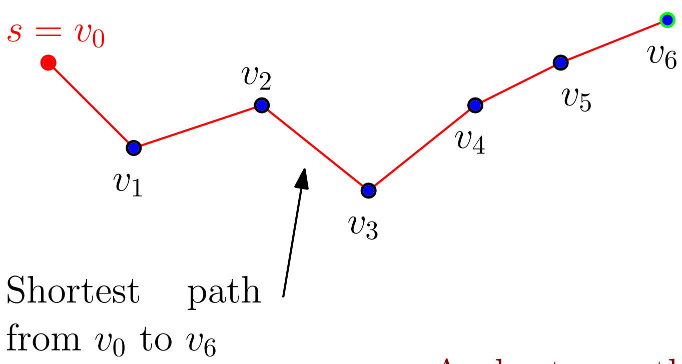
Why does **BFS** work?
BFS(s) explores nodes in increasing distance from s

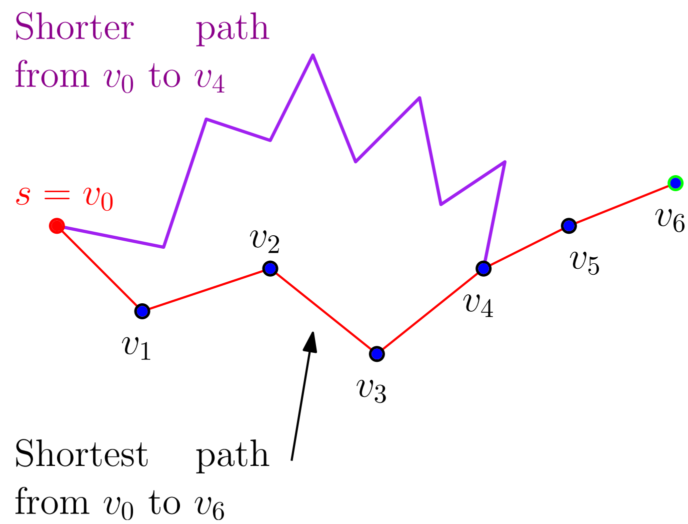
Lemma 4.3.1. *Let G be a directed graph with non-negative edge lengths. Let* $\mathrm{dist}(s,v)$ *denote the shortest path length from s to v. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \le i < k$:*

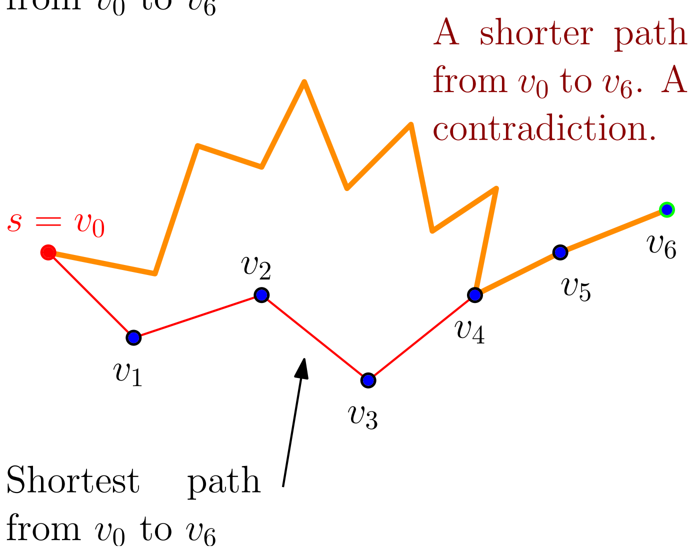
(A) $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from s to v_i*
(B) $\mathrm{dist}(s, v_i) \le \mathrm{dist}(s, v_k)$.

Proof: Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter path to v_k than $s = v_0 \to v_1 \ldots \to v_k$. ∎

4.3.1.4 A proof by picture







4.3.1.5 A Basic Strategy

Explore vertices in increasing order of distance from s:

(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

> Initialize for each node v, $\text{dist}(s, v) = \infty$
> Initialize $S = \emptyset$,
> **for** $i = 1$ to $|V|$ **do**
>   (* Invariant: S contains the $i - 1$ closest nodes to s *)
>   Among nodes in $V \setminus S$, find the node v that is the
> ith closest to s
>   Update $\text{dist}(s, v)$
> $S = S \cup \{v\}$

How can we implement the step in the for loop?

4.3.1.6 Finding the ith closest node

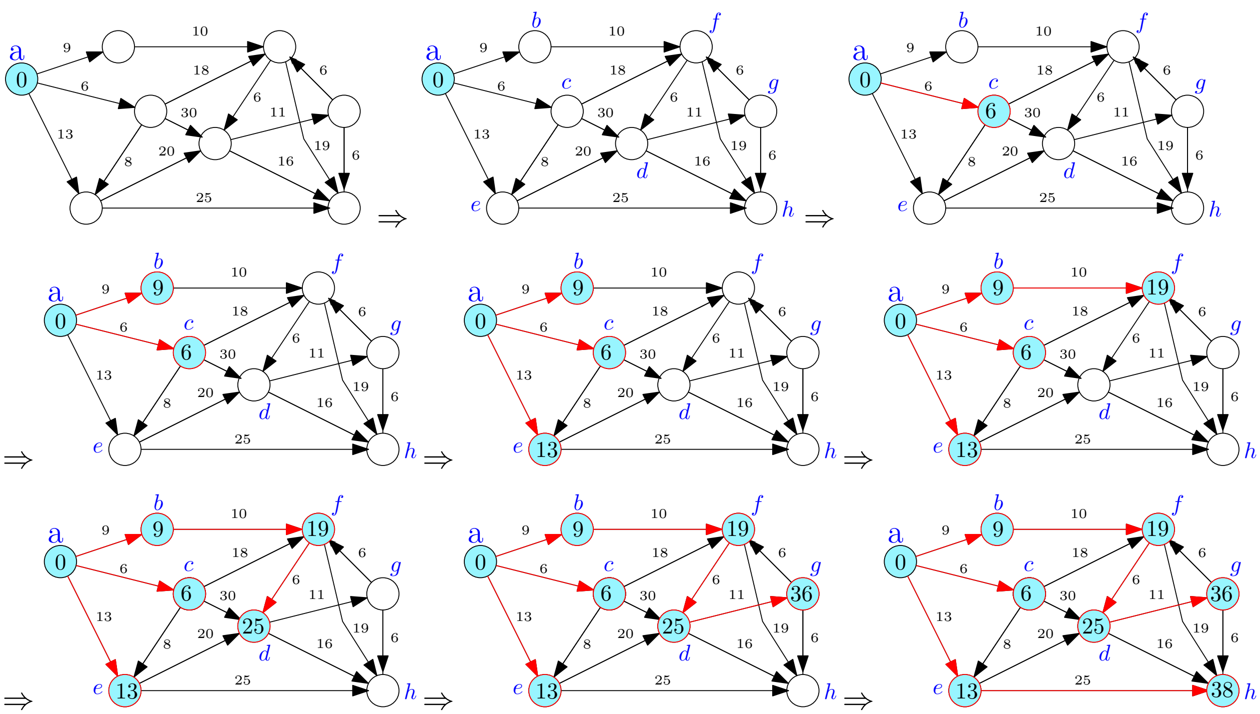
(A) S contains the $i - 1$ closest nodes to s
(B) Want to find the ith closest node from $V - S$.
What do we know about the ith closest node?

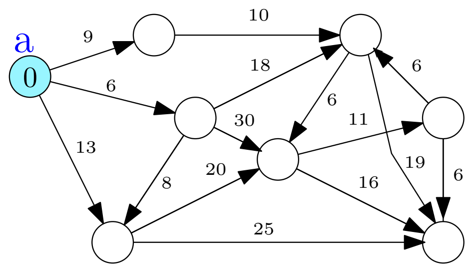
Claim 4.3.2. *Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to S.*

Proof: If P had an intermediate node u not in S then u will be closer to s than v. Implies v is not the ith closest node to s - recall that S already has the $i - 1$ closest nodes. ■

4.3.2 Finding the ith closest node repeatedly

4.3.2.1 An example





4.3.2.2 Finding the ith closest node

Corollary 4.3.3. *The ith closest node is adjacent to S.*

4.3.2.3 Finding the ith closest node

(A) S contains the $i-1$ closest nodes to s
(B) Want to find the ith closest node from $V - S$.
(C) For each $u \in \mathsf{V} \setminus S$ let $P(s, u, S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
(D) Let $d'(s, u)$ be the length of $P(s, u, S)$
(E) Observations: for each $u \in V - S$,
  (A) $\mathrm{dist}(s, u) \le d'(s, u)$ since we are constraining the paths
  (B) $d'(s, u) = \min_{a \in S}(\mathrm{dist}(s, a) + \ell(a, u))$ - Why?
(F) **Lemma 4.3.4.** *If v is the ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.*

4.3.2.4 Finding the ith closest node

Lemma 4.3.5. *Given:*
(A) S: Set of $i-1$ closest nodes to s.
(B) $d'(s, u) = \min_{x \in S}(\mathrm{dist}(s, x) + \ell(x, u))$
If v is an ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.

Proof: Let v be the ith closest node to s. Then there is a shortest path P from s to v that contains only nodes in S as intermediate nodes (see previous claim). Therefore $d'(s, v) = \mathrm{dist}(s, v)$. ■

4.3.2.5 Finding the ith closest node

Lemma 4.3.6. *If v is an ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.*

Corollary 4.3.7. *The ith closest node to s is the node $v \in V - S$ such that $d'(s, v) = \min_{u \in V - S} d'(s, u)$.*

Proof: For every node $u \in V - S$, $\mathrm{dist}(s, u) \le d'(s, u)$ and for the ith closest node v, $\mathrm{dist}(s, v) = d'(s, v)$. Moreover, $\mathrm{dist}(s, u) \ge \mathrm{dist}(s, v)$ for each $u \in V - S$. ■

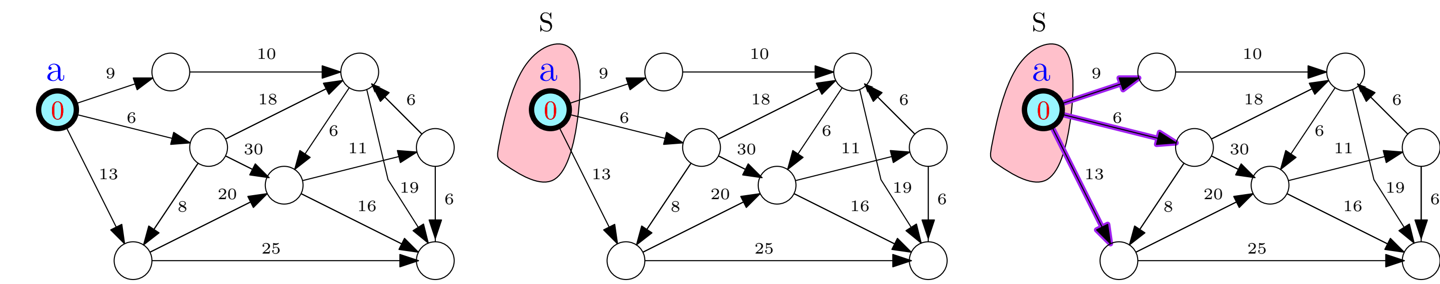
4.3.2.6 Candidate algorithm for shortest path

Initialize for each node v: $\mathrm{dist}(s, v) = \infty$
Initialize S = $\emptyset$, $d'(s, s) = 0$
for $i = 1$ to $|V|$ **do**
    (* Invariant: S contains the i-1 closest nodes to s *)
    (* Invariant: d'(s,u) is shortest path distance from u to s
     using only S as intermediate nodes*)
    Let v be such that $d'(s, v) = \min_{u \in V-S} d'(s, u)$
    $\mathrm{dist}(s, v) = d'(s, v)$
    $S = S \cup \{v\}$
    for each node u in $V \setminus S$ **do**
        $d'(s, u) \Leftarrow \min_{a \in S}\Big(\mathrm{dist}(s, a) + \ell(a, u)\Big)$

Correctness: By induction on i using previous lemmas.
Running time: $O(n \cdot (n + m))$ time.

(A) n outer iterations. In each iteration, $d'(s, u)$ for each u by scanning all edges out of nodes in S; $O(m + n)$ time/iteration.

4.3.2.7 Example



4.3.2.8 Improved Algorithm

(A) Main work is to compute the $d'(s, u)$ values in each iteration
(B) $d'(s, u)$ changes from iteration i to $i + 1$ only because of the node v that is added to S in iteration i.

Initialize for each node v, $\mathrm{dist}(s, v) = d'(s, v) = \infty$
Initialize S = $\emptyset$, d'(s,s) = 0
for $i = 1$ to $|V|$ **do**
    // S contains the $i - 1$ closest nodes to s,
    //   and the values of $d'(s, u)$ are current
    v be node realizing $d'(s, v) = \min_{u \in V-S} d'(s, u)$
    $\mathrm{dist}(s, v) = d'(s, v)$
    $S = S \cup \{v\}$
    Update $d'(s, u)$ for each u in $V - S$ as follows:
        $d'(s, u) = min\Big(d'(s, u),\ \mathrm{dist}(s, v) + \ell(v, u)\Big)$

Running time: $O(m + n^2)$ time.

(A) n outer iterations and in each iteration following steps
(B) updating $d'(s,u)$ after v added takes $O(deg(v))$ time so total work is $O(m)$ since a node enters S only once
(C) Finding v from $d'(s,u)$ values is $O(n)$ time

4.3.2.9 Dijkstra's Algorithm

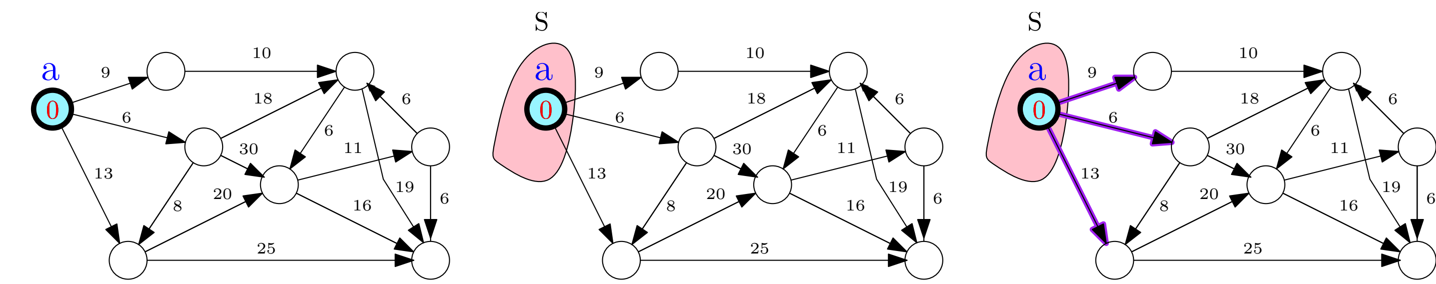
(A) eliminate $d'(s,u)$ and let $\text{dist}(s,u)$ maintain it
(B) update *dist* values after adding v by scanning edges out of v

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = {}, dist(s,s) = 0
for i = 1 to |V| do
    Let v be such that dist(s,v) = min_{u∈V−S} dist(s,u)
    S = S ∪ {v}
    for each u in Adj(v) do
        dist(s,u) = min(dist(s,u), dist(s,v) + ℓ(v,u))
```

Priority Queues to maintain *dist* values for faster running time

(A) Using heaps and standard priority queues: $O((m+n)\log n)$
(B) Using Fibonacci heaps: $O(m + n \log n)$.

4.3.2.10 Example: Dijkstra algorithm in action



4.3.3 Priority Queues

4.3.3.1 Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:
(A) **makePQ**: create an empty queue.
(B) **findMin**: find the minimum key in S.
(C) **extractMin**: Remove $v \in S$ with smallest key and return it.
(D) **insert**$(v, k(v))$: Add new element v with key $k(v)$ to S.
(E) **delete**(v): Remove element v from S.
(F) **decreaseKey**$(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \le k(v)$.

(G) **meld**: merge two separate priority queues into one.
All operations can be performed in $O(\log n)$ time.
decreaseKey is implemented via **delete** and **insert**.

4.3.3.2 Dijkstra's Algorithm using Priority Queues

```
Q ⇐ makePQ()
insert(Q, (s,0))
for each node u ≠ s do
    insert(Q, (u,∞))
S ⇐ ∅
for i = 1 to |V| do
    (v, dist(s,v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        decreaseKey([)]Q, (u, min(dist(s,u), dist(s,v) + ℓ(v,u))).
```

Priority Queue operations:
(A) $O(n)$ **insert** operations
(B) $O(n)$ **extractMin** operations
(C) $O(m)$ **decreaseKey** operations

4.3.3.3 Implementing Priority Queues via Heaps

Using Heaps Store elements in a heap based on the key value
(A) All operations can be done in $O(\log n)$ time
Dijkstra's algorithm can be implemented in $O((n + m)\log n)$ time.

4.3.3.4 Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps
(A) **extractMin**, **delete** in $O(\log n)$ time.
(B) **insert** in $O(1)$ *amortized* time.
(C) **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
(D) Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)
(A) Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
(B) Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

4.3.3.5 Shortest Path Tree

Dijkstra's algorithm finds the shortest path distances from s to V.
Question: How do we find the paths themselves?

```
Q = makePQ()
insert(Q, (s,0))
prev(s) ⇐ null
for each node u ≠ s do
    insert(Q, (u,∞) )
    prev(u) ⇐ null

S = ∅
for i = 1 to |V| do
    (v, dist(s,v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        if (dist(s,v) + ℓ(v,u) < dist(s,u) ) then
            decreaseKey(Q, (u, dist(s,v) + ℓ(v,u)) )
            prev(u) = v
```

4.3.3.6 Shortest Path Tree

Lemma 4.3.8. *The edge set $(u, \text{prev}(u))$ is the* reverse *of a shortest path tree rooted at s. For each u, the reverse of the path from u to s in the tree is a shortest path from s to u.*

Proof:[Proof Sketch.]
(A) The edge set $\{(u, \text{prev}(u)) \mid u \in V\}$ induces a directed in-tree rooted at s (Why?)
(B) Use induction on $|S|$ to argue that the tree is a shortest path tree for nodes in V.

■

4.3.3.7 Shortest paths to s

Dijkstra's algorithm gives shortest paths from s to all nodes in V.

How do we find shortest paths from all of V to s?

(A) In undirected graphs shortest path from s to u is a shortest path from u to s so there is no need to distinguish.
(B) In directed graphs, use Dijkstra's algorithm in G^{rev}!

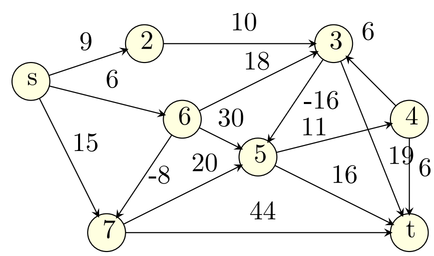
Chapter 5

Shortest Path Algorithms

OLD CS 473: Fundamental Algorithms, Spring 2015
February 3, 2015

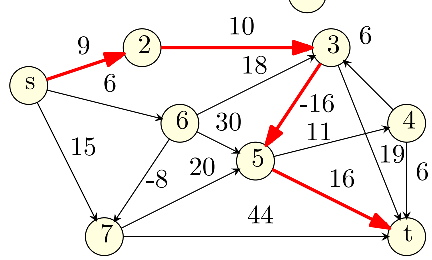
5.1 Shortest Paths with Negative Length Edges

5.1.0.8 Single-Source Shortest Paths with Negative Edge Lengths



Single-Source Shortest Path Problems **Input**: A *directed* graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Given nodes s, t find shortest path from s to t.
(B) Given node s find shortest path from s to all other nodes.

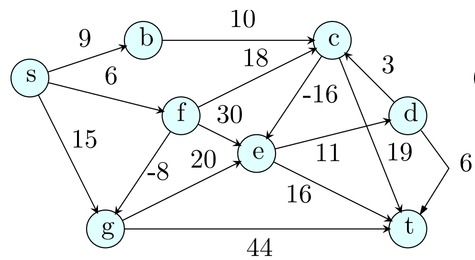


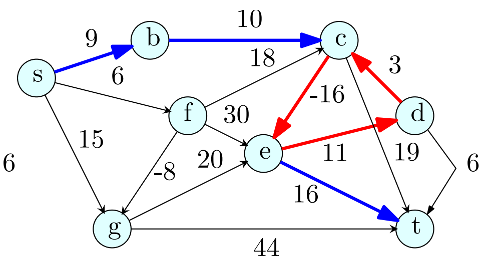
5.1.0.9 Negative Length Cycles

Definition 5.1.1. *A cycle C is a* **negative length cycle** *if the sum of the edge lengths of C is negative.*

5.1.0.10 Shortest Paths and Negative Cycles

(A) Given $G = (V, E)$ with edge lengths and s, t. Suppose
    (A) G has a negative length cycle C, and
    (B) s can reach C and C can reach t.
(B) **Question:** What is the shortest ***distance*** from s to t?





(C) Possible answers: Define shortest distance to be:
   (A) undefined, that is $-\infty$, OR
   (B) the length of a shortest ***simple*** path from s to t.

5.1.0.11 Shortest Paths and Negative Cycles

Lemma 5.1.2. *If there is an efficient algorithm to find a shortest simple $s \to t$ path in a graph with negative edge lengths, then there is an efficient algorithm to find the* longest *simple $s \to t$ path in a graph with positive edge lengths.*

Finding the $s \to t$ longest path is difficult. **NP-Hard**!

5.1.1 Shortest Paths with Negative Edge Lengths

5.1.1.1 Problems

Algorithmic Problems ***Input***: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

(A) Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
(B) Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
(C) Check if G has a negative length cycle or not.

5.1.2 Shortest Paths with Negative Edge Lengths

5.1.2.1 In Undirected Graphs

Note: With negative lengths, shortest path problems and negative cycle detection in undirected graphs cannot be reduced to directed graphs by bi-directing each undirected edge. Why?

Problem can be solved efficiently in undirected graphs but algorithms are different and more involved than those for directed graphs. Beyond the scope of this class. If interested, ask instructor for references.

5.1.2.2 Why Negative Lengths?

Several Applications

(A) Shortest path problems useful in modeling many situations — in some negative lengths are natural
(B) Negative length cycle can be used to find arbitrage opportunities in currency trading
(C) Important sub-routine in algorithms for more general problem: minimum-cost flow

5.1.3 Negative cycles

5.1.3.1 Application to Currency Trading

Currency Trading ***Input***: n currencies and for each ordered pair (a, b) the *exchange rate* for converting one unit of a into one unit of b.

Questions:

(A) Is there an arbitrage opportunity?
(B) Given currencies s, t what is the best way to convert s to t (perhaps via other intermediate currencies)?

5.1.4 Negative cycles

5.1.4.1 Application to Currency Trading

Concrete example:

(A) 1 Chinese Yuan = 0.1116 Euro
(B) 1 Euro = 1.3617 US dollar
(C) 1 US Dollar = 7.1 Chinese Yuan.

As such... Thus, if exchanging 1 \$ $\rightarrow$ Yuan $\rightarrow$ Euro $\rightarrow$ \$, we get: $0.1116 * 1.3617 * 7.1 = 1.07896\$$.

5.1.4.2 Reducing Currency Trading to Shortest Paths

(A) **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
(B) Create currency trading *directed* graph $G = (V, E)$:
    (A) For each currency i there is a node $v_i \in V$
    (B) $E = V \times V$: an edge for each pair of currencies
    (C) edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ **can be negative**
(C) **Exercise:** Verify that
    (A) There is an arbitrage opportunity if and only if G has a negative length cycle.
    (B) The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

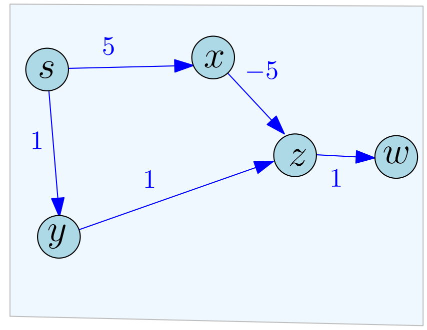
5.1.5 Reducing Currency Trading to Shortest Paths

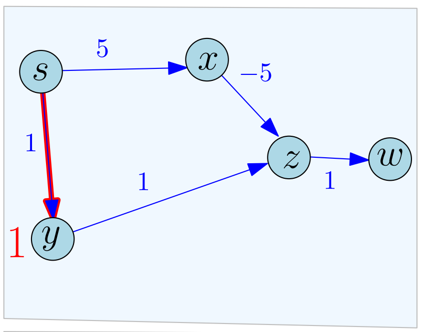
5.1.5.1 Math recall - relevant information

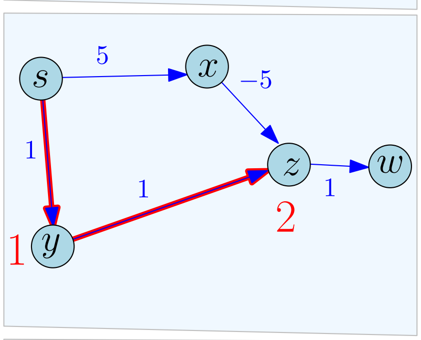
(A) $\log(\alpha_1 * \alpha_2 * \cdots * \alpha_k) = \log \alpha_1 + \log \alpha_2 + \cdots + \log \alpha_k$.
(B) $\log x > 0$ if and only if $x > 1$.

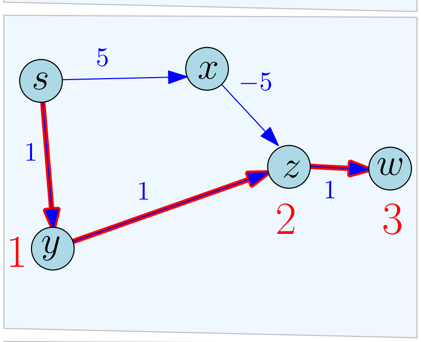
5.1.5.2 Dijkstra's Algorithm and Negative Lengths

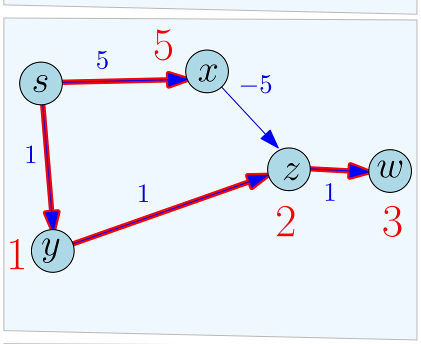
With negative cost edges, Dijkstra's algorithm fails

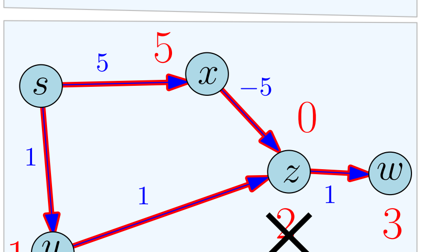












False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \to v_1 \to v_2 \ldots \to v_k$ is a shortest path from s to v_k then $dist(s, v_i) \leq dist(s, v_{i+1})$ for $0 \leq i < k$. Holds true only for non-negative edge lengths.

5.1.5.3 Shortest Paths with Negative Lengths

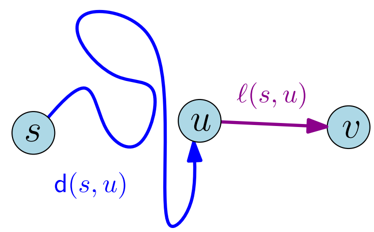
Lemma 5.1.3. *Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:*
(A) $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i
(B) ***False:*** *$dist(s, v_i) \leq dist(s, v_k)$ for $1 \leq i < k$.* ***Holds true only for non-negative edge lengths.***

Cannot explore nodes in increasing order of distance! We need a more basic strategy.

5.1.5.4 A Generic Shortest Path Algorithm

(A) Start with distance estimate for each node $d(s, u)$ set to ∞
(B) Maintain the invariant that there is an $s \to u$ path of length $d(s, u)$. Hence $d(s, u) \geq dist(s, u)$.
(C) Iteratively refine $d(s, \cdot)$ values until they reach the correct value $dist(s, \cdot)$ values at termination

Must hold that... $d(s, v) \leq d(s, u) + \ell(u, v)$



5.1.5.5 A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition 5.1.4. *Given distance estimates $d(s, u)$ for each $u \in V$, an edge $e = (u, v)$ is* **tense** *if $d(s, v) > d(s, u) + \ell(u, v)$.*

Relax$(e = (u, v))$
  if $(d(s, v) > d(s, u) + \ell(u, v))$ **then**
    $d(s, v) \Leftarrow d(s, u) + \ell(u, v)$

5.1.5.6 A Generic Shortest Path Algorithm

Invariant If a vertex u has value $d(s, u)$ associated with it, then there is a $s \rightsquigarrow u$ walk of length $d(s, u)$.

Proposition 5.1.5. **Relax** *maintains the invariant on $d(s, u)$ values.*

Proof: Indeed, if **Relax**$((u, v))$ changed the value of $d(s, v)$, then there is a walk to u of length $d(s, u)$ (by invariant), and there is a walk of length $d(s, u) + \ell(u, v)$ to v through u, which is the new value of $d(s, v)$. ∎

5.1.5.7 A Generic Shortest Path Algorithm

```
d(s,s) = 0
for each node u ≠ s do
    d(s,u) = ∞

while there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s,u) values
```

Technical assumption: If $e = (u, v)$ is an edge and $d(s, u) = d(s, v) = \infty$ then edge is not tense.

5.1.5.8 Properties of the generic algorithm

Proposition 5.1.6. *If u is not reachable from s then $d(s, u)$ remains at ∞ throughout the algorithm.*

5.1.5.9 Properties of the generic algorithm

Proposition 5.1.7. *If a negative length cycle C is reachable by s then there is always a tense edge and hence the algorithm never terminates.*

Proof Let $C = v_0, v_1, \dots, v_k$ be a negative length cycle. Suppose algorithm terminates. Since no edge of C was tense, for $i = 1, 2, \dots, k$ we have $d(s, v_i) \le d(s, v_{i-1}) + \ell(v_{i-1}, v_i)$ and $d(s, v_0) \le d(s, v_k) + \ell(v_k, v_0)$. Adding up all the inequalities we obtain that length of C is non-negative!

5.1.5.10 Proof in more detail...

$$
\begin{aligned}
d(s, v_1) &\le d(s, v_0) + \ell(v_0, v_1) \\
d(s, v_2) &\le d(s, v_1) + \ell(v_1, v_2) \\
&\dots \\
d(s, v_i) &\le d(s, v_{i-1}) + \ell(v_{i-1}, v_i) \\
&\dots \\
d(s, v_k) &\le d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
d(s, v_0) &\le d(s, v_k) + \ell(v_k, v_k)
\end{aligned}
$$

$$
\sum_{i=0}^{k} d(s, v_i) \le \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)
$$

$$0 \leq \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) = \text{len}(C).$$

C is a not a negative cycle. Contradiction.

5.1.5.11 Properties of the generic algorithm

Corollary 5.1.8. *If the algorithm terminates then there is no negative length cycle C that is reachable from s.*

5.1.5.12 Properties of the generic algorithm

Lemma 5.1.9. *If the algorithm terminates then $d(s,u) = dist(s,u)$ for each node u (and s cannot reach a negative cycle).*

Proof of lemma; see future slides.

5.1.6 Properties of the generic algorithm

5.1.6.1 If estimate distance from source too large, then ∃ tense edge...

Lemma 5.1.10. *Assume there is a path $\pi = v_1 \to v_2 \to \cdots \to v_k$ from $v_1 = s$ to $v_k = u$ (not necessarily simple!): $\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s,u)$.*

Then, there exists a tense edge in G.

Proof: Assume π is the shortest (in number of edges) such path, and observe that it must be that $\ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$. But then, we have that $d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k) = \ell(\pi) < d(s, v_k)$. Namely, $d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$ and the edge (v_{k-1}, v_k) is tense. ∎

$\implies$ If for any vertex u: $d(s,u) > \text{dist}(s,u)$ then the algorithm will continue working!

5.1.6.2 Generic Algorithm: Ordering Relax operations

```
d(s,s) = 0
for each node u ≠ s do
    d(s,u) = ∞

While there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s,u) values for u ∈ V(G)
```

Question: How do we pick edges to relax?

Observation: Suppose $s \to v_1 \to \ldots \to v_k$ is a shortest path.

If **Relax**(s, v_1), **Relax**(v_1, v_2), ..., **Relax**(v_{k-1}, v_k) are done in *order* then $d(s, v_k) = dist(s, v_k)$!

5.1.6.3 Ordering Relax operations

(A) **Observation:** Suppose $s \to v_1 \to \ldots \to v_k$ is a shortest path.

If **Relax**(s, v_1), **Relax**(v_1, v_2), $\ldots$, **Relax**(v_{k-1}, v_k) are done in *order* then $d(s, v_k) = dist(s, v_k)$! (Why?)

(B) We don't know the shortest paths so how do we know the order to do the Relax operations?

5.1.6.4 Ordering Relax operations

(A) We don't know the shortest paths so how do we know the order to do the Relax operations?
(B) We don't!
   (A) Relax *all* edges (even those not tense) in some arbitrary order
   (B) Iterate $|V| - 1$ times
   (C) First iteration will do **Relax**(s, v_1) (and other edges), second round **Relax**(v_1, v_2) and in iteration k we do **Relax**(v_{k-1}, v_k).

5.1.6.5 The Bellman-Ford (BellmanFord) Algorithm

```
BellmanFord:
    for each u ∈ V do
        d(s,u) ← ∞
    d(s,s) ← 0

    for i = 1 to |V| − 1 do
        for each edge e = (u,v) do
            Relax(e)

    for each u ∈ V do
        dist(s,u) ← d(s,u)
```

5.1.6.6 BellmanFord Algorithm: Scanning Edges

One possible way to scan edges in each iteration.

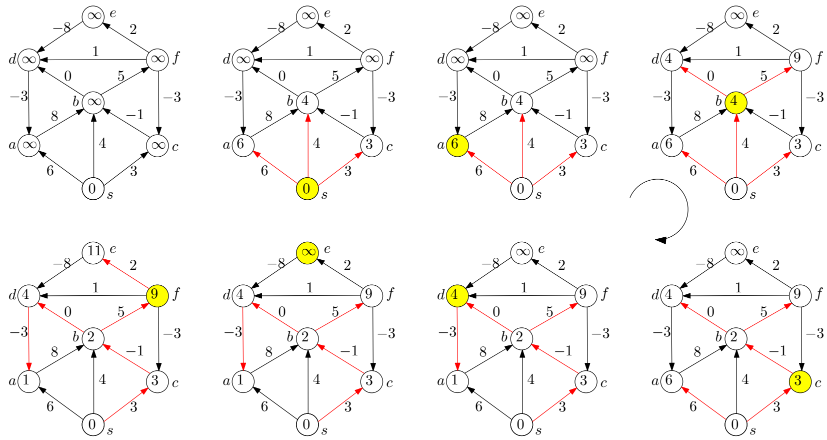


Figure 5.1: One iteration of **BellmanFord** that Relaxes all edges by processing nodes in the order s, a, b, c, d, e, f. Red edges indicate the prev pointers (in reverse)

```
Q is an empty queue
for each u ∈ V do
    d(s,u) = ∞
    enq(Q, u)
d(s,s) = 0

for i = 1 to |V| − 1 do
    for j = 1 to |V| do
        u = deq(Q)
        for each edge e in Adj(u) do
            Relax(e)
        enq(Q, u)

for each u ∈ V do
    dist(s,u) = d(s,u)
```

5.1.6.7 Example

5.1.6.8 Example

5.1.6.9 Correctness of the BellmanFord Algorithm

Lemma 5.1.11. *G: a directed graph with arbitrary edge lengths, v: a node in V s.t. there is a shortest path from s to v with i edges. Then, after i iterations of the loop in* **BellmanFord**, *$d(s,v) = dist(s,v)$*

Proof: By induction on i.

(A) Base case: $i = 0$. $d(s,s) = 0$ and $d(s,s) = dist(s,s)$.

(B) Induction Step: Let $s \to v_1 \ldots \to v_{i-1} \to v$ be a shortest path from s to v of i hops.

  (A) v_{i-1} has a shortest path from s of $i-1$ hops or less. (Why?). By induction, $d(s, v_{i-1}) = dist(s, v_{i-1})$ after $i-1$ iterations.

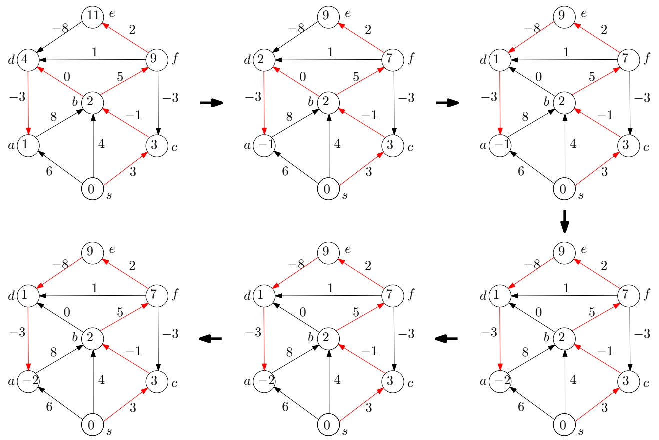


Figure 5.2: 6 iterations of **BellmanFord** starting with the first one from previous slide. No changes in 5th iteration and 6th iteration.

(B) In iteration i, **Relax**(v_{i-1}, v_i) sets $d(s, v_i) = \text{dist}(s, v_i)$.
(C) Note: Relax does not change $d(s, u)$ once $d(s, u) = \text{dist}(s, u)$.

■

5.1.6.10 Correctness of BellmanFord Algorithm

Corollary 5.1.12. *After $|V| - 1$ iterations of* **BellmanFord**, *$d(s, u) = dist(s, u)$ for any node u that has a shortest path from s.*

Note: If there is a negative cycle C such that s can reach C then we do not know whether $d(s, u) = dist(s, u)$ or not even if $dist(s, u)$ is well-defined.

Question: How do we know whether there is a negative cycle C reachable from s?

5.1.6.11 BellmanFord to detect Negative Cycles

```
for each u ∈ V do
        d(s,u) = ∞
d(s,s) = 0

for i = 1 to |V| − 1 do
        for each edge e = (u,v) do
            Relax(e)

for each edge e = (u,v) do
        if e = (u,v) is tense then
            Stop and output that s can reach
                    a negative length cycle

Output for each u ∈ V:  d(s,u)
```

5.1.6.12 Correctness

Lemma 5.1.13. *G has a negative cycle reachable from s if and only if there is a tense edge e after $|V|-1$ iterations of* **BellmanFord**.

Proof:[Proof Sketch.] G has no negative length cycle reachable from s implies that all nodes u have a shortest path from s. Therefore $d(s,u) = dist(s,u)$ after the $|V|-1$ iterations. Therefore, there cannot be any tense edges left.

If there is a negative cycle C then there is a tense edge after $|V|-1$ (in fact any number of) iterations. See lemma about properties of the generic shortest path algorithm. ■

5.2 Negative cycle detection

5.2.0.13 Finding the Paths and a Shortest Path Tree

```
BellmanFord:
    for each u ∈ V do
        d(s,u) = ∞
        prev(u) = null
    d(s,s) = 0
    for i = 1 to |V| − 1 do
        for each edge e = (u,v) do
            Relax(e)
    if there is a tense edge e then
        Output that s can reach a negative cycle C
    else
        for each u ∈ V do
            output d(s,u)
```

```
Relax(e = (u,v)):
    if (d(s,v) > d(s,u) + ℓ(u,v)) then
        d(s,v) = d(s,u) + ℓ(u,v)
        prev(v) = u
```

Note: prev pointers induce a shortest path tree.

5.2.0.14 Negative Cycle Detection

Negative Cycle Detection Given directed graph G with arbitrary edge lengths, does it have a negative length cycle?

(A) **BellmanFord** checks whether there is a negative cycle C that is reachable from a specific vertex s. There may negative cycles not reachable from s.

(B) Run **BellmanFord** $|V|$ times, once from each node u?

5.2.0.15 Negative Cycle Detection

(A) Add a new node s' and connect it to all nodes of G with zero length edges.
(B) **BellmanFord** from s' will fill find a negative length cycle if there is one.
(C) **Exercise:** why does this work?
(D) Negative cycle detection can be done with one **BellmanFord** invocation.

5.2.0.16 Running time for BellmanFord

(A) Input graph $G = (V, E)$ with $m = |E|$ and $n = |V|$.
(B) n outer iterations and m **Relax**() operations in each iteration. Each **Relax**() operation is $O(1)$ time.
(C) Total running time: $O(mn)$.

5.2.0.17 Dijkstra's Algorithm with Relax()

```
for each node u ≠ s do
    d(s,u) = ∞
d(s,s) = 0
S = ∅
while (S ≠ V) do
    Let v be node in V − S with min d value
    S = S ∪ {v}
    for each edge e in Adj(v) do
        Relax(e)
```

5.3 Shortest Paths in DAGs

5.3.0.18 Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input A directed **acyclic** graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Given nodes s, t find shortest path from s to t.
(B) Given node s find shortest path from s to all other nodes.

Simplification of algorithms for DAGs

(A) No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
(B) Can order nodes using topological sort

5.3.0.19 Algorithm for DAGs

(A) Want to find shortest paths from s. Ignore nodes not reachable from s.
(B) Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

(A) shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
(B) can find shortest paths in topological sort order.

5.3.0.20 Algorithm for DAGs

(A) Code:

```
ShortestPathDAG:
    for i = 1 to n do
        d(s, v_i) = ∞
    d(s, s) = 0

    for i = 1 to n − 1 do
        for each edge e in Adj(v_i) do
            Relax(e)

    return d(s, ·) values computed
```

(B) **Correctness:** induction on i and observation in previous slide.

(C) **Running time:** $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

5.3.0.21 Takeaway Points

(A) Shortest paths with potentially negative length edges arise in a variety of applications.

(B) Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).

(C) Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.

(D) In this case shortest walk = shortest path.

(E) Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.

(F) ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.

(G) If negative length cycle reachable from $s \implies$ no termination.

(H) Dijkstra's algorithm also instantiation of generic algorithm.

5.3.0.22 Points continued

(A) **BellmanFord** is instantiation of generic algorithm.

(B) ...in each iteration relaxes all the edges.

(C) Discovers negative length cycles if there is tense edge in the nth iteration.

(D) For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.

(E) Running time of **BellmanFord** algorithm is $O(nm)$.

(F) **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.

(G) If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.

(H) Can compute single-source shortest paths in DAG in linear time.

(I) Implies one can compute longest paths in a DAG in linear time.

5.3.0.23 Questions for a possible written quiz...

(A) Given a directed graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with n vertices and m edges, describe how to compute a cycle in G if such a cycle exist. What is the running time of your algorithm?

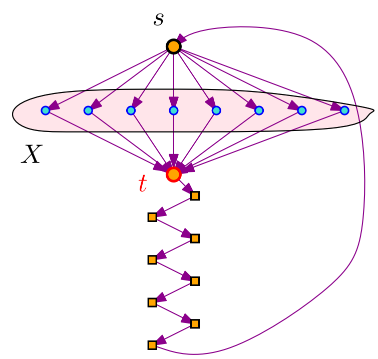
(B) As above, but assume edges have weights (negative or positive). Describe how to detect a negative cycle in G?

(C) Describe how to modify your algorithm from (B) so that it outputs the negative cycle.

5.4 Not for lecture

5.4.1 A shortest walk that visits all vertices...

5.4.1.1 ... in a graph might have to be of length $\Omega(n^2)$



Chapter 6

Reductions, Recursion and Divide and Conquer

OLD CS 473: Fundamental Algorithms, Spring 2015
February 5, 2015

6.1 Reductions and Recursion

6.1.0.2 Reduction

Reducing problem A to problem B:
(A) Algorithm for A uses algorithm for B as a *black box*

Q: How do you hunt a blue elephant? A: With a blue elephant gun.

Q: How do you hunt a red elephant? A: Hold his trunk shut until it turns blue, and then shoot him with the blue elephant gun.

Q: How do you shoot a white elephant? A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.

6.1.0.3 UNIQUENESS: Distinct Elements Problem

Problem Given an array A of n integers, are there any *duplicates* in A?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

6.1.0.4 Reduction to Sorting

(A) Code:

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
return NO
```

(B) **Running time:** $O(n)$ plus time to sort an array of n numbers
(C) **Key point:** algorithm uses sorting as a black box.

6.1.0.5 Two sides of Reductions

(A) Suppose problem A reduces to problem B
  (A) **Positive direction:** Algorithm for B implies an algorithm for A
  (B) **Negative direction:** Suppose there is no "efficient" algorithm for A then it implies no efficient algorithm for B (technical condition for reduction time necessary for this)
(B) **Example:** Distinct Elements reduces to Sorting in $O(n)$ time
  (A) An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
  (B) If there is *no* $o(n \log n)$ time algorithm for Distinct Elements problem then there is *no* $o(n \log n)$ time algorithm for Sorting.

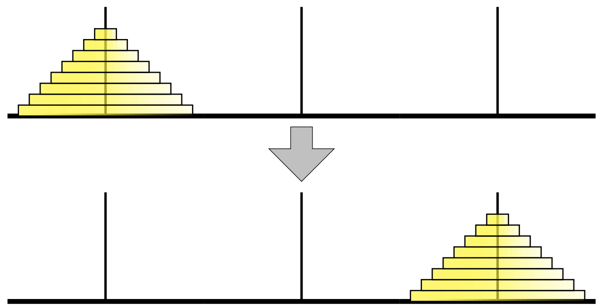
6.2 Recursion

6.2.0.6 Recursion

(A) **Reduction:** reduce one problem to another.
(B) **Recursion:** a special case of reduction
  (A) reduce problem to a *smaller* instance of *itself*
  (B) self-reduction
(C) Recursion as a reduction:
  (A) Problem instance of size n is reduced to *one or more* instances of size $n-1$ or less.
  (B) For termination, problem instances of small size are solved by some other method as *base cases*

6.2.0.7 Recursion

(A) Recursion is a powerful and fundamental technique.
(B) Basis for several other methods
  (A) Divide and conquer
  (B) Dynamic programming
  (C) Enumeration and branch and bound etc
  (D) Some classes of greedy algorithms



The Tower of Hanoi puzzle

(C) Makes proof of correctness easy (via induction)
(D) Recurrences arise in analysis

6.2.0.8 Selection Sort

(A) Sort a given array $A[1..n]$ of integers.
(B) Recursive version of Selection sort.
(C) Code:

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.
        Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

(D) $T(n)$: time for **SelectSort** on an n element array.
(E) $T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$
(F) $T(n) = \Theta(n^2)$.

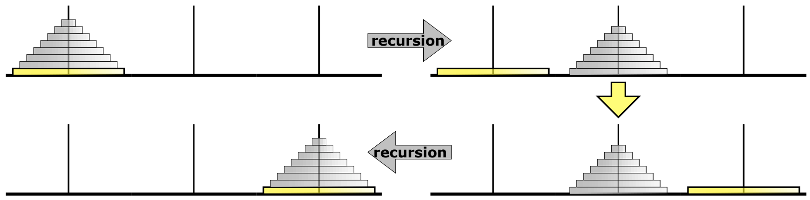
6.2.0.9 Tower of Hanoi

(A) Move stack of n disks from peg 0 to peg 2, one disk at a time.
(B) **Rule:** cannot put a larger disk on a smaller disk.
(C) **Question:** what is a strategy and how many moves does it take?

6.2.0.10 Tower of Hanoi via Recursion

6.2.0.11 Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```



The Tower of Hanoi algorithm; ignore everything but the bottom disk

$T(n)$: time to move n disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

6.2.0.12 Analysis

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2^2 T(n-2) + 2 + 1 \\ &= \ldots \\ &= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\ &= \ldots \\ &= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\ &= 2^{n-1} + 2^{n-2} + \ldots + 1 \\ &= (2^n - 1)/(2-1) = 2^n - 1 \end{aligned}$$

6.2.0.13 Non-Recursive Algorithms for Tower of Hanoi

(A) Pegs numbered $0, 1, 2$
(B) Non-recursive Algorithm 1:
  (A) Always move smallest disk forward if n is even, backward if n is odd.
  (B) Never move the same disk twice in a row.
  (C) Done when no legal move.
(C) Moves are exactly same as those of recursive algorithm. Prove by induction.

6.3 Divide and Conquer

6.3.0.14 Divide and Conquer Paradigm

(A) Divide and Conquer is a common and useful type of recursion Approach
  (A) Break problem instance into smaller instances - divide step
  (B) **Recursively** solve problem on smaller instances.

    (C) Combine solutions to smaller instances to obtain a solution to the original instance - conquer step
(B) **Question:** Why is this not plain recursion?
    (A) In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
    (B) There are many examples of this particular type of recursion that it deserves its own treatment.

6.4 Merge Sort

6.4.1 Merge Sort

6.4.1.1 Sorting

Input Given an array of n elements

Goal Rearrange them in ascending order

6.4.2 Merge Sort [von Neumann]

6.4.2.1 MergeSort

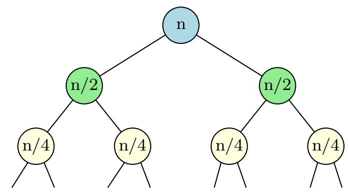
1. **Input:** Array $A[1 \ldots n]$
$$A\ L\ G\ O\ R\ I\ T\ H\ M\ S$$
2. Divide into subarrays $A[1 \ldots m]$ and $A[m+1 \ldots n]$, where $m = \lfloor n/2 \rfloor$
$$A\ L\ G\ O\ R \quad I\ T\ H\ M\ S$$
3. Recursively **MergeSort** $A[1 \ldots m]$ and $A[m+1 \ldots n]$
$$A\ G\ L\ O\ R \quad H\ I\ M\ S\ T$$
4. ¡5-¿Merge the sorted arrays
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

6.4.2.2 Merging Sorted Arrays

(A) Use a new array C to store the merged array
(B) Scan A and B from left-to-right, storing elements in C in order

¡1 $> A$ ¡2 $> G$ ¡3 $- 4 > L\ O\ R$ ¡1 $- 3 > H$ ¡4 $> I\ M\ S\ T$
$$A\ G\ H\ I\ L\ M\ O\ R\ S\ T$$

(C) Merge two arrays using only constantly more extra space is doable (in-place merge sort).
(D) **inplace_merge**: More complicated... Available in STL.



6.4.3 Analysis

6.4.3.1 Running Time

(A) $T(n)$: time for merge sort to sort an n element array
(B) $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
(C) What do we want as a solution to the recurrence?
(D) Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.
  (A) $T(n) = O(f(n))$ - upper bound.
  (B) $T(n) = \Omega(f(n))$ - lower bound

6.4.4 Solving Recurrences

6.4.4.1 Solving Recurrences: Some Techniques

(A) Some techniques:
  (A) Know some basic math: geometric series, logarithms, exponentials, elementary calculus.
  (B) Expand the recurrence and spot a pattern and use simple math.
  (C) **Recursion tree method** — imagine the computation as a tree.
  (D) **Guess and verify** — useful for proving upper and lower bounds even if not tight bounds
(B) **Albert Einstein:** "Everything should be made as simple as possible, but not simpler."
(C) Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

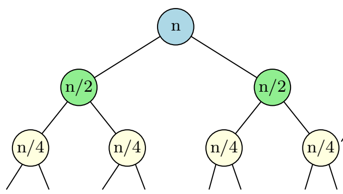
6.4.5 Recursion Trees

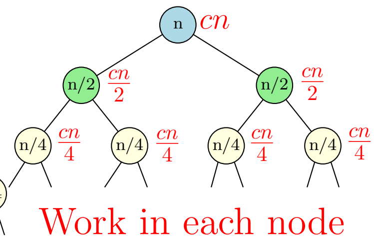
6.4.5.1 MergeSort: n is a power of 2

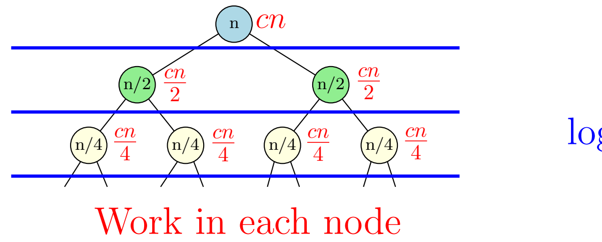
(A) Unroll the recurrence. $T(n) = 2T(n/2) + cn$
(B) Identify a pattern. At the ith level total work is cn.
(C) Sum over all levels. The number of levels is $\log n$. So total is $cn \log n = O(n \log n)$.

6.4.6 Recursion Trees

6.4.6.1 An illustrated example...







6.4.7 MergeSort Analysis

6.4.7.1 When n is not a power of 2

(A) When n is not a power of 2, the running time of **MergeSort** is expressed as

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

(B) $n_1 = 2^{k-1} < n \le 2^k = n_2$ (n_1, n_2 powers of 2).
(C) $T(n_1) < T(n) \le T(n_2)$ (Why?).
(D) $T(n) = \Theta(n \log n)$ since $n/2 \le n_1 < n \le n_2 \le 2n$.

6.4.7.2 Recursion Trees

MergeSort: n is not a power of 2

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

Observation: For any number x, $\lfloor x/2 \rfloor + \lceil x/2 \rceil = x$.

6.4.8 MergeSort Analysis

6.4.8.1 When n is not a power of 2: Guess and Verify

(A) If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
(B) Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
(C) *Verify?*
(D) proof by induction!
(E) **Induction Hypothesis:** $T(n) \le 2cn \log n$ for all $n \ge 1$
(F) **Base Case:** $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
(G) **Induction Step** Assume $T(k) \le 2ck \log k$ for all $k < n$ and prove it for $k = n$.

6.4.8.2 Induction Step

We have

$$\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\
&\leq 2c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor + 2c\lceil n/2 \rceil \log\lceil n/2 \rceil + cn \quad \text{(by induction)} \\
&\leq 2c\lfloor n/2 \rfloor \log\lceil n/2 \rceil + 2c\lceil n/2 \rceil \log\lceil n/2 \rceil + cn \\
&\leq 2c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) \log\lceil n/2 \rceil + cn \\
&\leq 2cn \log\lceil n/2 \rceil + cn \\
&\leq 2cn \log(2n/3) + cn \quad (\text{since } \lceil n/2 \rceil \leq 2n/3 \text{ for all } n \geq 2) \\
&\leq 2cn \log n + cn(1 - 2\log 3/2) \\
&\leq 2cn \log n + cn(\log 2 - \log 9/4) \\
&\leq 2cn \log n
\end{aligned}$$

6.4.8.3 Guess and Verify

The math worked out like magic!

Why was $2cn \log n$ chosen instead of say $4cn \log n$?

(A) Do not know upfront what constant to choose.
(B) Instead assume that $T(n) \leq \alpha cn \log n$ for some constant α.
α will be fixed later.
(C) Need to prove that for α large enough the algebra succeeds.
(D) In our case... need α such that $\alpha \log 3/2 > 1$.
(E) Typically, do the algebra with α and then show that it works...
... if α is chosen to be sufficiently large constant.

How do we know which function to guess? We don't so we try several "reasonable" functions. With practice and experience we get better at guessing the right function.

6.4.9 Guess and Verify

6.4.9.1 What happens if the guess is wrong?

(A) Guessed that the solution to the **MergeSort** recurrence is $T(n) = O(n)$.
(B) Try to prove by induction that $T(n) \leq \alpha cn$ for some const' α.
(C) **Induction Step:** attempt

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

But need to show that $T(n) \leq \alpha cn$!
(D) So guess does not work for **any** constant α. Suggests that our guess is incorrect.

6.4.9.2 Selection Sort vs Merge Sort

(A) Selection Sort spends $O(n)$ work to reduce problem from n to $n-1$ leading to $O(n^2)$ running time.
(B) Merge Sort spends $O(n)$ time *after* reducing problem to two instances of size $n/2$ each. Running time is $O(n \log n)$
(C) **Question:** Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say k arrays of size n/k each?

6.5 Quick Sort

6.5.0.3 Quick Sort

(A) **QuickSort** [Hoare]:
  (a) Pick a pivot element from array
  (b) Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
  (c) Linear scan of array it. Time is $O(n)$.
  (d) Recursively sort the subarrays, and concatenate them.
(B) Example:
  (A) array: 16, 12, 14, 20, 5, 3, 18, 19, 1
  (B) pivot: 16
  (C) split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
  (D) put them together with pivot in middle

6.5.0.4 Time Analysis

(A) Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
(B) If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
  (A) Theoretically, median can be found in linear time.
(C) Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

6.6 Fast Multiplication

6.7 The Problem

6.7.0.5 Multiplying Numbers

Problem Given two n-digit numbers x and y, compute their product.

Grade School Multiplication Compute “partial product” by multiplying each digit of y with x and adding the partial products.

$$
\begin{array}{r}
3141 \\
\underline{\times 2718} \\
25128 \\
3141 \\
21987 \\
\underline{6282} \\
8537238
\end{array}
$$

6.8 Algorithmic Solution

6.8.1 Grade School Multiplication

6.8.1.1 Time Analysis of Grade School Multiplication

(A) Each partial product: $\Theta(n)$
(B) Number of partial products: $\Theta(n)$
(C) Addition of partial products: $\Theta(n^2)$
(D) Total time: $\Theta(n^2)$

6.8.1.2 A Trick of Gauss

(A) Carl Friedrich Gauss: 1777–1855 “Prince of Mathematicians”
(B) Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$:

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

(C) How many multiplications do we need?
(D) Only 3! If we do extra additions and subtractions.
Compute $ac, bd, (a + b)(c + d)$. Then $(ad + bc) = (a + b)(c + d) - ac - bd$

6.8.2 Divide and Conquer Solution

6.8.2.1 Divide and Conquer

Assume n is a power of 2 for simplicity and numbers are in decimal.

(A) $x = x_{n-1}x_{n-2}\dots x_0$ and $y = y_{n-1}y_{n-2}\dots y_0$
(B) $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1}\dots x_{n/2}$ and $x_R = x_{n/2-1}\dots x_0$
(C) $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\dots y_{n/2}$ and $y_R = y_{n/2-1}\dots y_0$
Therefore

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

6.8.2.2 Example

$$
\begin{aligned}
1234 \times 5678 &= (100 \times 12 + 34) \times (100 \times 56 + 78) \\
&= 10000 \times 12 \times 56 \\
&\quad +100 \times (12 \times 78 + 34 \times 56) \\
&\quad +34 \times 78
\end{aligned}
$$

6.8.2.3 Time Analysis

$$
\begin{aligned}
xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\
&= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R
\end{aligned}
$$

4 recursive multiplications of number of size $n/2$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

6.8.3 Karatsuba's Algorithm

6.8.3.1 Improving the Running Time

$$
\begin{aligned}
xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\
&= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R
\end{aligned}
$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$. Time Analysis Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad\qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

6.8.3.2 State of the Art

Schönhage-Strassen 1971: $O(n \log n \log\log n)$ time using Fast-Fourier-Transform (FFT)

Martin Fürer 2007: $O(n \log n 2^{O(\log^* n)})$ time

Conjecture There is an $O(n \log n)$ time algorithm.

6.8.3.3 Analyzing the Recurrences

(A) Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
(B) Saving a multiplication: $T(n) = 3T(n/2)+O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

(A) In both cases, depth of recursion $L = \log n$.
(B) Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
(C) Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

6.8.3.4 Recursion tree analysis

Chapter 7

Recurrences, Closest Pair and Selection

OLD CS 473: Fundamental Algorithms, Spring 2015
February 10, 2015

7.1 Recurrences

7.1.0.5 Solving Recurrences

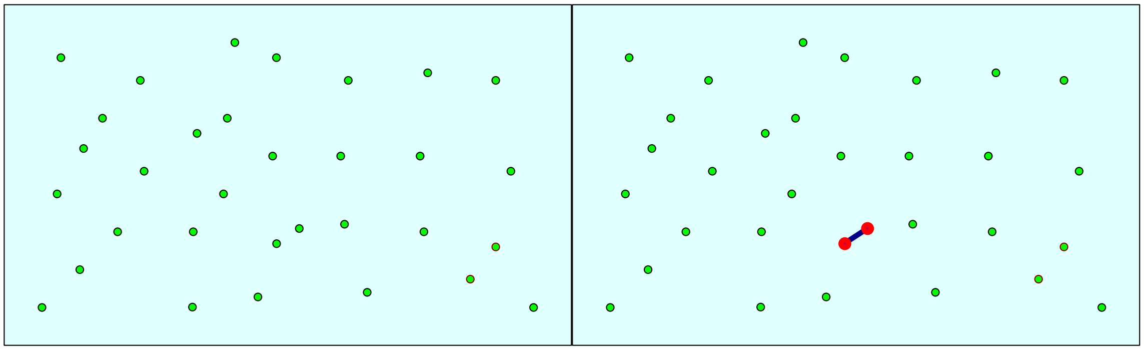
Two general methods:
(A) Recursion tree method: need to do sums
    (A) elementary methods, geometric series
    (B) integration
(B) Guess and Verify
    (A) guessing involves intuition, experience and trial & error
    (B) verification is via induction

7.1.0.6 Recurrence: Example I

(A) Consider $T(n) = 2T(n/2) + n/\lg n$.
(B) Construct recursion tree, and observe pattern.
(C) ith level has $n_i = 2^i$ nodes.
(D) problem size at node of level i is $n/2^i$.
(E) work at node of level i is $w_i = \frac{n}{2^i}/\lg \frac{n}{2^i}$.
(F) Total work at ith level is $n_i \cdot w_i = 2^i \cdot \frac{n}{2^i}/\lg \frac{n}{2^i} = n/\lg \frac{n}{2^i}$
(G) Summing over all levels $T(n) = \sum_{i=0}^{\lg n-1} n_i \cdot w_i = \sum_{i=0}^{\lg n-1} \frac{n}{\lg \frac{n}{2^i}} = n \sum_{i=0}^{\lg n-1} \frac{1}{\lg n - i} = n \sum_{j=1}^{\lg n} \frac{1}{j} = nH_{\lg n} = \Theta(n \log\log n)$

7.1.0.7 Recurrence: Example II

(A) Consider...



(B) What is the depth of recursion? $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.
(C) Number of levels: $n^{2^{-L}} = 2$ means $L = \log\log n$.
(D) Number of children at each level is 1, work at each node is 1
(E) Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log\log n)$.

7.1.0.8 Recurrence: Example III

(A) Consider $T(n) = \sqrt{n}T(\sqrt{n}) + n$.
(B) Using recursion trees: number of levels $L = \log\log n$
(C) Work at each level? Root is n, next level is $\sqrt{n} \times \sqrt{n} = n$, so on. Can check that each level is n.
(D) Thus, $T(n) = \Theta(n \log\log n)$

7.1.0.9 Recurrence: Example IV

(A) Consider $T(n) = T(n/4) + T(3n/4) + n$.
(B) Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
(C) Total work in any level is at most n. Total work in any level without leaves is exactly n.
(D) Highest leaf is at level $\log_4 n$ and lowest leaf is at level $\log_{4/3} n$
(E) Thus, $n \log_4 n \leq T(n) \leq n \log_{4/3} n$, which means $T(n) = \Theta(n \log n)$

7.2 Closest Pair

7.2.1 The Problem

7.2.1.1 Closest Pair - the problem

Input Given a set S of n points on the plane

Goal Find $p, q \in S$ such that $d(p, q)$ is minimum

7.2.1.2 Applications

(A) Basic primitive used in graphics, vision, molecular modelling
(B) Ideas used in solving nearest neighbor, Voronoi diagrams, Euclidean MST

7.2.2 Algorithmic Solution

7.2.2.1 Algorithm: Brute Force

(A) Compute distance between every pair of points and find minimum.
(B) Takes $O(n^2)$ time.
(C) Can we do better?

7.2.3 Special Case

7.2.3.1 Closest Pair: 1-d case

Input Given a set S of n points on a line

Goal Find $p, q \in S$ such that $d(p, q)$ is minimum

Algorithm
(A) Sort points based on coordinate
(B) Compute the distance between successive points, keeping track of the closest pair.
Running time $O(n \log n)$
(A) Can we do this in better running time?
(B) Can reduce Distinct Elements Problem (see lecture 1) to this problem in $O(n)$ time. Do you see how?

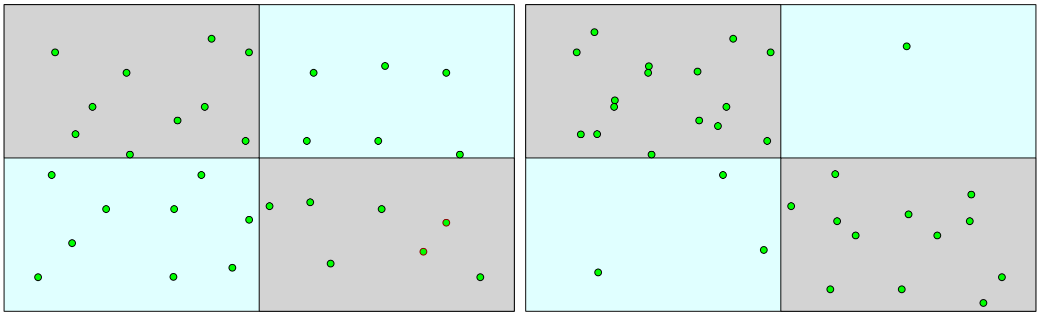
7.2.3.2 Generalizing 1-d case

(A) Can we generalize 1-d algorithm to 2-d?
(B) Sort according to x or y-coordinate??
(C) No easy generalization.

7.2.4 Divide and Conquer

7.2.4.1 First Attempt

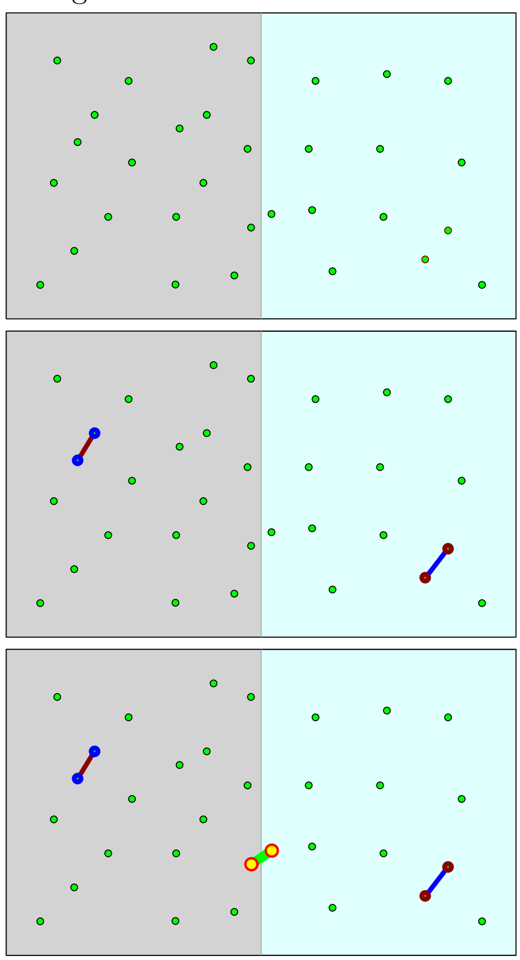
Divide and Conquer I
(A) Partition into 4 quadrants of roughly equal size.
(B) Find closest pair in each quadrant recursively.
(C) Combine solutions.
(D) But... How to partition the points in a balanced way?

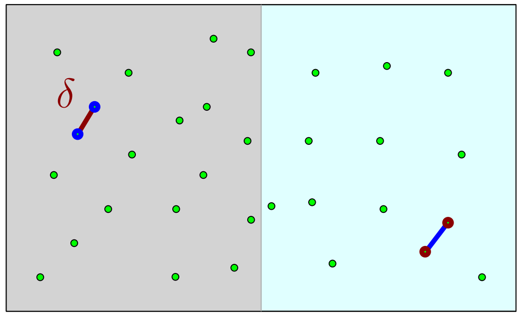


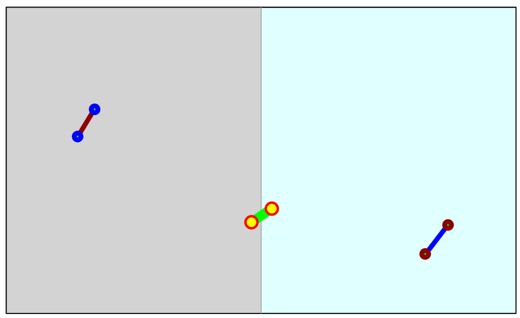
7.2.4.2 New Algorithm

Divide and Conquer II

(A) Divide the set of points into two equal parts via vertical line.
(B) Find closest pair in each half recursively.
(C) Find closest pair with one point in each half
(D) Return the best pair among the above 3 solutions







7.2.5 Towards a fast solution

7.2.5.1 New Algorithm

Divide and Conquer II

(A) ¡2-3¿Divide the set of points into two equal parts via vertical line
(B) Find closest pair in each half recursively
(C) ¡4-¿Find closest pair with one point in each half
(D) Return the best pair among the above 3 solutions

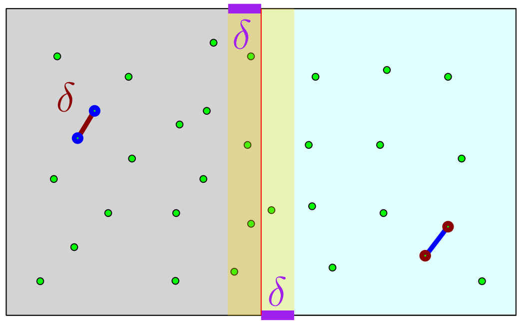
(A) Sort points based on x-coordinate and pick the median. Time $= O(n \log n)$
(B) How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

7.2.5.2 Combining Partial Solutions

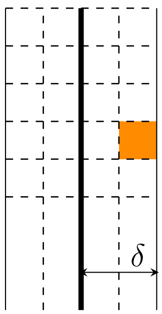
(A) Does it take $O(n^2)$ to combine solutions?
(B) Let δ be the distance between closest pairs, where both points belong to the same half.

7.2.5.3 Combining Partial Solutions

(A) Let δ be the distance between closest pairs, where both points belong to the same half.
(B) Need to consider points within δ of dividing line



7.2.5.4 Sparsity of Band

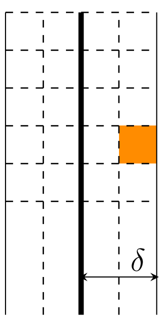


Divide the band into square boxes of size $\delta/2$

Lemma 7.2.1. *Each box has at most one point*

Proof: If not, then there are a pair of points (both belonging to one half) that are at most $\sqrt{2}\delta/2 < \delta$ apart! ■

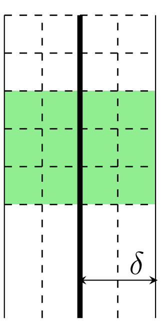
7.2.5.5 Searching within the Band



Lemma 7.2.2. *Suppose a, b are both in the band $d(a, b) < \delta$ then a, b have at most two rows of boxes between them.*

Proof: Each row of boxes has height $\delta/2$. If more than two rows then $d(a, b) > 2 \cdot \delta/2$! ■

7.2.5.6 Searching within the Band



Corollary 7.2.3. *Order points according to their y-coordinate. If p, q are such that $d(p, q) < \delta$ then p and q are within 11 positions in the sorted list.*

Proof:
(A) ≤ 2 points between them if p and q in same row.
(B) ≤ 6 points between them if p and q in two consecutive rows.
(C) ≤ 10 points between if p and q one row apart.
(D) $\implies$ More than ten points between them in the sorted y order than p and q are more than two rows apart.
(E) $\implies$ $d(p, q) > \delta$. A contradiction. ■

■

7.2.5.7 The Algorithm

ClosestPair(P):
1. <2-3>Find vertical line L splits P into equal halves: P_1 and P_2
2. $\delta_1 \leftarrow$ **ClosestPair**(P_1).
3. $\delta_2 \leftarrow$ **ClosestPair**(P_2).
4. $\delta = \min(\delta_1, \delta_2)$
5. <4-5>Delete points from P further than δ from L
6. <6-7>Sort P based on y-coordinate into an array A
7. <8-9>**for** $i = 1$ to $|A| - 1$ **do**
    - <8-9>**for** $j = i + 1$ to $\min\{i + 11, |A|\}$ **do**
        - <8-9>**if** ($dist(A[i], A[j]) < \delta$) update δ and closest pair

(A) Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
(B) Step 5 takes $O(n)$ time.
(C) Step 6 takes $O(n \log n)$ time
(D) Step 7 takes $O(n)$ time.

7.2.6 Running Time Analysis

7.2.6.1 Running Time

The running time of the algorithm is given by

$$T(n) \leq 2T(n/2) + O(n \log n)$$

Thus, $T(n) = O(n \log^2 n)$. Improved Algorithm Avoid repeated sorting of points in band: two options
(A) Sort all points by y-coordinate and store the list. In conquer step use this to avoid sorting
(B) Each recursive call returns a list of points sorted by their y-coordinates. Merge in conquer step in linear time.
Analysis: $T(n) \leq 2T(n/2) + O(n) = O(n \log n)$

7.3 Selecting in Unsorted Lists

7.3.1 Quick Sort

7.3.1.1 Quick Sort

Quick Sort [Hoare]
(A) ¡4¿Pick a pivot element from array
(B) ¡2-3¿Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
(C) Recursively sort the subarrays, and concatenate them.
Example:
(A) array: 16, 12, 14, 20, 5, 3, 18, 19, 1

(B) pivot: 16
(C) split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
(D) put them together with pivot in middle

7.3.1.2 Time Analysis

(A) Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
(B) If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
    (A) Theoretically, median can be found in linear time.
(C) Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

7.3.2 Selection

7.3.2.1 Problem - Selection

Input Unsorted array A of n integers

Goal Find the jth smallest number in A (*rank* j number)

Example 7.3.1. $A = \{4, 6, 2, 1, 5, 8, 7\}$ *and* $j = 4$. *The* j*th smallest element is* 5.

Median: $j = \lfloor (n+1)/2 \rfloor$

7.3.3 Naïve Algorithm

7.3.3.1 Algorithm I

(A) Sort the elements in A
(B) Pick jth element in sorted order
  Time taken $= O(n \log n)$
  Do we need to sort? Is there an $O(n)$ time algorithm?

7.3.3.2 Algorithm II

If j is small or $n - j$ is small then
(A) Find j smallest/largest elements in A in $O(jn)$ time. (How?)
(B) Time to find median is $O(n^2)$.

7.3.4 Divide and Conquer

7.3.4.1 Divide and Conquer Approach

(A) Pick a pivot element a from A
(B) Partition A based on a.
$A_{\text{less}} = \{x \in A \mid x \leq a\}$ and $A_{\text{greater}} = \{x \in A \mid x > a\}$
(C) $|A_{\text{less}}| = j$: return a
(D) $|A_{\text{less}}| > j$: recursively find jth smallest element in A_{less}
(E) $|A_{\text{less}}| < j$: recursively find kth smallest element in A_{greater} where $k = j - |A_{\text{less}}|$.

7.3.4.2 Time Analysis

(A) Steps:
  (A) Partitioning step: $O(n)$ time to scan A
  (B) How do we choose pivot? Recursive running time?
(B) Suppose we always choose pivot to be $A[1]$.
(C) Say A is sorted in increasing order and $j = n$.
(D) Exercise: show that algorithm takes $\Omega(n^2)$ time.

7.3.4.3 A Better Pivot

(A) Suppose: pivot ℓth smallest element where $n/4 \leq \ell \leq 3n/4$.
(B) That is pivot is *approximately* in the middle of A.
(C) $\implies n/4 \leq |A_{\text{less}}| \leq 3n/4$ *and* $n/4 \leq |A_{\text{greater}}| \leq 3n/4$.
(D) If we apply recursion,

$$T(n) \leq T(3n/4) + O(n)$$

Implies $T(n) = O(n)$!
(E) How do we find such a pivot?
(F) Randomly? This works!
Analysis a little bit later.
(G) Can we choose pivot deterministically?

7.3.5 Median of Medians

7.3.6 Divide and Conquer Approach

7.3.6.1 A game of medians

Idea

(A) Break input A into many subarrays: $L_1, \ldots L_k$.

(B) Find median m_i in each subarray L_i.
(C) Find the median x of the medians $m_1, \ldots, m_k$.
(D) Intuition: The median x should be close to being a good median of all the numbers in A.
(E) Use x as pivot in previous algorithm.

But we have to be...

More specific...
(A) Size of each group?
(B) How to find median of medians?

7.3.7 Choosing the pivot

7.3.7.1 A clash of medians

(A) Partition array A into $\lceil n/5 \rceil$ lists of 5 items each.
$L_1 = \left\{ A[1], A[2], \ldots, A[5] \right\}$, $L_2 = \left\{ A[6], \ldots, A[10] \right\}$, ..., $L_i = \left\{ A[5i+1], \ldots, A[5i-4] \right\}$, ..., $L_{\lceil n/5 \rceil} = \left\{ A[5\lceil n/5 \rceil - 4, \ldots, A[n] \right\}$.
(B) For $i = 1, \ldots, n/5$: compute median b_i of L_i
(C) ...using brute-force in $O(1)$ time. Total $O(n)$ time
(D) Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$
(E) Find median b of B

Lemma 7.3.2. *Median of B is an* approximate *median of A. That is, if b is used a pivot to partition A, then $|A_{less}| \le 7n/10 + 6$* and *$|A_{greater}| \le 7n/10 + 6$.*

7.3.8 Algorithm for Selection

7.3.8.1 A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉, where L_i = {A[5i-4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less| > j)
        return select(A_less, j)
    else
        return select(A_greater, j - |A_less|)
```

How do we find median of B? Recursively!

7.3.9 Running time of deterministic median selection

7.3.9.1 A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(1) = 1$$

Exercise: show that $T(n) = O(n)$

Lemma 7.3.3. *For $T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$, it holds that $T(n) = O(n)$.*

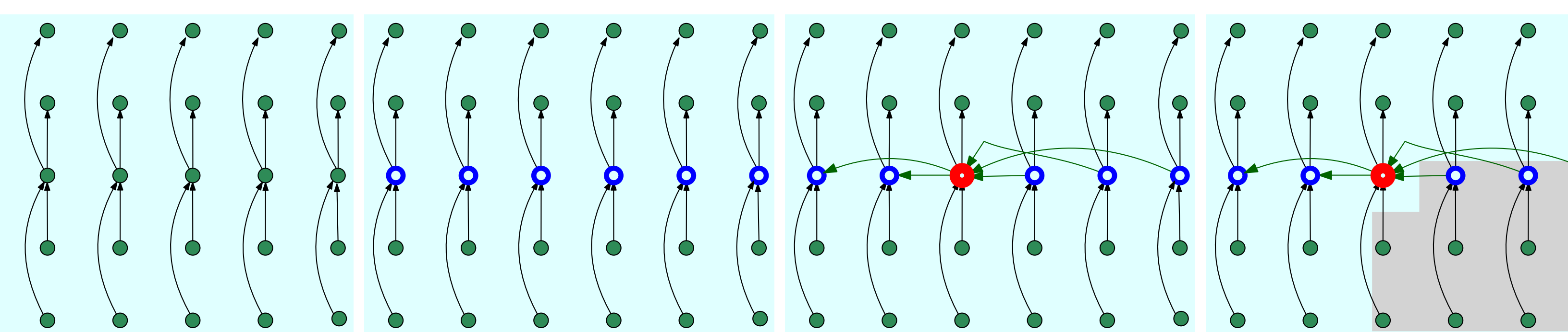
Proof: We claim that $T(n) \leq cn$, for some constant c. We have that $T(i) \leq c$ for all $i = 1, \ldots, 1000$, by picking c to be sufficiently large. This implies the base of the induction. Similarly, we can assume that the $O(n)$ in the above recurrence is smaller than $cn/100$, by picking c to be sufficiently large.

So, assume the claim holds for any $i < n$, and we will prove it for n. By induction, we have

$$\begin{aligned} T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n) \\ &\leq c(n/5 + 1) + c(7n/10 + 6) + cn/100 \\ &= cn(1/5 + 7/10 + 1/100 + 1/n + 6/n) \leq cn, \end{aligned}$$

for $n > 1000$. ■

7.3.9.2 Median of Medians: The movie



7.3.9.3 Median of Medians: Proof of Lemma

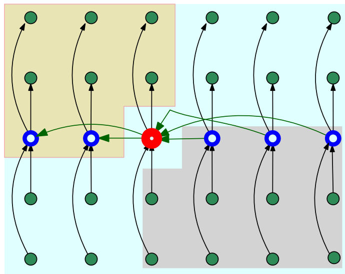


Figure 7.1: Shaded elements are all greater than b

Proposition 7.3.4. *There are at least $3n/10 - 6$ elements greater than the median of medians b.*

Proof: At least half of the $\lceil n/5 \rceil$ groups have at least 3 elements larger than b, except for last group and the group containing b. So b is less than

$$3(\lceil (1/2)\lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$$

■

7.3.9.4 Median of Medians: Proof of Lemma

Proposition 7.3.5. *There are at least $3n/10 - 6$ elements greater than the median of medians b.*

Corollary 7.3.6. $|A_{less}| \leq 7n/10 + 6$.

Via symmetric argument,

Corollary 7.3.7. $|A_{greater}| \leq 7n/10 + 6$.

7.3.9.5 Questions to ponder

(A) Why did we choose lists of size 5? Will lists of size 3 work?
(B) Write a recurrence to analyze the algorithm's running time if we choose a list of size k.

7.3.9.6 Median of Medians Algorithm

Due to:M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?
All except Vaughn Pratt!

7.3.9.7 Takeaway Points

(A) Recursion tree method and guess and verify are the most reliable methods to analyze recursions in algorithms.
(B) Recursive algorithms naturally lead to recurrences.
(C) Some times one can look for certain type of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.

Chapter 8

Binary Search, Introduction to Dynamic Programming

OLD CS 473: Fundamental Algorithms, Spring 2015
February 12, 2015

8.1 Exponentiation, Binary Search

8.2 Exponentiation

8.2.0.8 Exponentiation

(A) The problem:

Input Two numbers: a and integer $n \geq 0$

Goal Compute a^n

(B) Obvious algorithm:

```
SlowPow(a,n):
        x = 1;
        for i = 1 to n do
            x = x*a
        Output x
```

(C) $O(n)$ multiplications.

8.2.0.9 Fast Exponentiation

(A) **Observation:** $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.
(B) Implementation:

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,⌊n/2⌋)
    x = x * x
    if (n is odd) then
        x = x * a
    return x
```

(C) $T(n)$: number of multiplications for n

$$T(n) \leq T(\lfloor n/2 \rfloor) + 2$$

(D) $T(n) = \Theta(\log n)$

8.2.0.10 Complexity of Exponentiation

(A) **Question:** Is **SlowPow**() a polynomial time algorithm? **FastPow**?
(B) Input size: $O(\log a + \log n)$
(C) Output size:
(D) ... $O(n \log a)$.
(E) Not necessarily polynomial in input size!
(F) Both **SlowPow** and **FastPow** are polynomial in output size.

8.2.0.11 Exponentiation modulo a given number

(A) Exponentiation in applications:

Input Three integers: a, $n \geq 0$, $p \geq 2$ (typically a prime).

Goal Compute $a^n \mod p$.

(B) Input size: $\Theta(\log a + \log n + \log p)$.
(C) Output size: $O(\log p)$ and hence polynomial in input size.
(D) **Observation:** $xy \mod p = ((x \mod p)(y \mod p)) \mod p$

8.2.0.12 Exponentiation modulo a given number

(A) Problem:

Input Three integers: a, $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \mod p$

(B) Implementation:

```
FastPowMod(a,n,p):
    if (n = 0) return 1
    x = FastPowMod(a,⌊n/2⌋,p)
    x = x * x mod p
    if (n is odd)
        x = x * a mod p
    return x
```

(C) **FastPowMod** is a polynomial time algorithm.
(D) **SlowPowMod** is not (why?).

8.3 Binary Search

8.3.0.13 Binary Search in Sorted Arrays

Input Sorted array A of n numbers and number x

Goal Is x in A?

```
BinarySearch(A[a..b], x):
        if (b − a < 0) return NO
        mid = A[⌊(a + b)/2⌋]
        if (x = mid) return YES
        if (x < mid)
            return BinarySearch(A[a..⌊(a + b)/2⌋ − 1], x)
        else
            return BinarySearch(A[⌊(a + b)/2⌋ + 1..b],x)
```

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.
Observation: After k steps, size of array left is $n/2^k$

8.3.0.14 Another common use of binary search

(A) **Optimization version:** find solution of best (say minimum) value
(B) **Decision version:** is there a solution of value at most a given value v?
Reduce optimization to decision (may be easier to think about):
(A) Given instance I compute upper bound $U(I)$ on best value
(B) Compute lower bound $L(I)$ on best value
(C) Do binary search on interval $[L(I), U(I)]$ using decision version as black box
(D) $O(\log(U(I) - L(I)))$ calls to decision version if $U(I), L(I)$ are integers

8.3.0.15 Example

(A) **Problem:** shortest paths in a graph.
(B) **Decision version:** given G with non-negative integer edge lengths, nodes s, t and bound B, is there an s-t path in G of length at most B?
(C) **Optimization version:** find the length of a shortest path between s and t in G.
Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

8.3.0.16 Example continued

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?
(A) Let U be maximum edge length in G.
(B) Minimum edge length is L.
(C) s-t shortest path length is at most $(n - 1)U$ and at least L.
(D) Apply binary search on the interval $[L, (n - 1)U]$ via the algorithm for the decision problem.

(E) $O(\log((n-1)U - L))$ calls to the decision problem algorithm sufficient. Polynomial in input size.
(F) Assuming all numbers are integers.

8.4 Introduction to Dynamic Programming

8.4.0.17 Recursion

(A) Reduction...

Reduction: Reduce one problem to another

(B) Recursion...

Recursion

A special case of reduction

(A) reduce problem to a *smaller* instance of *itself*
(B) self-reduction

(C) Problem instance of size n is reduced to one or more instances of size $n-1$ or less.
(D) For termination, problem instances of small size are solved by some other method as ***base cases***.

8.4.0.18 Recursion in Algorithm Design

(A) ***Tail Recursion***: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Interval scheduling, MST algorithms, etc.
(B) ***Divide and Conquer***: Problem reduced to multiple ***independent*** sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

Examples: Closest pair, deterministic median selection, quick sort.

(C) ***Dynamic Programming***: problem reduced to multiple (typically) *dependent or overlapping* sub-problems. Use ***memoization*** to avoid recomputation of common solutions leading to *iterative bottom-up* algorithm.

8.5 Fibonacci Numbers

8.5.0.19 Fibonacci Numbers

(A) Fibonacci numbers defined by recurrence:

$$F(n) = F(n-1) + F(n-2) \text{ and } F(0) = 0, F(1) = 1.$$

(B) These numbers have many interesting and amazing properties.
A journal *The Fibonacci Quarterly*!

(C) $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $(1+\sqrt{5})/2 \simeq 1.618$.
(D) $\lim_{n\to\infty} F(n+1)/F(n) = \phi$

8.5.0.20 Recursive Algorithm for Fibonacci Numbers

Question: Given n, compute $F(n)$.

```
Fib(n):
        if (n = 0)
            return 0
        else if (n = 1)
            return 1
        else
            return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$T(n) = T(n-1) + T(n-2) + 1 \text{ and } T(0) = T(1) = 0$$

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

The number of additions is exponential in n. Can we do better?

8.5.0.21 An iterative algorithm for Fibonacci numbers

```
FibIter(n):
        if (n = 0) then
            return 0
        if (n = 1) then
            return 1
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] ⇐ F[i − 1] + F[i − 2]
        return F[n]
```

What is the running time of the algorithm? $O(n)$ additions.

8.5.0.22 What is the difference?

(A) Recursive algorithm is computing the same numbers again and again.
(B) Iterative algorithm is storing computed values and building bottom up the final value. **Memoization**.
(C) Dynamic programming...

Dynamic Programming: Finding a recursion that can be *effectively/efficiently* memoized.

(D) Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

8.5.0.23 Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (Fib(n) was previously computed)
            return stored value of Fib(n)
        else
            return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)

8.5.0.24 Automatic explicit memoization

Initialize table/array M of size n such that $M[i] = -1$ for $i = 0, \ldots, n$.

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (M[n] ≠ −1) (* M[n] has stored value of Fib(n) *)
            return M[n]
        M[n] ⇐ Fib(n − 1) + Fib(n − 2)
        return M[n]
```

Need to know upfront the number of subproblems to allocate memory

8.5.0.25 Automatic implicit memoization

Initialize a (dynamic) dictionary data structure D to empty

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (n is already in D)
            return value stored with n in D
        val ⇐ Fib(n − 1) + Fib(n − 2)
        Store (n, val) in D
        return val
```

8.5.0.26 Explicit vs Implicit Memoization

(A) Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.
(B) Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system.
  (A) Need to pay overhead of data-structure.
  (B) Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.

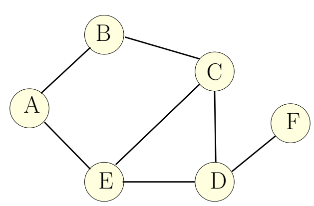
8.5.0.27 Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?
(A) input is n and hence input size is $\Theta(\log n)$
(B) output is $F(n)$ and output size is $\Theta(n)$. Why?
(C) Hence output size is exponential in input size so no polynomial time algorithm possible!
(D) Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
(E) Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.

8.6 Brute Force Search, Recursion and Backtracking

8.6.0.28 Maximum Independent Set in a Graph

Definition 8.6.1. *Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an **independent set** (also called a stable set) if for there are no edges between nodes in S. That is, if $u, v \in S$ then $(u, v) \notin E$.*

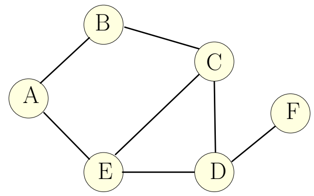


Some independent sets in graph above:

8.6.0.29 Maximum Independent Set Problem

Input Graph $G = (V, E)$

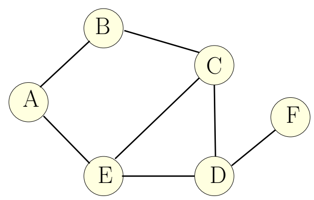
Goal Find maximum sized independent set in G



8.6.0.30 Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

Goal Find maximum weight independent set in G



8.6.0.31 Maximum Weight Independent Set Problem

(A) No one knows an *efficient* (polynomial time) algorithm for this problem.
(B) Problem is **NP-Complete** and it is *believed* that there is no polynomial time algorithm.
(C) Naive algorithm:

Brute-force algorithm: Try all subsets of vertices.

8.6.0.32 Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)
    Output max
```

Running time: suppose G has n vertices and m edges
(A) 2^n subsets of V
(B) checking each subset S takes $O(m)$ time
(C) total time is $O(m2^n)$

8.6.0.33 A Recursive Algorithm

(A) $V = \{v_1, v_2, \ldots, v_n\}$: vertices.
(B) For a vertex u let $N(u)$ be the of all neighboring vertics.
(C) We have that:

Observation 8.6.2. v_n*: Vertex in the graph.*
One of the following two cases is true

Case 1 v_n *is in* some *maximum independent set.*

Case 2 v_n *is in* no *maximum independent set.*

(D) Implementation:

```
RecursiveMIS(G):
    if G is empty then Output 0
    a = RecursiveMIS(G − v_n)
    b = w(v_n) + RecursiveMIS(G − v_n − N(v_n))
    Output max(a, b)
```

8.6.1 Recursive Algorithms

8.6.1.1 ..for Maximum Independent Set

(A) **Running time:**

$$T(n) = T(n-1) + T\Big(n - 1 - deg(v_n)\Big) + O(1 + deg(v_n))$$

(B) where $deg(v_n)$ is the degree of v_n. $T(0) = T(1) = 1$ is base case.
(C) Worst case is when $deg(v_n) = 0$ when the recurrence becomes

$$T(n) = 2T(n-1) + O(1)$$

(D) Solution to this is $T(n) = O(2^n)$.

8.6.1.2 Backtrack Search via Recursion

(A) Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem)
(B) Simple recursive algorithm computes/explores the whole tree blindly in some order.
(C) Backtrack search is a way to explore the tree intelligently to prune the search space
   (A) Some subproblems may be so simple that we can stop the recursive algorithm and solve it directly by some other method
   (B) Memoization to avoid recomputing same problem
   (C) Stop the recursion at a subproblem if it is clear that there is no need to explore further.
   (D) Leads to a number of heuristics that are widely used in practice although the worst case running time may still be exponential.

8.6.1.3 Example

Chapter 9

Dynamic Programming

OLD CS 473: Fundamental Algorithms, Spring 2015
February 17, 2015

9.1 Longest Increasing Subsequence

9.1.1 Longest Increasing Subsequence

9.1.1.1 Sequences

Definition 9.1.1. **Sequence**: *an ordered list* $a_1, a_2, \ldots, a_n$. **Length** *of a sequence is number of elements in the list.*

Definition 9.1.2. $a_{i_1}, \ldots, a_{i_k}$ *is a* **subsequence** *of* $a_1, \ldots, a_n$ *if* $1 \le i_1 < i_2 < \ldots < i_k \le n$.

Definition 9.1.3. *A sequence is* **increasing** *if* $a_1 < a_2 < \ldots < a_n$. *It is* **non-decreasing** *if* $a_1 \le a_2 \le \ldots \le a_n$. *Similarly* **decreasing** *and* **non-increasing**.

9.1.2 Sequences

9.1.2.1 Example...

Example 9.1.4. *(A) Sequence:* $6, 3, 5, 2, 7, 8, 1, 9$
(B) Subsequence of above sequence: $5, 2, 1$
(C) Increasing sequence: $3, 5, 9, 17, 54$
(D) Decreasing sequence: $34, 21, 7, 5, 1$
(E) Increasing subsequence of the first sequence: $2, 7, 9$.

9.1.2.2 Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an ***increasing subsequence*** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example 9.1.5. *(A) Sequence: 6, 3, 5, 2, 7, 8, 1*
(B) Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
(C) Longest increasing subsequence: 3, 5, 7, 8

9.1.2.3 Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array A

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|

    Output max
```

Running time: $O(n2^n)$.
2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

9.1.3 Recursive Approach: Take 1

9.1.3.1 LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

(A) **Case 1:** Does not contain $A[n]$ in which case LIS($A[1..n]$) = LIS($A[1..(n-1)]$)
(B) **Case 2:** contains $A[n]$ in which case LIS($A[1..n]$) is not so clear.

Observation 9.1.6. *if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller.*

9.1.3.2 Recursive Approach: Take 1

```
algLIS(A[1..n]):
    if (n = 0) then return 0
    m = algLIS(A[1..(n − 1)])
    B is subsequence of A[1..(n − 1)] with
            only elements less than A[n]
    (* let h be size of B, h ≤ n − 1 *)
    m = max(m, 1 + algLIS(B[1..h]))
    Output m
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(n)$.

Easy to see that $T(n)$ is $O(n2^n)$.

9.1.3.3 Recursive Approach: Take 2

LIS($A[1..n]$):
(A) **Case 1:** Does not contain $A[n]$ in which case LIS($A[1..n]$) = LIS($A[1..(n-1)]$)
(B) **Case 2:** contains $A[n]$ in which case LIS($A[1..n]$) is not so clear.

Observation 9.1.7. *(A) Case 2: find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than $A[n]$.*
(B) Generalization **LIS_smaller**($A[1..n], x$)*: longest increasing subsequence in A, all numbers in sequence is $\leq x$.*

9.1.3.4 Recursive Approach: Take 2

LIS_smaller($A[1..n], x$) : length of longest increasing subsequence in $A[1..n]$ with all numbers in subsequence less than x

```
LIS_smaller(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_smaller(A[1..(n-1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_smaller(A[1..(n-1)], A[n]))
    Output m
```

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(1)$.
Question: Is there any advantage?

9.1.3.5 Recursive Algorithm: Take 2

Observation 9.1.8. *The number of* different *subproblems generated by* **LIS_smaller**($A[1..n], x$) *is $O(n^2)$.*

Memoization the recursive algorithm leads to an $O(n^2)$ running time!
Question: What are the recursive subproblem generated by **LIS_smaller**($A[1..n], x$)?
(A) For $0 \leq i < n$ **LIS_smaller**($A[1..i], y$) where y is either x or one of $A[i+1], \ldots, A[n]$.

Observation 9.1.9. *Previous recursion also generates only $O(n^2)$ subproblems. Slightly harder to see.*

9.1.3.6 Recursive Algorithm: Take 3

Definition 9.1.10. **LISEnding**($A[1..n]$)*: length of longest increasing sub-sequence that* ends *in $A[n]$.*

Question: can we obtain a recursive expression?

$$\textbf{LISEnding}(A[1..n]) = \max_{i:A[i]<A[n]} ([)]1 + \textbf{LISEnding}(A[1..i])$$

9.1.3.7 Recursive Algorithm: Take 3

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))

    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1...i])
```

Question: How many distinct subproblems generated by **LIS_ending_alg**$(A[1..n])$? n.

9.1.3.8 Iterative Algorithm via Memoization

Compute the values **LIS_ending_alg**$(A[1..i])$ iteratively in a bottom up fashion.

```
LIS_ending_alg(A[1..n]):
    Array L[1..n]  (* L[i] = value of LIS_ending_alg(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L
```

```
LIS(A[1..n]):
        L = LIS_ending_alg(A[1..n])
        return the maximum value in L
```

9.1.3.9 Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion
Running time: $O(n^2)$, **Space:** $\Theta(n)$

9.1.3.10 Example

Example 9.1.11. *(A) Sequence: 6, 3, 5, 2, 7, 8, 1*
(B) Longest increasing subsequence: 3, 5, 7, 8

(A) $L[i]$ is value of longest increasing subsequence ending in $A[i]$
(B) Recursive algorithm computes $L[i]$ from $L[1]$ to $L[i-1]$
(C) Iterative algorithm builds up the values from $L[1]$ to $L[n]$

9.1.3.11 Memoizing LIS_smaller

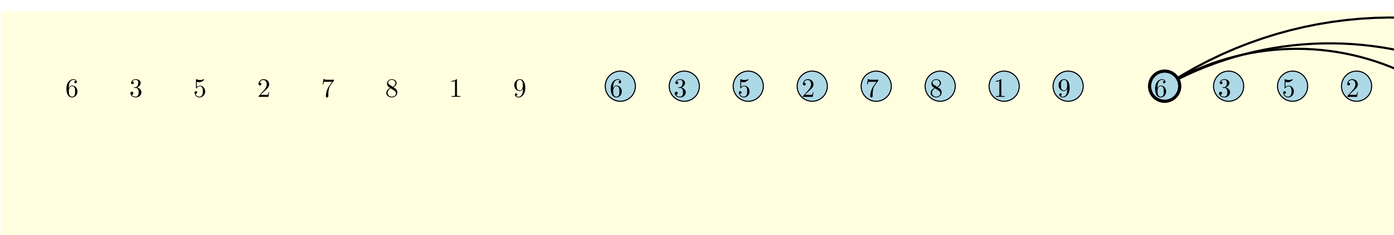
```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *)
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i-1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i-1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)
Running time: $O(n^2)$, **Space:** $\Theta(n^2)$

9.1.4 Longest increasing subsequence

9.1.4.1 Another way to get quadratic time algorithm

Input sequence: $6, 3, 5, 2, 7, 8, 1, 9$.



Longest increasing subsequence: $3, 5, 7, 8, 9$.

9.1.5 Longest increasing subsequence

9.1.5.1 Another way to get quadratic time algorithm

(A) $G = (\{s, 1, \dots, n\}, \{\})$: directed graph.
  (A) $\forall i, j$: If $i < j$ and $A[i] < A[j]$ then
    add the edge $i \to j$ to G.
  (B) $\forall i$: Add $s \to i$.
(B) The graph G is a DAG. LIS corresponds to longest path in G starting at s.
(C) We know how to compute this in $O(|V(G)| + |E(G)|) = O(n^2)$.

(D) Comment: One can compute LIS in $O(n \log n)$ time with a bit more work.

9.1.5.2 Dynamic Programming

(A) Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
(B) Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.
(C) Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation. This leads to an explicit algorithm.
(D) Optimize the resulting algorithm further

9.2 Weighted Interval Scheduling

9.2.1 Weighted Interval Scheduling

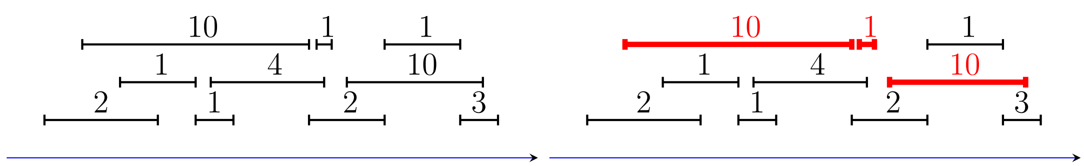
9.2.2 The Problem

9.2.2.1 Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

(A) Two jobs with overlapping intervals cannot both be scheduled!



9.2.3 Greedy Solution

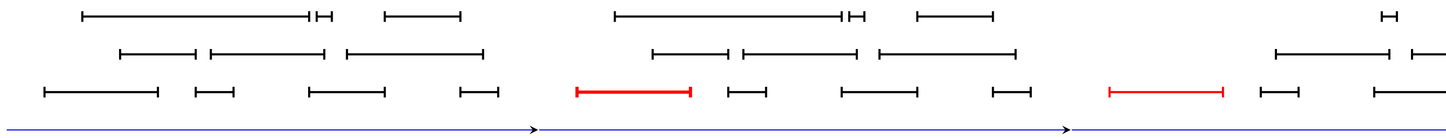
9.2.4 Interval Scheduling

9.2.4.1 Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

Goal Schedule as many jobs as possible.

(A) Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).



9.2.4.2 Greedy Strategies

(A) Earliest finish time first
(B) Largest weight/profit first
(C) Largest weight to length ratio first
(D) Shortest length first
(E) ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don't work!

9.2.5 Reduction to...

9.2.5.1 Max Weight Independent Set Problem

(A) Given weighted interval scheduling instance I create an instance of max weight independent set on a graph $G(I)$ as follows.
  (A) For each interval i create a vertex v_i with weight w_i.
  (B) Add an edge between v_i and v_j if i and j overlap.
(B) **Claim:** max weight independent set in $G(I)$ has weight equal to max weight set of intervals in I that do not overlap

9.2.6 Reduction to...

9.2.6.1 Max Weight Independent Set Problem

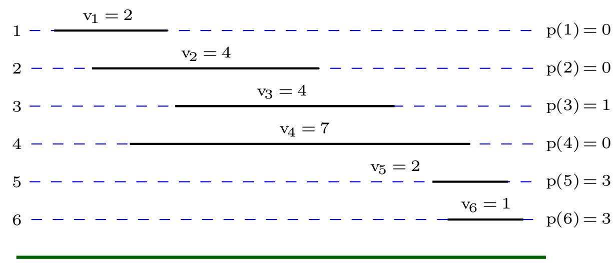
(A) There is a reduction from **Weighted Interval Scheduling** to **Independent Set**.
(B) Can use structure of original problem for efficient algorithm?
(C) **Independent Set** in general is NP-Complete.

We do not know an efficient (polynomial time) algorithm for independent set! Can we take advantage of the interval structure to find an efficient algorithm?

9.2.7 Recursive Solution

9.2.7.1 Conventions

Definition 9.2.1. *(A) Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \le f_j$*
(B) Define $p(j)$ to be the largest i (less than j) such that job i and job j are not in conflict

Example 9.2.2. 

9.2.7.2 Towards a Recursive Solution

Observation 9.2.3. *Consider an optimal schedule $\mathcal{O}$*

[¡+-¿]

Case $n \in \mathcal{O}$: *None of the jobs between n and $p(n)$ can be scheduled. Moreover $\mathcal{O}$ must contain an optimal schedule for the first $p(n)$ jobs.*

Case $n \notin \mathcal{O}$: *$\mathcal{O}$ is an optimal schedule for the first $n-1$ jobs.*

9.2.7.3 A Recursive Algorithm

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule(n):
    if n = 0 then return 0
    if n = 1 then return w(v_1)
    O_p(n) ← Schedule(p(n))
    O_{n-1} ← Schedule(n − 1)
    if (O_p(n) + w(v_n) < O_{n-1}) then
        O_n = O_{n-1}
    else
        O_n = O_p(n) + w(v_n)
    return O_n
```

Time Analysis Running time is $T(n) = T(p(n)) + T(n-1) + O(1)$ which is ...

9.2.7.4 Bad Example

Running time on this instance is

$$T(n) = T(n-1) + T(n-2) + O(1) = \Theta(\phi^n)$$

where $\phi \approx 1.618$ is the golden ratio.

(Because... $T(n)$ is the n Fibonacci number.)

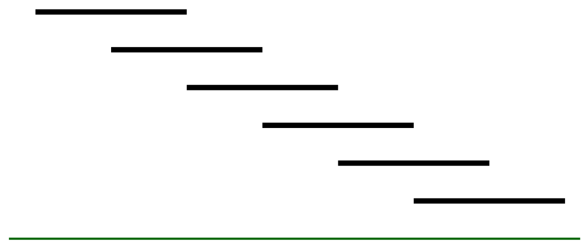


Figure 9.1: Bad instance for recursive algorithm

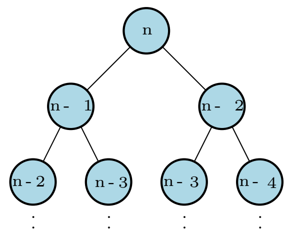


Figure 9.2: Label of node indicates size of sub-problem. Tree of sub-problems grows very quickly

9.2.7.5 Analysis of the Problem

9.2.8 Dynamic Programming

9.2.8.1 Memo(r)ization

Observation 9.2.4. *(A) Number of different sub-problems in recursive algorithm is $O(n)$; they are $O_1, O_2, \ldots, O_{n-1}$*
(B) Exponential time is due to recomputation of solutions to sub-problems

Solution Store optimal solution to different sub-problems, and perform recursive call **only** if not already computed.

9.2.8.2 Recursive Solution with Memoization

> **schdIMem**(j)
>   **if** $j = 0$ **then return** 0
>   **if** $M[j]$ `is defined then (* sub-problem already solved *)`
>   **return** $M[j]$
>   **if** $M[j]$ `is not defined` **then**
> $M[j] = max\Big(w(v_j) + \textbf{schdIMem}(p(j)), \quad \textbf{schdIMem}(j-1)\Big)$
>   **return** $M[j]$

Time Analysis

¡+-¿ Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one $M[\cdot]$

¡+-¿ Initially no entry of $M[]$ is filled; at the end all entries of $M[]$ are filled

¡+-¿ So total time is $O(n)$ (Assuming input is presorted...)

9.2.8.3 Automatic Memoization

Fact Many functional languages (like LISP) automatically do memoization for recursive function calls!

9.2.8.4 Back to Weighted Interval Scheduling

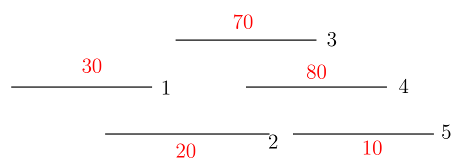
Iterative Solution

$M[0] = 0$
for $i = 1$ `to` n **do**
    $M[i] = \max\Big(w(v_i) + M[p(i)], M[i-1]\Big)$

M: table of subproblems

(A) Implicitly dynamic programming fills the values of M.
(B) Recursion determines order in which table is filled up.
(C) Think of decomposing problem first (recursion) and then worry about setting up table — this comes naturally from recursion.

9.2.8.5 Example



$p(5) = 2$, $p(4) = 1$, $p(3) = 1$, $p(2) = 0$, $p(1) = 0$

9.2.9 Computing Solutions

9.2.9.1 Computing Solutions + First Attempt

(A) Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max( w(v_i) + M[p(i)], M[i-1] )
    if w(v_i) + M[p(i)] < M[i-1] then
        S[i] = S[i-1]
    else
        S[i] = S[p(i)] ∪ {i}
```

(B) Naïvely updating $S[]$ takes $O(n)$ time
(C) Total running time is $O(n^2)$
(D) Using pointers and linked lists running time can be improved to $O(n)$.

9.2.9.2 Computing Implicit Solutions

Observation 9.2.5. *Solution can be obtained from $M[]$ in $O(n)$ time, without any additional information*

```
findSolution( j )
    if (j = 0) then return empty schedule
    if (v_j + M[p(j)] > M[j-1]) then
        return findSolution(p(j)) ∪ {j}
    else
        return findSolution(j-1)
```

Makes $O(n)$ recursive calls, so **findSolution** *runs in $O(n)$ time.*

9.2.9.3 Computing Implicit Solutions

A generic strategy for computing solutions in dynamic programming:

(A) Keep track of the *decision* in computing the optimum value of a sub-problem. decision space depends on recursion
(B) Once the optimum values are computed, go back and use the decision values to compute an optimum solution.

Question: What is the decision in computing $M[i]$?

A: Whether to include i or not.

9.2.9.4 Computing Implicit Solutions

```
    M[0] = 0
    for i = 1 to n do
        M[i] = max(v_i + M[p(i)], M[i-1])
        if (v_i + M[p(i)] > M[i-1]) then
            Decision[i] = 1 (* 1:  i included in solution M[i] *)
        else
            Decision[i] = 0 (* 0:  i not included in solution M[i] *)

    S = ∅,  i = n
    while (i > 0) do
        if (Decision[i] = 1) then
            S = S ∪ {i}
            i = p(i)
        else
            i = i - 1
return S
```

Chapter 10

More Dynamic Programming

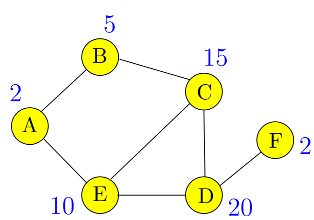
OLD CS 473: Fundamental Algorithms, Spring 2015
February 19, 2015

10.1 Maximum Weighted Independent Set in Trees

10.1.0.5 Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in G



Maximum weight independent set in above graph: $\{B, D\}$

10.1.0.6 Maximum Weight Independent Set in a Tree

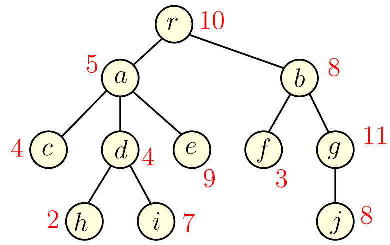
Input Tree $T = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in T

Maximum weight independent set in above tree: ??

10.1.0.7 Towards a Recursive Solution

(A) For an arbitrary graph G:
  (A) Number vertices as $v_1, v_2, \ldots, v_n$
  (B) Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).



- (C) Saw that if graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

(B) What about a tree?

(C) Natural candidate for v_n is root r of T?

10.1.0.8 Towards a Recursive Solution

(A) Natural candidate for v_n is root r of T?

(B) Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

Case $r \in \mathcal{O}$: None of the children of r can be in $\mathcal{O}$. $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r.

(C) Subproblems?

(D) Subtrees of T hanging at nodes in T.

10.1.0.9 A Recursive Solution

(A) $T(u)$: subtree of T hanging at node u.

(B) $OPT(u)$: max weighted independent set value in $T(u)$.

(C) $OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$

10.1.0.10 Iterative Algorithm

(A) Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u

(B) What is an ordering of nodes of a tree T to achieve above?

(C) Post-order traversal of a tree.

10.1.0.11 Iterative Algorithm

(A) Code:

MIS-Tree(T):
 Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
 for $i = 1$ to n **do**

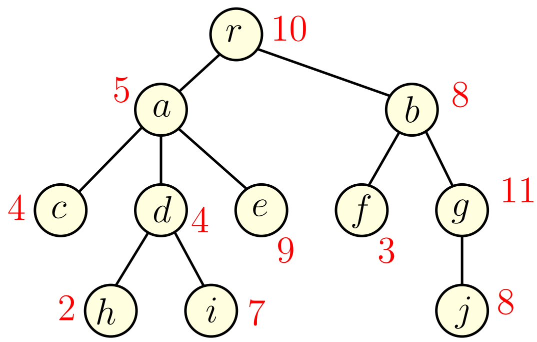
$$M[v_i] = \max \begin{pmatrix} \sum_{v_j \texttt{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \texttt{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

 return $M[v_n]$ (* Note: v_n is the root of T *)

(B) **Space:** $O(n)$ to store the value at each node of T.

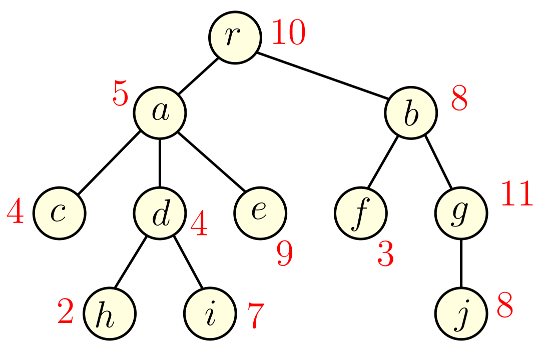
(C) **Running time:**
 (A) Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
 (B) Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

10.1.0.12 Example



10.1.0.13 Dominating set

Definition 10.1.1. $G = (V, E)$. *The set $X \subseteq V$ is a* **dominating set**, *if any vertex $v \in V$ is either in X or is adjacent to a vertex in X.*



Problem 10.1.2. *Given weights on vertices, compute the* **minimum** *weight dominating set in G.*

Dominating Set is **NP-Hard**!

10.2 DAGs and Dynamic Programming

10.2.0.14 Recursion and DAGs

Observation 10.2.1. *Let A be a recursive algorithm for problem Π. For each instance I of Π there is an associated* DAG *$G(I)$.*

(A) Create directed graph $G(I)$ as follows...
(B) For each sub-problem in the execution of A on I create a node.
(C) If sub-problem v *depends* on or *recursively calls* sub-problem u add *directed* edge (u, v) to graph.
(D) $G(I)$ is a DAG. Why? If $G(I)$ has a cycle then A will not terminate on I.

10.2.1 Iterative Algorithm for...

10.2.1.1 Dynamic Programming and DAGs

Observation 10.2.2. *An iterative algorithm B obtained from a recursive algorithm A for a problem Π does the following:*

> *For each instance I of Π, it computes a topological sort of $G(I)$ and evaluates sub-problems according to the topological ordering.*

(A) Sometimes the DAG $G(I)$ can be obtained directly without thinking about the recursive algorithm A
(B) In some cases (**not all**) the computation of an optimal solution reduces to a shortest/longest path in DAG $G(I)$
(C) Topological sort based shortest/longest path computation is dynamic programming!

10.2.2 A quick reminder...

10.2.2.1 A Recursive Algorithm for weighted interval scheduling

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule(n):
        if n = 0 then return 0
        if n = 1 then return w(v_1)
        O_p(n) ←Schedule(p(n))
        O_{n-1} ←Schedule(n - 1)
        if (O_p(n) + w(v_n) < O_{n-1}) then
            O_n = O_{n-1}
        else
            O_n = O_p(n) + w(v_n)
        return O_n
```

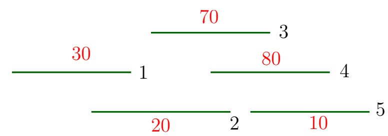
10.2.3 Weighted Interval Scheduling via...

10.2.3.1 Longest Path in a DAG

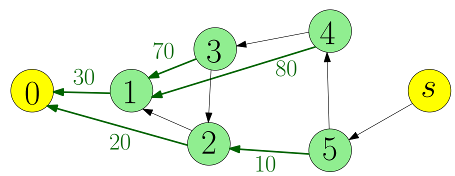
Given intervals, create a DAG as follows:
(A) Create one node for each interval, plus a dummy sink node 0 for interval 0, plus a dummy source node s.

(B) For each interval i add edge $(i, p(i))$ of the length/weight of v_i.
(C) Add an edge from s to n of length 0.
(D) For each interval i add edge $(i, i-1)$ of length 0.

10.2.3.2 Example



$p(5) = 2$, $p(4) = 1$, $p(3) = 1$, $p(2) = 0$, $p(1) = 0$



10.2.3.3 Relating Optimum Solution

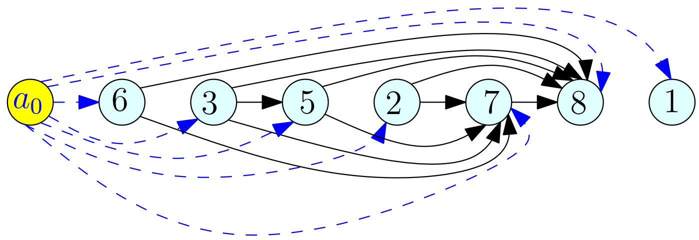
(A) Given interval problem instance I let $G(I)$ denote the DAG constructed as described.
(B) We have...

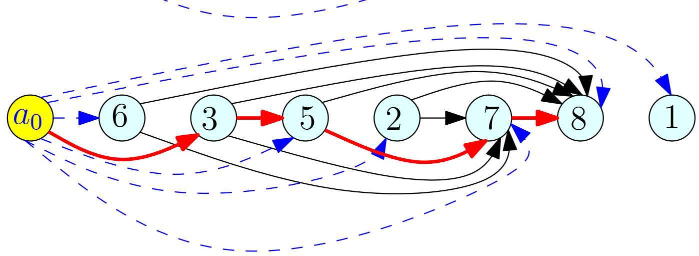
Claim 10.2.3. *Optimum solution to weighted interval scheduling instance I is given by* longest *path from s to 0 in $G(I)$.*

(C) Assuming claim is true,
  (A) If I has n intervals, DAG $G(I)$ has $n+2$ nodes and $O(n)$ edges. Creating $G(I)$ takes $O(n \log n)$ time: to find $p(i)$ for each i. How?
  (B) Longest path can be computed in $O(n)$ time — recall $O(m+n)$ algorithm for shortest/longest paths in DAGs.

10.2.3.4 DAG for Longest Increasing Sequence

Given sequence $a_1, a_2, \ldots, a_n$ create DAG as follows:
(A) add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
(B) for each i there is a node v_i
(C) if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
(D) find longest path from v_0





10.3 Edit Distance and Sequence Alignment

10.3.0.5 Spell Checking Problem

(A) Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?
(B) What does nearness mean?
(C) **Question:** Given two strings $x_1 x_2 \dots x_n$ and $y_1 y_2 \dots y_m$ what is a *distance* between them?
(D) **Edit Distance**: minimum number of "edits" to transform x into y.

10.3.0.6 Edit Distance

Definition 10.3.1. ***Edit distance*** *between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X.*

Example 10.3.2. *The edit distance between FOOD and MONEY is at most* 4*:*

$$\underline{\mathrm{F}}\mathrm{OOD} \to \mathrm{MO}\underline{\mathrm{O}}\mathrm{D} \to \mathrm{MON}\underline{\mathrm{O}}\mathrm{D} \to \mathrm{MONE}\underline{\mathrm{D}} \to \mathrm{MONEY}$$

10.3.0.7 Edit Distance: Alternate View

Alignment Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F	O	O		D
M	O	N	E	Y

Formally, an **alignment** is a set M of pairs (i, j) such that each index appears at most once, and there is no "crossing": $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1,1), (2,2), (3,3), (4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

10.3.0.8 Edit Distance Problem

Problem Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

10.3.0.9 Applications

(A) Spell-checkers and Dictionaries
(B) Unix `diff`
(C) DNA sequence alignment … but, we need a new metric

10.3.0.10 Similarity Metric

Definition 10.3.3. *For two strings X and Y, the cost of alignment M is*
*(A) [**Gap penalty**] For each gap in the alignment, we incur a cost δ.*
*(B) [**Mismatch cost**] For each pair p and q that have been matched in M, we incur cost α_{pq}; typically $\alpha_{pp} = 0$.*
Edit distance is special case when $\delta = \alpha_{pq} = 1$.

10.3.0.11 An Example

Example 10.3.4.

$$\begin{array}{c|c|c|c|c|c|c|c|c|c|} o & & c & u & r & r & a & n & c & e \\ o & c & c & u & r & r & e & n & c & e \end{array} \qquad Cost \;= \delta + \alpha_{ae}$$

Alternative:

$$\begin{array}{c|c|c|c|c|c|c|c|c|c|c} o & & c & u & r & r & & a & n & c & e \\ o & c & c & u & r & r & e & & n & c & e \end{array} \qquad Cost \;= 3\delta$$

Or a really stupid solution (delete string, insert other string):

$$\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} o & c & u & r & r & a & n & c & e & & & & & & & & & & \\ & & & & & & & & & o & c & c & u & r & r & e & n & c & e \end{array}$$

Cost $= 19\delta$.

10.3.0.12 Sequence Alignment

Input Given two words X and Y, and gap penalty δ and mismatch costs α_{pq}

Goal Find alignment of minimum cost

10.3.1 Edit distance

10.3.1.1 Basic observation

(A) Let $X = \alpha x$ and $Y = \beta y$.
(B) α, β: strings.x and y single characters.
(C) Optimal edit distance between X and Y as alignment. Consider last column of alignment of the two strings:

$$\begin{array}{|c||c|} \hline \alpha & x \\ \hline \beta & y \\ \hline \end{array} \quad \text{or} \quad \begin{array}{|c||c|} \hline \alpha & x \\ \hline \beta y & \\ \hline \end{array} \quad \text{or} \quad \begin{array}{|c||c|} \hline \alpha x & \\ \hline \beta & y \\ \hline \end{array}$$

(D) **Observation 10.3.5.** *Prefixes must have optimal alignment!*

10.3.1.2 Problem Structure

Observation 10.3.6. *Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$. If (m, n) are not matched then either the mth position of X remains unmatched or the nth position of Y remains unmatched.*

(A) **Case** x_m and y_n are matched.
  (A) Pay mismatch cost $\alpha_{x_m y_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$
(B) **Case** x_m is unmatched.
  (A) Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$
(C) **Case** y_n is unmatched.
  (A) Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

10.3.1.3 Subproblems and Recurrence

Optimal Costs Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\text{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base Cases: $\text{Opt}(i, 0) = \delta \cdot i$ and $\text{Opt}(0, j) = \delta \cdot j$

10.3.1.4 Dynamic Programming Solution

for `all` i **do** $M[i, 0] = i\delta$
for `all` j **do** $M[0, j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i, j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

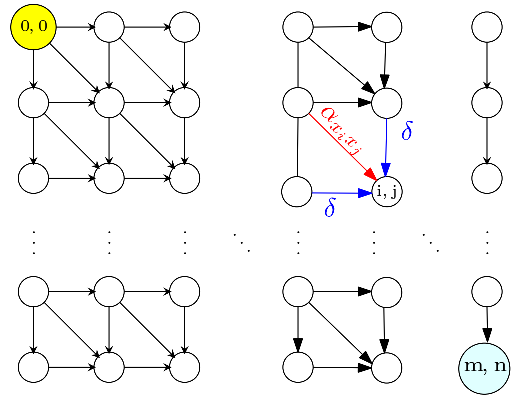


Figure 10.1: Iterative algorithm in previous slide computes values in row order. Optimal value is a shortest path from $(0,0)$ to (m,n) in DAG.

(A) Running time is $O(mn)$.
(B) Space used is $O(mn)$.

10.3.1.5 Matrix and DAG of Computation

10.3.1.6 Sequence Alignment in Practice

(A) Typically the DNA sequences that are aligned are about 10^5 letters long!
(B) So about 10^{10} operations and 10^{10} bytes needed
(C) The killer is the 10GB storage
(D) Can we reduce space requirements?

10.3.1.7 Optimizing Space

(A) Recall

$$M(i,j) = \min \begin{cases} \alpha_{x_i y_j} + M(i-1,j-1), \\ \delta + M(i-1,j), \\ \delta + M(i,j-1) \end{cases}$$

(B) Entries in jth column only depend on $(j-1)$st column and earlier entries in jth column
(C) Only store the current column and the previous column reusing space; $N(i,0)$ stores $M(i,j-1)$ and $N(i,1)$ stores $M(i,j)$

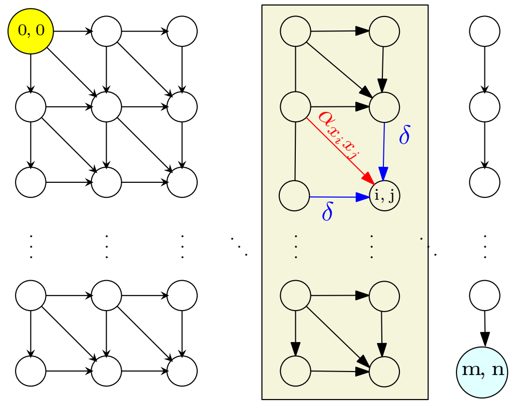


Figure 10.2: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.

10.3.1.8 Computing in column order to save space

10.3.1.9 Space Efficient Algorithm

for `all` i **do** $N[i, 0] = i\delta$
for $j = 1$ `to` n **do**
    $N[0, 1] = j\delta$ `(* corresponds to` $M(0, j)$ `*)`
    for $i = 1$ `to` m **do**

$$N[i, 1] = \min \begin{cases} \alpha_{x_i y_j} + N[i-1, 0] \\ \delta + N[i-1, 1] \\ \delta + N[i, 0] \end{cases}$$

    for $i = 1$ `to` m **do**
        `Copy` $N[i, 0] = N[i, 1]$

Analysis Running time is $O(mn)$ and space used is $O(2m) = O(m)$

10.3.1.10 Analyzing Space Efficiency

(A) From the $m \times n$ matrix M we can construct the actual alignment (exercise)
(B) Matrix N computes cost of optimal alignment but no way to construct the actual alignment
(C) Space efficient computation of alignment? More complicated algorithm — see text book.

10.3.1.11 Takeaway Points

(A) Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.
(B) Given a recursive algorithm there is a natural DAG associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this DAG.
(C) The space required to evaluate the answer can be reduced in some cases by a careful examination of that dependency DAG of the subproblems and keeping only a subset of

the DAG at any time.

Chapter 11

More Dynamic Programming

OLD CS 473: Fundamental Algorithms, Spring 2015
February 26, 2015

11.1 All Pairs Shortest Paths

11.1.0.12 Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Given nodes s, t find shortest path from s to t.
(B) Given node s find shortest path from s to all other nodes.
(C) Find shortest paths for all pairs of nodes.

11.1.0.13 Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Given nodes s, t find shortest path from s to t.
(B) Given node s find shortest path from s to all other nodes.

Dijkstra's algorithm for non-negative edge lengths. Running time: $O((m+n)\log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $O(nm)$.

11.1.0.14 All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Find shortest paths for all pairs of nodes.
(A) Apply single-source algorithms n times, once for each vertex.
(B) Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
(C) Arbitrary edge lengths: $O(n^2 m)$.
$\Theta(n^4)$ if $m = \Omega(n^2)$.
(D) **Q:** Can we do better?

11.1.0.15 Shortest Paths and Recursion

(A) Compute the shortest path distance from s to t recursively?
(B) What are the smaller sub-problems?
(C) ***Prefix property***:

Lemma 11.1.1. *Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is shortest path from s to v_k then for $1 \le i < k$:*
(A) $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is shortest path from s to v_i

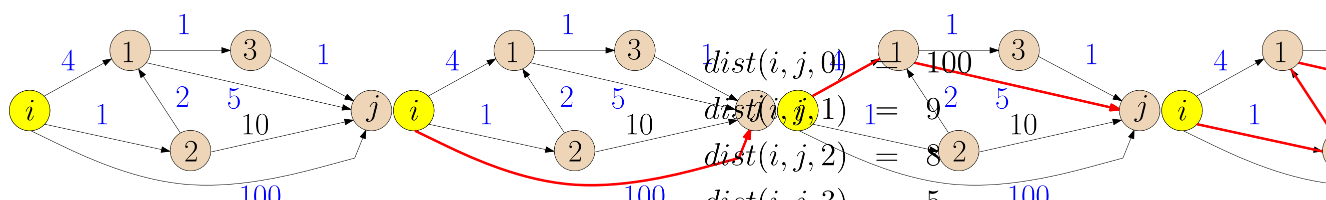
(D) Sub-problem idea: paths of fewer hops/edges

11.1.0.16 Hop-based Recur': Single-Source Shortest Paths

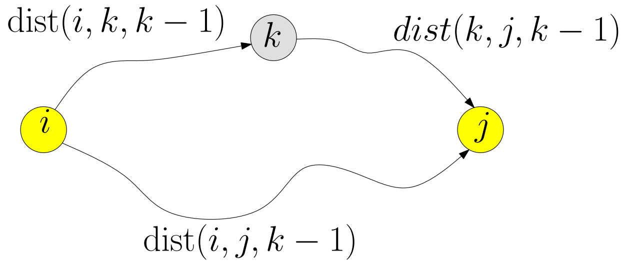
(A) Single-source problem: fix source s.
(B) $OPT(v, k)$**:** shortest path dist. from s to v using at most k edges.
(C) Note: $dist(s, v) = OPT(v, n-1)$. Recursion for $OPT(v, k)$:
(D) $OPT(v,k) = \min \begin{cases} \min_{u \in V}(OPT(u, k-1) + c(u, v)). \\ OPT(v, k-1) \end{cases}$
Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise ∞
(E) Leads to Bellman-Ford algorithm — see text book.
(F) $OPT(v, k)$ values are also of independent interest: shortest paths with at most k hops.

11.1.0.17 All-Pairs: Recursion on index of intermediate nodes

(A) Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
(B) $dist(i, j, k)$: shortest path distance between v_i and v_j among all paths in which the largest index of an *intermediate node* is at most k



11.1.0.18 All-Pairs: Recursion on index of intermediate nodes



$$dist(i,j,k) = \min \begin{cases} dist(i,j,k-1) \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

Base case: $dist(i,j,0) = c(i,j)$ if $(i,j) \in E$, otherwise ∞
Correctness: If $i \to j$ shortest path goes through k then k occurs only once on the path — otherwise there is a negative length cycle.

11.1.1 Floyd-Warshall Algorithm

11.1.1.1 for All-Pairs Shortest Paths

```
Check if G has a negative cycle // Bellman-Ford: O(mn) time
if there is a negative cycle then return ``Negative cycle''

for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = c(i, j) (* c(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min { dist(i, j, k − 1),
                                  dist(i, k, k − 1) + dist(k, j, k − 1)
```

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$, **Space:** $\Theta(n^3)$.

11.1.2 Floyd-Warshall Algorithm

11.1.2.1 for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

```
for i = 1 to n do
   for j = 1 to n do
        dist(i,j,0) = c(i,j)   (* c(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)
not edge, 0 if i = j *)

for k = 1 to n do
   for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min(dist(i,j,k-1), dist(i,k,k-1) + dist(k,j,k-1))

for i = 1 to n do
   if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Correctness: exercise

11.1.2.2 Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

(A) Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices

(B) With array Next, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

11.1.3 Floyd-Warshall Algorithm

11.1.3.1 Finding the Paths

```
for i = 1 to n do
   for j = 1 to n do
        dist(i,j,0) = c(i,j) (* c(i,j) = ∞ if (i,j) not edge, 0 if i = j *)
        Next(i,j) = -1
for k = 1 to n do
   for i = 1 to n do
        for j = 1 to n do
            if (dist(i,j,k-1) > dist(i,k,k-1) + dist(k,j,k-1)) then
                dist(i,j,k) = dist(i,k,k-1) + dist(k,j,k-1)
                Next(i,j) = k

for i = 1 to n do
   if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Exercise: Given $Next$ array and any two vertices i, j describe an $O(n)$ algorithm to find a i-j shortest path.

11.1.3.2 Summary of results on shortest paths

Single vertex		
No negative edges	Dijkstra	$O(n \log n + m)$
Edges cost might be negative But no negative cycles	Bellman Ford	$O(nm)$

All Pairs Shortest Paths	No negative edges	n * Dijkstra	$O(n^2 \log n + nm)$
	No negative cycles	n * Bellman Ford	$O(n^2 m) = O(n^4)$
	No negative cycles	Floyd-Warshall	$O(n^3)$

11.2 Knapsack

11.2.0.3 Knapsack Problem

Input Given a Knapsack of capacity W lbs. and n objects with ith object having weight w_i and value v_i; assume W, w_i, v_i are all positive integers

Goal Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.

11.2.0.4 Knapsack Example

Example 11.2.1.

Item	I_1	I_2	I_3	I_4	I_5
Value	*1*	*6*	*18*	*22*	*28*
Weight	*1*	*2*	*5*	*6*	*7*

If $W = 11$*, the best is* $\{I_3, I_4\}$ *giving value 40.*

Special Case When $v_i = w_i$, the Knapsack problem is called the **Subset Sum Problem**.

11.2.0.5 Greedy Approach

(A) Pick objects with greatest value
   (A) Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
(B) Pick objects with smallest weight
   (A) Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick $\{1\}$, but the optimal is $\{2\}$
(C) Pick objects with largest v_i/w_i ratio
   (A) Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
   (B) Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to W.

11.2.0.6 Towards a Recursive Solution

First guess: $\mathrm{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

Observation 11.2.2. *Consider an optimal solution $\mathcal{O}$ for $1, \ldots, i$*

Case item $i \notin \mathcal{O}$ *$\mathcal{O}$ is an optimal solution to items 1 to $i-1$*

Case item $i \in \mathcal{O}$ *Then $\mathcal{O} - \{i\}$ is an optimum solution for items 1 to $n-1$ in knapsack of capacity $W - w_i$.*
Subproblems depend also on remaining *capacity. Cannot write subproblem only in terms of* $\mathrm{Opt}(1), \ldots, \mathrm{Opt}(i-1)$.

$\mathrm{Opt}(i, w)$: optimum profit for items 1 to i in knapsack of size w
Goal: compute $\mathrm{Opt}(n, W)$

11.2.0.7 Dynamic Programming Solution

Definition 11.2.3. *Let $\mathrm{Opt}(i, w)$ be the optimal way of picking items from 1 to i, with total weight not exceeding w.*

$$\mathrm{Opt}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \mathrm{Opt}(i-1, w) & \text{if } w_i > w \\ \max \begin{cases} \mathrm{Opt}(i-1, w) \\ \mathrm{Opt}(i-1, w - w_i) + v_i \end{cases} & \text{otherwise} \end{cases}$$

11.2.0.8 An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max(M[i-1, w], M[i-1, w-w_i] + v_i)
```

Running Time

(A) Time taken is $O(nW)$

(B) Input has size $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but “pseudo-polynomial”!

11.2.0.9 Knapsack Algorithm and Polynomial time

(A) Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
(B) Running time of dynamic programming algorithm: $O(nW)$.
(C) Not a polynomial time algorithm.
(D) Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
(E) Algorithm is called a ***pseudo-polynomial*** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the ***combinatorial size*** of problem.
(F) Knapsack is **NP-Hard** if numbers are not polynomial in n.

11.3 Traveling Salesman Problem

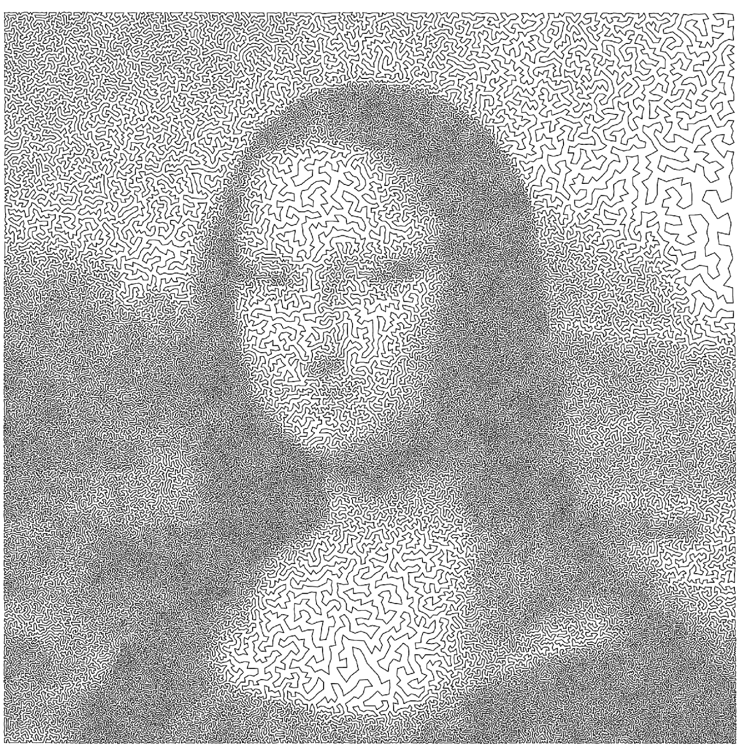
11.3.0.10 Traveling Salesman Problem

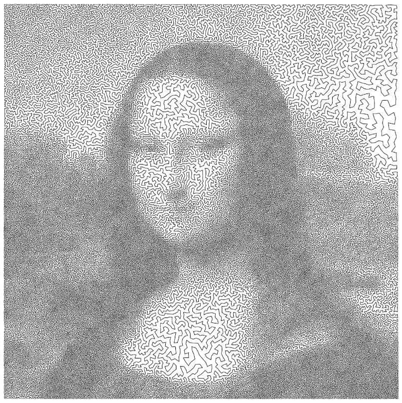
Input A graph $G = (V, E)$ with non-negative edge costs/lengths. $c(e)$ for edge e

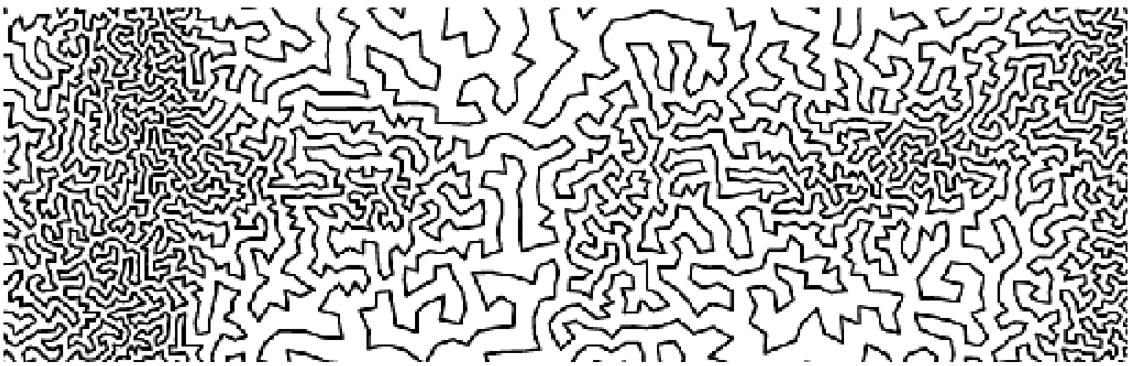
Goal Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is **NP-Hard**.

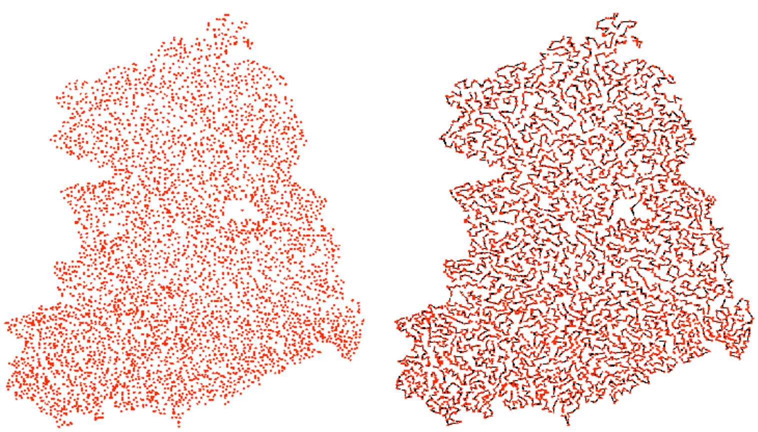
11.3.0.11 Drawings using TSP







11.3.0.12 Example: optimal tour for cities of a country (which one?)



11.3.0.13 An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?

11.3.0.14 Towards a Recursive Solution

(A) Order vertices as $v_1, v_2, \ldots, v_n$

(B) $OPT(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to S. Want $OPT(V)$.

Can we compute $OPT(S)$ recursively?

(A) Say $v \in S$. What are the two neighbors of v in optimum tour in S?

(B) If u, w are neighbors of v in an optimum tour of S then removing v gives an optimum *path* from u to w visiting all nodes in $S - \{v\}$.

Path from u to w is not a recursive subproblem! Need to find a more general problem to allow recursion.

11.3.0.15 A More General Problem: TSP Path

Input A graph $G = (V, E)$ with non-negative edge costs/lengths($c(e)$ for edge e) and two nodes s, t

Goal Find a path from s to t of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

(A) $OPT(u, v, S)$**:** optimum TSP Path from u to v in the graph restricted to S (here $u, v \in S$).

11.3.1 A More General Problem: TSP Path

11.3.1.1 Continued...

What is the next node in the optimum path from u to v? Suppose it is w. Then what is $OPT(u, v, S)$?

$$OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$$

We do not know w! So try all possibilities for w.

11.3.1.2 A Recursive Solution

$OPT(u, v, S) = \min_{w \in S, w \neq u, v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)$

What are the subproblems for the original problem $OPT(s, t, V)$?

$OPT(u, v, S)$ for $u, v \in S$, $S \subseteq V$.

How many subproblems?

(A) number of distinct subsets S of V is at most 2^n

(B) number of pairs of nodes in a set S is at most n^2

(C) hence number of subproblems is $O(n^2 2^n)$

Exercise: Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Disadvantage of dynamic programming solution: memory!

11.3.1.3 Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

(A) How to come up with the recursion?
(B) How to recognize that dynamic programming may apply?

11.3.1.4 Some Tips

(A) Problems where there is a *natural* linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
(B) Problems involving trees: recursion based on subtrees.
(C) More generally:
    (A) Problem admits a natural recursive divide and conquer
    (B) If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
    (C) If optimal solution depends on all pieces then can apply dynamic programming if *interface/interaction* between pieces is *limited*. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.

11.3.1.5 Examples

(A) Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
(B) Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
(C) Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the sutrees?
(D) Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!
(E) Knapsack: Split items into two sets of half each. What is the interaction?

Chapter 12

Greedy Algorithms

OLD CS 473: Fundamental Algorithms, Spring 2015
March 3, 2015

12.1 Problems and Terminology

12.2 Problem Types

12.2.0.6 Problem Types

(A) **Decision Problem**: Is the input a YES or NO input?
Example: Given graph G, nodes s, t, is there a path from s to t in G?
(B) **Search Problem:** Find a *solution* if input is a YES input.
Example: Given graph G, nodes s, t, find an s-t path.
(C) **Optimization Problem:** Find a *best* solution among all solutions for the input.
Example: Given graph G, nodes s, t, find a shortest s-t path.

12.2.0.7 Terminology

(A) A **problem** Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an **instance**.
(B) The **size** of an instance I is the number of bits in its representation.
(C) I: instance. $sol(I)$: set of **feasible solutions** to I.
(D) *Implicit assumption:* given I, $y \in \Sigma^*$, one can check (efficiently!) if $y \in sol(I)$.
(E) $\implies$ Problem is in **NP**. (More on this later in the course.)
(F) Optimization problems:
$\forall$ solution $s \in sol(I)$ has associated **value**.
(G) *Implicit assumption:* given s, can compute value efficiently.

12.2.0.8 Problem Types

Given instance I...
(A) **Decision Problem**: Output whether $sol(I) = \emptyset$ or not.

(B) **Search Problem:** Compute solution $s \in sol(I)$ if $sol(I) \neq \emptyset$.
(C) **Optimization Problem:** Given I,
  (A) Minimization problem. Find solution $s \in sol(I)$ of min value.
  (B) Maximization problem. Find solution $s \in sol(I)$ of max value.
  (C) Notation:
  $opt(I)$: denote the value of an optimum solution or some fixed optimum solution.

12.3 Greedy Algorithms: Tools and Techniques

12.3.0.9 What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:
(A) Do the right thing. Locally.
(B) Make decisions incrementally in small steps *no backtracking*.
(C) Decision at each step based on improving *local or current* state in myopic fashion. ... without considering the *global* situation.
(D) ***myopia***: lack of understanding or foresight.
(E) Decisions often based on some fixed and simple *priority* rules.

12.3.0.10 Pros and Cons of Greedy Algorithms

Pros:
(A) Usually (too) easy to design greedy algorithms
(B) Easy to implement and often run fast since they are simple
(C) Several important cases where they are effective/optimal
(D) Lead to first-cut heuristic when problem not well understood

Cons:
(A) **Very often** greedy algorithms don't work.
(B) Easy to lull oneself into believing they work
(C) Many greedy algorithms possible for a problem and no structured way to find effective ones.

CS 473: Every greedy algorithm needs a proof of correctness

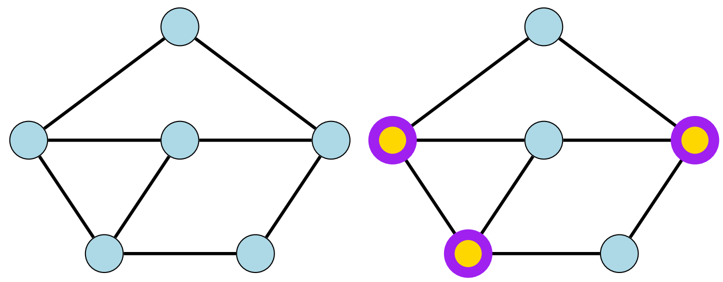
12.3.0.11 Greedy Algorithm Types

(A) Crude classification:
  (A) **Non-adaptive:** fix ordering of decisions a priori and stick with it.
  (B) **Adaptive:** make decisions adaptively but greedily/locally at each step.
(B) Plan:
  (A) See several examples
  (B) Pick up some proof techniques

12.3.0.12 Vertex Cover

Given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, a set of vertices S is:

(A) A ***vertex cover*** if every $e \in E$ has at least one endpoint in S.

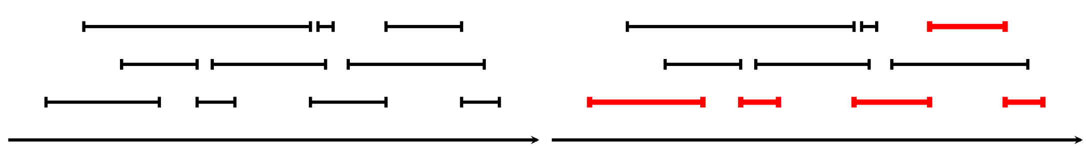


Natural algorithms for computing vertex cover?

12.4 Interval Scheduling

12.4.0.13 Interval Scheduling

Problem 12.4.1 (Interval Scheduling).
Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*
Goal: *Schedule as many jobs as possible*
(A) Two jobs with overlapping intervals cannot both be scheduled!



12.4.1 The Algorithm

12.4.1.1 Greedy Template

R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled *)
while R is not empty **do**
<2->choose $i \in R$
add i to X
remove from R all requests that overlap with i
return the set X

Main task: Decide the order in which to process requests in R

12.4.1.2 Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

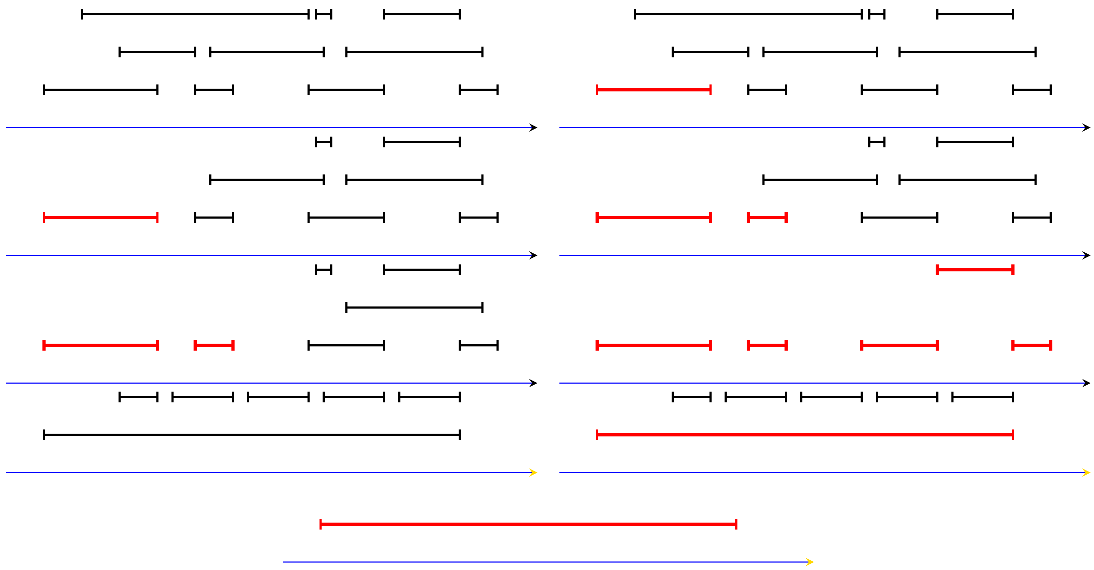
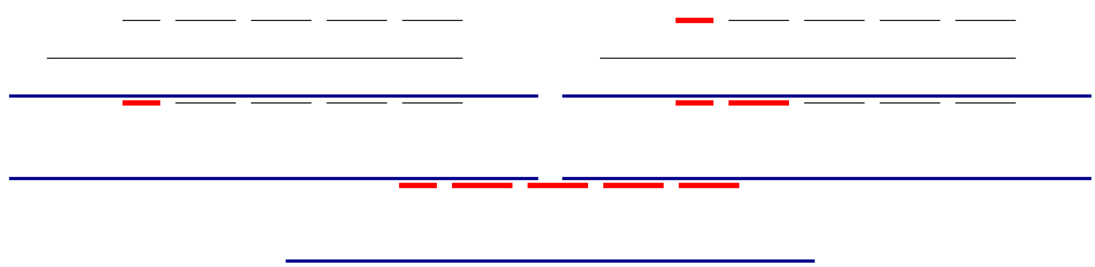


Figure 12.1: Counter example for earliest start time



12.4.1.3 Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

12.4.1.4 Fewest Conflicts

Process jobs in that have the fewest “conflicts” first.

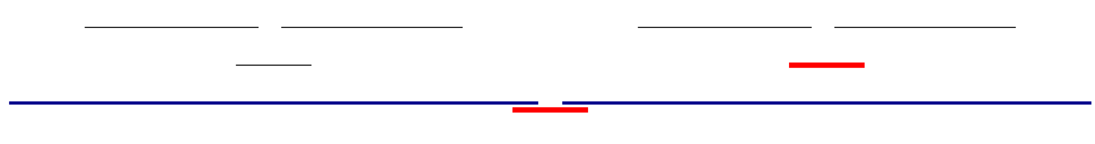


Figure 12.2: Counter example for smallest processing time

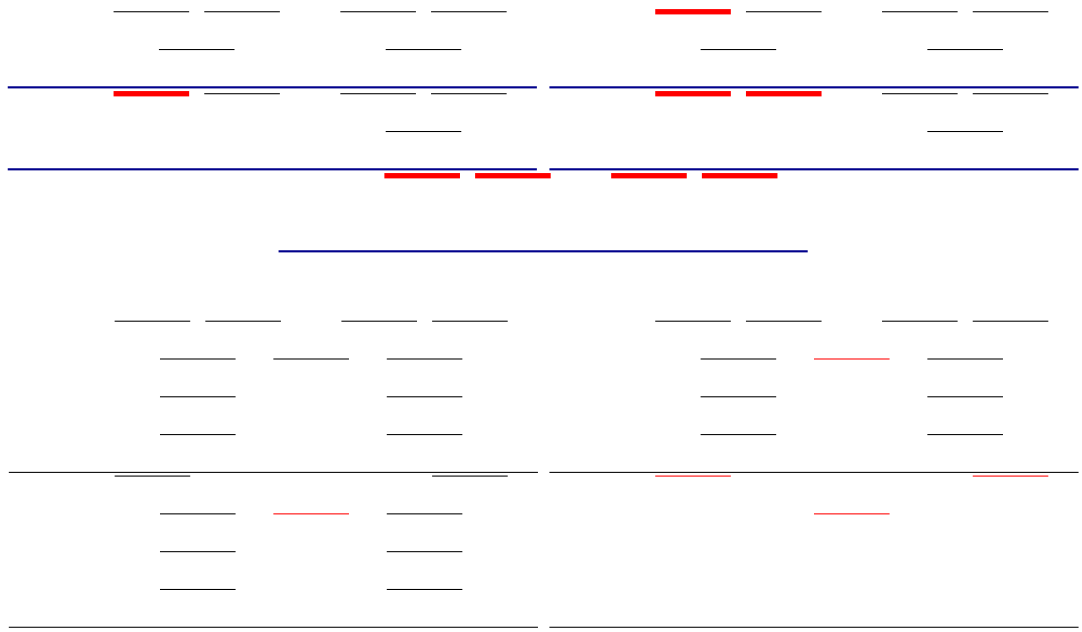


Figure 12.3: Counter example for fewest conflicts

12.4.1.5 Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

12.4.2 Correctness

12.4.2.1 Optimal Greedy Algorithm

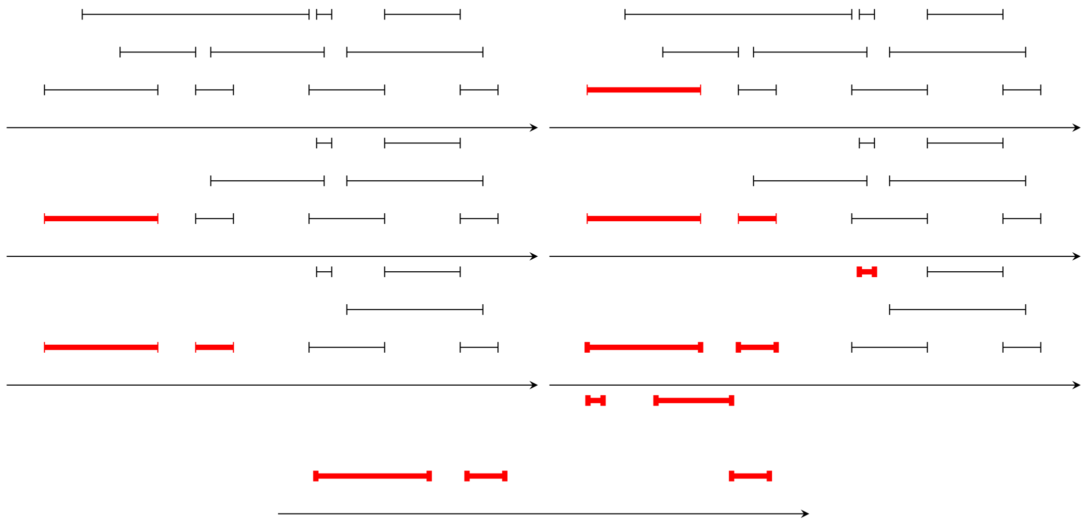
```
R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled *)
while R is not empty
    choose i ∈ R such that finishing time of i is least
    add i to X
    remove from R all requests that overlap with i
return X
```

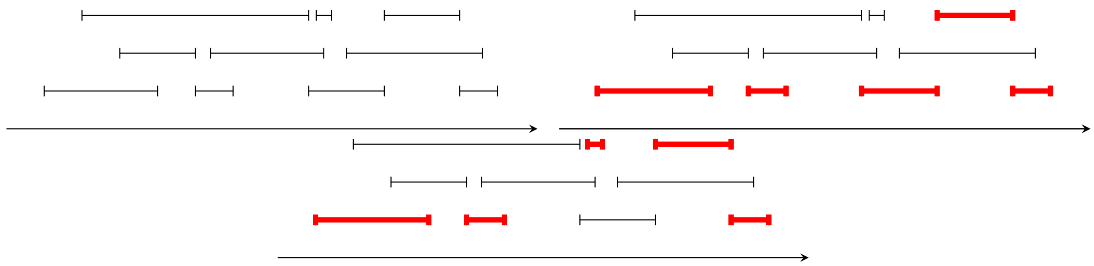
Theorem 12.4.2. *The greedy algorithm that picks jobs in the order of their finishing times is optimal.*

12.4.2.2 Proving Optimality

(A) **Correctness:** Clearly the algorithm returns a set of jobs that does not have any conflicts

(B) For a set of requests R, let O be an optimal set and let X be the set returned by the greedy algorithm. Then $O = X$? Not likely! Instead we will show that $|O| = |X|$





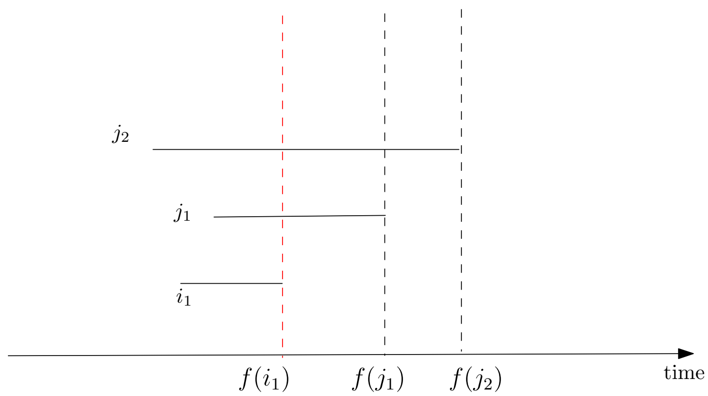


Figure 12.4: Since i_1 has the earliest finish time, any interval that conflicts with it does so at $f(i_1)$. This implies j_1 and j_2 conflict.

12.4.2.3 Proof of Optimality: Key Lemma

Lemma 12.4.3. *Let i_1 be first interval picked by Greedy. There exists an optimum solution that contains i_1.*

Proof: Let O be an *arbitrary* optimum solution. If $i_1 \in O$ we are done.

Claim: If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1. (proof later)
(A) Form a new set O' by removing j_1 from O and adding i_1, that is $O' = (O - \{j_1\}) \cup \{i_1\}$.
(B) From claim, O' is a *feasible* solution (no conflicts).
(C) Since $|O'| = |O|$, O' is also an optimum solution and it contains i_1.

∎

12.4.2.4 Proof of Claim

Claim 12.4.4. *If $i_1 \notin O$ then there is exactly one interval $j_1 \in O$ that conflicts with i_1.*

Proof:
(A) Suppose $j_1, j_2 \in O$ such that $j_1 \neq j_2$ and both j_1 and j_2 conflict with i_1.
(B) Since i_1 has earliest finish time, j_1 and i_1 overlap at $f(i_1)$.
(C) For same reason j_2 also overlaps with i_1 at $f(i_1)$.
(D) Implies that j_1, j_2 overlap at $f(i_1)$ contradicting the feasibility of O.
See figure in next slide. ∎

12.4.2.5 Figure for proof of Claim

12.4.2.6 Proof of Optimality of Earliest Finish Time First

Proof:[Proof by Induction on number of intervals] **Base Case:** $n = 1$. Trivial since Greedy picks one interval.
Induction Step: Assume theorem holds for $i < n$.

Let I be an instance with n intervals
I': I with i_1 and all intervals that overlap with i_1 removed
$G(I), G(I')$: Solution produced by Greedy on I and I'
From Lemma, there is an optimum solution O to I and $i_1 \in O$.
Let $O' = O - \{i_1\}$. O' is a solution to I'.

$$\begin{aligned}|G(I)| &= 1 + |G(I')| \quad \text{(from Greedy description)} \\ &\leq 1 + |O'| \quad \text{(By induction, } G(I') \text{ is optimum for } I'\text{)} \\ &= |O|\end{aligned}$$

■

12.4.3 Running Time

12.4.3.1 Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled *)
while R is not empty
    <3>choose i ∈ R such that finishing time of i is least
    <4>if i does not overlap with requests in X
        add i to X
    <5>remove i from R
return the set X
```

(A) Presort all requests based on finishing time. $O(n \log n)$ time
(B) Now choosing least finishing time is $O(1)$
(C) Keep track of the finishing time of the last request added to A. Then check if starting time of i later than that
(D) Thus, checking non-overlapping is $O(1)$
(E) Total time $O(n \log n + n) = O(n \log n)$

12.4.4 Extensions and Comments

12.4.4.1 Comments

(A) Interesting Exercise: smallest interval first picks at least half the optimum number of intervals.
(B) All requests need not be known at the beginning. Such *online* algorithms are a subject of research.

12.4.5 Interval Partitioning

12.4.6 The Problem

12.4.6.1 Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

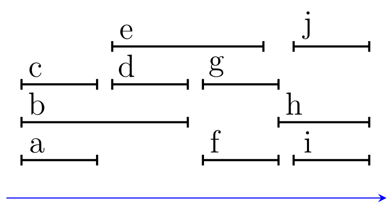


Figure 12.5: A schedule requiring 4 classrooms

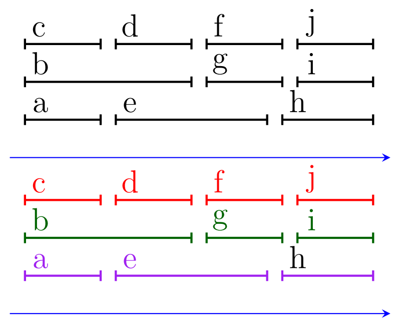


Figure 12.6: A schedule requiring 3 classrooms

12.4.7 The Algorithm

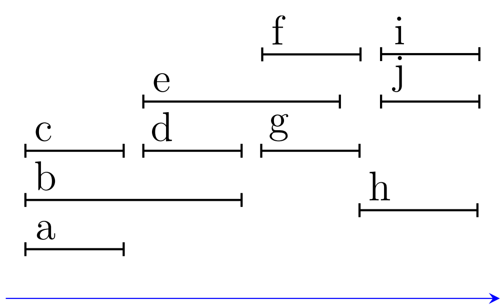
12.4.7.1 Greedy Algorithm

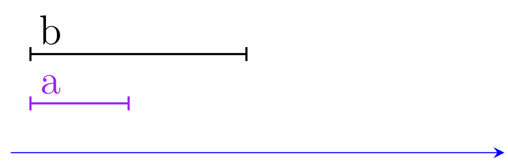
```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty do
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1
                and schedule lecture i in d + 1
        d = d + 1
```

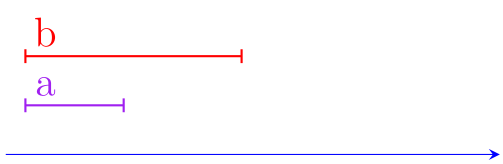
What order should we process requests in? According to start times (breaking ties arbitrarily)

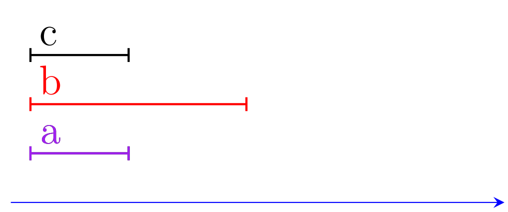
12.4.8 Example of algorithm execution

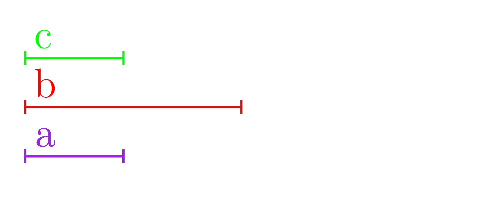
12.4.8.1 “Few things are harder to put up with than a good example.” – Mark Twain

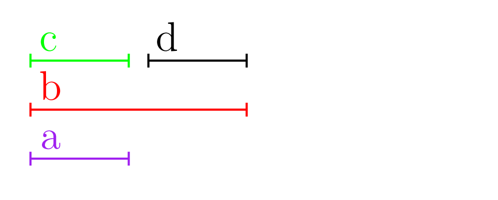


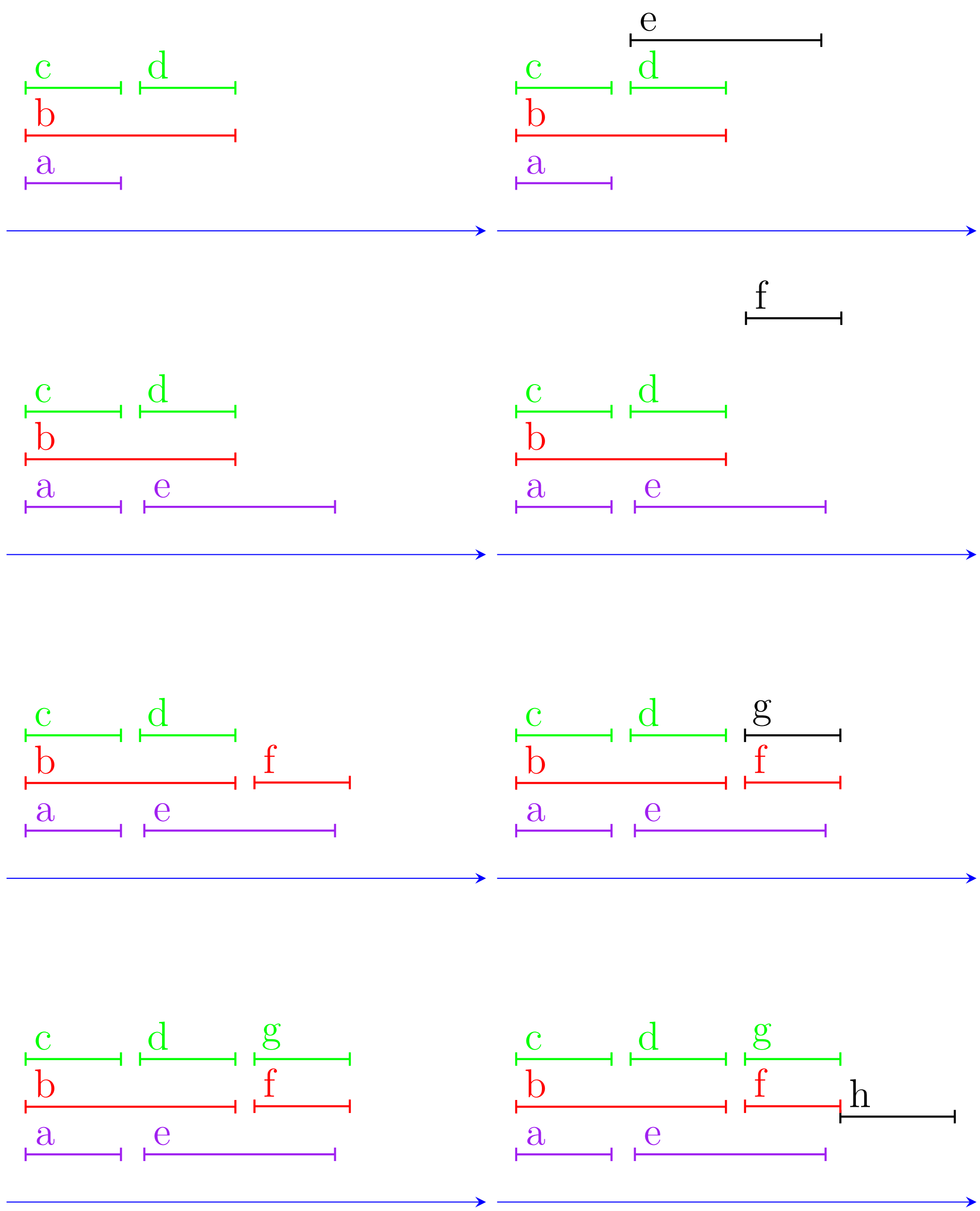


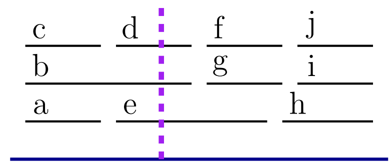


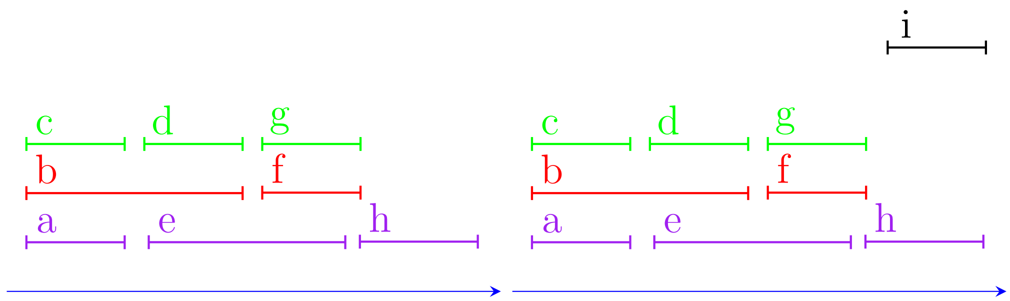


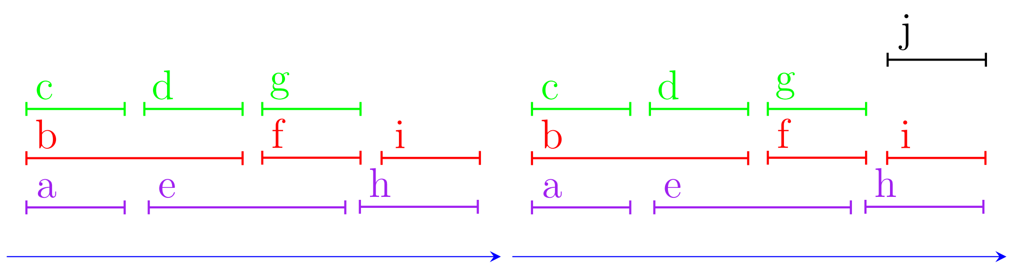


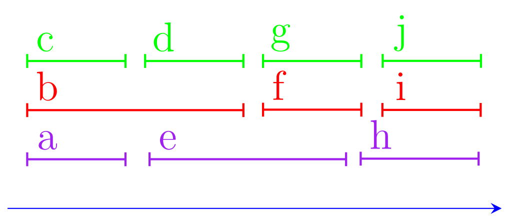



c d
b
a
e
c d
b
a
c d
b
a e
f
c d
b
a e
c d
b f
a e
c d g
b f
a e
c d g
b f
a e
c d g
b f h
a e







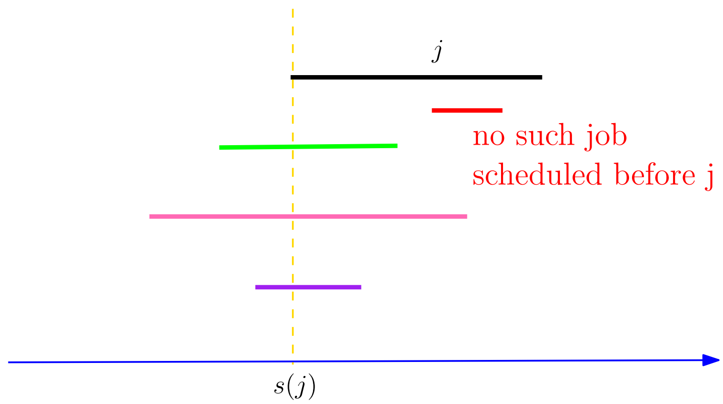


12.4.9 Correctness

12.4.9.1 Depth of Lectures

Definition 12.4.5. *(A) For a set of lectures R, k are said to be* ***in conflict*** *if there is some time t such that there are k lectures going on at time t.*

(B) The ***depth*** *of a set of lectures R is the maximum number of lectures in conflict at any time.*



12.4.9.2 Depth and Number of Class-rooms

Lemma 12.4.6. *For any set R of lectures, the number of class-rooms required is at least the depth of R.*

Proof: All lectures that are in conflict must be scheduled in different rooms. ∎

12.4.9.3 Number of Class-rooms used by Greedy Algorithm

Lemma 12.4.7. *Let d be the depth of the set of lectures R. The number of class-rooms used by the greedy algorithm is d.*

Proof:

(A) Suppose the greedy algorithm uses more that d rooms. Let j be the first lecture that is scheduled in room $d + 1$.

(B) Since we process lectures according to start times, there are d lectures that start (at or) before j and which conflict with j.

(C) Thus, *at the start time of j*, there are at least $d+1$ lectures in conflict, which contradicts the fact that the depth is d.

∎

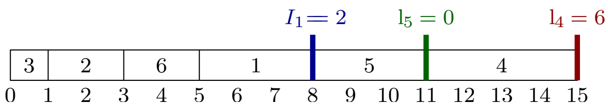
12.4.9.4 Figure

12.4.9.5 Correctness

Observation 12.4.8. *The greedy algorithm does not schedule two overlapping lectures in the same room.*

Theorem 12.4.9. *The greedy algorithm is correct and uses the optimal number of class-rooms.*

	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15



12.4.10 Running Time

12.4.10.1 Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    <1-2>choose i ∈ R such that start time of i is earliest
    <3->if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i in d + 1
        d = d + 1
```

(A) Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
(B) Keep track of the finish time of last lecture in each room.
(C) ¡4¿Checking conflict takes $O(d)$ time.¡5¿With priority queues, checking conflict takes $O(\log d)$ time.
(D) Total time ¡4¿$= O(n \log n + nd)$¡5¿$= O(n \log n + n \log d) = O(n \log n)$

12.5 Scheduling to Minimize Lateness

12.5.1 The Problem

12.5.1.1 Scheduling to Minimize Lateness

(A) Given jobs with deadlines and processing times to be scheduled on a single resource.
(B) If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time. d_i: deadline.
(C) The lateness of a job is $l_i = \max(0, f_i - d_i)$.
(D) Schedule all jobs such that $L = \max l_i$ is **minimized**.

12.5.1.2 A Simpler Feasibility Problem

(A) Given jobs with deadlines and processing times to be scheduled on a single resource.
(B) If a job i starts at time s_i then it will finish at time $f_i = s_i + t_i$, where t_i is its processing time.

(C) Schedule all jobs such that each of them completes before its deadline (in other words $L = \max_i l_i = 0$).

Definition 12.5.1. *A schedule is **feasible** if all jobs finish before their deadline.*

12.5.2 The Algorithm

12.5.2.1 Greedy Template

```
Initially R is the set of all requests
curr_time = 0
while R is not empty do
    <2->choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        return ``no feasible schedule''

return ``found feasible schedule''
```

Main task: Decide the order in which to process jobs in R

12.5.2.2 Three Algorithms

(A) Shortest job first — sort according to t_i.
(B) Shortest slack first — sort according to $d_i - t_i$.
(C) EDF = Earliest deadline first — sort according to d_i.
Counter examples for first two: exercise

12.5.2.3 Earliest Deadline First

Theorem 12.5.2. *Greedy with EDF rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.
Idle time: time during which machine is not working.

Lemma 12.5.3. *If there is a feasible schedule then there is one with no idle time before all jobs are finished.*

12.5.2.4 Inversions

Definition 12.5.4. *A schedule S is said to have an **inversion** if there are jobs i and j such that S schedules i before j, but $d_i > d_j$.*

Claim 12.5.5. *If a schedule S has an inversion then there is an inversion between two* adjacently *scheduled jobs.*

Proof: exercise.

12.5.2.5 Main Lemma

Lemma 12.5.6. *If there is a feasible schedule, then there is one with no inversions.*

Proof:[Proof Sketch] Let S be a schedule with minimum number of inversions.
(A) If S has 0 inversions, done.
(B) Suppose S has one or more inversions. By claim there are two adjacent jobs i and j that define an inversion.
(C) Swap positions of i and j.
(D) New schedule is still feasible. (Why?)
(E) New schedule has one fewer inversion — contradiction!

■

12.5.2.6 Back to Minimizing Lateness

Goal: schedule to minimize $L = \max_i l_i$.

How can we use algorithm for simpler feasibility problem?
Given a lateness bound L, can we check if there is a schedule such that $\max_i l_i \leq L$?
Yes! Set $d'_i = d_i + L$ for each job i. Use feasibility algorithm with new deadlines.
How can we find *minimum* L? Binary search!

12.5.2.7 Binary search for finding minimum lateness

```
L = L_min = 0
L_max = Σ_i t_i // why is this sufficient?
While L_min < L_max do
    L = ⌊(L_max + L_min)/2⌋
    check if there is a feasible schedule with lateness L
    if ``yes'' then L_max = L
    else L_min = L + 1
end while
return L
```

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$
(A) $O(n \log n)$ for feasibility test (sort by deadlines)
(B) $O(\log T)$ calls to feasibility test in binary search

12.5.2.8 Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $d'_i = d_i + L$.

Greedy with EDF schedules the jobs in the same order for all L!!!
Maybe there is a direct greedy algorithm for minimizing maximum lateness?

12.5.2.9 Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time − d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

12.5.2.10 Greedy Analysis: Overview

(A) **Greedy's first step leads to an optimum solution.** Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.

(B) **Greedy algorithm stays ahead.** Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.

(C) **Structural property of solution.** Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.

(D) **Exchange argument.** Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

12.5.2.11 Takeaway Points

(A) Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.

(B) *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.

(C) Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.

Chapter 13

Greedy Algorithms for Minimum Spanning Trees

OLD CS 473: Fundamental Algorithms, Spring 2015
March 5, 2015

13.1 Greedy Algorithms: Minimum Spanning Tree

13.2 Minimum Spanning Tree

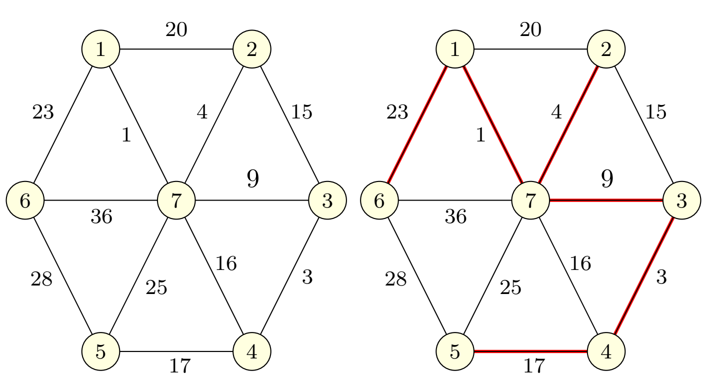
13.2.1 The Problem

13.2.1.1 Minimum Spanning Tree

Input Connected graph $G = (V, E)$ with edge costs

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost of all edges in T is smallest

(A) T is the **minimum spanning tree** (MST) of G



13.2.1.2 Applications

(A) Network Design
  (A) Designing networks with minimum cost but maximum connectivity
(B) Approximation algorithms
  (A) Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
(C) Cluster Analysis

13.2.2 The Algorithms

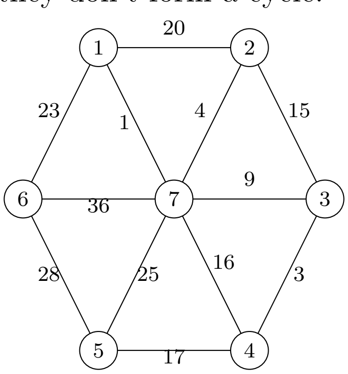
13.2.2.1 Greedy Template

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty do
    choose i ∈ E
    if (i satisfies condition)
        add i to T
return the set T
```

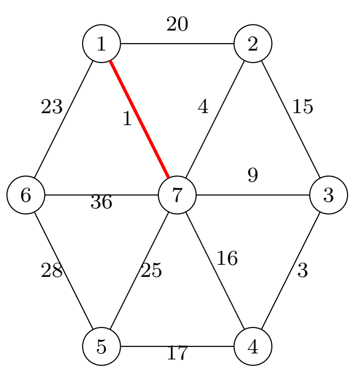
Main Task: In what order should edges be processed? When should we add edge to spanning tree?

13.2.2.2 Kruskal's Algorithm

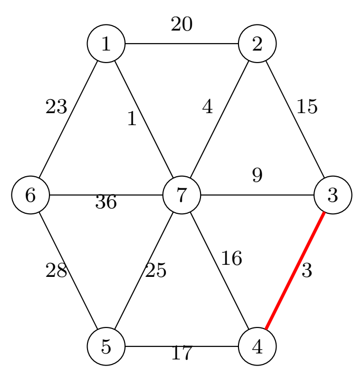
Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.



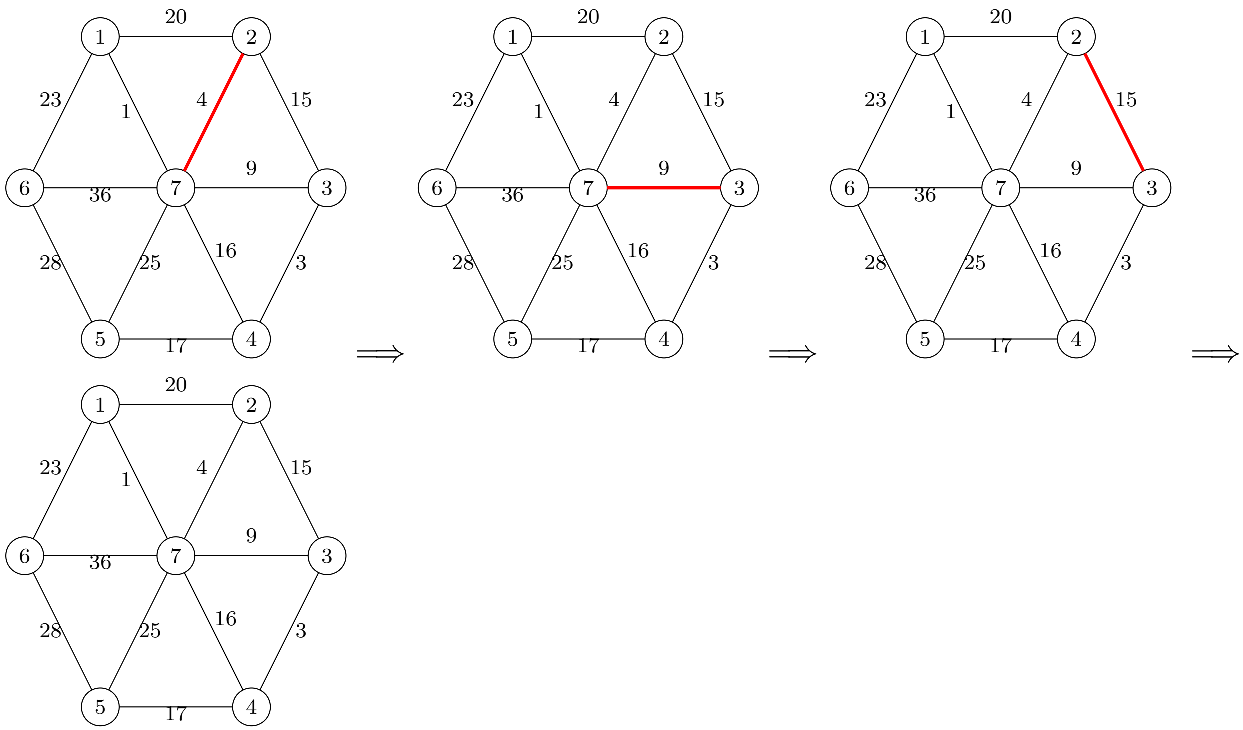
$\Longrightarrow$



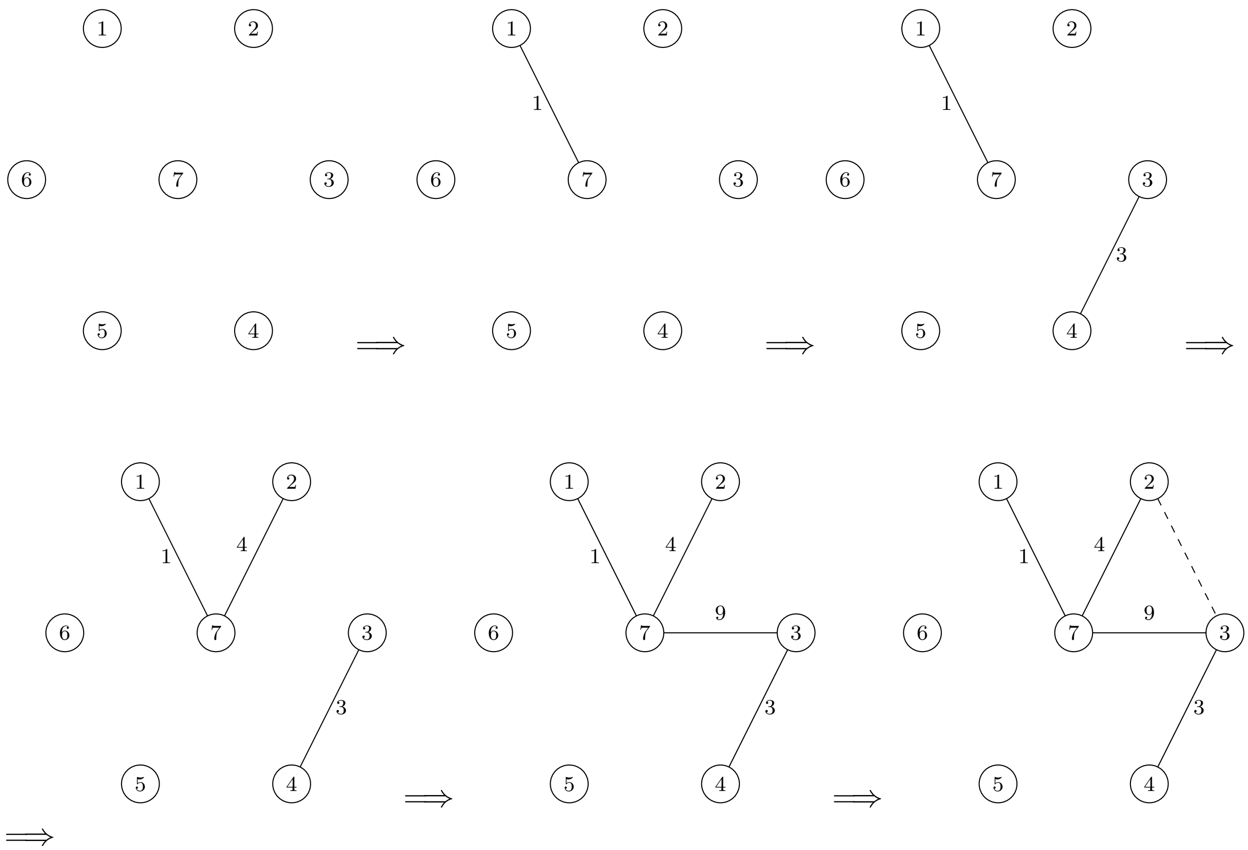
$\Longrightarrow$

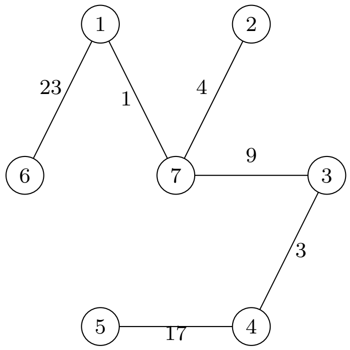


$\Longrightarrow$



MST of G:

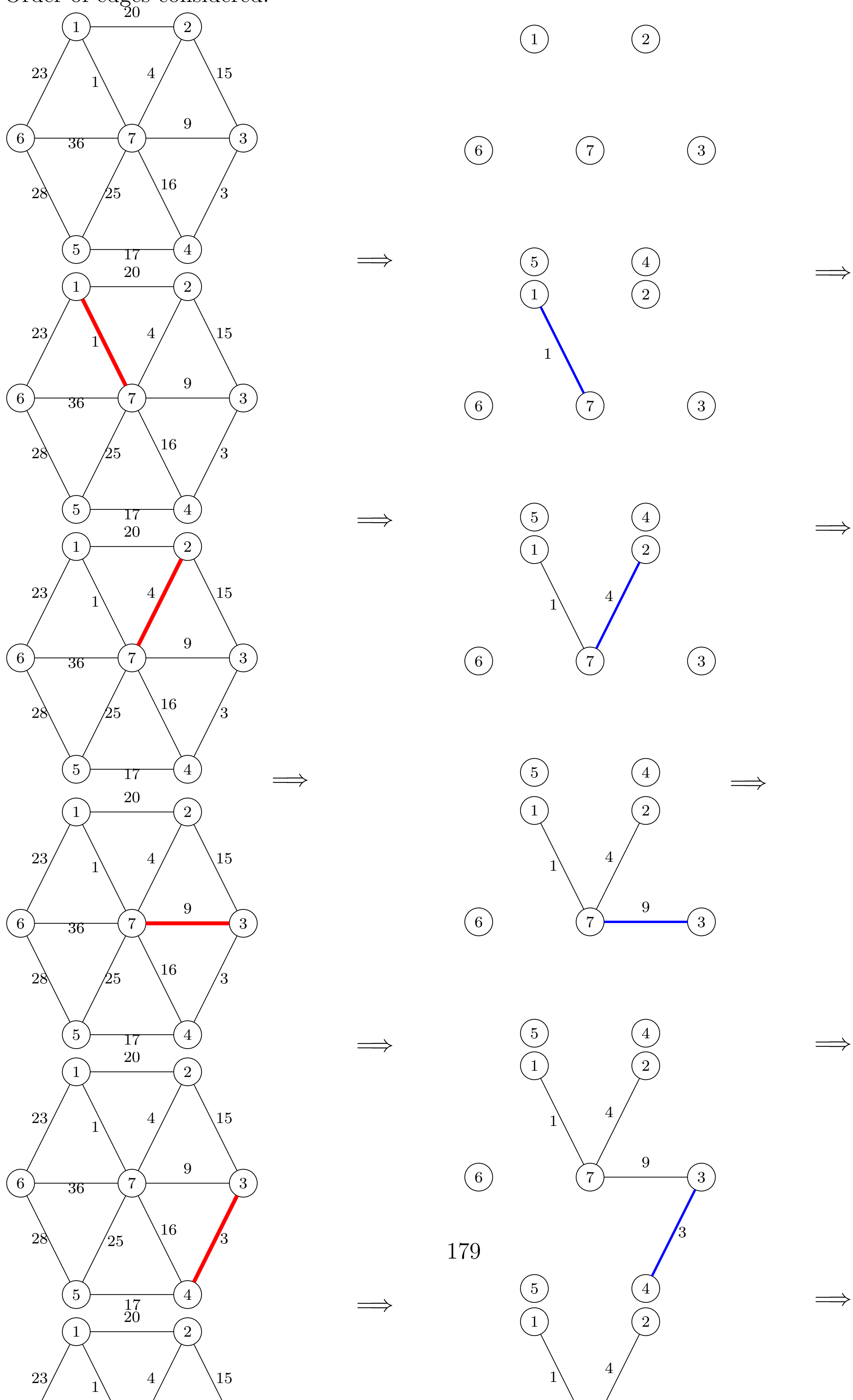




13.2.2.3 Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

Order of edges considered:



13.2.2.4 Reverse Delete Algorithm

```
Initially E is the set of all edges in G
T is E (* T will store edges of a MST *)
while E is not empty do
    choose i ∈ E of largest cost
    if removing i does not disconnect T then
        remove i from T
return the set T
```

Returns a minimum spanning tree.

13.2.3 Correctness

13.2.3.1 Correctness of MST Algorithms

(A) Many different MST algorithms
(B) All of them rely on some basic properties of MSTs, in particular the ***Cut Property*** to be seen shortly.

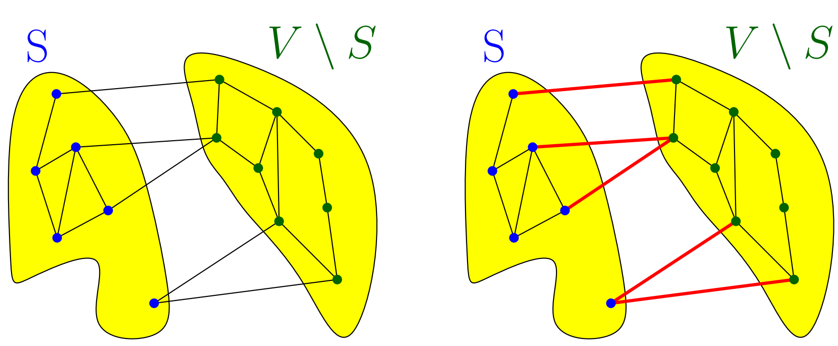
13.2.4 Assumption

13.2.4.1 And for now ...

Assumption 13.2.1. *Edge costs are distinct, that is no two edge costs are equal.*

13.2.4.2 Cuts

Definition 13.2.2. *(A) $G = (V, E)$: graph. A* **cut** *is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.*
(B) Edges having an endpoint on both sides are the **edges of the cut**.
(C) A cut edge is **crossing** *the cut.*



13.2.4.3 Safe and Unsafe Edges

Definition 13.2.3. *An edge $e = (u, v)$ is a **safe** edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).*

Definition 13.2.4. *An edge $e = (u, v)$ is an **unsafe** edge if there is some cycle C such that e is the unique maximum cost edge in C.*

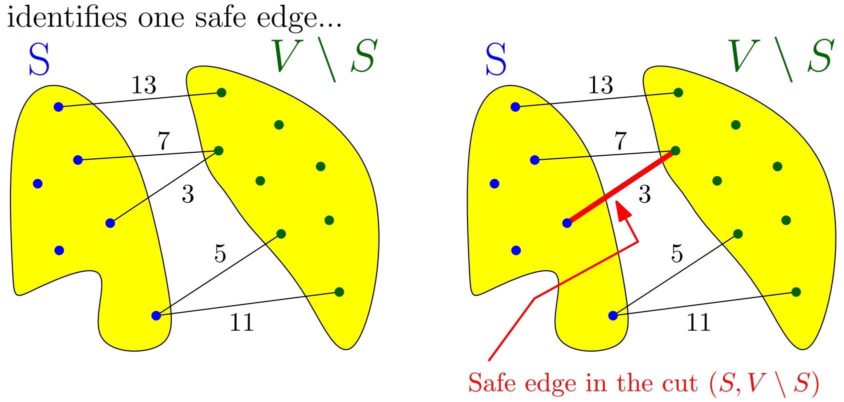
Proposition 13.2.5. *If edge costs are distinct then every edge is either safe or unsafe.*

Proof: Exercise. ■

13.2.5 Safe edge

13.2.5.1 Example...

(A) Every cut identifies one safe edge...

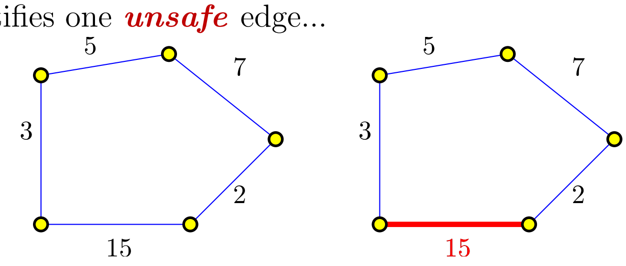


(B) ...the cheapest edge in the cut.
(C) **Note:** An edge e may be a safe edge for *many* cuts!

13.2.6 Unsafe edge

13.2.6.1 Example...

(A) Every cycle identifies one ***unsafe*** edge...



(B) ...the most expensive edge in the cycle.

13.2.6.2 Example

And all safe edges are in the MST in this case...

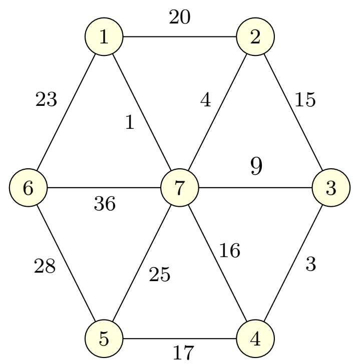


Figure 13.1: Graph with unique edge costs. Safe edges are red, rest are unsafe.

13.2.6.3 Key Observation: Cut Property

Lemma 13.2.6. *If e is a safe edge then every minimum spanning tree contains e.*

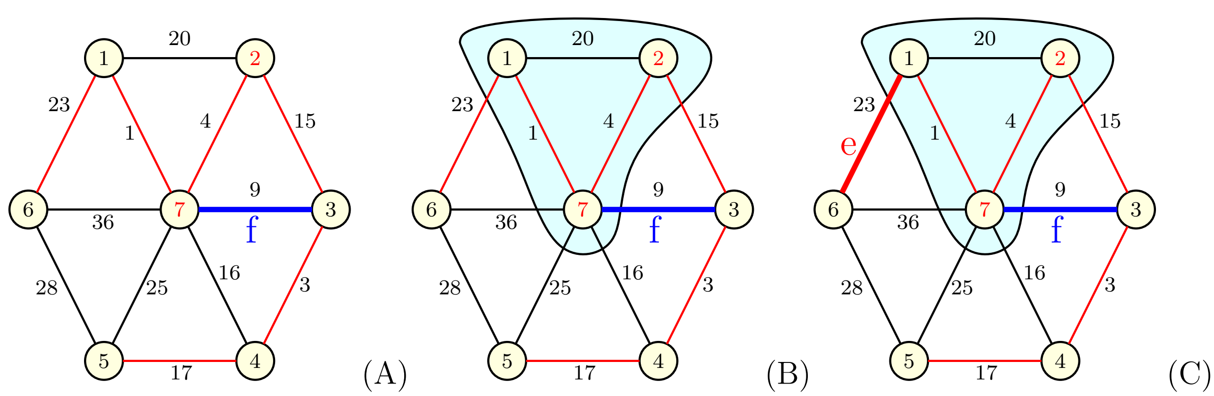
Proof:

(A) Suppose (for contradiction) e is not in MST T.
(B) Since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S.
(C) Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
(D) Since $c_f > c_e$, $T' = (T \setminus \{f\}) \cup \{e\}$ is a spanning tree of lower cost!
(E) **Error:** T' **may not** be a spanning tree!!

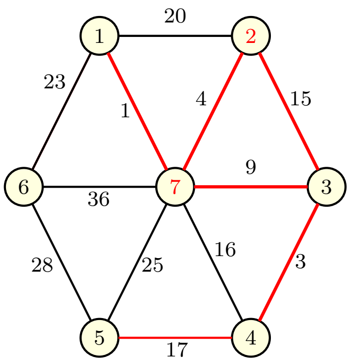
■

13.2.7 Error in Proof: Example

13.2.7.1 Problematic example. $S = \{1, 2, 7\}$, $e = (7, 3)$, $f = (1, 6)$. $T - f + e$ is not a spanning tree.



(A) (B) (C)



(D)

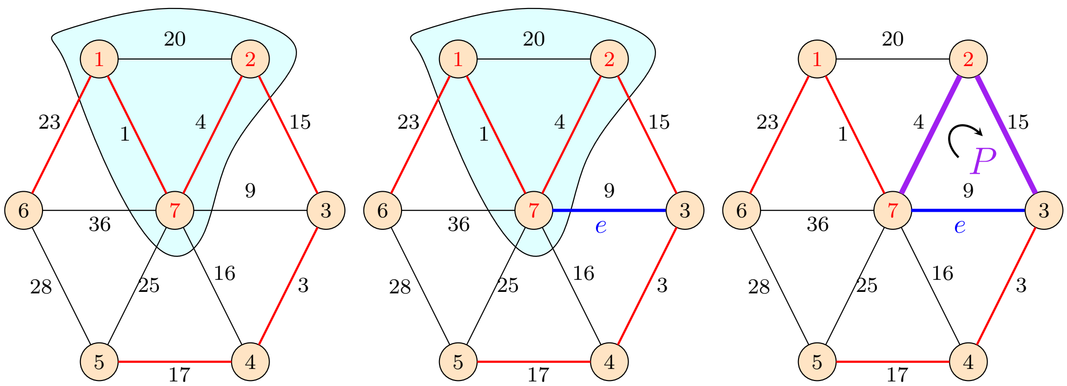
(A) (A) Consider adding the edge f.

(B) (B) It is safe because it is the cheapest edge in the cut.

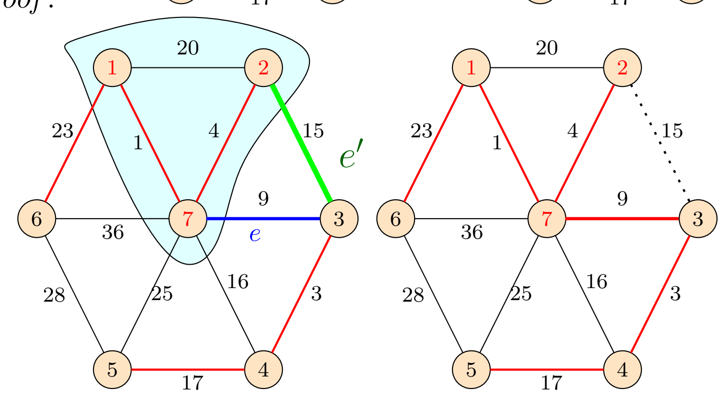
(C) (C) Lets throw out the edge e currently in the spanning tree which is more expensive than f and is in the same cut. Put it f instead...

(D) (D) New graph of selected edges is not a tree anymore. BUG.

13.2.7.2 Proof of Cut Property



Proof:



(A) Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.

(B) T is spanning tree: there is a unique path P from v to w in T

(C) 4- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$

(D) $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?)

∎

13.2.7.3 Proof of Cut Property (contd)

Observation 13.2.7. *$T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree.*

Proof: T' is connected.

Removed $e' = (v', w')$ from T but v' and w' are connected by the path $P - f + e$ in T'. Hence T' is connected if T is.

T' is a tree

T' is connected and has $n - 1$ edges (since T had $n - 1$ edges) and hence T' is a tree

∎

13.2.7.4 Safe Edges form a Tree

Lemma 13.2.8. *Let G be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.*

Proof:

(A) Suppose not. Let S be a connected component in the graph induced by the safe edges.

(B) Consider the edges crossing S, there must be a safe edge among them since edge costs are distinct and so we must have picked it.

∎

13.2.7.5 Safe Edges form an MST

Corollary 13.2.9. *Let G be a connected graph with distinct edge costs, then set of safe edges form the **unique** MST of G.*

Consequence: Every correct MST algorithm when G has unique edge costs includes exactly the safe edges.

13.2.7.6 Cycle Property

Lemma 13.2.10. *If e is an unsafe edge then no MST of G contains e.*

Proof: Exercise. See text book. ∎

Note: Cut and Cycle properties hold even when edge costs are not distinct. Safe and unsafe definitions do not rely on distinct cost assumption.

13.2.7.7 Correctness of Prim's Algorithm

Prim's Algorithm Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof:[Proof of correctness]
(A) If e is added to tree, then e is safe and belongs to every MST.
  (A) Let S be the vertices connected by edges in T when e is added.
  (B) e is edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.
(B) Set of edges output is a spanning tree
  (A) Set of edges output forms a connected graph: by induction, S is connected in each iteration and eventually $S = V$.
  (B) Only safe edges added and they do not have a cycle

■

13.2.7.8 Correctness of Kruskal's Algorithm

Kruskal's Algorithm Pick edge of lowest cost and add if it does not form a cycle with existing edges.

Proof:[Proof of correctness]
(A) If $e = (u, v)$ is added to tree, then e is safe
  (A) When algorithm adds e let S and S' be the connected components containing u and v respectively
  (B) e is the lowest cost edge crossing S (and also S').
  (C) If there is an edge e' crossing S and has lower cost than e, then e' would come before e in the sorted order and would be added by the algorithm to T
(B) Set of edges output is a spanning tree : exercise

■

13.2.7.9 Correctness of Reverse Delete Algorithm

Reverse Delete Algorithm Consider edges in decreasing cost and remove an edge if it does not disconnect the graph

Proof:[Proof of correctness] Argue that only unsafe edges are removed (see text book). ■

13.2.7.10 When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge
Formal argument: Order edges lexicographically to break ties
(A) $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
(B) Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or $(c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A)$

(C) Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's, Kruskal, and Reverse Delete Algorithms are optimal with respect to lexicographic ordering.

13.2.7.11 Edge Costs: Positive and Negative

(A) Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
(B) Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
(C) Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

13.3 Data Structures for MST: Priority Queues and Union-Find

13.4 Data Structures

13.4.1 Implementing Prim's Algorithm

13.4.2 Implementing Prim's Algorithm

13.4.2.1 Implementing Prim's Algorithm

```
Prim_ComputeMST
    E is the set of all edges in G
    S = {1}
    T is empty (* T will store edges of a MST *)
    <2>while S ≠ V do
        <3>pick e = (v, w) ∈ E such that
            v ∈ S and w ∈ V − S
            e has minimum cost
        T = T ∪ e
        S = S ∪ w
    return the set T
```

Analysis
(A) Number of iterations $= O(n)$, where n is number of vertices
(B) Picking e is $O(m)$ where m is the number of edges
(C) Total time $O(nm)$

13.4.3 Implementing Prim's Algorithm

13.4.3.1 More Efficient Implementation

```
Prim_ComputeMST
    E is the set of all edges in G
    S = {1}
    T is empty (* T will store edges of a MST *)
    for v ∉ S, a(v) = min_{w∈S} c(w, v)
    for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
    while S ≠ V do
        pick v with minimum a(v)
        T = T ∪ {(e(v), v)}
        S = S ∪ {v}
        update arrays a and e
    return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$.

13.4.4 Priority Queues

13.4.4.1 Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations

(A) **makeQ**: create an empty queue

(B) **findMin**: find the minimum key in S

(C) **extractMin**: Remove $v \in S$ with smallest key and return it

(D) **add**$(v, k(v))$: Add new element v with key $k(v)$ to S

(E) **Delete**(v): Remove element v from S

(F) **decreaseKey** $(v, k'(v))$: *decrease* key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$

(G) **meld**: merge two separate priority queues into one

13.4.4.2 Prim's using priority queues

```
    E is the set of all edges in G
    S = {1}
    T is empty (* T will store edges of a MST *)
    for v ∉ S, a(v) = min_{w∈S} c(w, v)
    for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
    while S ≠ V do
        <2>pick v with minimum a(v)
        T = T ∪ {(e(v), v)}
        S = S ∪ {v}
        <3>update arrays a and e
    return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$

(A) Requires $O(n)$ **extractMin** operations
(B) Requires $O(m)$ **decreaseKey** operations

13.4.4.3 Running time of Prim's Algorithm

$O(n)$ **extractMin** operations and $O(m)$ **decreaseKey** operations
(A) Using standard Heaps, **extractMin** and **decreaseKey** take $O(\log n)$ time. Total: $O((m+n)\log n)$
(B) Using Fibonacci Heaps, $O(\log n)$ for **extractMin** and $O(1)$ (amortized) for **decreaseKey**. Total: $O(n \log n + m)$.

Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?

13.4.5 Implementing Kruskal's Algorithm

13.4.5.1 Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        <2-3>choose e ∈ E of minimum cost
        <4-5>if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

(A) Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
(B) Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
(C) Total time $O(m \log m) + O(mn) = O(mn)$

13.4.5.2 Implementing Kruskal's Algorithm Efficiently

```
Kruskal_ComputeMST
    Sort edges in E based on cost
    T is empty (* T will store edges of a MST *)
    each vertex u is placed in a set by itself
    while E is not empty do
        pick e = (u,v) ∈ E of minimum cost
        <2->if u and v belong to different sets
            add e to T
            <2->merge the sets containing u and v
    return the set T
```

Need a data structure to check if two elements belong to same set and to merge two sets.

13.4.6 Union-Find Data Structure

13.4.6.1 Union-Find Data Structure

Data Structure Store disjoint sets of elements that supports the following operations

(A) **makeUnionFind**(S) returns a data structure where each element of S is in a separate set

(B) **find**(u) returns the *name* of set containing element u. Thus, u and v belong to the same set if and only if **find**(u) = **find**(v)

(C) **union**(A, B) merges two sets A and B. Here A and B are the names of the sets. Typically the name of a set is some element in the set.

13.4.6.2 Implementing Union-Find using Arrays and Lists

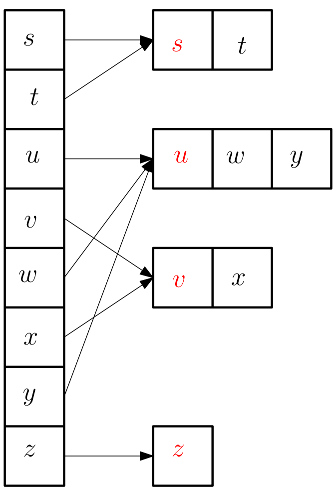
Using lists

(A) Each set stored as list with a name associated with the list.

(B) For each element $u \in S$ a pointer to the its set. Array for pointers: `component`[u] is pointer for u.

(C) **makeUnionFind** (S) takes $O(n)$ time and space.

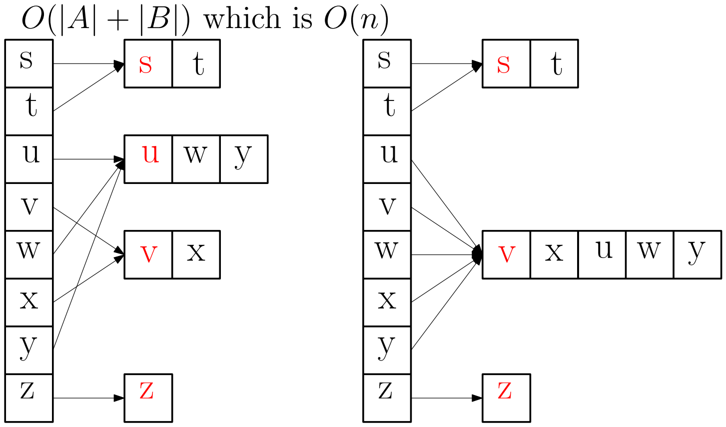
13.4.6.3 Example



13.4.6.4 Implementing Union-Find using Arrays and Lists

(A) **find**(u) reads the entry `component`[u]: $O(1)$ time

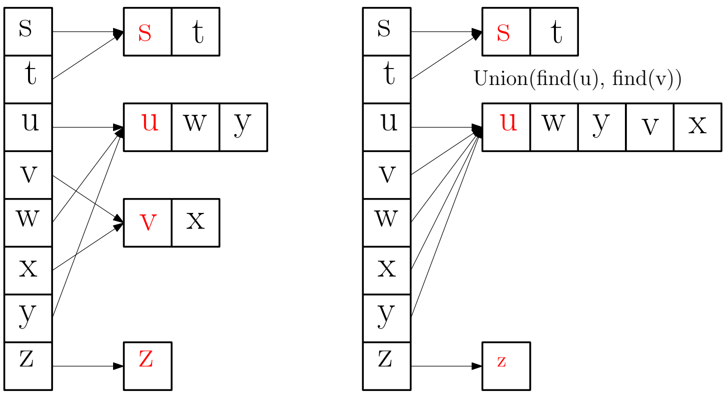
(B) **union**(A,B) involves updating the entries `component`[u] for all elements u in A and B: $O(|A| + |B|)$ which is $O(n)$



13.4.6.5 Improving the List Implementation for Union

New Implementation As before use `component`$[u]$ to store set of u.
Change to **union**(A,B):
(A) with each set, keep track of its size
(B) assume $|A| \le |B|$ for now
(C) Merge the list of A into that of B: $O(1)$ time (linked lists)
(D) Update `component`$[u]$ only for elements in the smaller set A
(E) Total $O(|A|)$ time. Worst case is still $O(n)$.
find still takes $O(1)$ time

13.4.6.6 Example



The smaller set (list) is appended to the largest set (list)

13.4.6.7 Improving the List Implementation for Union

Question Is the improved implementation provably better or is it simply a nice heuristic?

Theorem 13.4.1. *Any sequence of k* **union** *operations, starting from* **makeUnionFind**(S) *on set S of size n, takes at most $O(k \log k)$.*

Corollary 13.4.2. *Kruskal's algorithm can be implemented in $O(m \log m)$ time.*

Sorting takes $O(m \log m)$ time, $O(m)$ finds take $O(m)$ time and $O(n)$ unions take $O(n \log n)$ time.

13.4.6.8 Amortized Analysis

Why does theorem work?
Key Observation **union**(A,B) takes $O(|A|)$ time where $|A| \le |B|$. Size of new set is $\ge 2|A|$. Cannot double too many times.

13.4.6.9 Proof of Theorem

Proof:
(A) Any union operation involves at most 2 of the original one-element sets; thus at least $n - 2k$ elements have never been involved in a union

(B) Also, maximum size of any set (after k unions) is $2k$
(C) **union**(A,B) takes $O(|A|)$ time where $|A| \le |B|$.
(D) *Charge* each element in A constant time to *pay* for $O(|A|)$ time.
(E) How much does any element get charged?
(F) If `component`$[v]$ is updated, set containing v *doubles* in size
(G) `component`$[v]$ is updated at most $\log 2k$ times
(H) Total number of updates is $2k \log 2k = O(k \log k)$

■

13.4.6.10 Improving Worst Case Time

Better data structure Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.
(A) **find**(u): Traverse from u to the root
(B) **union**(A, B): Make root of A (smaller set) point to root of B. Takes $O(1)$ time.

13.4.6.11 Details of Implementation

Each element $u \in S$ has a pointer parent(u) to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

```
union(component(u), component(v))
        (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
```

13.4.6.12 Analysis

Theorem 13.4.3. *The forest based implementation for a set of size n, has the following complexity for the various operations:* **makeUnionFind** *takes $O(n)$,* **union** *takes $O(1)$, and* **find** *takes $O(\log n)$.*

Proof:
(A) **find**(u) depends on the height of tree containing u.
(B) Height of u increases by at most 1 only when the set containing u changes its name.
(C) If height of u increases then size of the set containing u (at least) doubles.
(D) Maximum set size is n; so height of any tree is at most $O(\log n)$.

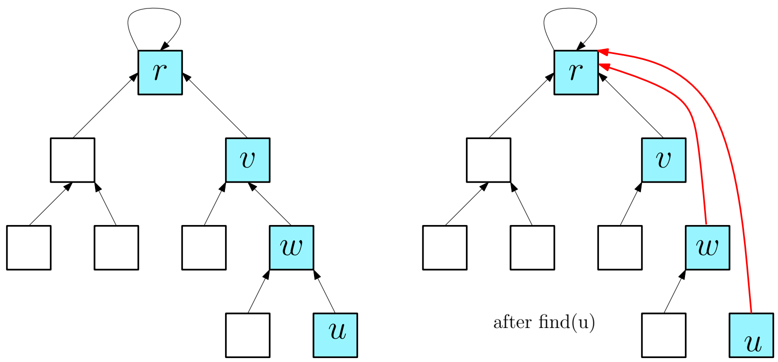
■

13.4.6.13 Further Improvements: Path Compression

Observation 13.4.4. *Consecutive calls of* **find**(u) *take* $O(\log n)$ *time each, but they traverse the same sequence of pointers.*

Idea: Path Compression Make all nodes encountered in the **find**(u) point to root.

13.4.6.14 Path Compression: Example



13.4.6.15 Path Compression

```
find(u):
    if (parent(u) ≠ u) then
        parent(u) = find(parent(u))
    return parent(u)
```

Question Does Path Compression help?
Yes!

Theorem 13.4.5. *With Path Compression,* k *operations (***find** *and/or* **union***) take* $O(k\alpha(k, \min\{k, n\}))$ *time where* α *is the* **inverse Ackermann function***.*

13.4.6.16 Ackermann and Inverse Ackermann Functions

Ackermann function $A(m, n)$ defined for $m, n \geq 0$ recursively

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

$A(3, n) = 2^{n+3} - 3$
$A(4, 3) = 2^{65536} - 3$

$\alpha(m, n)$ is inverse Ackermann function defined as

$$\alpha(m,n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

For **all practical** purposes $\alpha(m, n) \leq 5$

13.4.6.17 Lower Bound for Union-Find Data Structure

Amazing result:

Theorem 13.4.6 (Tarjan). *For* **Union-Find***,* ***any*** *data structure in the pointer model requires* $\Omega(m\alpha(m,n))$ *time for* m *operations.*

13.4.6.18 Running time of Kruskal's Algorithm

Using Union-Find data structure:
(A) $O(m)$ **find** operations (two for each edge)
(B) $O(n)$ **union** operations (one for each edge added to T)
(C) Total time: $O(m \log m)$ for sorting plus $O(m\alpha(n))$ for union-find operations. Thus $O(m \log m)$ time despite the improved Union-Find data structure.

13.4.6.19 Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $O(n \log n + m)$.
If m is $O(n)$ then running time is $\Omega(n \log n)$.

Question Is there a linear time ($O(m + n)$ time) algorithm for MST?
(A) $O(m \log^* m)$ time [**Fredman and Tarjan, 1987**].
(B) $O(m + n)$ time using bit operations in RAM model [**Fredman and Willard, 1994**].
(C) $O(m + n)$ expected time (randomized algorithm) [**Karger et al., 1995**].
(D) $O((n + m)\alpha(m, n))$ time [**Chazelle, 2000**].
(E) Still open: Is there an $O(n + m)$ time deterministic algorithm in the comparison model?

Chapter 14

Introduction to Randomized Algorithms: QuickSort and QuickSelect

OLD CS 473: Fundamental Algorithms, Spring 2015
March 10, 2015

14.1 Introduction to Randomized Algorithms

14.2 Introduction

14.2.0.20 Randomized Algorithms

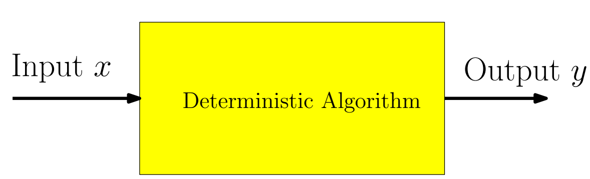
14.2.0.21 Example: Randomized QuickSort

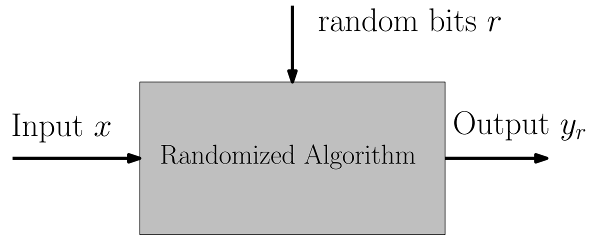
QuickSort [Hoare, 1962]

(A) Pick a pivot element from array
(B) Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
(C) Recursively sort the subarrays, and concatenate them.

Randomized **QuickSort**

(A) Pick a pivot element *uniformly at random* from the array
(B) Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
(C) Recursively sort the subarrays, and concatenate them.





14.2.0.22 Example: Randomized Quicksort

Recall: **QuickSort** can take $\Omega(n^2)$ time to sort array of size n.

Theorem 14.2.1. *Randomized* **QuickSort** *sorts a given array of length n in $O(n \log n)$* expected *time.*

Note: On *every* input randomized **QuickSort** takes $O(n \log n)$ time in expectation. On *every* input it may take $\Omega(n^2)$ time with some small probability.

14.2.0.23 Example: Verifying Matrix Multiplication

Problem Given three $n \times n$ matrices A, B, C is $AB = C$?

Deterministic algorithm:

(A) Multiply A and B and check if equal to C.
(B) Running time? $O(n^3)$ by straight forward approach. $O(n^{2.37})$ with fast matrix multiplication (complicated and impractical).

14.2.0.24 Example: Verifying Matrix Multiplication

Problem Given three $n \times n$ matrices A, B, C is $AB = C$?

Randomized algorithm:

(A) Pick a random $n \times 1$ vector r.
(B) Return the answer of the equality $ABr = Cr$.
(C) Running time? $O(n^2)$!

Theorem 14.2.2. *If $AB = C$ then the algorithm will always say YES. If $AB \neq C$ then the algorithm will say YES with probability at most $1/2$. Can repeat the algorithm 100 times independently to reduce the probability of a false positive to $1/2^{100}$.*

14.2.0.25 Why randomized algorithms?

(A) Many applications: algorithms, data structures and CS.
(B) In some cases only known algorithms are randomized or randomness is provably necessary.
(C) Often randomized algorithms are (much) simpler and/or more efficient.
(D) Several deep connections to mathematics, physics etc.
(E) ...
(F) Lots of fun!

14.2.0.26 Where do I get random bits?

Question: Are true random bits available in practice?
(A) Buy them!
(B) CPUs use physical phenomena to generate random bits.
(C) Can use pseudo-random bits or semi-random bits from nature. Several fundamental unresolved questions in complexity theory on this topic. Beyond the scope of this course.
(D) In practice pseudo-random generators work quite well in many applications.
(E) The model is interesting to think in the abstract and is very useful even as a theoretical construct. One can *derandomize* randomized algorithms to obtain deterministic algorithms.

14.2.0.27 Average case analysis vs Randomized algorithms

Average case analysis:
(A) Fix a deterministic algorithm.
(B) Assume inputs comes from a probability distribution.
(C) Analyze the algorithm's *average* performance over the distribution over inputs.

Randomized algorithms:
(A) Algorithm uses random bits in addition to input.
(B) Analyze algorithms *average* performance over the given input where the average is over the random bits that the algorithm uses.
(C) On each input behaviour of algorithm is random. Analyze worst-case over all inputs of the (average) performance.

14.3 Basics of Discrete Probability

14.3.0.28 Discrete Probability

We restrict attention to finite probability spaces.

Definition 14.3.1. *A discrete probability space is a pair* $(\Omega, \mathbf{Pr})$ *consists of finite set* Ω *of* elementary events *and function* $p : \Omega \to [0, 1]$ *which assigns a probability* $\mathbf{Pr}[\omega]$ *for each* $\omega \in \Omega$ *such that* $\sum_{\omega\in\Omega} \mathbf{Pr}[\omega] = 1$.

Example 14.3.2. *An unbiased coin.* $\Omega = \{H, T\}$ *and* $\mathbf{Pr}[H] = \mathbf{Pr}[T] = 1/2$.

Example 14.3.3. *A* 6*-sided unbiased die.* $\Omega = \{1, 2, 3, 4, 5, 6\}$ *and* $\mathbf{Pr}[i] = 1/6$ *for* $1 \leq i \leq 6$.

14.3.1 Discrete Probability

14.3.1.1 And more examples

Example 14.3.4. *A biased coin.* $\Omega = \{H, T\}$ *and* $\mathbf{Pr}[H] = 2/3, \mathbf{Pr}[T] = 1/3$.

Example 14.3.5. *Two independent unbiased coins.* $\Omega = \{HH, TT, HT, TH\}$ *and* $\mathbf{Pr}[HH] = \mathbf{Pr}[TT] = \mathbf{Pr}[HT] = \mathbf{Pr}[TH] = 1/4$*.*

Example 14.3.6. *A pair of (highly) correlated dice.*
$\Omega = \{(i,j) \mid 1 \leq i \leq 6, 1 \leq j \leq 6\}$.
$\mathbf{Pr}[i,i] = 1/6$ *for* $1 \leq i \leq 6$ *and* $\mathbf{Pr}[i,j] = 0$ *if* $i \neq j$*.*

14.3.1.2 Events

Definition 14.3.7. *Given a probability space* $(\Omega, \mathbf{Pr})$ *an* **event** *is a subset of* Ω*. In other words an event is a collection of elementary events. The probability of an event* A*, denoted by* $\mathbf{Pr}[A]$*, is* $\sum_{\omega \in A} \mathbf{Pr}[\omega]$*.*

Definition 14.3.8. *The* **complement event** *of an event* $A \subseteq \Omega$ *is the event* $\Omega \setminus A$ *frequently denoted by* $\bar{A}$*.*

14.3.2 Events

14.3.2.1 Examples

Example 14.3.9. *A pair of independent dice.* $\Omega = \{(i,j) \mid 1 \leq i \leq 6, 1 \leq j \leq 6\}$.

(A) Let A *be the event that the sum of the two numbers on the dice is even.*
Then $A = \left\{ (i,j) \in \Omega \;\middle|\; (i+j) \text{ is even} \right\}$.
$\mathbf{Pr}[A] = |A|/36 = 1/2$.

(B) Let B *be the event that the first die has* 1*. Then* $B = \left\{ (1,1), (1,2), (1,3), (1,4), (1,5), (1,6) \right\}$.
$\mathbf{Pr}[B] = 6/36 = 1/6$.

14.3.2.2 Independent Events

Definition 14.3.10. *Given a probability space* $(\Omega, \mathbf{Pr})$ *and two events* A, B *are* **independent** *if and only if* $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A]\,\mathbf{Pr}[B]$*. Otherwise they are* dependent*. In other words* A, B *independent implies one does not affect the other.*

Example 14.3.11. *Two coins.* $\Omega = \{HH, TT, HT, TH\}$ *and* $\mathbf{Pr}[HH] = \mathbf{Pr}[TT] = \mathbf{Pr}[HT] = \mathbf{Pr}[TH] = 1/4$*.*

(A) A *is the event that the first coin is heads and* B *is the event that second coin is tails.* A, B *are independent.*

(B) A *is the event that the two coins are different.* B *is the event that the second coin is heads.* A, B *independent.*

14.3.3 Independent Events

14.3.3.1 Examples

Example 14.3.12. A *is the event that both are not tails and* B *is event that second coin is heads.* A, B *are dependent.*

14.3.4 Union bound

14.3.4.1 The probability of the union of two events, is $\leq$ the probability of the sum of their probabilities.

Lemma 14.3.13. *For any two events $\mathcal{E}$ and $\mathcal{F}$, we have that* $\mathbf{Pr}\left[\mathcal{E} \cup \mathcal{F}\right] \leq \mathbf{Pr}\left[\mathcal{E}\right] + \mathbf{Pr}\left[\mathcal{F}\right]$.

Proof: Consider $\mathcal{E}$ and $\mathcal{F}$ to be a collection of elmentery events (which they are). We have

$$\begin{aligned}\mathbf{Pr}\left[\mathcal{E} \cup \mathcal{F}\right] &= \sum_{x \in \mathcal{E} \cup \mathcal{F}} \mathbf{Pr}[x] \\ &\leq \sum_{x \in \mathcal{E}} \mathbf{Pr}[x] + \sum_{x \in \mathcal{F}} \mathbf{Pr}[x] = \mathbf{Pr}\left[\mathcal{E}\right] + \mathbf{Pr}\left[\mathcal{F}\right].\end{aligned}$$

■

14.3.4.2 Random Variables

Definition 14.3.14. *Given a probability space $(\Omega, \mathbf{Pr})$ a (real-valued) random variable X over Ω is a function that maps each elementary event to a real number. In other words $X : \Omega \to \mathbb{R}$.*

Example 14.3.15. *A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\mathbf{Pr}[i] = 1/6$ for $1 \leq i \leq 6$.*
(A) $X : \Omega \to \mathbb{R}$ where $X(i) = i \mod 2$.
(B) $Y : \Omega \to \mathbb{R}$ where $Y(i) = i^2$.

Definition 14.3.16. *A* **binary random variable** *is one that takes on values in $\{0, 1\}$.*

14.3.4.3 Indicator Random Variables

Special type of random variables that are quite useful.

Definition 14.3.17. *Given a probability space $(\Omega, \mathbf{Pr})$ and an event $A \subseteq \Omega$ the indicator random variable X_A is a binary random variable where $X_A(\omega) = 1$ if $\omega \in A$ and $X_A(\omega) = 0$ if $\omega \notin A$.*

Example 14.3.18. *A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\mathbf{Pr}[i] = 1/6$ for $1 \leq i \leq 6$. Let A be the even that i is divisible by 3. Then $X_A(i) = 1$ if $i = 3, 6$ and 0 otherwise.*

14.3.4.4 Expectation

Definition 14.3.19. *For a random variable X over a probability space $(\Omega, \mathbf{Pr})$ the* **expectation** *of X is defined as $\sum_{\omega \in \Omega} \mathbf{Pr}[\omega] X(\omega)$. In other words, the expectation is the average value of X according to the probabilities given by $\mathbf{Pr}[\cdot]$.*

Example 14.3.20. *A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\mathbf{Pr}[i] = 1/6$ for $1 \leq i \leq 6$.*
(A) $X : \Omega \to \mathbb{R}$ where $X(i) = i \mod 2$. Then $\mathbf{E}[X] = 1/2$.
(B) $Y : \Omega \to \mathbb{R}$ where $Y(i) = i^2$. Then $\mathbf{E}[Y] = \sum_{i=1}^{6} \frac{1}{6} \cdot i^2 = 91/6$.

14.3.4.5 Expectation

Proposition 14.3.21. *For an indicator variable X_A, $\mathbf{E}[X_A] = \mathbf{Pr}[A]$.*

Proof:

$$\begin{aligned}\mathbf{E}[X_A] &= \sum_{y\in\Omega} X_A(y)\,\mathbf{Pr}[y]\\ &= \sum_{y\in A} 1\cdot\mathbf{Pr}[y] + \sum_{y\in\Omega\setminus A} 0\cdot\mathbf{Pr}[y]\\ &= \sum_{y\in A}\mathbf{Pr}[y]\\ &= \mathbf{Pr}[A]\,.\end{aligned}$$

■

14.3.4.6 Linearity of Expectation

Lemma 14.3.22. *Let X, Y be two random variables (not necessarily independent) over a probability space $(\Omega, \mathbf{Pr})$. Then $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.*

Proof:

$$\begin{aligned}\mathbf{E}[X+Y] &= \sum_{\omega\in\Omega}\mathbf{Pr}[\omega]\,(X(\omega)+Y(\omega))\\ &= \sum_{\omega\in\Omega}\mathbf{Pr}[\omega]\,X(\omega) + \sum_{\omega\in\Omega}\mathbf{Pr}[\omega]\,Y(\omega) = \mathbf{E}[X] + \mathbf{E}[Y]\,.\end{aligned}$$

■

Corollary 14.3.23. $\mathbf{E}[a_1X_1 + a_2X_2 + \ldots + a_nX_n] = \sum_{i=1}^n a_i\,\mathbf{E}[X_i]$.

14.4 Analyzing Randomized Algorithms

14.4.0.7 Types of Randomized Algorithms

Typically one encounters the following types:

(A) ***Las Vegas randomized algorithms:*** for a given input x output of algorithm is always correct but the running time is a random variable. In this case we are interested in analyzing the *expected* running time.

(B) ***Monte Carlo randomized algorithms:*** for a given input x the running time is deterministic but the output is random; correct with some probability. In this case we are interested in analyzing the *probability* of the correct output (and also the running time).

(C) Algorithms whose running time and output may both be random variables.

14.4.0.8 Analyzing Las Vegas Algorithms

Deterministic algorithm Q for a problem Π:

(A) Let $Q(x)$ be the time for Q to run on input x of length $|x|$.
(B) Worst-case analysis: run time on worst input for a given size n.

$$T_{wc}(n) = \max_{x:|x|=n} Q(x).$$

Randomized algorithm R for a problem Π:

(A) Let $R(x)$ be the time for Q to run on input x of length $|x|$.
(B) $R(x)$ is a random variable: depends on random bits used by R.
(C) $\mathbf{E}[R(x)]$ is the expected running time for R on x
(D) Worst-case analysis: expected time on worst input of size n

$$T_{rand-wc}(n) = \max_{x:|x|=n} \mathbf{E}[Q(x)] .$$

14.4.0.9 Analyzing Monte Carlo Algorithms

Randomized algorithm M for a problem Π:

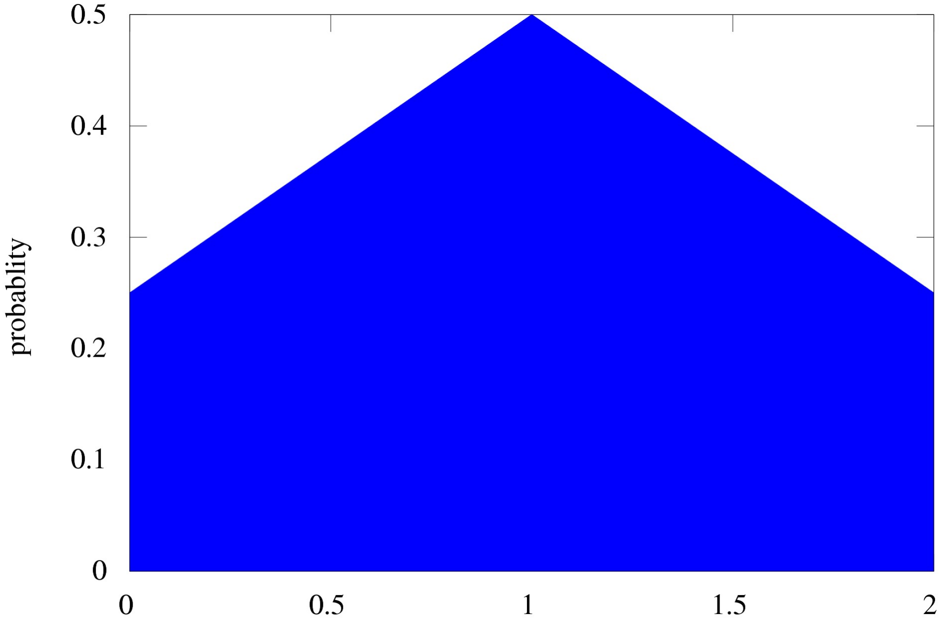
(A) Let $M(x)$ be the time for M to run on input x of length $|x|$. For Monte Carlo, assumption is that run time is deterministic.
(B) Let $\mathbf{Pr}[x]$ be the probability that M is correct on x.
(C) $\mathbf{Pr}[x]$ is a random variable: depends on random bits used by M.
(D) Worst-case analysis: success probability on worst input

$$P_{rand-wc}(n) = \min_{x:|x|=n} \mathbf{Pr}[x] .$$

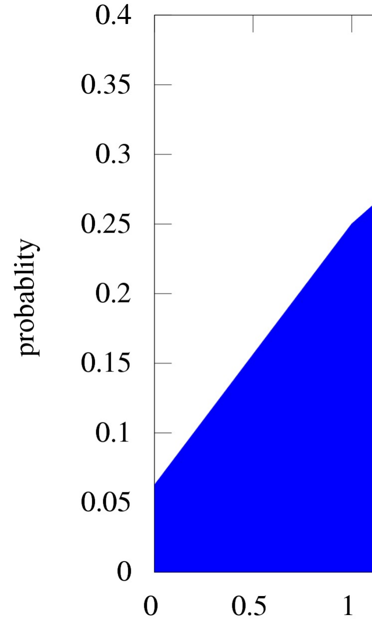
14.5 Why does randomization help?

14.5.0.10 Massive randomness.. Is not that random.

Consider flipping a fair coin n times independently, head given 1, tail gives zero. How many heads? ...we get a binomial distribution.

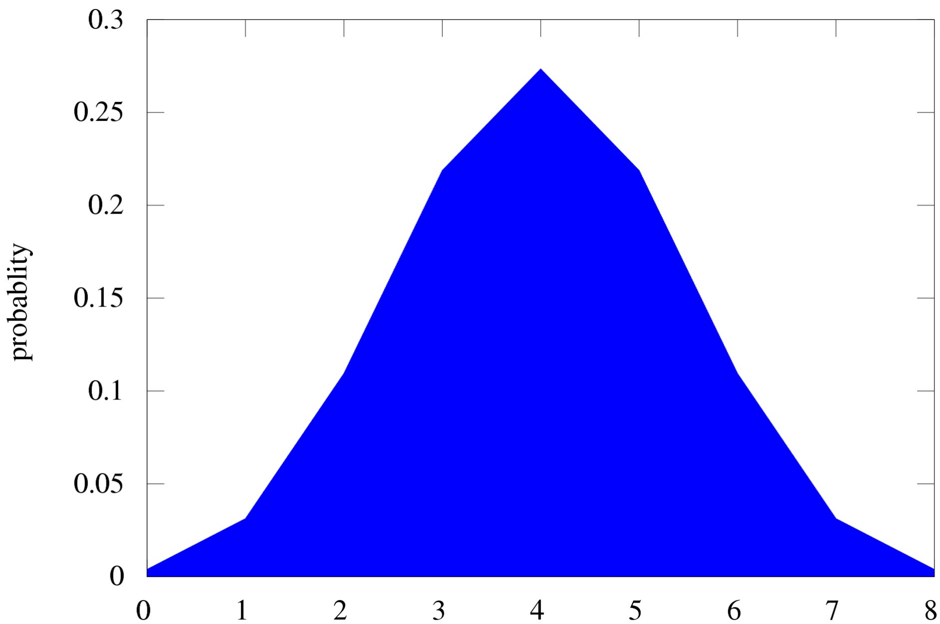


probablity
0.5
0.4
0.3
0.2
0.1
0
0
0.5
1
1.5
2

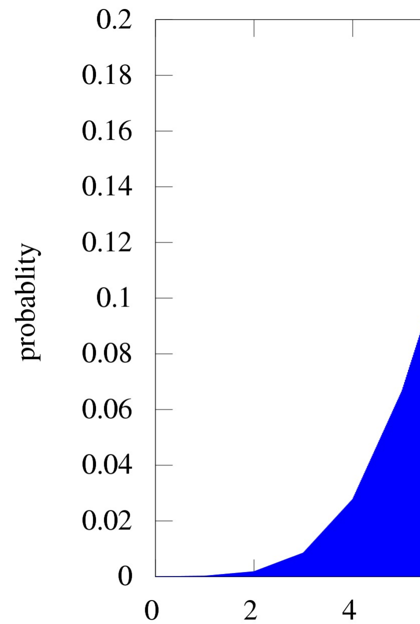


probablity
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
0.5
1

n = 2

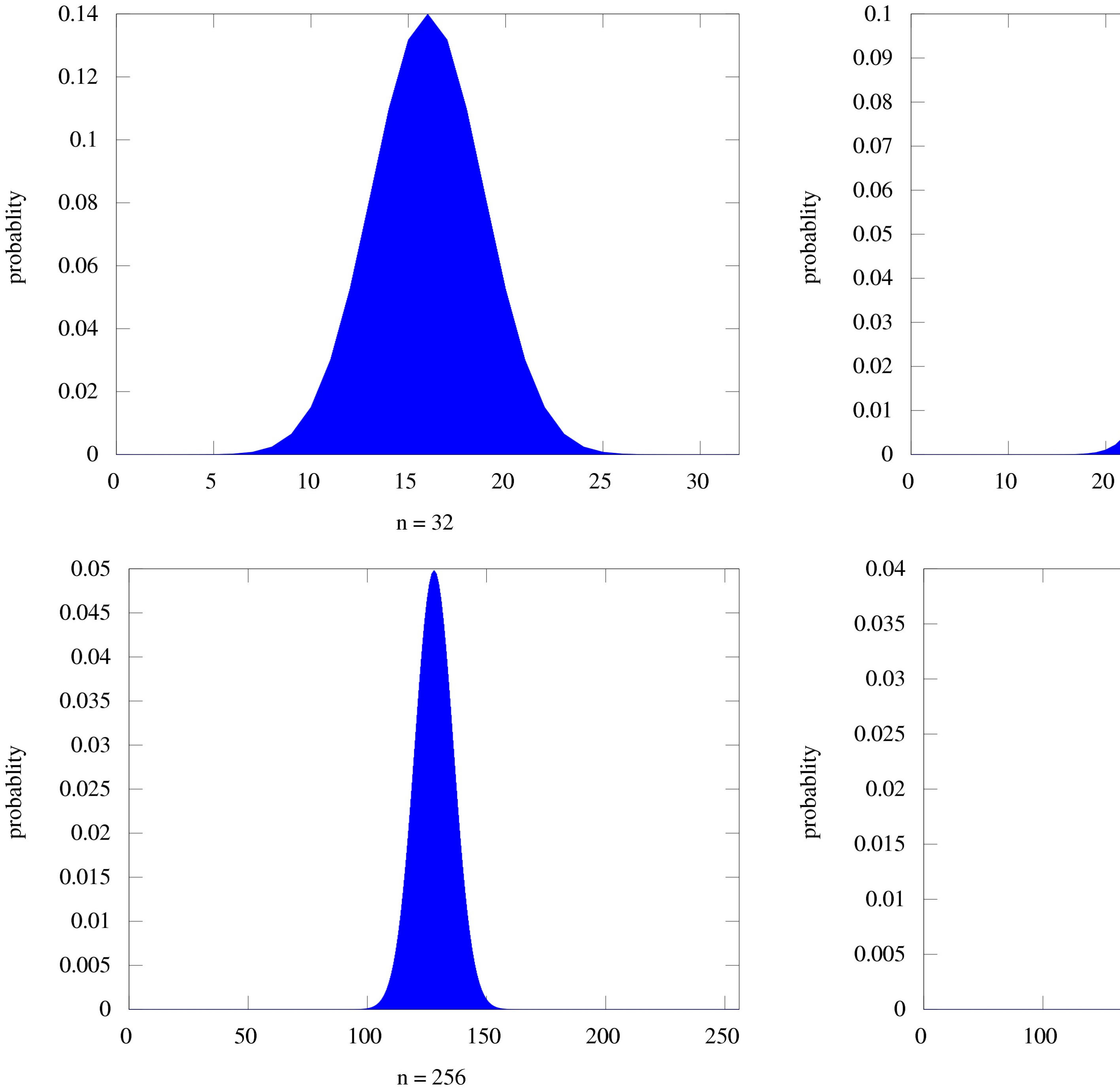


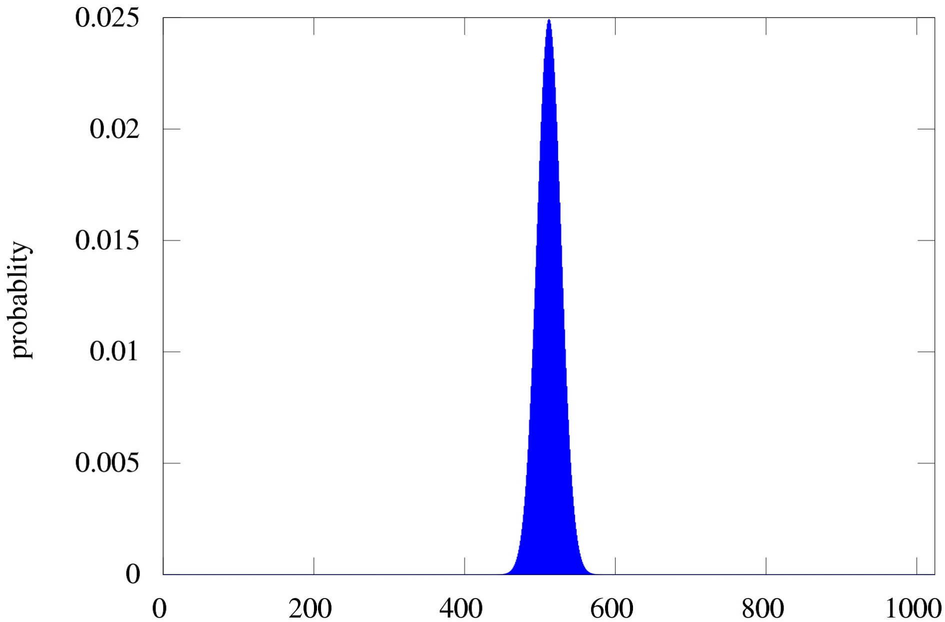
probablity
0.3
0.25
0.2
0.15
0.1
0.05
0
0
1
2
3
4
5
6
7
8



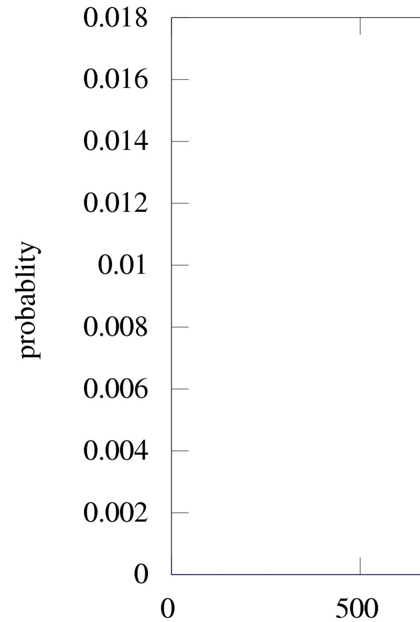
probablity
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
2
4

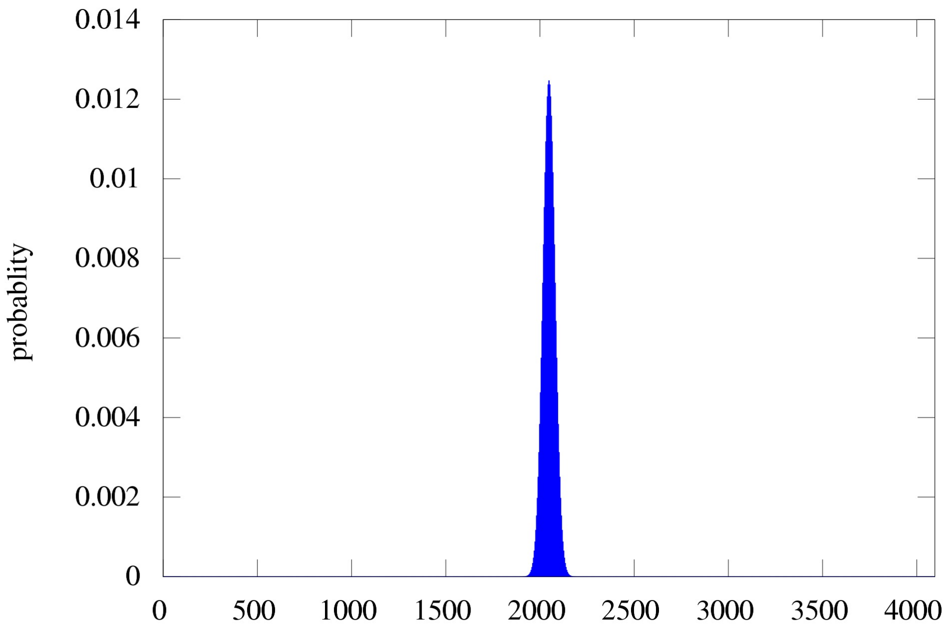
n = 8


probablity
n = 32
probablity
probablity
n = 256
probablity

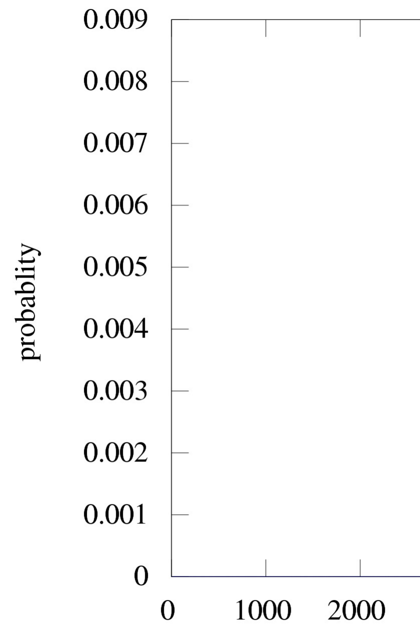

probablity
0.025
0.02
0.015
0.01
0.005
0
0
200
400
600
800
1000

n = 1024

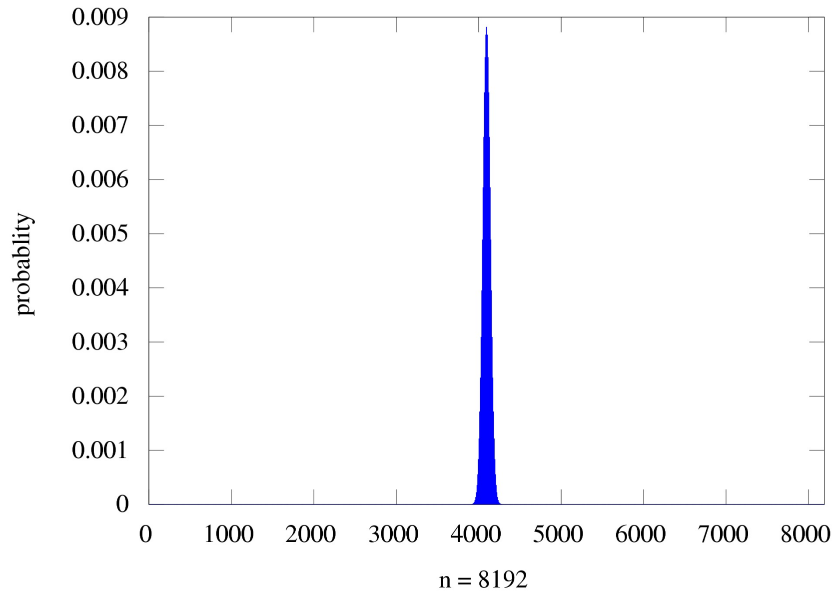

probablity
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
500


probablity
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
500
1000
1500
2000
2500
3000
3500
4000

n = 4096


probablity
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
1000
2000

14.5.0.11 Massive randomness.. Is not that random.



This is known as ***concentration of mass***.

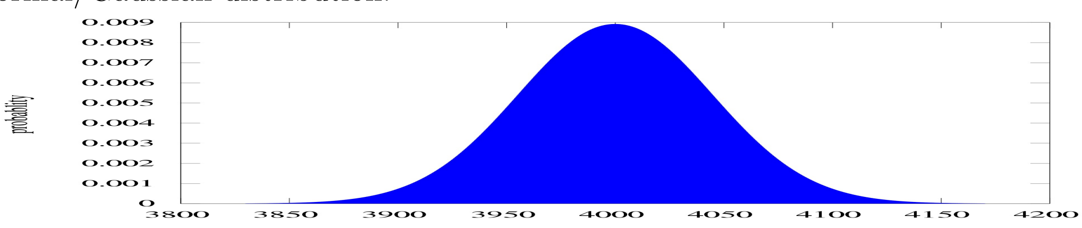
This is a very special case of the ***law of large numbers***.

14.5.1 Side note...

14.5.1.1 Law of large numbers (weakest form)...

Informal statement of law of large numbers

For n large enough, the middle portion of the binomial distribution looks like (converges to) the normal/Gaussian distribution.



14.5.1.2 Massive randomness.. Is not that random.

Intuitive conclusion

Randomized algorithm are unpredictable in the tactical level, but very predictable in the strategic level.

14.5.1.3 Binomial distribution

X_n = numbers of heads when flipping a coin n times.

Claim

$\mathbf{Pr}\Big[X_n = i\Big] = \frac{\binom{n}{i}}{2^n}$.

Where: $\binom{n}{k} = \frac{n!}{(n-k)!k!}$.

Indeed, $\binom{n}{i}$ is the number of ways to choose i elements out of n elements (i.e., pick which i coin flip come up heads).

Each specific such possibility (say 0100010...) had probability $1/2^n$.

We are interested in the bad event $\mathbf{Pr}[X_n \leq n/4]$ (way too few heads). We are going to prove this probability is tiny.

14.5.2 Binomial distribution

14.5.2.1 Playing around with binomial coefficients

Lemma 14.5.1. *$n! \geq (n/e)^n$.*

Proof:

$$\frac{n^n}{n!} \leq \sum_{i=0}^{\infty} \frac{n^i}{i!} = e^n,$$

by the Taylor expansion of $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. This implies that $(n/e)^n \leq n!$, as required. ■

14.5.3 Binomial distribution

14.5.3.1 Playing around with binomial coefficients

Lemma 14.5.2. *For any $k \leq n$, we have $\binom{n}{k} \leq \left(\frac{ne}{k}\right)^k$.*

Proof:

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{(n-k)!k!} = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!} \\ &\leq \frac{n^k}{k!} \leq \frac{n^k}{\left(\frac{k}{e}\right)^k} = \left(\frac{ne}{k}\right)^k.\end{aligned}$$

since $k! \geq (k/e)^k$ (by previous lemma). ■

14.5.4 Binomial distribution

14.5.4.1 Playing around with binomial coefficients

$$\mathbf{Pr}\Big[X_n \leq \frac{n}{4}\Big] = \sum_{k=0}^{n/4} \frac{1}{2^n}\binom{n}{k} = \frac{1}{2^n}\sum_{k=0}^{n/4}\binom{n}{k} \leq \frac{1}{2^n} 2 \cdot \binom{n}{n/4}$$

For $k \leq n/4$ the above sequence behave like a geometric variable.

$$\binom{n}{k+1} \Big/ \binom{n}{k} = \frac{n!}{(k+1)!(n-k-1)!} \Big/ \frac{n!}{(k)!(n-k)!}$$
$$= \frac{n-k}{k+1} \geq \frac{(3/4)n}{n/4+1} \geq 2.$$

14.5.5 Binomial distribution

14.5.5.1 Playing around with binomial coefficients

$$\mathbf{Pr}\Big[X_n \leq \frac{n}{4}\Big] \leq \frac{1}{2^n} 2 \cdot \binom{n}{n/4} \leq \frac{1}{2^n} 2 \cdot \left(\frac{ne}{n/4}\right)^{n/4} \leq 2 \cdot \left(\frac{4e}{2^4}\right)^{n/4}$$
$$\leq 2 \cdot 0.68^{n/4}.$$

We just proved the following theorem.

Theorem 14.5.3. *Let X_n be the random variable which is the number of heads when flipping an unbiased coin independently n times. Then*

$$\mathbf{Pr}\Big[X_n \leq \frac{n}{4}\Big] \leq 2 \cdot 0.68^{n/4} \text{ and } \mathbf{Pr}\Big[X_n \geq \frac{3n}{4}\Big] \leq 2 \cdot 0.68^{n/4}.$$

14.6 Randomized Quick Sort and Selection

14.7 Randomized Quick Sort

14.7.0.2 Randomized QuickSort

Randomized **QuickSort**

(A) Pick a pivot element *uniformly at random* from the array.
(B) Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
(C) Recursively sort the subarrays, and concatenate them.

14.7.0.3 Example

(A) array: 16, 12, 14, 20, 5, 3, 18, 19, 1

14.7.0.4 Analysis via Recurrence

(A) Given array A of size n, let $Q(A)$ be number of comparisons of randomized **QuickSort** on A.
(B) Note that $Q(A)$ is a random variable.
(C) Let A^i_{left} and A^i_{right} be the left and right arrays obtained if:
pivot is of rank i in A.

$$Q(A) = n + \sum_{i=1}^{n} \mathbf{Pr}\Big[\text{pivot has rank } i\Big] \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\right).$$

Since each element of A has probability exactly of $1/n$ of being chosen:

$$Q(A) = n + \sum_{i=1}^{n} \frac{1}{n} \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\right).$$

14.7.0.5 Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We have, for any A:

$$Q(A) = n + \sum_{i=1}^{n} \mathbf{Pr}\Big[\text{pivot has rank } i\Big] \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\right)$$

Therefore, by linearity of expectation:

$$\mathbf{E}\Big[Q(A)\Big] = n + \sum_{i=1}^{n} \mathbf{Pr}\left[\begin{array}{c}\text{pivot is}\\ \text{of rank } i\end{array}\right] \left(\mathbf{E}\Big[Q(A^i_{\text{left}})\Big] + \mathbf{E}\Big[Q(A^i_{\text{right}})\Big]\right).$$

$$\Rightarrow \quad \mathbf{E}\Big[Q(A)\Big] \le n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

14.7.0.6 Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} \mathbf{E}[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We derived:

$$\mathbf{E}\Big[Q(A)\Big] \le n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

Note that above holds for any A of size n. Therefore

$$\max_{A:|A|=n} \mathbf{E}[Q(A)] = T(n) \le n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

14.7.0.7 Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i) \right)$$

with base case $T(1) = 0$.

Lemma 14.7.1. $T(n) = O(n \log n)$.

Proof: (Guess and) Verify by induction. ∎

Chapter 15

Randomized Algorithms: QuickSort and QuickSelect

OLD CS 473: Fundamental Algorithms, Spring 2015
March 12, 2015

15.1 Slick analysis of QuickSort

15.1.0.8 A Slick Analysis of QuickSort

(A) Let $Q(A)$ be number of comparisons done on input array A:
  (A) R_{ij}: event that rank i element is compared with rank j element, for $1 \leq i < j \leq n$.
  (B) X_{ij} is the indicator random variable for R_{ij}. That is, $X_{ij} = 1$ if rank i is compared with rank j element, otherwise 0.

(B) $Q(A) = \sum_{1 \leq i < j \leq n} X_{ij}$.

(C) By linearity of expectation,

$$\mathbf{E}\Big[Q(A)\Big] = \mathbf{E}\left[\sum_{1 \leq i < j \leq n} X_{ij}\right] = \sum_{1 \leq i < j \leq n} \mathbf{E}\Big[X_{ij}\Big]$$

$$= \sum_{1 \leq i < j \leq n} \mathbf{Pr}\Big[R_{ij}\Big].$$

15.1.0.9 A Slick Analysis of QuickSort

$\boxed{R_{ij} = \text{rank } i \text{ element is compared with rank } j \text{ element.}}$

Question: What is $\mathbf{Pr}[R_{ij}]$?

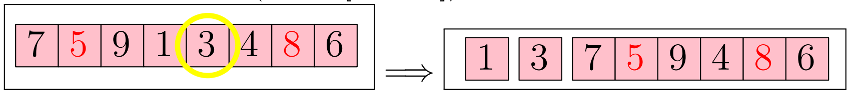
7	5	9	1	3	4	8	6

With ranks:

7	5	9	1	3	4	8	6
6	4	8	1	2	3	7	5

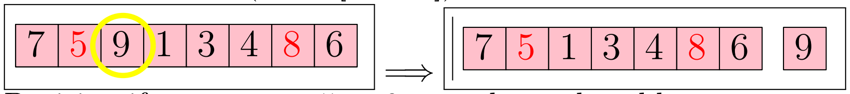
As such, probability of comparing 5 to 8 is $\mathbf{Pr}[R_{4,7}]$.

(A) If pivot too small (say 3 [rank 2]). Partition and call recursively:



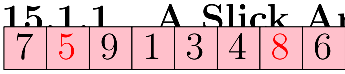
Decision if to compare 5 to 8 is moved to subproblem.

(B) If pivot too large (say 9 [rank 8]):



Decision if to compare 5 to 8 moved to subproblem.

15.1.1 A Slick Analysis of QuickSort

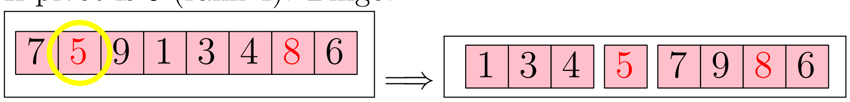


6 4 8 1 2 3 7 5

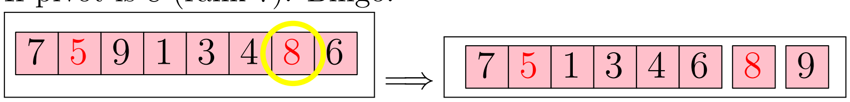
As such, probability of comparing 5 to 8 is $\mathbf{Pr}[R_{4,7}]$.

15.1.1.1 Question: What is $\mathbf{Pr}[R_{i,j}]$?

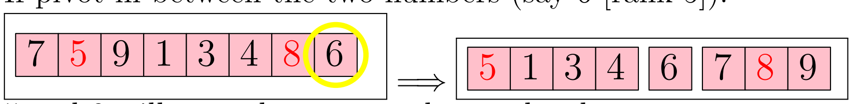
(A) If pivot is 5 (rank 4). Bingo!



(B) If pivot is 8 (rank 7). Bingo!



(C) If pivot in between the two numbers (say 6 [rank 5]):



5 and 8 will never be compared to each other.

15.1.2 A Slick Analysis of QuickSort

15.1.2.1 Question: What is $\mathbf{Pr}[R_{i,j}]$?

Conclusion:

$R_{i,j}$ happens $\iff$:

$$\boxed{\text{ith or jth ranked element is the first pivot out of the elements of rank } i, i+1, i+2, \ldots, j}$$

How to analyze this? Thinking acrobatics!

(A) Assign every element in array random priority (say in $[0,1]$).
(B) Choose pivot to be element with lowest priority in subproblem.
(C) Equivalent to picking pivot uniformly at random
(as **QuickSort** do).

15.1.3 A Slick Analysis of QuickSort

15.1.3.1 Question: What is $\mathbf{Pr}[R_{i,j}]$?

(A) Choosing a pivot using priorities
  (A) Assign every element in array is a random priority (in $[0, 1]$).
  (B) pivot = the element with lowest priority in subproblem.
(B) $\implies R_{i,j}$ happens if either i or j have lowest priority out of elements in rank $i \dots j$,
(C) There are $k = j - i + 1$ relevant elements.
(D) $\mathbf{Pr}\left[R_{i,j}\right] = \dfrac{2}{k} = \dfrac{2}{j - i + 1}$.

15.1.3.2 A Slick Analysis of QuickSort

Question: What is $\mathbf{Pr}[R_{ij}]$?

Lemma 15.1.1. $\mathbf{Pr}\left[R_{ij}\right] = \frac{2}{j-i+1}$.

Proof
(A) $a_1, \dots, a_i, \dots, a_j, \dots, a_n$: elements of A in sorted order. Let $S = \{a_i, a_{i+1}, \dots, a_j\}$
(B) **Observation:** If pivot is chosen outside S then all of S either in left or right recursive subproblem.
(C) **Observation:** a_i and a_j separated when a pivot is chosen from S for the first time. Once separated never to meet again. $\implies$ a_i and a_j will not be compared.

15.1.4 A Slick Analysis of QuickSort

15.1.4.1 Continued...

Lemma 15.1.2. $\mathbf{Pr}\left[R_{ij}\right] = \frac{2}{j-i+1}$.

Proof:
(A) Let $a_1, \dots, a_i, \dots, a_j, \dots, a_n$ be sort of A.
(B) Let $S = \{a_i, a_{i+1}, \dots, a_j\}$
(C) **Observation:** a_i is compared with $a_j \iff$ either a_i or a_j is chosen as a pivot from S at separation.
(D) **Observation:** Given: Pivot chosen from S.
The probability that it is a_i or a_j is exactly
$2/|S| = 2/(j - i + 1)$ since the pivot is chosen uniformly at random from the array.

■

15.1.5 A Slick Analysis of QuickSort

15.1.5.1 Continued...

$$\mathbf{E}\left[Q(A)\right] = \sum_{1 \le i < j \le n} \mathbf{E}[X_{ij}] = \sum_{1 \le i < j \le n} \mathbf{Pr}[R_{ij}] .$$

Lemma 15.1.3. $\mathbf{Pr}[R_{ij}] = \frac{2}{j-i+1}$.

$$
\begin{aligned}
\mathbf{E}\Big[Q(A)\Big] &= \sum_{1 \le i < j \le n} \mathbf{Pr}\Big[R_{ij}\Big] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \qquad = 2\sum_{i=1}^{n-1} \sum_{i<j}^{n} \frac{1}{j-i+1} \\
&\le 2\sum_{i=1}^{n-1} (H_{n-i+1} - 1) \ \le\ 2 \sum_{1 \le i < n} H_n \\
&\le 2nH_n = O(n \log n)
\end{aligned}
$$

15.2 Quick sort with high probability

15.2.1 Yet another analysis of QuickSort

15.2.1.1 You should never trust a man who has only one way to spell a word

(A) Consider element e in the array.
(B) $S_1, S_2, \dots, S_k$: subproblems e participates in during **QuickSort** execution:
(C) **Definition**
e is lucky in the jth iteration if $|S_j| \le (3/4)\,|S_{j-1}|$.
(D) **Key observation**: The event that e is lucky in jth iteration...
(E) ... is independent of the event that e is lucky in kth iteration, (If $j \ne k$)
(F) $X_j = 1 \iff e$ is lucky in the jth iteration.

15.2.2 Yet another analysis of QuickSort

15.2.2.1 Continued...

Claim

$\mathbf{Pr}[X_j = 1] = 1/2$.

Proof:
(A) X_j determined by j recursive subproblem.
(B) Subproblem has $n_{j-1} = |X_{j-1}|$ elements.
(C) jth pivot rank $\in [n_{j-1}/4, (3/4)n_{j-1}] \implies e$ lucky in jth iter.
(D) Prob. e is lucky $\ge |[n_{j-1}/4, (3/4)n_{j-1}]| \,/ n_{j-1} = 1/2$.

■

Observation

If $X_1 + X_2 + \dots X_k = \lceil \log_{4/3} n \rceil$ then e subproblem is of size one. Done!

15.2.3 Yet another analysis of QuickSort

15.2.3.1 Continued...

Observation

Probability e participates in $\geq k = 40\lceil \log_{4/3} n \rceil$ subproblems. Is equal to

$$\begin{aligned}
&\mathbf{Pr}\Big[X_1 + X_2 + \dots + X_k \leq \lceil \log_{4/3} n \rceil\Big] \\
&\qquad \leq \mathbf{Pr}[X_1 + X_2 + \dots + X_k \leq k/4] \\
&\qquad \leq 2 \cdot 0.68^{k/4} \leq 1/n^5.
\end{aligned}$$

Conclusion

QuickSort takes $O(n \log n)$ time with high probability.

15.3 Randomized Selection

15.3.0.2 Randomized Quick Selection

Input Unsorted array A of n integers

Goal Find the jth smallest number in A (*rank* j number)

Randomized Quick Selection

(A) Pick a pivot element *uniformly at random* from the array
(B) Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
(C) Return pivot if rank of pivot is j.
(D) Otherwise recurse on one of the arrays depending on j and their sizes.

15.3.0.3 Algorithm for Randomized Selection

Assume for simplicity that A has distinct elements.

```
QuickSelect(A, j):
    Pick pivot x uniformly at random fr
    Partition A into A_less, x, and A_grea
    if (|A_less| = j - 1) then
        return x
    if (|A_less| ≥ j) then
        return QuickSelect(A_less, j)
    else
        return QuickSelect(A_greater, j -
```

15.3.0.4 QuickSelect analysis

(A) $S_1, S_2, \ldots, S_k$ be the subproblems considered by the algorithm.
Here $|S_1| = n$.

(B) S_i would be ***successful*** if $|S_i| \leq (3/4)\,|S_{i-1}|$

(C) Y_1 = number of recursive calls till first successful iteration.
Clearly, total work till this happens is $O(Y_1 n)$.

(D) n_i = size of the subproblem immediately after the $(i-1)$th successful iteration.

(E) Y_i = number of recursive calls after the $(i-1)$th successful call, till the ith successful iteration.

(F) Running time is $O(\sum_i n_i Y_i)$.

15.3.0.5 QuickSelect analysis

Example

S_i = subarray used in ith recursive call

$|S_i|$ = size of this subarray

Red indicates successful iteration.

Inst'	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
$\|S_i\|$	100	70	60	50	40	30	25	5	2
Succ'	$Y_1 = 2$		$Y_2 = 4$				$Y_3 = 2$		$Y_4 = 1$
$n_i =$	$n_1 = 100$		$n_2 = 60$				$n_3 = 25$		$n_4 = 2$

(A) All the subproblems after $(i-1)$th successful iteration till ith successful iteration have size $\leq n_i$.

(B) Total work: $O(\sum_i n_i Y_i)$.

15.3.0.6 QuickSelect analysis

(A) Total work: $O(\sum_i n_i Y_i)$.

(B) $n_i \leq (3/4)n_{i-1} \leq (3/4)^{i-1}n$.

(C) Y_i is a random variable with geometric distribution
Probability of $Y_i = k$ is $1/2^i$.

(D) $\mathbf{E}[Y_i] = 2$.

(E) As such, expected work is proportional to

$$\mathbf{E}\left[\sum_i n_i Y_i\right] = \sum_i \mathbf{E}\left[n_i Y_i\right] \leq \sum_i \mathbf{E}\left[(3/4)^{i-1} n Y_i\right]$$
$$= n \sum_i (3/4)^{i-1}\,\mathbf{E}\left[Y_i\right] = n \sum_{i=1} (3/4)^{i-1} 2 \leq 8n.$$

15.3.0.7 QuickSelect analysis

Theorem 15.3.1. *The expected running time of* **QuickSelect** *is* $O(n)$.

15.3.1 QuickSelect analysis via recurrence

15.3.1.1 Analysis via Recurrence

(A) Given array A of size n let $Q(A)$ be number of comparisons of randomized selection on A for selecting rank j element.
(B) Note that $Q(A)$ is a random variable
(C) Let A^i_{less} and A^i_{greater} be the left and right arrays obtained if pivot is rank i element of A.
(D) Algorithm recurses on A^i_{less} if $j < i$ and recurses on A^i_{greater} if $j > i$ and terminates if $j = i$.

$$\begin{aligned} Q(A) &= n + \sum_{i=1}^{j-1} \mathbf{Pr}[\text{pivot has rank } i]\, Q(A^i_{\text{greater}}) \\ &\quad + \sum_{i=j+1}^{n} \mathbf{Pr}[\text{pivot has rank } i]\, Q(A^i_{\text{less}}) \end{aligned}$$

15.3.1.2 Analyzing the Recurrence

As in **QuickSort** we obtain the following recurrence where $T(n)$ is the worst-case expected time.

$$T(n) \leq n + \frac{1}{n}\left(\sum_{i=1}^{j-1} T(n-i) + \sum_{i=j}^{n} T(i-1)\right).$$

Theorem 15.3.2. $T(n) = O(n)$.

Proof: (Guess and) Verify by induction (see next slide). ∎

15.3.1.3 Analyzing the recurrence

Theorem 15.3.3. $T(n) = O(n)$.

Prove by induction that $T(n) \leq \alpha n$ for some constant $\alpha \geq 1$ to be fixed later.
Base case: $n = 1$, we have $T(1) = 0$ since no comparisons needed and hence $T(1) \leq \alpha$.
Induction step: Assume $T(k) \leq \alpha k$ for $1 \leq k < n$ and prove it for $T(n)$. We have by the recurrence:

$$\begin{aligned} T(n) &\leq n + \frac{1}{n}\left(\sum_{i=1}^{j-1} T(n-i) + \sum_{i=j^n} T(i-1)\right) \\ &\leq n + \frac{\alpha}{n}\left(\sum_{i=1}^{j-1} (n-i) + \sum_{i=j}^{n} (i-1)\right) \quad \text{by applying induction} \end{aligned}$$

15.3.1.4 Analyzing the recurrence

$$
\begin{aligned}
T(n) &\leq n + \frac{\alpha}{n}(\sum_{i=1}^{j-1}(n-i) + \sum_{i=j}^{n}(i-1)) \\
&\leq n + \frac{\alpha}{n}((j-1)(2n-j)/2 + (n-j+1)(n+j-2)/2) \\
&\leq n + \frac{\alpha}{2n}(n^2 + 2nj - 2j^2 - 3n + 4j - 2) \\
& \quad \text{above expression maximized when } j = (n+1)/2\text{: calculus} \\
&\leq n + \frac{\alpha}{2n}(3n^2/2 - n) \quad \text{substituting } (n+1)/2 \text{ for } j \\
&\leq n + 3\alpha n/4 \\
&\leq \alpha n \quad \text{for any constant } \alpha \geq 4
\end{aligned}
$$

15.3.1.5 Comments on analyzing the recurrence

(A) Algebra looks messy but intuition suggest that the median is the hardest case and hence can plug $j = n/2$ to simplify without calculus

(B) Analyzing recurrences comes with practice and after a while one can see things more intuitively

John Von Neumann:

Young man, in mathematics you don't understand things. You just get used to them.

Chapter 16

Hashing

OLD CS 473: Fundamental Algorithms, Spring 2015
March 17, 2015

16.1 Hash Tables

16.2 Introduction

16.2.0.6 Dictionary Data Structure

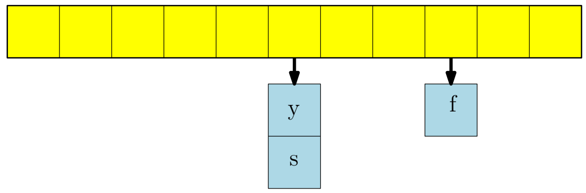
(A) $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
(B) Data structure to store a subset $S \subseteq \mathcal{U}$
(C) **Operations:**
  (A) **Search**/**lookup**: given $x \in \mathcal{U}$ is $x \in S$?
  (B) **Insert**: given $x \notin S$ add x to S.
  (C) **Delete**: given $x \in S$ delete x from S
(D) ***Static*** structure: S given in advance or changes very infrequently, main operations are lookups.
(E) ***Dynamic*** structure: S changes rapidly so inserts and deletes as important as lookups.

16.2.0.7 Dictionary Data Structures

Common solutions:
(A) Static:
  (A) Store S as a *sorted* array
  (B) **Lookup**: Binary search in $O(\log |S|)$ time (comparisons)
(B) Dynamic:
  (A) Store S in a *balanced* binary search tree
  (B) Lookup, Insert, Delete in $O(\log |S|)$ time (comparisons)

16.2.0.8 Dictionary Data Structures II

(A) **Question:** "Should Tables be Sorted?"
(also title of famous paper by Turing award winner Andy Yao)
(B) Hashing is a widely used & powerful technique for dictionaries.
(C) **Motivation:**



- (A) Universe $\mathcal{U}$ may not be (naturally) totally ordered.
- (B) Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
- (C) Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

16.2.0.9 Hashing and Hash Tables

- (A) Hash Table data structure:
  - (A) A (hash) table/array T of size m (the table ***size***).
  - (B) A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$.
  - (C) Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T.
- (B) Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?
- (C) ...

  Ideal situation:
  - (A) Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
  - (B) **Lookup**: Given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!
- (D) Collisions unavoidable. Several different techniques to handle them.

16.2.0.10 Handling Collisions: Chaining

- (A) **Collision:** $h(x) = h(y)$ for some $x \neq y$.
- (B) **Chaining** to handle collisions:
  - (A) For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list
  - (B) **Lookup**: to find if $y \in \mathcal{U}$ is in T, check the linked list at $T[h(y)]$. Time proportion to size of linked list.
- (C) This is also known as ***Open hashing***.

16.2.0.11 Handling Collisions

Several other techniques:

- (A) Open addressing.
  Every element has a list of places it can be (in certain order). Check in this order.
- (B) ...
- (C) Cuckoo hashing.
  Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick stored value to its other location. Repeat till stable. if no stability then rebuild table.
- (D) Others.

16.2.0.12 Understanding Hashing

(A) Does hashing give $O(1)$ time per operation for dictionaries?
(B) **Questions:**
   (A) Complexity of evaluating h on a given element?
   (B) Relative sizes of the universe $\mathcal{U}$ and the set to be stored S.
   (C) Size of table relative to size of S.
   (D) Worst-case vs average-case vs randomized (expected) time?
   (E) How do we choose h?

16.2.0.13 Understanding Hashing

(A) Considerations:
   (A) Complexity of evaluating h on a given element? Should be small.
   (B) Relative sizes of the universe $\mathcal{U}$ and the set to be stored S: typically $|\mathcal{U}| \gg |S|$.
   (C) Size of table relative to size of S. The ***load factor*** of T is the ratio n/t where $n = |S|$ and $m = |T|$. Typically n/t is a small constant smaller than 1.
   Also known as the ***fill factor***.
(B) Main and interrelated questions:
   (A) Worst-case vs average-case vs randomized (expected) time?
   (B) How do we choose h?

16.2.0.14 Single hash function

(A) $\mathcal{U}$: universe (very large).
(B) Assume $N = |\mathcal{U}| \gg m$ where m is size of table T. In particular assume $N \geq m^2$ (very conservative).
(C) Fix hash function $h : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
(D) N items hashed to m slots. By pigeon hole principle there is some $i \in \{0, \ldots, m-1\}$ such that $N/m \geq m$ elements of $\mathcal{U}$ get hashed to i (!).
(E) Implies that there is a set $S \subseteq \mathcal{U}$ where $|S| = m$ such that all of S hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

16.2.0.15 Picking a hash function

(A) How to pick functions?
   (A) Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
   (B) Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.
(B) Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$
   (A) $\mathcal{H}$ is a ***family*** of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
   (B) h is chosen *randomly* from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
   (C) Randomized guarantee: should have the property that for any *fixed* set $S \subseteq \mathcal{U}$ of size m the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of h.

16.2.0.16 Picking a hash function II

(A) **Question:** Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?
(B) (A) Too many functions! A random function has high complexity!
    # of functions: $M = m^{|\mathcal{U}|}$.
    Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.
(C) **Question:** Are there good and compact families $\mathcal{H}$?
    (A) Yes... But what it means for $\mathcal{H}$ to be good and compact.

16.3 Universal Hashing

16.3.0.17 Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?
(A) Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \le i < m$.
(B) Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
(C) Second property is stronger than the first and the crucial issue.

Definition 16.3.1. *A family hash function $\mathcal{H}$ is* **2-universal** *if for all distinct $x, y \in \mathcal{U}$, $Pr[h(x) = h(y)] = 1/m$ where m is the table size.*

Note: The set of all hash functions satisfies stronger properties!

16.3.0.18 Analyzing Uniform Hashing

(A) T is hash table of size m.
(B) $S \subseteq \mathcal{U}$ is a ***fixed*** set of size $\le m$.
(C) h is chosen randomly from a uniform hash family $\mathcal{H}$.
(D) x is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $x \notin S$.

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

16.3.0.19 Analyzing Uniform Hashing

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?
(A) The time to look up x is the size of the list at $T[h(x)]$: same as the number of elements in S that collide with x under h.
(B) Let $\ell(x)$ be this number. We want $\mathrm{E}[\ell(x)]$
(C) For $y \in S$ let A_y be the even that x, y collide and D_y be the corresponding indicator variable.

16.3.1 Analyzing Uniform Hashing

16.3.1.1 Continued...

Number of elements colliding with x: $\ell(x) = \sum_{y \in S} D_y$.

$$
\begin{aligned}
\Rightarrow \mathrm{E}[\ell(x)] &= \sum_{y \in S} \mathrm{E}[D_y] \qquad \text{linearity of expectation} \\
&= \sum_{y \in S} Pr[h(x) = h(y)] \\
&= \sum_{y \in S} \frac{1}{m} \qquad \text{since } \mathcal{H} \text{ is a uniform hash family} \\
&= |S|/m \\
&\leq 1 \quad \text{if } |S| \leq m
\end{aligned}
$$

16.3.1.2 Analyzing Uniform Hashing

(A) **Question:** What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

(B) **Answer:** $O(n/m)$.

(C) Comments:

  (A) $O(1)$ expected time also holds for insertion.

  (B) Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.

  (C) ***Worst-case***: look up time can be large! How large? $\Omega(\log n / \log \log n)$
  [Lower bound holds even under stronger assumptions.]

16.3.2 Rehashing, amortization and...

16.3.2.1 ... making the hash table dynamic

Previous analysis assumed fixed S of size $\simeq m$.

Question: What happens as items are inserted and deleted?

(A) If $|S|$ grows to more than cm for some constant c then hash table performance clearly degrades.

(B) If $|S|$ stays around $\simeq m$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

(A) Choose a new table size based on current number of elements in table.

(B) Choose a new random hash function and rehash the elements.

(C) Discard old table and hash function.

Question: When to rebuild? How expensive?

16.3.2.2 Rebuilding the hash table

(A) Start with table size m where m is some estimate of $|S|$ (can be some large constant).

(B) If $|S|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.

(C) If $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say 10), rebuild.

The *amortize* cost of rebuilding to previously performed operations. Rebuilding ensures $O(1)$ expected analysis holds even when S changes. Hence $O(1)$ expected look up/insert/delete time *dynamic* data dictionary data structure!

16.3.2.3 Some math required...

Lemma 16.3.2. *Let p be a prime number,*
x: an integer number in $\{1, \ldots, p-1\}$.
$\implies$ *There exists a unique y s.t. $xy = 1 \mod p$.*

In other words: For every element there is a unique inverse.
$\implies$ $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ when working module p is a field.

16.3.2.4 Proof of lemma

Claim 16.3.3. *Let p be a prime number. For any $\alpha, \beta, i \in \{1, \ldots, p-1\}$ s.t. $\alpha \neq \beta$, we have that $\alpha i \neq \beta i \mod p$.*

Proof: Assume for the sake of contradiction $\alpha i = \beta i \mod p$. Then

$$\begin{aligned} i(\alpha - \beta) = 0 \quad \bmod p \\ \implies \quad & p \text{ divides } \; i(\alpha - \beta) \\ \implies \quad & p \text{ divides } \; \alpha - \beta \\ \implies \quad & \alpha - \beta = 0 \\ \implies \quad & \alpha = \beta. \end{aligned}$$

And that is a contradiction. ■

16.3.3 Proof of lemma...

16.3.3.1 Uniqueness.

Lemma 16.3.4. *Let p be a prime number,*
x: an integer number in $\{1, \ldots, p-1\}$.
$\implies$ *There exists a unique y s.t. $xy = 1 \mod p$.*

Proof: Assume the lemma is false and there are two distinct numbers $y, z \in \{1, \ldots, p-1\}$ such that

$$xy = 1 \mod p \quad \text{and} \quad xz = 1 \mod p.$$

But this contradicts the above claim (set $i = x$, $\alpha = y$ and $\beta = z$). ■

16.3.4 Proof of lemma...

16.3.4.1 Existence

Proof: By claim, for any $\alpha \in \{1, \ldots, p-1\}$ we have that $\{\alpha * 1 \mod p, \alpha * 2 \mod p, \ldots, \alpha * (p-1) \mod p\} = \{1, 2, \ldots, p-1\}$.

$\implies$ There exists a number $y \in \{1, \ldots, p-1\}$ such that $\alpha y = 1 \mod p$. ∎

16.3.4.2 Constructing Universal Hash Families

Parameters: $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$

(A) Choose a ***prime*** number $p \geq N$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.

(B) For $a, b \in \mathbb{Z}_p$, $a \neq 0$, define the hash function $h_{a,b}$ as $h_{a,b}(x) = ((ax + b) \mod p) \mod m$.

(C) Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem 16.3.5. *$\mathcal{H}$ is a 2-universal hash family.*

Comments:

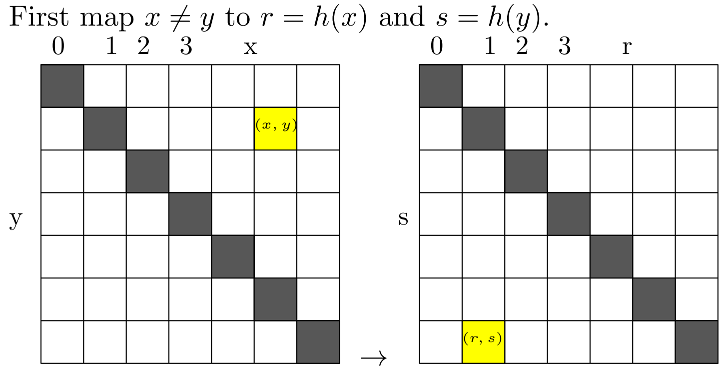
(A) Hash family is of small size, easy to sample from.

(B) Easy to store a hash function (a, b have to be stored) and evaluate it.

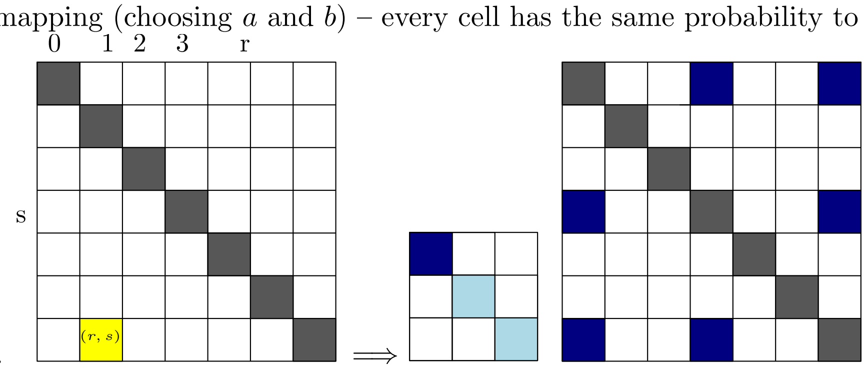
16.3.4.3 What the is going on?

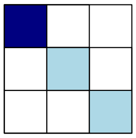
$h_{a,b}(x) = ((ax + b) \mod p) \mod m$

First map $x \neq y$ to $r = h(x)$ and $s = h(y)$.

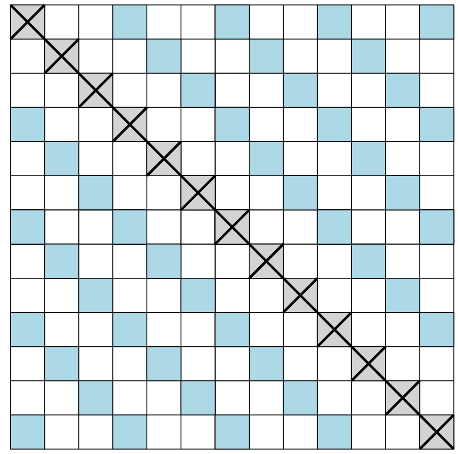


This is a random uniform mapping (choosing a and b) – every cell has the same probability to be the target, for fixed x and y.



$\Longrightarrow$ 

(A) First part of mapping maps (x, y) to a random location $(h_{a,b}(x), h_{a,b}(y))$ in the "matrix".
(B) $(h_{a,b}(x), h_{a,b}(y))$ is not on main diagonal.
(C) All blue locations are "bad" – map by mod m to a location of collusion.
(D) But... at most $1/m$ fraction of allowable locations in the matrix are bad.



16.3.4.4 Constructing Universal Hash Families

Theorem 16.3.6. *$\mathcal{H}$ is a (2)-universal hash family.*

Proof: Fix $x, y \in \mathcal{U}$. What is the probability they will collide if h is picked randomly from $\mathcal{H}$?
(A) Let a, b be *bad* for x, y if $h_{a,b}(x) = h_{a,b}(y)$.
(B) **Claim:** Number of bad pairs is at most $p(p-1)/m$.
(C) Total number of hash functions is $p(p-1)$ and hence probability of a collision is $\leq 1/m$. ■

16.3.4.5 Some Lemmas

Lemma 16.3.7. *If $x \neq y$ then for any $a, b \in \mathbb{Z}_p$ such that $a \neq 0$, we have*
$$ax + b \mod p \neq ay + b \mod p.$$

Proof: If $ax + b \mod p = ay + b \mod p$ then $a(x - y) \mod p = 0$ and $a \neq 0$ and $(x - y) \neq 0$. However, a and $(x - y)$ cannot divide p since p is prime and $a < p$ and $(x - y) < p$. ■

16.3.4.6 Some Lemmas

Lemma 16.3.8. *If $x \neq y$ then for each (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ there is exactly **one** a, b such that* $ax + b \mod p = r$ *and* $ay + b \mod p = s$

Proof: Solve the two equations:

$$ax + b = r \mod p \qquad \text{and} \qquad ay + b = s \mod p$$

We get $a = \frac{r-s}{x-y} \mod p$ and $b = r - ax \mod p$. ■

16.3.4.7 Understanding the hashing

Once we fix a and b, and we are given a value x, we compute the hash value of x in two stages:
(A) ***Compute***: $r \leftarrow (ax + b) \mod p$.
(B) ***Fold***: $r' \leftarrow r \mod m$

Collision...

Given two values x and y they might collide because of either steps.

Lemma 16.3.9. *# not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.*

16.3.4.8 Folding numbers

Lemma 16.3.10. *# not equal pairs of $\mathbb{Z}_p \times \mathbb{Z}_p$ that are folded to the same number is $p(p-1)/m$.*

Proof: Consider a pair $(x, y) \in \{0, 1, \ldots, p-1\}^2$ s.t. $x \neq y$. Fix x:
(A) There are $\lceil p/m \rceil$ values of y that fold into x. That is

$$x \mod m = y \mod m.$$

(B) One of them is when $x = y$.
(C) $\implies$ # of colliding pairs $(\lceil p/m \rceil - 1)p \leq (p-1)p/m$

■

16.3.5 Proof of Claim

16.3.5.1 # of bad pairs is $p(p-1)/m$

Proof: Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.
(A) Let $ax + b \mod p = r$ and $ay + b \mod = s \mod p$.
(B) Collision if and only if $r = s \mod m$.
(C) (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p-1$ and $r = s \mod m$ is $p(p-1)/m$.
(D) From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

■

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

16.3.5.2 Perfect Hashing

(A) **Question:** Can we make look up time $O(1)$ in worst case?
(B) Yes, for static dictionaries but then space usage is $O(m)$ only in expectation.

16.3.5.3 Practical Issues

Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement for other objects need to map objects in some fashion to integers (via representation)

- Practical methods for various important cases such as vectors, strings are studied extensively. See `http://en.wikipedia.org/wiki/Universal_hashing` for some pointers.

- Recent important paper briding theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See `http://en.wikipedia.org/wiki/Tabulation_hashing`

16.3.5.4 Bloom Filters

(A) **Hashing:**
  (A) To insert x in dictionary store x in table in location $h(x)$
  (B) To lookup y in dictionary check contents of location $h(y)$
(B) **Bloom Filter:** tradeoff space for false positives
  (A) Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
  (B) To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
  (C) To lookup y if bit in location $h(y)$ is 1 say yes, else no.

16.3.5.5 Bloom Filters

(A) **Bloom Filter:** tradeoff space for false positives
  (A) To insert x in dictionary set *bit* to 1 in location $h(x)$ (initially all bits are set to 0)
  (B) To lookup y if bit in location $h(y)$ is 1 say yes, else no
  (C) No false negatives but false positives possible due to collisions
(B) Reducing false positives:
  (A) Pick k hash functions $h_1, h_2, \ldots, h_k$ *independently*
  (B) To insert x for $1 \leq i \leq k$ set bit in location $h_i(x)$ in table i to 1
  (C) To lookup y compute $h_i(y)$ for $1 \leq i \leq k$ and say yes only if each bit in the corresponding location is 1, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with k independent hash function it is α^k.

16.3.5.6 Take away points

(A) Hashing is a powerful and important technique for dictionaries. Many practical applications.
(B) Randomization fundamental to understanding hashing.
(C) Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
(D) Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
(E) Many applications in practice.

Chapter 17

Network Flows

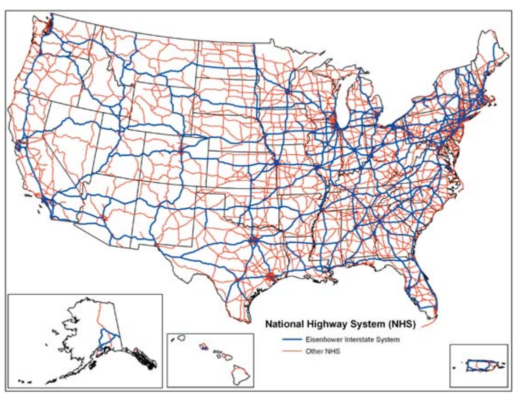
OLD CS 473: Fundamental Algorithms, Spring 2015
March 19, 2015

17.0.5.7 Everything flows

Panta rei – everything flows (literally).
Heraclitus (535–475 BC)

17.1 Network Flows: Introduction and Setup

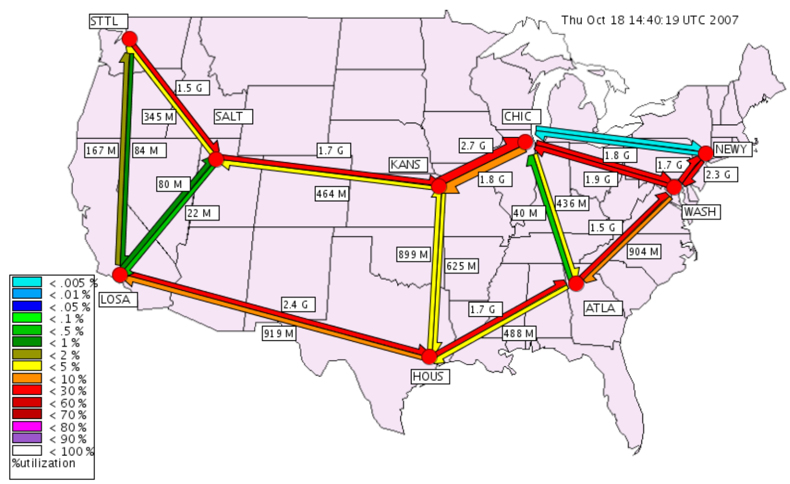
17.1.0.8 Transportation/Road Network



17.1.0.9 Internet Backbone Network

17.1.0.10 Common Features of Flow Networks

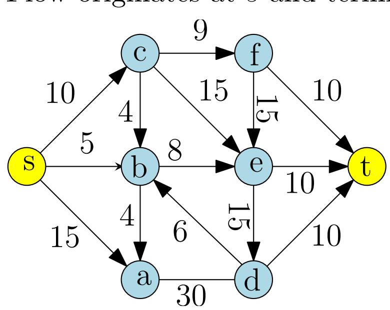
(A) ***Network*** represented by a (directed) *graph* $G = (V, E)$.
(B) Each edge e has a ***capacity*** $c(e) \geq 0$ that limits amount of *traffic* on e.
(C) *Source(s)* of traffic/data.
(D) *Sink(s)* of traffic/data.
(E) Traffic *flows* from sources to sinks.
(F) Traffic is *switched/interchanged* at nodes.
(G) ***Flow*** abstract term to indicate stuff (traffic/data/etc) that *flows* from sources to sinks.



17.1.0.11 Single Source/Single Sink Flows

Simple setting:

(A) Single source s and single sink t.
(B) Every other node v is an ***internal*** node.
(C) Flow originates at s and terminates at t.



(A) Each edge e has a capacity $c(e) \geq 0$.
(B) Sometimes assume:
Source $s \in V$ has no incoming edges, and sink $t \in V$ has no outgoing edges.

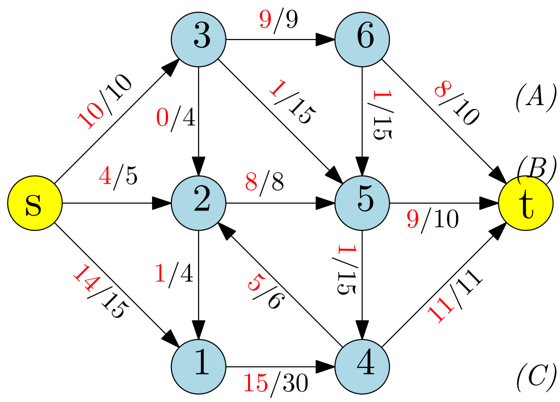
(A) **Assumptions:** All capacities are integer, and every vertex has at least one edge incident to it.

17.1.0.12 Definition of Flow

(A) Two ways to define flows...
(B) edge based, or
(C) path based.
(D) Essentially equivalent but have different uses.
(E) Edge based definition is more compact.

17.1.0.13 Edge Based Definition of Flow

Definition 17.1.1. **Flow** *in network* $G = (V, E)$, *is function* $f : E \to \mathbb{R}^{\geq 0}$ *s.t.*



(A) ***Capacity Constraint:*** *For each edge e, $f(e) \leq c(e)$.*

(B) ***Conservation Constraint:*** *For each vertex $v \neq s, t$.*

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

(C) ***Value of flow***= *(total flow out of source) − (total flow in to source).*

Figure 17.1: Flow with value.

17.1.0.14 Flow...

Conservation of flow law is also known as ***Kirchhoff's law***.

17.1.0.15 More Definitions and Notation

Notation

(A) The inflow into a vertex v is $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$ and the outflow is $f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$

(B) For a set of vertices A, $f^{\text{in}}(A) = \sum_{e \text{ into } A} f(e)$. Outflow $f^{\text{out}}(A)$ is defined analogously

Definition 17.1.2. *For a network $G = (V, E)$ with source s, the* **value** *of flow f is defined as $v(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$.*

17.1.0.16 A Path Based Definition of Flow

Intuition: Flow goes from source s to sink t along a path.

$\mathcal{P}$: set of all paths from s to t. $|\mathcal{P}|$ can be *exponential* in n.

Definition 17.1.3 (Flow by paths.). *A* **flow** *in network $G = (V, E)$, is function $f : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ s.t.*

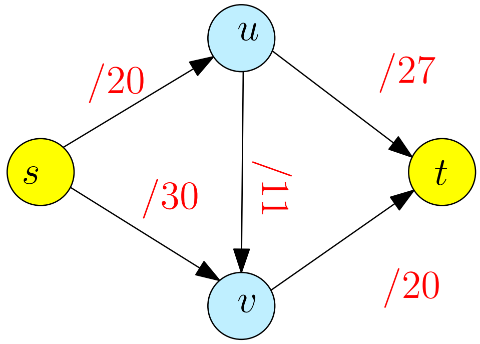
(A) ***Capacity Constraint:*** *For each edge e, total flow on e is $\leq c(e)$.*

$$\sum_{p \in \mathcal{P} : e \in p} f(p) \leq c(e)$$

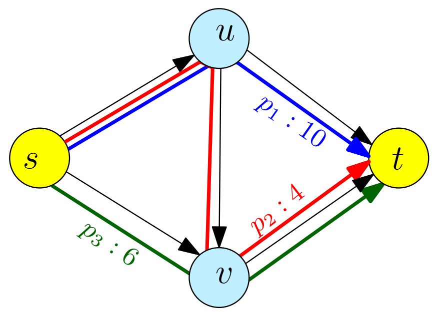
(B) ***Conservation Constraint:*** *No need! Automatic.*

Value of flow: $\sum_{p \in \mathcal{P}} f(p)$.

17.1.0.17 Example



$\mathcal{P} = \{p_1, p_2, p_3\}$
$p_1 : s \to u \to t$
$p_2 : s \to u \to v \to t$
$p_3 : s \to v \to t$
$f(p_1) = 10, f(p_2) = 4, f(p_3) = 6$

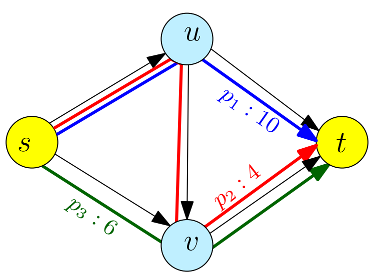


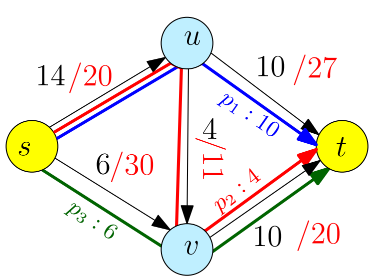
17.1.0.18 Path based flow implies edge based flow

Lemma 17.1.4. *Given a path based flow* $f : \mathcal{P} \to \mathbb{R}^{\geq 0}$ *there is an edge based flow* $f' : E \to \mathbb{R}^{\geq 0}$ *of the same value.*

Proof: For each edge e define $f'(e) = \sum_{p:e \in p} f(p)$.
Exercise: Verify capacity and conservation constraints for f'.
Exercise: Verify that value of f and f' are equal ∎

17.1.0.19 Example





$\mathcal{P} = \{p_1, p_2, p_3\}$
$p_1 : s \to u \to t$
$p_2 : s \to u \to v \to t$
$p_3 : s \to v \to t$
$f(p_1) = 10, f(p_2) = 4, f(p_3) = 6$
$f'(s \to u) = 14$
$f'(u \to v) = 4$
$f'(s \to v) = 6$
$f'(u \to t) = 10$
$f'(v \to t) = 10$

17.1.1 Flow Decomposition

17.1.1.1 Edge based flow to Path based Flow

Lemma 17.1.5. *Given an edge based flow $f_1 : E \to \mathbb{R}^{\geq 0}$, there is a path based flow $f : \mathcal{P} \to \mathbb{R}^{\geq 0}$ of same value. Moreover, f assigns non-negative flow to at most m paths where $|E| = m$ and $|V| = n$. Given f_1, the path based flow can be computed in $O(mn)$ time.*

17.1.2 Flow Decomposition

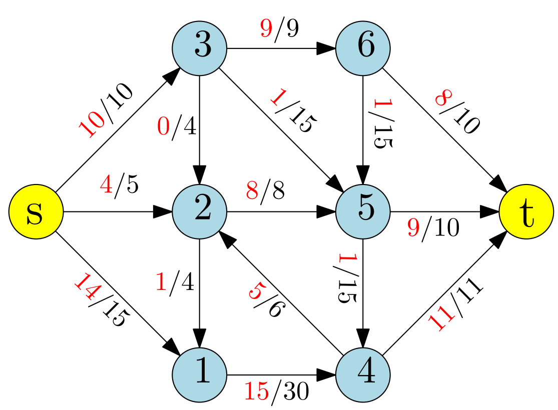
17.1.2.1 Edge based flow to Path based Flow

Proof:[Proof Idea]

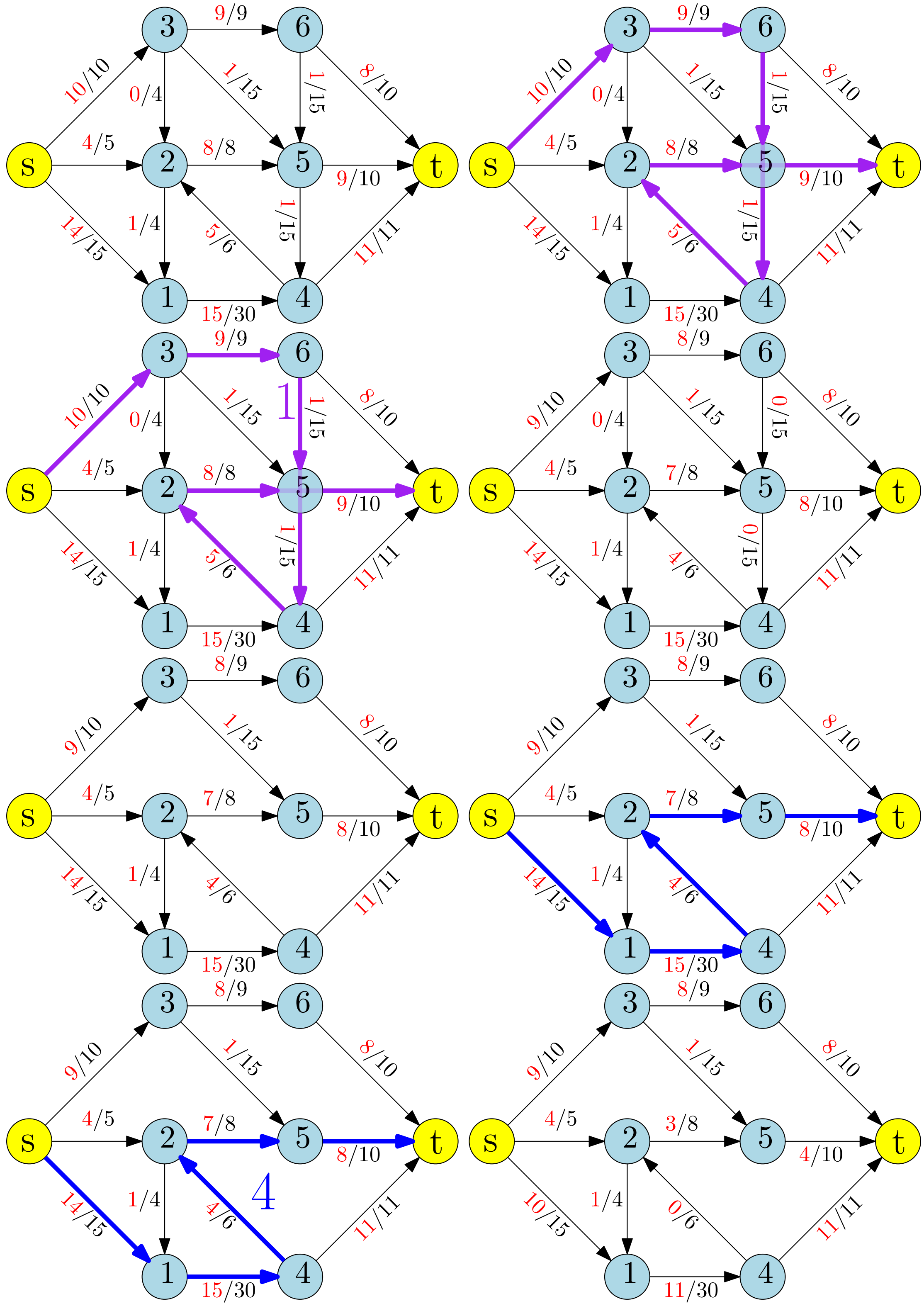
(A) Remove all edges with $f_1(e) = 0$.
(B) Find a path p from s to t.
(C) Assign $f(p)$ to be $\min_{e \in p} f_1(e)$.
(D) Reduce $f_1(e)$ for all $e \in p$ by $f(p)$.
(E) Repeat until no path from s to t.
(F) In each iteration at least on edge has flow reduced to zero.
(G) Hence, at most m iterations. Can be implemented in $O(m(m+n))$ time. $O(mn)$ time requires care.

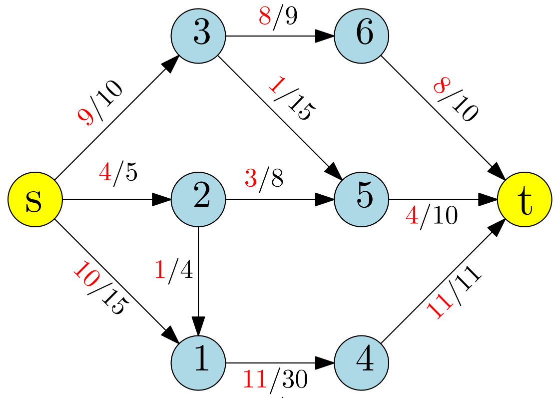
■

17.1.2.2 Example

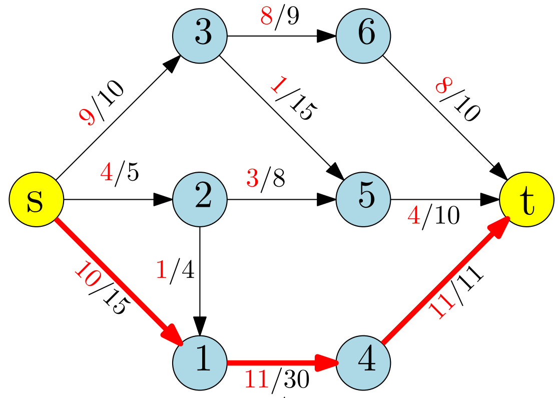


17.1.2.3 Example

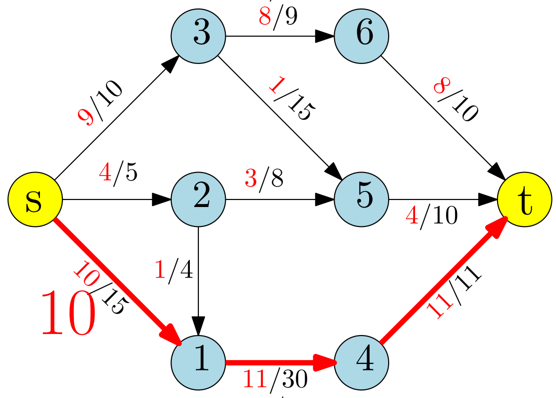




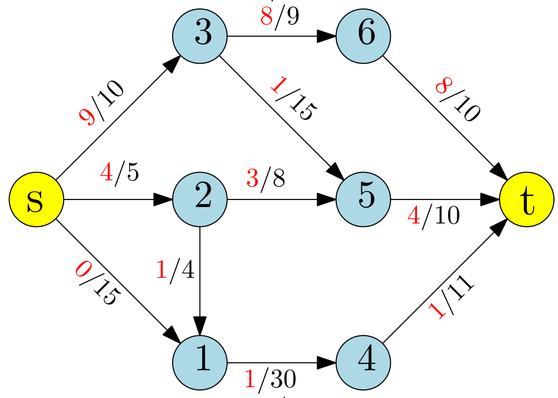
s
3
2
1
6
5
4
t
9/10
4/5
10/15
8/9
1/15
3/8
1/4
8/10
4/10
11/11
11/30



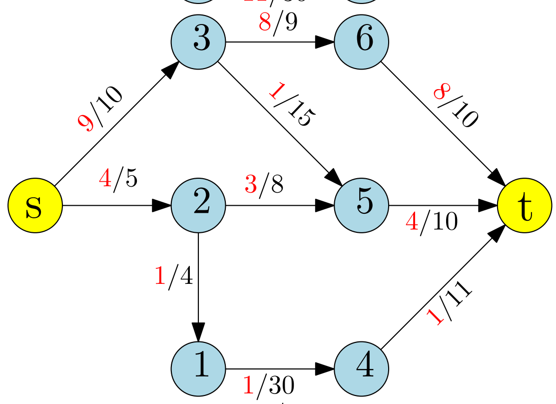
s
3
2
1
6
5
4
t
9/10
4/5
10/15
8/9
1/15
3/8
1/4
8/10
4/10
11/11
11/30



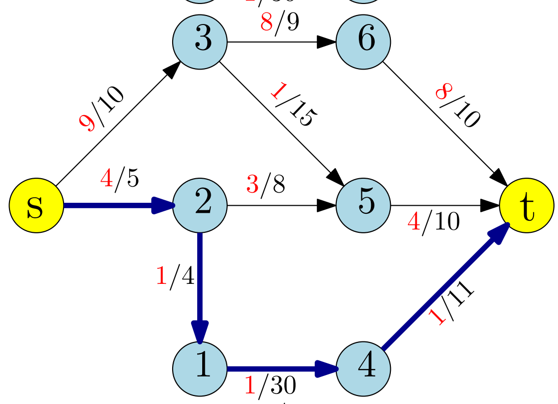
s
3
2
1
6
5
4
t
9/10
4/5
10/15
10
8/9
1/15
3/8
1/4
8/10
4/10
11/11
11/30



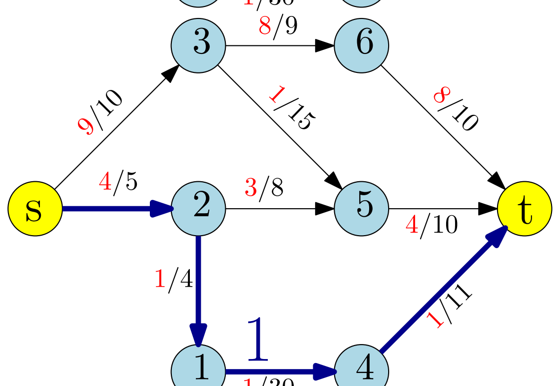
s
3
2
1
6
5
4
t
9/10
4/5
0/15
8/9
1/15
3/8
1/4
8/10
4/10
1/11
1/30



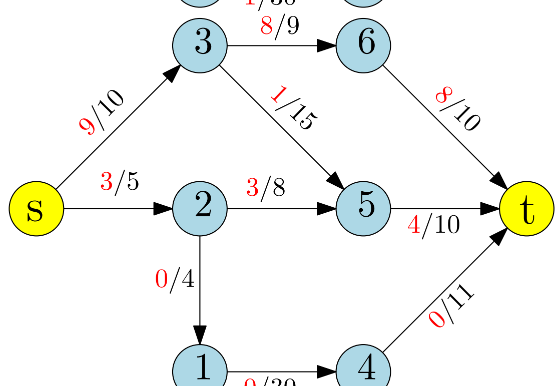
s
3
2
1
6
5
4
t
9/10
4/5
8/9
1/15
3/8
1/4
8/10
4/10
1/11
1/30



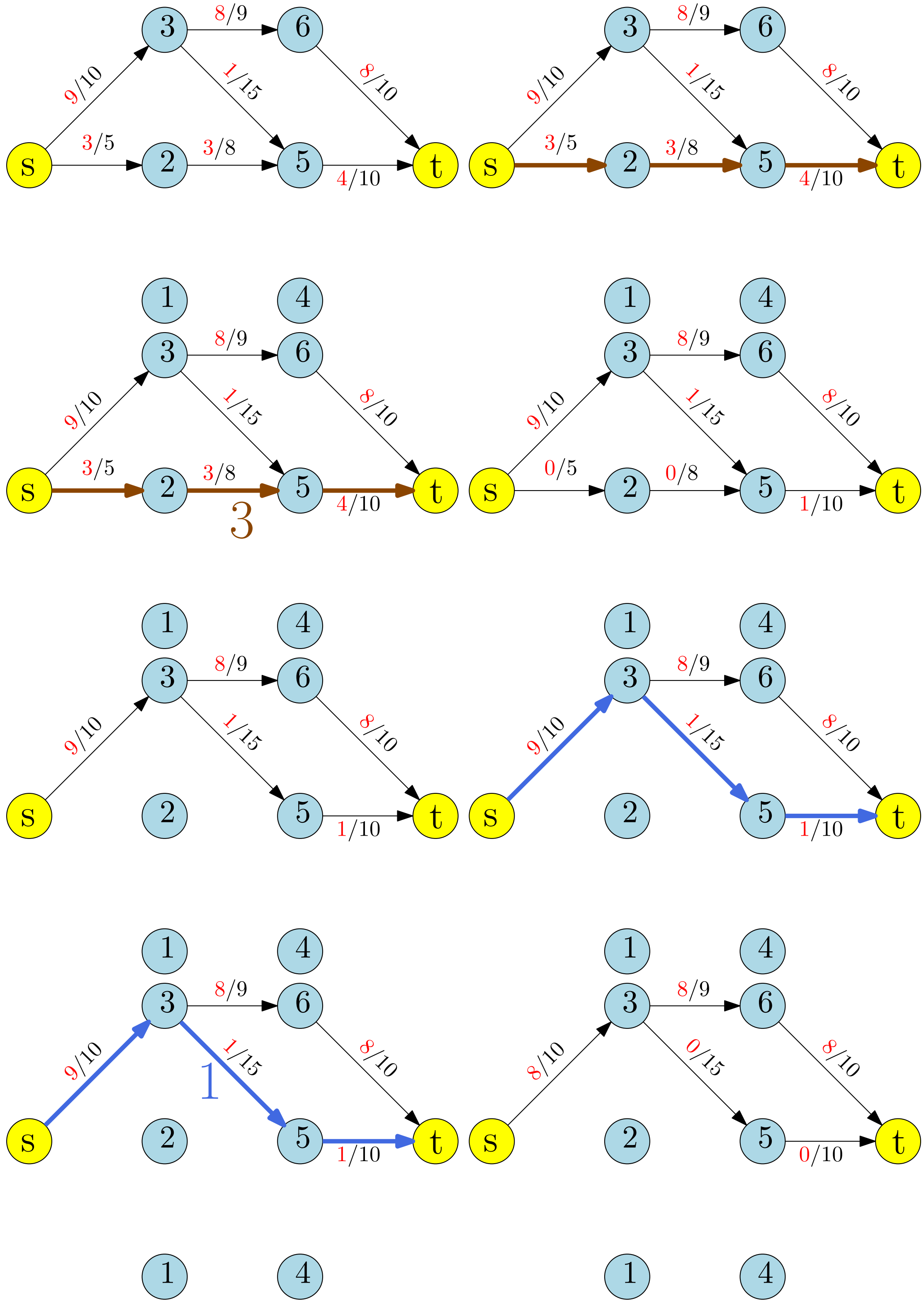
s
3
2
1
6
5
4
t
9/10
4/5
8/9
1/15
3/8
1/4
8/10
4/10
1/11
1/30

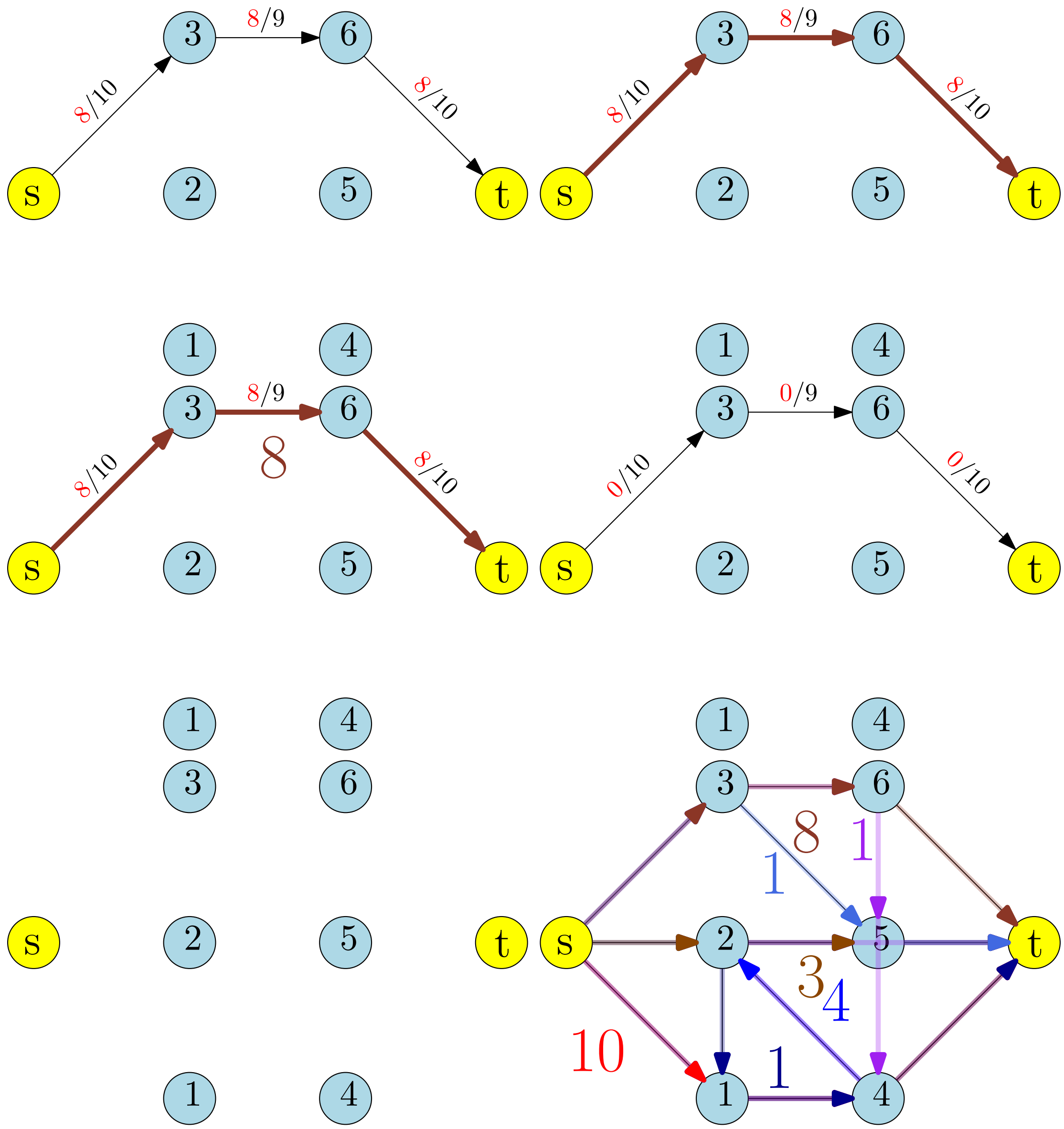


s
3
2
1
6
5
4
t
9/10
4/5
8/9
1/15
3/8
1/4
1
8/10
4/10
1/11
1/30



s
3
2
1
6
5
4
t
9/10
3/5
8/9
1/15
3/8
0/4
8/10
4/10
0/11
0/30

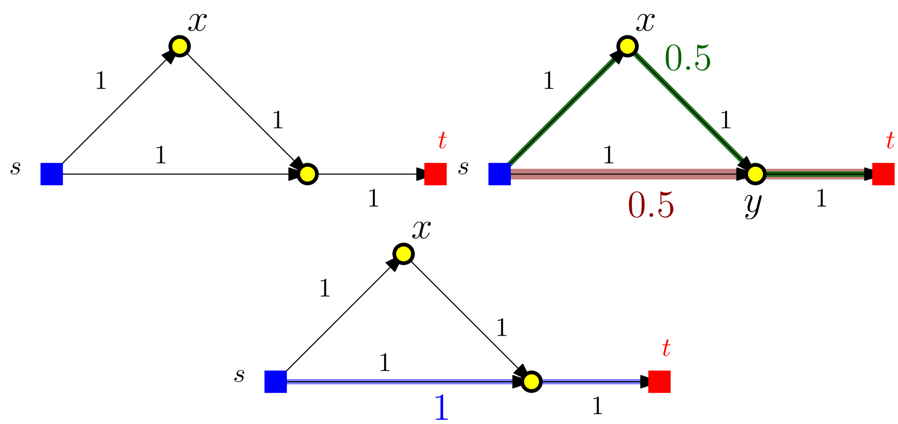

s 3 6 2 5 t 9/10 8/9 1/15 8/10 3/5 3/8 4/10
s 3 6 2 5 t 9/10 8/9 1/15 8/10 3/5 3/8 4/10
1 4 s 3 6 2 5 t 9/10 8/9 1/15 8/10 3/5 3/8 4/10 3
1 4 s 3 6 2 5 t 9/10 8/9 1/15 8/10 0/5 0/8 1/10
1 4 s 3 6 2 5 t 9/10 8/9 1/15 8/10 1/10
1 4 s 3 6 2 5 t 9/10 8/9 1/15 8/10 1/10
1 4 s 3 6 2 5 t 9/10 8/9 1/15 8/10 1/10 1
1 4 s 3 6 2 5 t 8/10 8/9 0/15 8/10 0/10
1 4
1 4



8/10
8/9
8/10
0/10
0/9
0/10
8
1
1
3
4
10
1

17.1.3 Path flow decomposition

17.1.3.1 Do not have to be efficient...



17.1.3.2 Edge vs Path based Definitions of Flow

(A) Edge based flows:
  (A) *compact* representation, only m values to be specified, and
  (B) need to check flow conservation explicitly at each internal node.
(B) Path flows:
  (A) in some applications, paths more natural,
  (B) not compact,
  (C) no need to check flow conservation constraints.
(C) Equivalence shows that we can go back and forth easily.

17.1.3.3 The Maximum-Flow Problem

(A) The ***network flow problem***: Problem

  Input A network G with capacity c and source s and sink t.

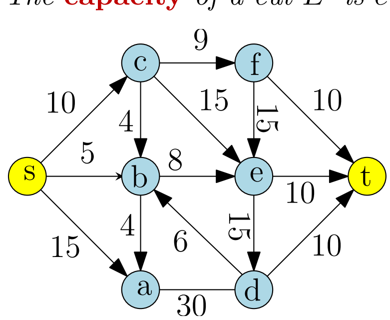
  Goal Find flow of *maximum* value.

(B) **Question:** Given a flow network, what is an *upper bound* on the maximum flow between source and sink?

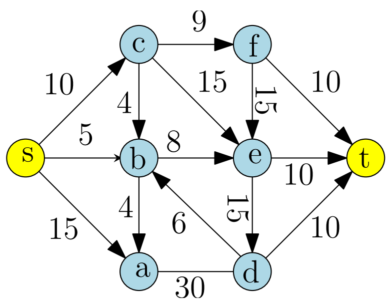
17.1.3.4 Cuts

Definition 17.1.6 (s-t cut). *Given a flow network an* **s-t cut** *is a set of edges* $E' \subset E$ *such that removing* E' disconnects s *from* t*: in other words there is no directed* $s \to t$ *path in* $E - E'$.

The **capacity** *of a cut* E' *is* $c(E') = \sum_{e \in E'} c(e)$.

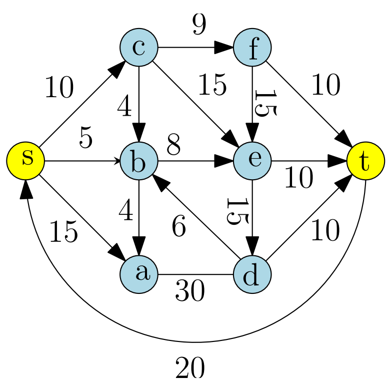


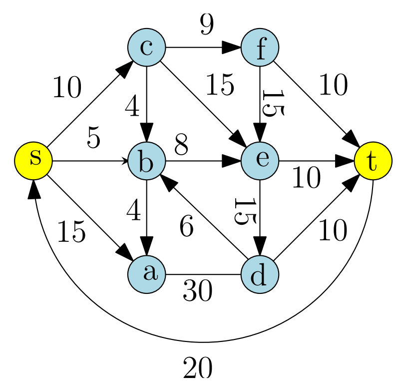
Caution:
(A) Cut may leave $t \to s$ paths!
(B) There might be many s-t cuts.



17.1.4 $s - t$ cuts

17.1.4.1 A death by a thousand cuts





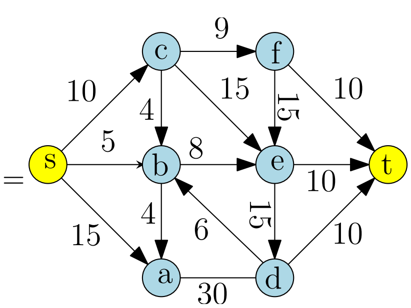
17.1.4.2 Minimal Cut

Definition 17.1.7 (Minimal s-t cut.). *Given a s-t flow network $G = (V, E)$, $E' \subseteq E$ is a* **minimal cut** *if for all $e \in E'$, if $E' \setminus \{e\}$ is not a cut.*

Observation: given a cut E', can check efficiently whether E' is a minimal cut or not. How?

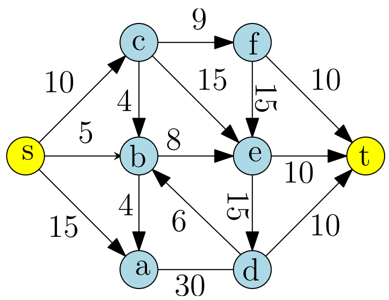
17.1.4.3 Cuts as Vertex Partitions

(A) Let $A \subset V$ such that
  (A) $s \in A$, $t \notin A$, and
  (B) $B = V \setminus A$ (hence $t \in B$).
(B) The ***cut*** (A, B) is the set of edges (A, B) = $\{(u, v) \in E \mid u \in A, v \in B\}$.
  Cut (A, B) is set of edges leaving A.



Claim 17.1.8. *(A, B) is an s-t cut.*

Proof: Let P be any $s \to t$ path in G. Since t is not in A, P has to leave A via some edge (u, v) in (A, B). ∎



17.1.4.4 Cuts as Vertex Partitions

Lemma 17.1.9. *Suppose E' is an s-t cut. Then there is a cut (A, B) such that $(A, B) \subseteq E'$.*

Proof: E' is an s-t cut implies no path from s to t in $(V, E - E')$.

(A) Let A be set of all nodes reachable by s in $(V, E - E')$.
(B) Since E' is a cut, $t \notin A$.
(C) $(A, B) \subseteq E'$. Why?If some edge $(u, v) \in (A, B)$ is not in E' then v will be reachable by s and should be in A, hence a contradiction.

■

Corollary 17.1.10. *Every* minimal *s-t cut E' is a cut of the form (A, B).*

17.1.4.5 Minimum Cut

Definition 17.1.11. *Given a flow network an s-t* **minimum** *cut is a cut E' of smallest capacity among all s-t cuts.*

Observation: exponential number of s-t cuts and no "easy" algorithm to find a minimum cut.

17.1.4.6 The Minimum-Cut Problem

Problem

Input A flow network G

Goal Find the capacity of a *minimum* s-t cut

17.1.4.7 Flows and Cuts

Lemma 17.1.12. *For any s-t cut E',* **maximum** *s-t flow $\leq$ capacity of E'.*

Proof:

(A) Formal proof easier with path based definition of flow.
(B) Suppose $f : \mathcal{P} \to \mathbb{R}^{\geq 0}$ is a max-flow.
(C) Every path $p \in \mathcal{P}$ contains an edge $e \in E'$. Why?
(D) Assign each path $p \in \mathcal{P}$ to exactly one edge $e \in E'$.
(E) Let $\mathcal{P}_e$ be paths assigned to $e \in E'$. Then

$$v(f) = \sum_{p \in \mathcal{P}} f(p) = \sum_{e \in E'} \sum_{p \in \mathcal{P}_e} f(p) \leq \sum_{e \in E'} c(e).$$

■

17.1.4.8 Flows and Cuts

Lemma 17.1.13. *For any s-t cut E',* **maximum** *s-t flow $\leq$ capacity of E'.*

Corollary 17.1.14. *Maximum s-t flow $\leq$ minimum s-t cut.*

17.1.4.9 Max-Flow Min-Cut Theorem

Theorem 17.1.15. *In any flow network:*

$$\Bigl(\textit{value of maximum s-t flow}\Bigr) = \Bigl(\textit{capacity of minimum s-t cut}\Bigr).$$

(A) Can compute minimum-cut from maximum flow and vice-versa!
(B) Proof coming shortly.
(C) Many applications:
  (A) optimization
  (B) graph theory
  (C) combinatorics

17.1.4.10 The Maximum-Flow Problem

Problem

Input A network G with capacity c and source s and sink t.

Goal Find flow of *maximum* value from s to t.

Exercise: Given G, s, t as above, show that one can remove all edges into s and all edges out of t without affecting the flow value between s and t.

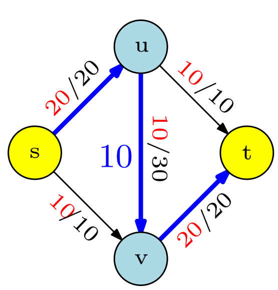
Chapter 18

Network Flow Algorithms

OLD CS 473: Fundamental Algorithms, Spring 2015
March 31, 2015

18.1 Algorithm(s) for Maximum Flow

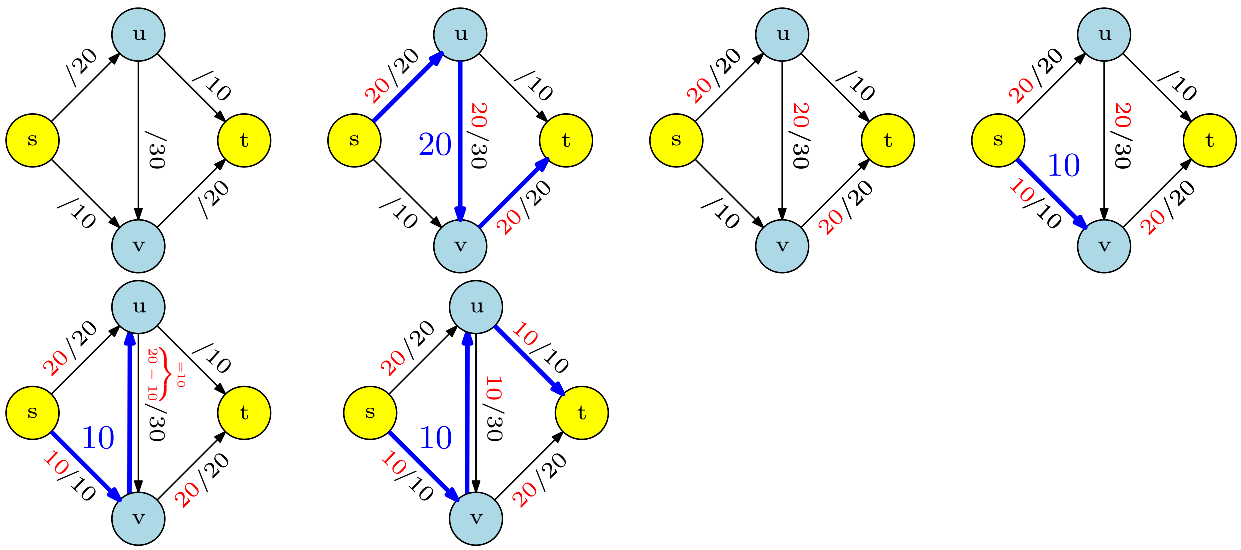
18.1.0.11 Greedy Approach



(A) Begin with $f(e) = 0$ for each edge.
(B) Find a s-t path P with $f(e) < c(e)$ for every edge $e \in P$.
(C) ***Augment*** flow along this path.
(D) Repeat augmentation for as long as possible.

18.1.1 Greedy Approach: Issues

18.1.1.1 Issues = What is this nonsense?



(A) Begin with $f(e) = 0$ for each edge
(B) Find a s-t path P with $f(e) < c(e)$ for every edge $e \in P$
(C) Augment flow along this path
(D) Repeat augmentation for as long as possible.

(A) Greedy can get stuck in sub-optimal flow!
(B) Need to "push-back" flow along edge (u, v).

18.2 Ford-Fulkerson Algorithm

18.2.1 Residual Graph

18.2.1.1 The "leftover" graph

Definition 18.2.1. *For a network $G = (V, E)$ and flow f, the* **residual graph** *$G_f = (V', E')$ of G with respect to f is*

(A) $V' = V$,
(B) **Forward Edges***: For each edge $e \in E$ with $f(e) < c(e)$, we add $e \in E'$ with capacity $c(e) - f(e)$.*
(C) **Backward Edges***: For each edge $e = (u, v) \in E$ with $f(e) > 0$, we add $(v, u) \in E'$ with capacity $f(e)$.*

18.2.1.2 Residual Graph Example

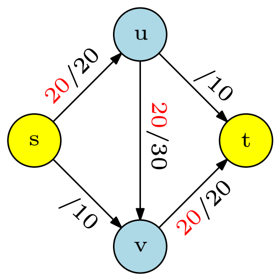


Figure 18.1: Flow on edges is indicated in red

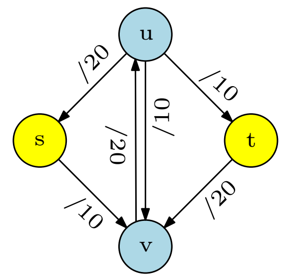


Figure 18.2: Residual Graph

18.2.1.3 Residual capacity

(A) f flow f in network G.
(B) c capacities on the edges.
(C) The ***residual capacity*** is the leftover capacity on each edge. Formally:

$$c_f\Big((u,v)\Big) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in \mathsf{E}(\mathsf{G}) \\ -f(v,u) & (v,u) \in \mathsf{E}(\mathsf{G}) \end{cases}$$

(D) ...assumed that G does not contain both (u,v) and (v,u).
(E) G_f with c_f is a new instance of network flow!

18.2.1.4 Residual graph properties

(A) **Observation:** Residual graph captures the "residual" problem exactly.
(B) Flow in residual graph improves overall flow:

Lemma 18.2.2. *Let f be a flow in G and G_f be the residual graph. If f' is a flow in G_f then $f+f'$ is a flow in G of value $v(f) + v(f')$.*

(C) If there is a bigger flow, we will find it:

Lemma 18.2.3. *Let f and f' be two flows in G with $v(f') \geq v(f)$. Then there is a flow f'' of value $v(f')$-$v(f)$ in G_f.*

(D) Definition of + and - for flows is intuitive and the above lemmas are easy in some sense but a bit messy to formally prove.

18.2.1.5 Residual Graph Property: Implication

Recursive algorithm for finding a maximum flow:

MaxFlow(G, s, t):
    if `the flow from` s `to` t `is` 0 **then**
        return 0
    `Find any flow` f `with` $v(f) > 0$ `in` G
    `Recursively compute a maximum flow` f' `in` G_f
    `Output the flow` f+f'

Iterative algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    Start with flow f that is 0 on all edges
    while there is a flow f' in G_f with v(f') > 0 do
        f = f+f'
        Update G_f

    Output f
```

18.2.1.6 Residual capacity of an augmenting path

(A) f current flow in G_f.
(B) π: A path π in residual graph G_f.
(C) c_f: Residual capacities in G_f.
(D) The ***residual capacity*** of π is

$$c_f(\pi) = \min_{\mathsf{e} \in \mathsf{E}(\pi)} c_f(\mathsf{e}).$$

(E) $c_f(\pi)$ = maximum amount of flow that can be pushed on π in G_f without violating capacities (i.e., c_f).

18.2.1.7 Ford-Fulkerson Algorithm

```
algFordFulkerson
    for every edge e, f(e) = 0
    G_f is residual graph of G with respect to f
    while G_f has a simple s-t path do
        let P be simple s-t path in G_f
        f = augment(f, P)
        Construct new residual graph G_f.
```

```
augment(f,P)
    let b be bottleneck capacity,
        i.e., min capacity of edges in P (in G_f)
    for each edge (u,v) in P do
        if e = (u,v) is a forward edge then
            f(e) = f(e) + b
        else (* (u,v) is a backward edge *)
            let e = (v,u) (* (v,u) is in G *)
            f(e) = f(e) - b
    return f
```

18.3 Correctness and Analysis

18.3.1 Termination

18.3.1.1 Properties about Augmentation: Flow

Lemma 18.3.1. *If f is a flow and P is a simple s-t path in G_f, then $f' = \texttt{augment}(f, P)$ is also a flow.*

Proof: Verify that f' is a flow. Let b be augmentation amount.

(A) **Capacity constraint:** If $(u, v) \in P$ is a forward edge then $f'(e) = f(e)+b$ and $b \leq c(e)-f(e)$. If $(u, v) \in P$ is a backward edge, then letting $e = (v, u)$, $f'(e) = f(e) - b$ and $b \leq f(e)$. Both cases $0 \leq f'(e) \leq c(e)$.

(B) **Conservation constraint:** Let v be an internal node. Let e_1, e_2 be edges of P incident to v. Four cases based on whether e_1, e_2 are forward or backward edges. Check cases (see fig next slide).

■

18.3.2 Properties of Augmentation

18.3.2.1 Conservation Constraint

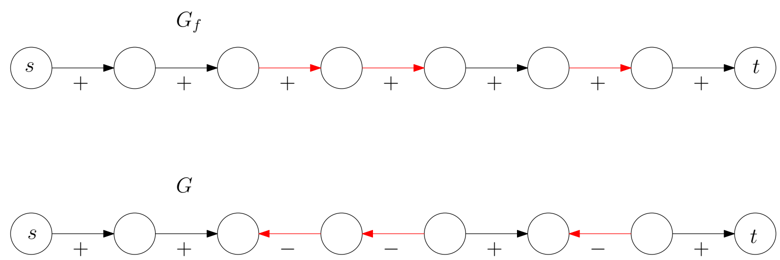


Figure 18.3: Augmenting path P in G_f and corresponding change of flow in G. Red edges are backward edges.

18.3.3 Properties of Augmentation

18.3.3.1 Integer Flow

Lemma 18.3.2. *At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., $f(e)$, for all edges e) and the residual capacities in G_f are integers.*

Proof: Initial flow and residual capacities are integers. Suppose lemma holds for j iterations. Then in $(j + 1)$st iteration, minimum capacity edge b is an integer, and so flow after augmentation is an integer. ■

18.3.3.2 Progress in Ford-Fulkerson

Proposition 18.3.3. *Let f be a flow and f' be flow after one augmentation. Then $v(f) < v(f')$.*

Proof: Let P be an augmenting path, i.e., P is a simple s-t path in residual graph. We have the following.

(A) First edge e in P must leave s.

(B) Original network G has no incoming edges to s; hence e is a forward edge.

(C) P is simple and so never returns to s.

(D) Thus, value of flow increases by the flow on edge e.

■

18.3.3.3 Termination proof for integral flow

Theorem 18.3.4. *Let C be the minimum cut value; in particular $C \leq \sum_{e \text{ out of } s} c(e)$. Ford-Fulkerson algorithm terminates after finding at most C augmenting paths.*

Proof: The value of the flow increases by at least 1 after each augmentation. Maximum value of flow is at most C. ■

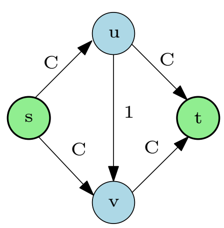
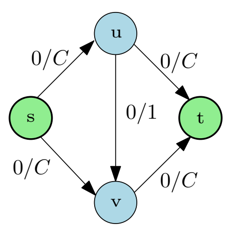
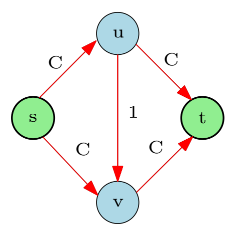
Running time
(A) Number of iterations $\leq C$.
(B) Number of edges in $G_f \leq 2m$.
(C) Time to find augmenting path is $O(n + m)$.
(D) Running time is $O(C(n + m))$ (or $O(mC)$).

18.3.3.4 Efficiency of Ford-Fulkerson

Running time $= O(mC)$ is not polynomial. Can the running time be as $\Omega(mC)$ or is our analysis weak?

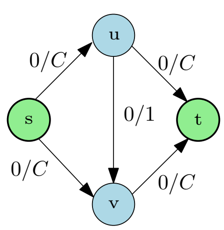
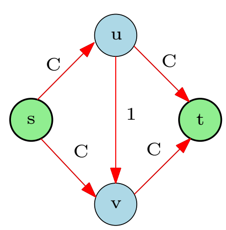
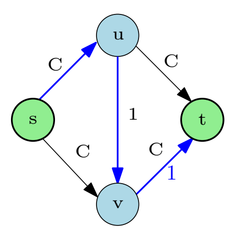
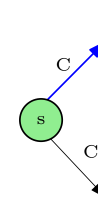
18.3.4 Efficiency of Ford-Fulkerson

18.3.4.1 Flip-flop 1

		
Input network + capacities.	Initial empty flow	Residual graph (Boring)

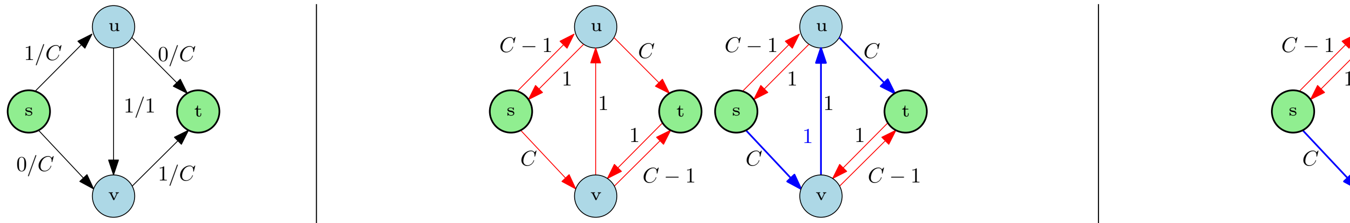
18.3.5 Efficiency of Ford-Fulkerson

18.3.5.1 Flip-flop 2

	 	
f_0: Initial empty flow	Residual graph Augmenting path	Augmenting path

18.3.6 Efficiency of Ford-Fulkerson

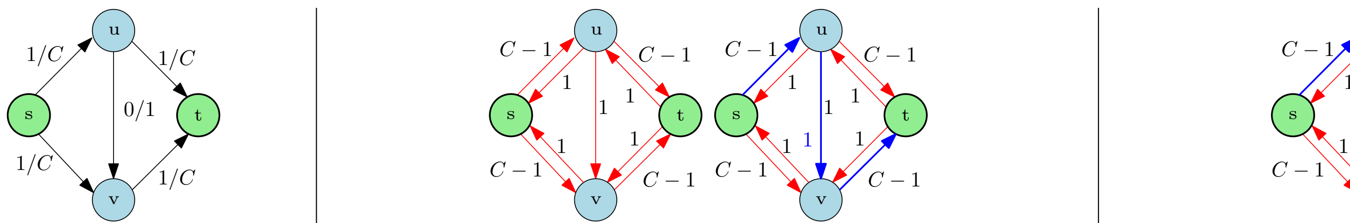
18.3.6.1 Flip-flop 3



f_1: current flow	G_{f_1}: Residual graph Augmenting path	Augmenting path

18.3.7 Efficiency of Ford-Fulkerson

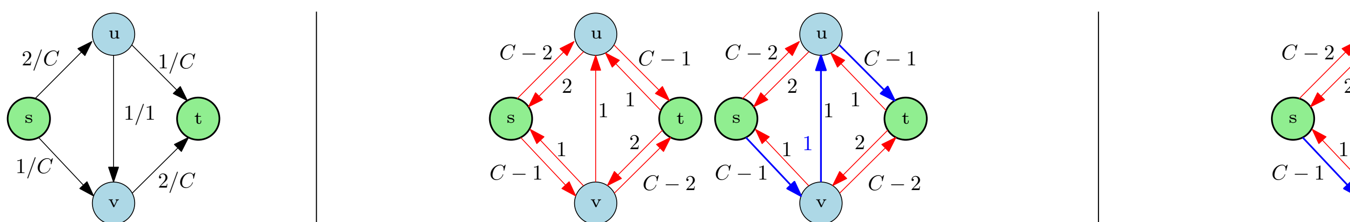
18.3.7.1 Flip-flop 4



f_2: current flow	G_{f_2}: Residual graph Augmenting path	Augmenting path

18.3.8 Efficiency of Ford-Fulkerson

18.3.8.1 Flip-flop 5



f_3: current flow	G_{f_3}: Residual graph Augmenting path	Augmenting path

And so it continues for $2C$ iterations...

18.3.8.2 Efficiency of Ford-Fulkerson

(A) Running time $= O(mC)$ is not polynomial.
(B) Can the running time be as $\Omega(mC)$ or is our analysis weak?
(C) Previous example shows this is tight!.
(D) Ford-Fulkerson can take $\Omega(C)$ iterations.

18.3.9 Correctness

18.3.10 Correctness of Ford-Fulkerson

18.3.10.1 Why the augmenting path approach works

(A) **Question:** When the algorithm terminates, is the flow computed the maximum s-t flow?
(B) Proof idea: show a cut of value equal to the flow. Also shows that maximum flow is equal to minimum cut!

18.3.10.2 Recalling Cuts

(A) Definition:

Definition 18.3.5. *Given a flow network an* **s-t cut** *is a set of edges* $E' \subset E$ *such that removing* E' disconnects s *from* t*: in other words there is no directed* $s \to t$ *path in* $E - E'$*.* **Capacity** *of cut* E' *is* $\sum_{e \in E'} c(e)$*.*

(B) ***Vertex cut***: Let $A \subset V$ such that
  (A) $s \in A$, $t \notin A$, and
  (B) $B = V \setminus -A$ and hence $t \in B$.
(C) Define $(A, B) = \{(u, v) \in E \mid u \in A, v \in B\}$

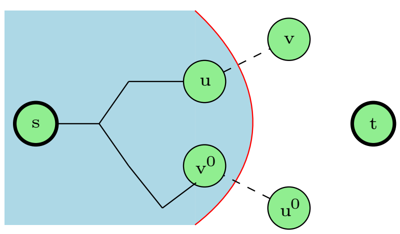
Claim 18.3.6. (A, B) *is an* s-t *cut.*

(D) Recall: Every *minimal* s-t cut E' is a cut of the form (A, B).

18.3.10.3 Ford-Fulkerson Correctness

Lemma 18.3.7. *If there is no* s-t *path in* G_f *then there is some cut* (A, B) *such that* $v(f) = c(A, B)$

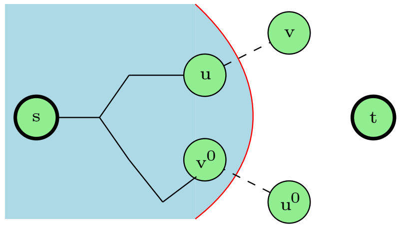
Proof: Let A be all vertices reachable from s in G_f; $B = V \backslash A$.



(A) $s \in A$ and t
(B) If $e = (u, v)$
(**saturated**
in G_f.

18.3.10.4 Lemma Proof Continued

Proof:

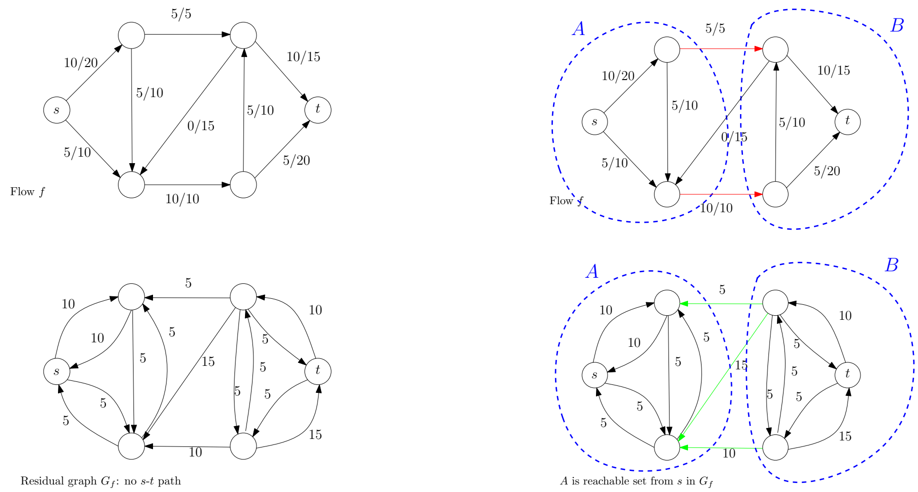


(A) If $e = (u', v') \in G$ with $u' \in B$ and $v' \in A$, then $f(e) = 0$ because otherwise u' is reachable from s in G_f
(B) Thus,

$$
\begin{aligned}
v(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \\
&= f^{\text{out}}(A) - 0 \\
&= c(A, B) - 0 \\
&= c(A, B).
\end{aligned}
$$

■

18.3.10.5 Example



18.3.10.6 Ford-Fulkerson Correctness

Theorem 18.3.8. *The flow returned by the algorithm is the maximum flow.*

Proof:
(A) For any flow f and s-t cut (A, B), $v(f) \leq c(A, B)$.
(B) For flow f^* returned by algorithm, $v(f^*) = c(A^*, B^*)$ for some s-t cut (A^*, B^*).
(C) Hence, f^* is maximum.

■

18.3.10.7 Max-Flow Min-Cut Theorem and Integrality of Flows

Theorem 18.3.9. *For any network G, the value of a maximum s-t flow is equal to the capacity of the minimum s-t cut.*

Proof: Ford-Fulkerson algorithm terminates with a maximum flow of value equal to the capacity of a (minimum) cut. ■

18.3.10.8 Max-Flow Min-Cut Theorem and Integrality of Flows

Theorem 18.3.10. *For any network G with integer capacities, there is a maximum s-t flow that is integer valued.*

Proof: Ford-Fulkerson algorithm produces an integer valued flow when capacities are integers. ■

18.4 Polynomial Time Algorithms

18.4.0.9 Efficiency of Ford-Fulkerson

(A) Running time $= O(mC)$ is not polynomial.
(B) Can the upper bound be achieved?
(C) Yes - saw an example.

18.4.0.10 Polynomial Time Algorithms

(A) **Question:** Is there a polynomial time algorithm for max-flow?
(B) **Question:** Is there a variant of Ford-Fulkerson that leads to a polynomial time algorithm? Can we choose an augmenting path in some clever way?
(C) Yes! Two variants.
    (A) Choose the augmenting path with largest bottleneck capacity.
    (B) Choose the shortest augmenting path.

18.4.1 Capacity Scaling Algorithm

18.4.1.1 Augmenting Paths with Large Bottleneck Capacity

(A) Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.
(B) How do we find path with largest bottleneck capacity?
    (A) Assume we know Δ the bottleneck capacity
    (B) Remove all edges with residual capacity $\leq \Delta$
    (C) Check if there is a path from s to t
    (D) Do binary search to find largest Δ
    (E) Running time: $O(m \log C)$
(C) Can we bound the number of augmentations? Can show that in $O(m \log C)$ augmentations the algorithm reaches a max flow. This leads to an $O(m^2 \log^2 C)$ time algorithm.

18.4.1.2 Augmenting Paths with Large Bottleneck Capacity

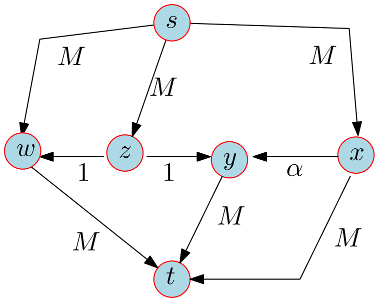
(A) How do we find path with largest bottleneck capacity?
    (A) Max bottleneck capacity is one of the edge capacities. Why?
    (B) Can do binary search on the edge capacities. First, sort the edges by their capacities and then do binary search on that array as before.
    (C) Algorithm's running time is $O(m \log m)$.
    (D) Different algorithm that also leads to $O(m \log m)$ time algorithm by adapting Prim's algorithm.

18.4.1.3 Removing Dependence on C

(A) **[Dinic, 1970]**, **[Edmonds and Karp, 1972]**
    Picking augmenting paths with fewest number of edges yields a $O(m^2 n)$ algorithm, i.e., independent of C. Such an algorithm is called a **strongly polynomial** time algorithm since the running time does not depend on the numbers (assuming RAM model). (Many implementation of Ford-Fulkerson would actually use shortest augmenting path if they use **BFS** to find an s-t path).
(B) Further improvements can yield algorithms running in $O(mn \log n)$, or $O(n^3)$.

18.5 Not for lecture: Non-termination of Ford-Fulkerson

18.5.0.4 Ford-Fulkerson runs in vain



(A) M: large positive integer.
(B) $\alpha = (\sqrt{5} - 1)/2 \approx 0.618$.
(C) $\alpha < 1$,
(D) $1 - \alpha < \alpha$.
(E) Maximum flow in this network is: $2M + 1$.

18.5.0.5 Some algebra...

For $\alpha = \dfrac{\sqrt{5}-1}{2}$:

$$
\begin{aligned}
\alpha^2 &= ([])\frac{\sqrt{5}-1}{2}^2 = \frac{1}{4}([])\sqrt{5}-1^2 = \frac{1}{4}([])5 - 2\sqrt{5} + 1 \\
&= 1 + \frac{1}{4}([])2 - 2\sqrt{5} \\
&= 1 + \frac{1}{2}([])1 - \sqrt{5} \\
&= 1 - \frac{\sqrt{5}-1}{2} \\
&= 1 - \alpha.
\end{aligned}
$$

18.5.0.6 Some algebra...

Claim 18.5.1. *Given:* $\alpha = (\sqrt{5} - 1)/2$ *and* $\alpha^2 = 1 - \alpha$.

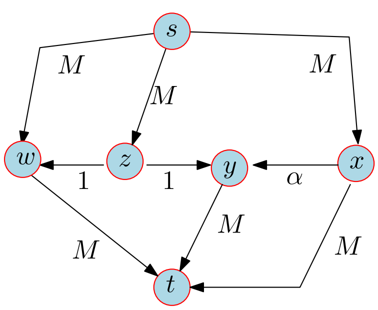
$$\implies \forall i \qquad \alpha^i - \alpha^{i+1} = \alpha^{i+2}$$

Proof:

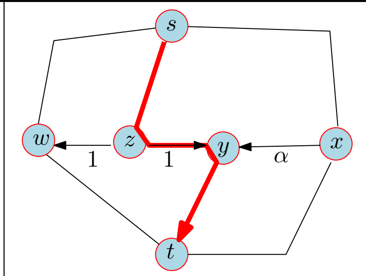
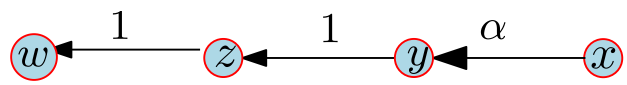
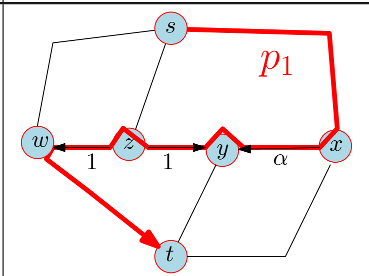
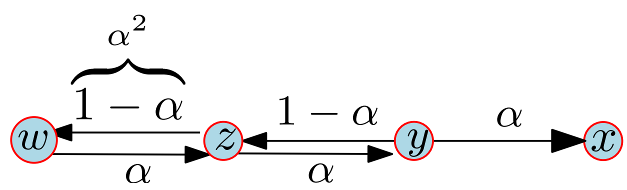
$$\alpha^i - \alpha^{i+1} = \alpha^i(1 - \alpha) = \alpha^i \alpha^2 = \alpha^{i+2}.$$

■

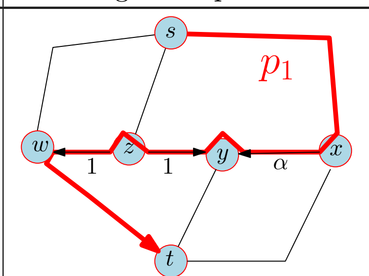
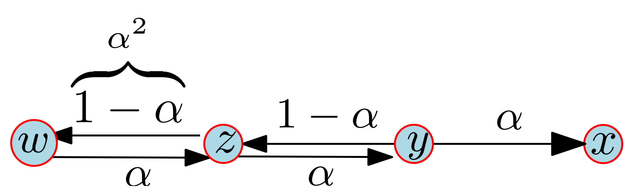
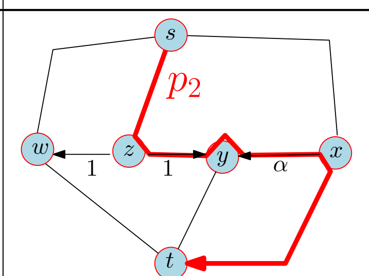
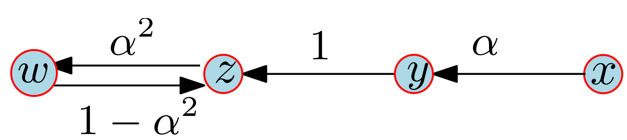
18.5.0.7 The network



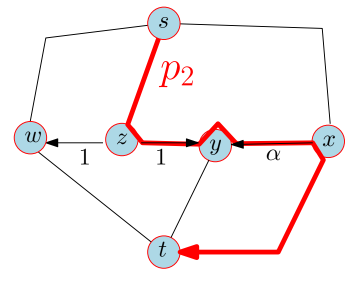
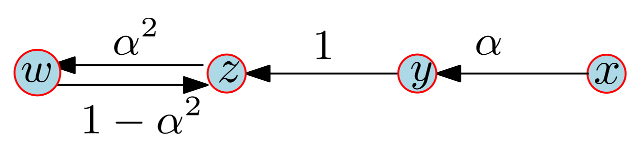
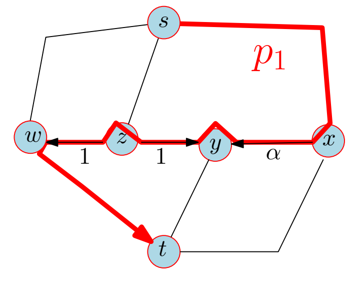
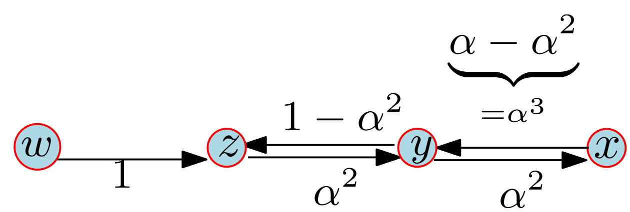
18.5.0.8 Let it flow...

#	Augment. path π	c_π	New residual network
0.		1	
1.		α	

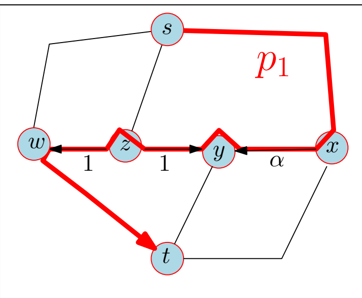
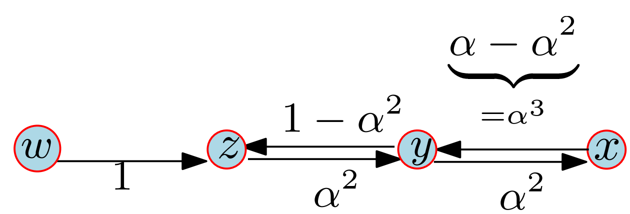
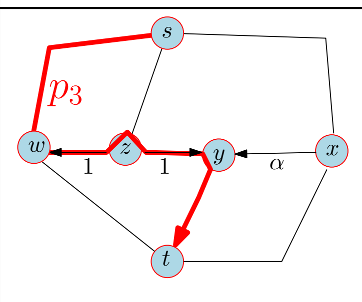
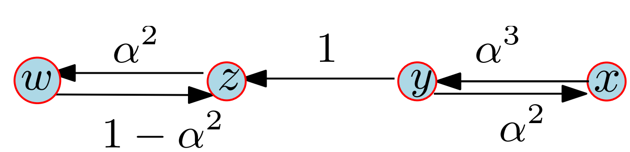
18.5.0.9 Let it flow II

#	Augment. path π	c_π	New residual network
1.		α	
2.		α	

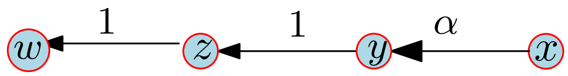
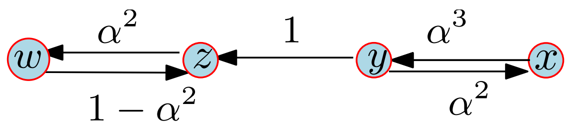
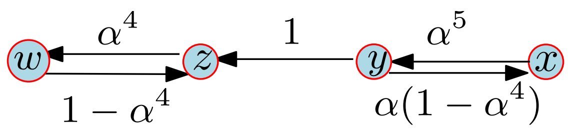
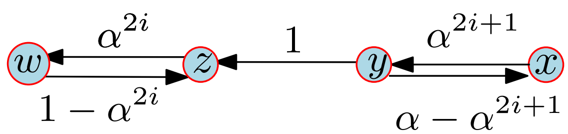
18.5.0.10 Let it flow II

2.		α^2	
3.		α^2	

18.5.0.11 Let it flow III

3.		α^2	
4.		α^2	

18.5.0.12 Let it flow III

moves	Residual network after
0	
moves $0,(1,2,3,4)$	
moves $0,(1,2,3,4)^2$	
$0.(1,2,3,4)^i$	

Namely, the algorithm never terminates.

Chapter 19

Applications of Network Flows

OLD CS 473: Fundamental Algorithms, Spring 2015
April 2, 2015

19.0.1 Important Properties of Flows

19.0.1.1 Network flow, what we know...

(A) G: Network flow with n vertices and m edges.
(B) **algFordFulkerson** computes max-flow if capacities are integers.
(C) If total capacity is C, running time of **algFordFulkerson** is $O(mC)$.
(D) **algFordFulkerson** is not polynomial time.
(E) **algFordFulkerson** might not terminate if capacities are real numbers.
(F) ...see end of the slides in previous lectures for detailed example.

19.1 Edmonds-Karp algorithm

19.1.0.2 Edmonds-Karp algorithm

algEdmondsKarp
    for every edge e, $f(e) = 0$
    G_f is residual graph of G with respect to f
    while G_f has a simple s-t path **do**
        Perform **BFS** in G_f
        P: shortest s-t path in G_f
        $f =$ **augment**(f, P)
        Construct new residual graph G_f.

Theorem 19.1.1. *Given a network flow G with n vertices and m edges, and capacities that are real numbers, the algorithm* **algEdmondsKarp** *computes the maximum flow in G.*
The running time is $O(m^2 n)$.

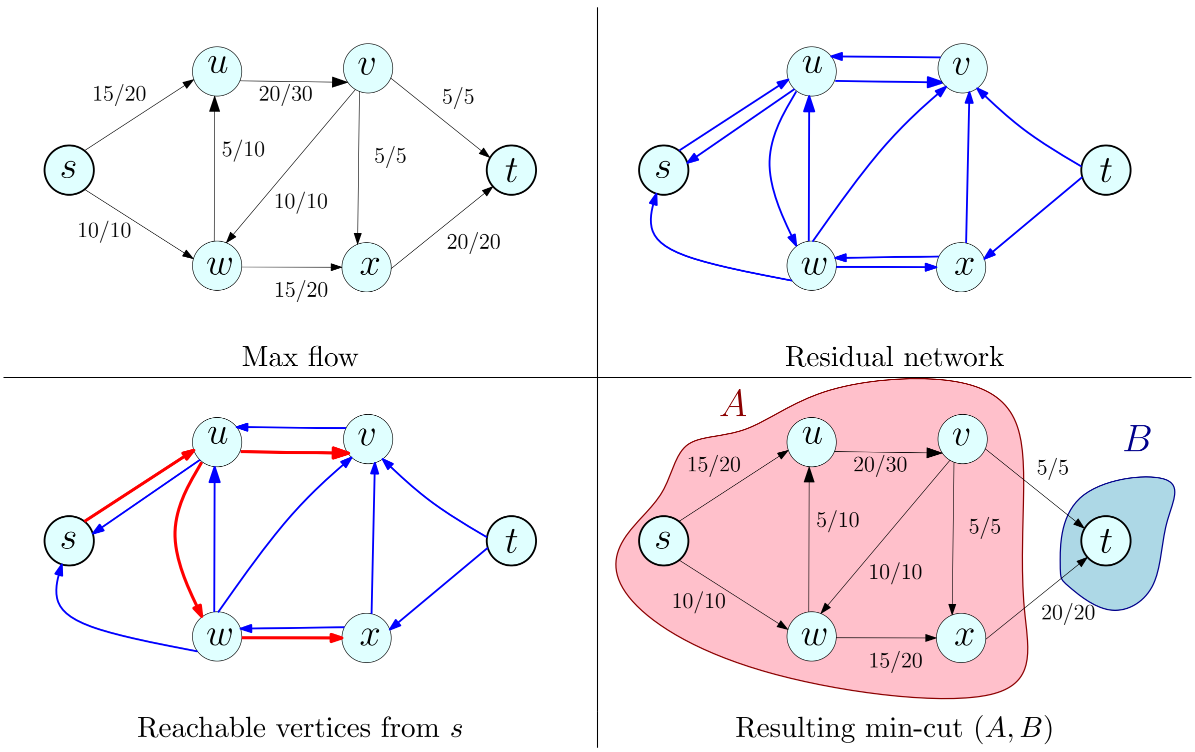
19.1.1 Computing a minimum cut...

19.1.1.1 Finding a Minimum Cut

(A) **Question:** How do we find an actual minimum s-t cut?

(B) Proof gives the algorithm!
   (A) Compute an s-t maximum flow f in G
   (B) Obtain the residual graph G_f
   (C) Find the nodes A reachable from s in G_f
   (D) Output the cut $(A, B) = \{(u, v) \mid u \in A, v \in B\}$. **Note:** The cut is found in G while A is found in G_f
(C) Running time is essentially the same as finding a maximum flow.
(D) **Note:** Given G and a flow f there is a linear time algorithm to check if f is a maximum flow and if it is, outputs a minimum cut. How?

19.1.1.2 Min cut from max-flow



19.1.1.3 Network Flow: Facts to Remember

Flow network: directed graph G, capacities c, source s, sink t.

(A) Maximum s-t flow can be computed:
   (A) Using Ford-Fulkerson algorithm in $O(mC)$ time when capacities are integral and C is an upper bound on the flow.
   (B) Using variant of algorithm, in $O(m^2 \log C)$ time, when capacities are integral. (Polynomial time.)
   (C) Using Edmonds-Karp algorithm, in $O(m^2 n)$ time, when capacities are rational (strongly polynomial time algorithm).

19.1.2 Network Flow

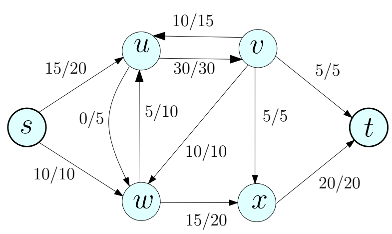
19.1.2.1 Even more facts to remember

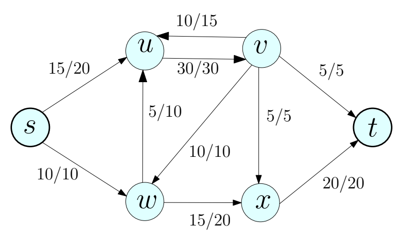
(A) If capacities are integral then there is a maximum flow that is integral and above algorithms give an integral max flow. This is known as ***integrality of flow***.
(B) Given a flow of value v, can decompose into $O(m+n)$ flow paths of same total value v. Integral flow implies integral flow on paths.
(C) Maximum flow is equal to the minimum cut and minimum cut can be found in $O(m+n)$ time given any maximum flow.

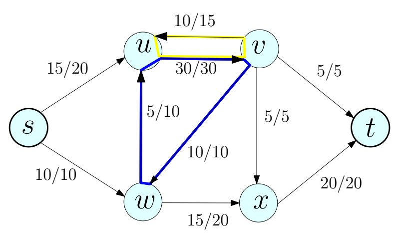
19.1.2.2 Paths, Cycles and Acyclicity of Flows

Definition 19.1.2. *Given a flow network $G = (V, E)$ and a flow $f : E \to \mathbb{R}^{\geq 0}$ on the edges, the* **support** *of f is the set of edges $E' \subseteq E$ with non-zero flow on them. That is, $E' = \{e \in E \mid f(e) > 0\}$.*

Question: Given a flow f, can there by cycles in its support?







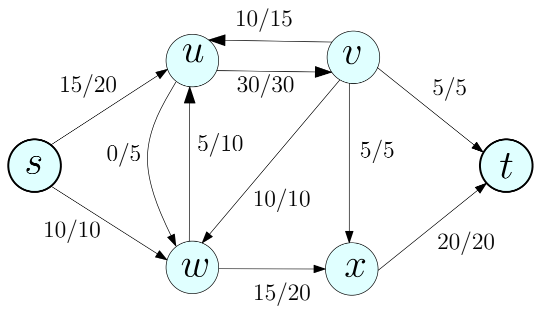
19.1.2.3 Acyclicity of Flows

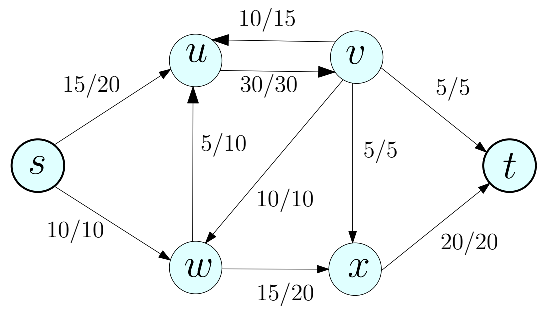
Proposition 19.1.3. *In any flow network, if f is a flow then there is another flow f' such that the support of f' is an* acyclic *graph and $v(f') = v(f)$. Further if f is an integral flow then so is f'.*

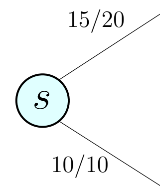
Proof:
(A) $E' = \{e \in E \mid f(e) > 0\}$, support of f.
(B) Suppose there is a directed cycle C in E'
(C) Let e' be the edge in C with least amount of flow
(D) For each $e \in C$, reduce flow by $f(e')$. Remains a flow. Why?
(E) Flow on e' is reduced to 0.
(F) Claim: Flow value from s to t does not change. Why?
(G) Iterate until no cycles

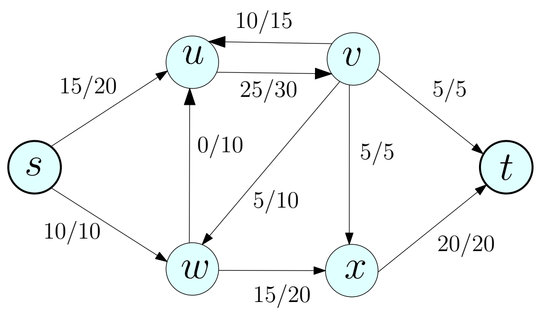
■

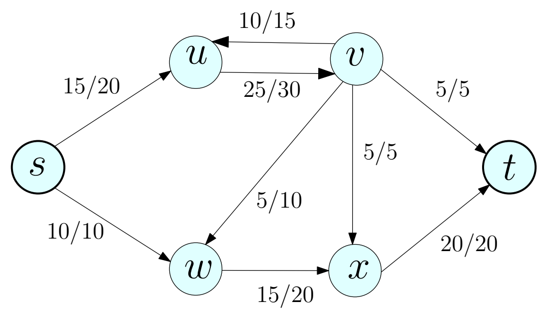
19.1.2.4 Example

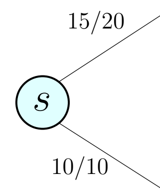


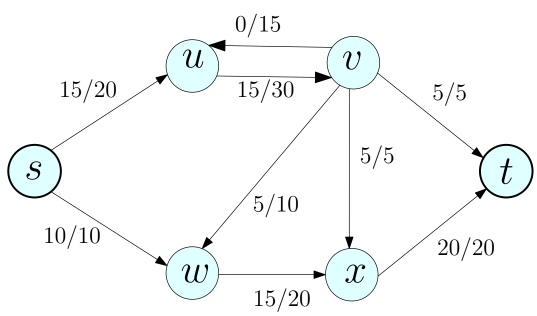


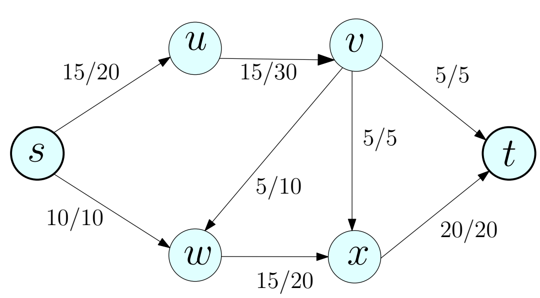


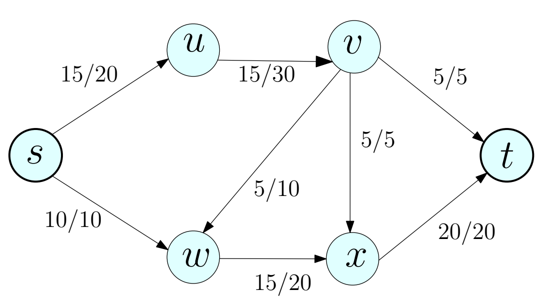




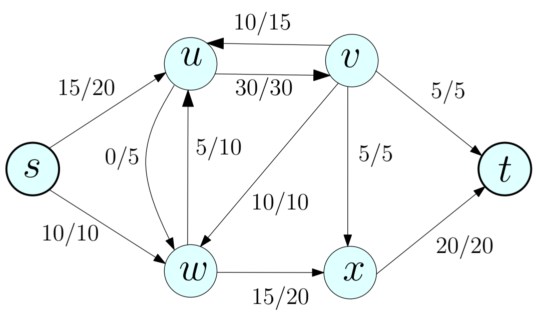








Throw away edge with no flow on itFind a cycle in the support/flowReduce flow on cycle as much as possibleThrow away edge with no flow on itFind a cycle in the support/flowReduce flow on cycle as much as possibleThrow away edge with no flow on itViola!!! An equivalent flow with no cycles in it. Original flow:



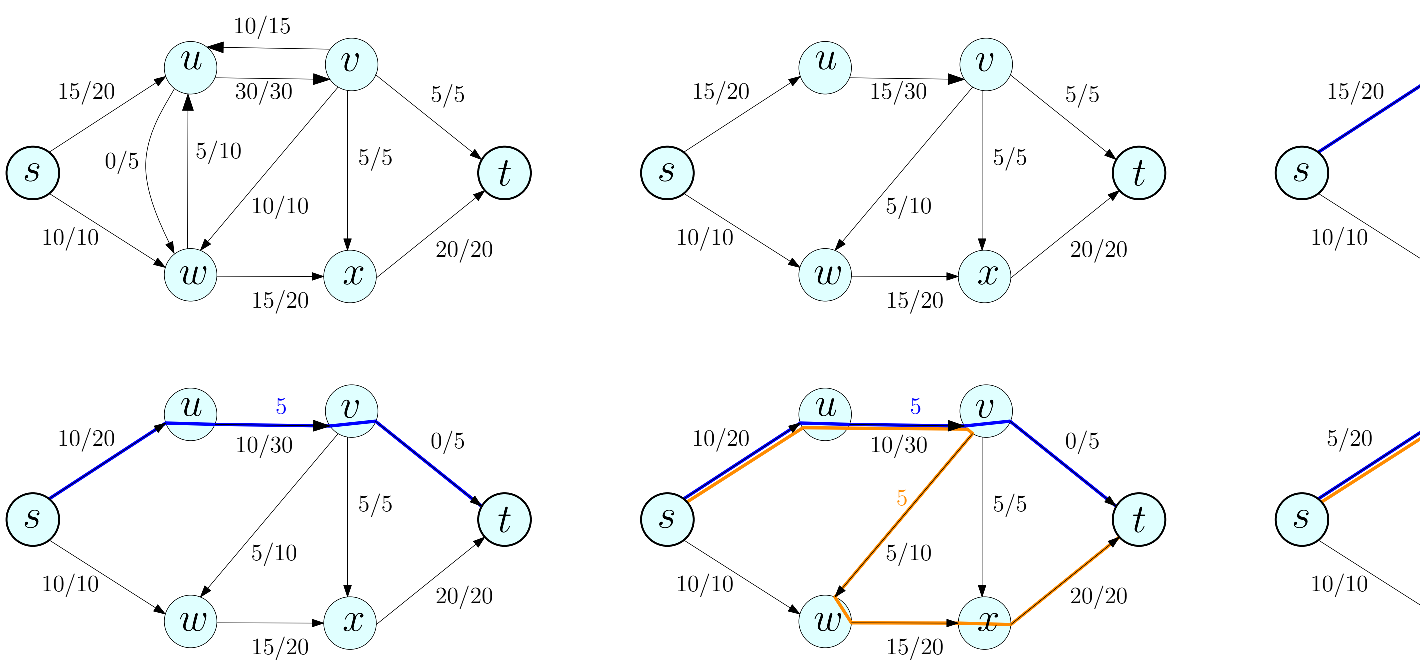
19.1.2.5 Flow Decomposition

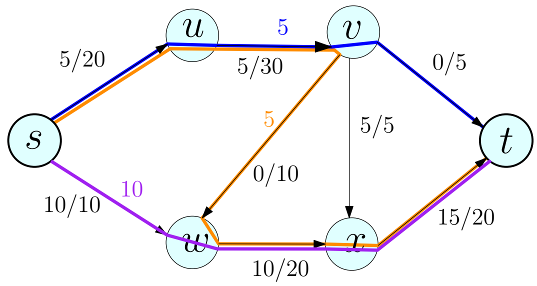
Lemma 19.1.4. *Given an edge based flow $f : E \to \mathbb{R}^{\geq 0}$, there exists a collection of paths $\mathcal{P}$ and cycles $\mathcal{C}$ and an assignment of flow to them $f' : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}^{\geq 0}$ such that:*
(A) $|\mathcal{P} \cup \mathcal{C}| \leq m$
(B) for each $e \in E$, $\sum_{P \in \mathcal{P}: e \in P} f'(P) + \sum_{C \in \mathcal{C}: e \in C} f'(C) = f(e)$
(C) $v(f) = \sum_{P \in \mathcal{P}} f'(P)$.
(D) if f is integral then so are $f'(P)$ and $f'(C)$ for all P and C

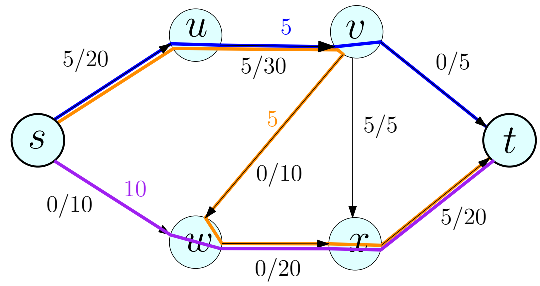
Proof:[Proof Idea]
(A) Remove all cycles as in previous proposition.
(B) Next, decompose into paths as in previous lecture.
(C) Exercise: verify claims.

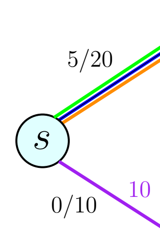
■

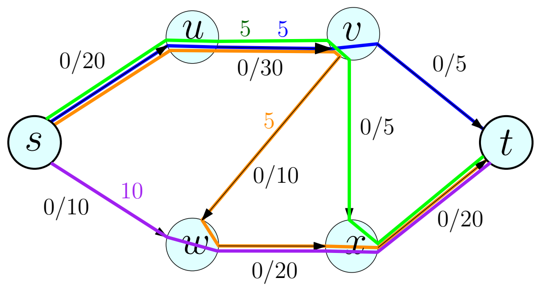
19.1.2.6 Example











Find cycles as shown beforeFind a source to sink path, and push max flow along it (5 unites)Compute remaining flowFind a source to sink path, and push max flow along it (5 unites). Edges with 0 flow on them can not be used as they are no longer in the support of the flow.Compute remaining flowFind a source to sink path, and push max flow along it (10 unites). Compute remaining flowFind a source to sink path, and push max flow along it (5 unites). Compute remaining flowNo flow remains in the graph. We fully decomposed the flow into flow on paths. Together with the cycles, we get a decomposition of the original flow into m flows on paths and cycles.

19.1.2.7 Flow Decomposition

Lemma 19.1.5. *Given an edge based flow $f : E \to \mathbb{R}^{\geq 0}$, there exists a collection of paths $\mathcal{P}$ and cycles $\mathcal{C}$ and an assignment of flow to them $f' : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}^{\geq 0}$ such that:*

(A) $|\mathcal{P} \cup \mathcal{C}| \leq m$
(B) for each $e \in E$, $\sum_{P \in \mathcal{P}: e \in P} f'(P) + \sum_{C \in \mathcal{C}: e \in C} f'(C) = f(e)$
(C) $v(f) = \sum_{P \in \mathcal{P}} f'(P)$.
(D) if f is integral then so are $f'(P)$ and $f'(C)$ for all P and C.

Above flow decomposition can be computed in $O(m^2)$ time.

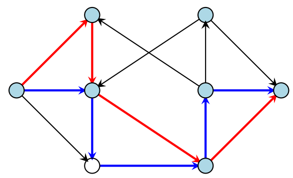
19.2 Network Flow Applications I

19.2.1 Edge Disjoint Paths

19.2.2 Directed Graphs

19.2.2.1 Edge-Disjoint Paths in Directed Graphs

Definition 19.2.1.



A set of paths is **edge disjoint** *if no two paths sh*
edge.

Problem Given a directed graph with two special vertices s and t, find the *maximum* number of edge disjoint paths from s to t. **Applications:** Fault tolerance in routing — edges/nodes in networks can fail. Disjoint paths allow for planning backup routes in case of failures.

19.2.3 Reduction to Max-Flow

19.2.3.1 Reduction to Max-Flow

Problem Given a directed graph G with two special vertices s and t, find the maximum number of edge disjoint paths from s to t. Reduction Consider G as a flow network with edge capacities 1, and compute max-flow.

19.2.3.2 Correctness of Reduction

Lemma 19.2.2. *If G has k edge disjoint paths $P_1, P_2, \ldots, P_k$ then there is an s-t flow of value k in G.*

Proof: Set $f(e) = 1$ if e belongs to one of the paths $P_1, P_2, \ldots, P_k$; other-wise set $f(e) = 0$. This defines a flow of value k. ■

19.2.3.3 Correctness of Reduction

Lemma 19.2.3. *If G has a flow of value k then there are k edge disjoint paths between s and t.*

Proof:

(A) Capacities are all 1 and hence there is integer flow of value k, that is $f(e) = 0$ or $f(e) = 1$ for each e.
(B) Decompose flow into paths.
(C) Flow on each path is either 1 or 0.
(D) Hence there are k paths $P_1, P_2, \ldots, P_k$ with flow of 1 each.
(E) Paths are edge-disjoint since capacities are 1.

■

19.2.3.4 Running Time

Theorem 19.2.4. *The number of edge disjoint paths in G can be found in $O(mn)$ time.*

Proof:
(A) Set capacities of edges in G to 1.
(B) Run Ford-Fulkerson algorithm.
(C) Maximum value of flow is n and hence run-time is $O(nm)$.
(D) Decompose flow into k paths ($k \leq n$).
Takes $O(k \times m) = O(km) = O(mn)$ time.

■

Remark Algorithm computes set of edge-disjoint paths realizing opt. solution.

19.2.4 Menger's Theorem

19.2.4.1 Menger's Theorem

Theorem 19.2.5 ([Menger, 1927]). *Let G be a directed graph. The minimum number of edges whose removal disconnects s from t (the minimum-cut between s and t) is equal to the maximum number of edge-disjoint paths in G between s and t.*

Proof: Maxflow-mincut theorem and integrality of flow. ■

Menger proved his theorem before Maxflow-Mincut theorem! Maxflow-Mincut theorem is a generalization of Menger's theorem to capacitated graphs.

19.2.5 Undirected Graphs

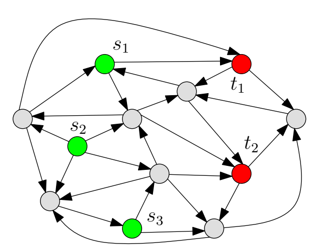
19.2.5.1 Edge Disjoint Paths in Undirected Graphs

(A) The problem: Problem Given an **undirected** graph G, find the maximum number of edge disjoint paths in G
(B) Reduction:
  (A) create **directed** graph H by adding directed edges (u, v) and (v, u) for each edge uv in G.
  (B) compute maximum s-t flow in H.
(C) **Problem:** Both edges (u, v) and (v, u) may have non-zero flow!
(D) **Not a Problem!** Can assume maximum flow in H is acyclic and hence cannot have non-zero flow on both (u, v) and (v, u). Reduction works. See book for more details.

19.2.6 Multiple Sources and Sinks

19.2.6.1 Multiple Sources and Sinks

(A) Input:
  (A) A directed graph G with edge capacities $c(e)$.
  (B) Source nodes $s_1, s_2, \ldots, s_k$.
  (C) Sink nodes $t_1, t_2, \ldots, t_\ell$.
  (D) Sources and sinks are *disjoint*.



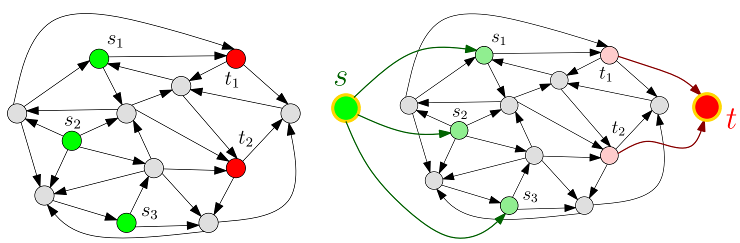
(A) **Maximum Flow**: Send as much flow as possible from the sources to the sinks. *Sinks don't care which source they get flow from.*
(B) **Minimum Cut**: Find a minimum capacity set of edge E' such that removing E' disconnects every source from every sink.

19.2.6.2 Multiple Sources and Sinks: Formal Definition

(A) Input:
  (A) A directed graph G with edge capacities $c(e)$.
  (B) Source nodes $s_1, s_2, \ldots, s_k$.
  (C) Sink nodes $t_1, t_2, \ldots, t_\ell$.
  (D) Sources and sinks are *disjoint*.
(B) A function $f : E \to \mathbb{R}^{\geq 0}$ is a ***flow*** if:
  (A) For each $e \in E$, $f(e) \leq c(e)$, and
  (B) for each v which is not a source or a sink $f^{\text{in}}(v) = f^{\text{out}}(v)$.
(C) **Goal:** max $\sum_{i=1}^{k}(f^{\text{out}}(s_i) - f^{\text{in}}(s_i))$, that is, flow out of sources.

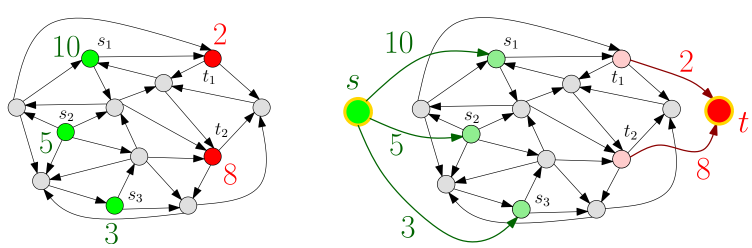
19.2.6.3 Reduction to Single-Source Single-Sink

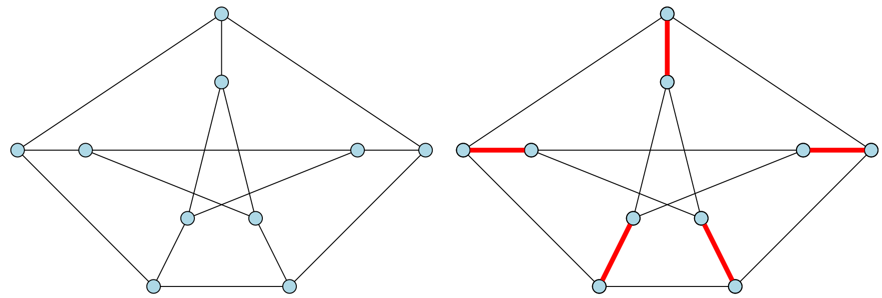
(A) Add a *source* node s and a sink node t.
(B) Add edges $(s, s_1), (s, s_2), \ldots, (s, s_k)$.
(C) Add edges $(t_1, t), (t_2, t), \ldots, (t_\ell, t)$.
(D) Set the capacity of the new edges to be ∞.



19.2.6.4 Supplies and Demands

(A) A further generalization:
  (A) source s_i has a supply of $S_i \geq 0$
  (B) since t_j has a demand of $D_j \geq 0$ units
(B) **Question:** is there a flow from source to sinks such that supplies are not exceeded and demands are met?
(C) Formally: additional constraints that $f^{\text{out}}(s_i) - f^{\text{in}}(s_i) \leq S_i$ for each source s_i and $f^{\text{in}}(t_j) - f^{\text{out}}(t_j) \geq D_j$ for each sink t_j.





19.2.7 Bipartite Matching

19.2.8 Definitions

19.2.8.1 Matching

Problem 19.2.6 (Matching).

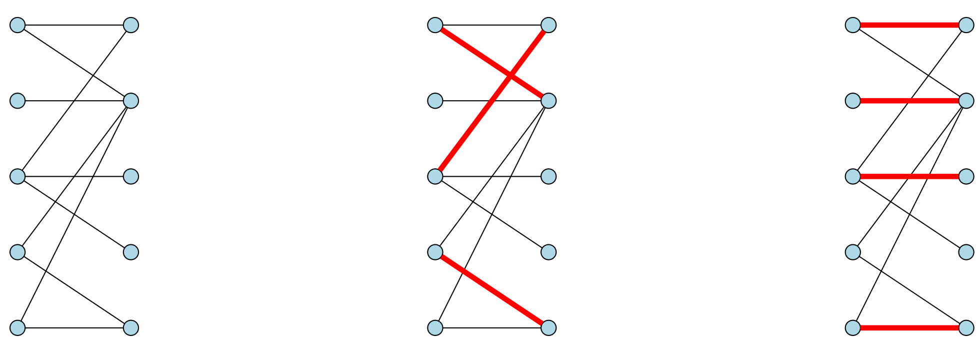
Input: *Given a (undirected) graph $G = (V, E)$.*
Goal: *Find a matching of maximum cardinality.*

(A) A matching is $M \subseteq E$ such that at most one edge in M is incident on any vertex

19.2.8.2 Bipartite Matching

Problem 19.2.7 (Bipartite matching).

Input: *Given a bipartite graph $G = (L \cup R, E)$.*
Goal: *Find a matching of maximum cardinality*

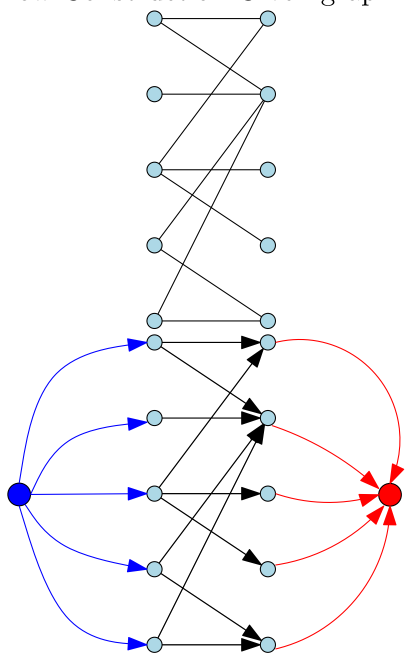


Maximum matching has 4 edges

19.2.9 Reduction of bipartite matching to max-flow

19.2.9.1 Reduction of bipartite matching to max-flow

Max-Flow Construction Given graph $G = (L \cup R, E)$ create flow-network $G' = (V', E')$ as follows:



(A) $V' = L \cup R \cup \{s, t\}$ where s and t are the new source and sink.
(B) Direct all edges in E from L to R, and add edges from s to all vertices in L and from each vertex in R to t.
(C) Capacity of every edge is 1.

19.2.9.2 Correctness: Matching to Flow

Proposition 19.2.8. *If G has a matching of size k then G' has a flow of value k.*

Proof: Let M be matching of size k. Let $M = \{(u_1, v_1), \ldots, (u_k, v_k)\}$. Consider following flow f in G':
(A) $f(s, u_i) = 1$ and $f(v_i, t) = 1$ for $1 \le i \le k$
(B) $f(u_i, v_i) = 1$ for $1 \le i \le k$
(C) for all other edges flow is zero.

Verify that f is a flow of value k (because M is a matching). ■

19.2.9.3 Correctness: Flow to Matching

Proposition 19.2.9. *If G' has a flow of value k then G has a matching of size k.*

Proof: Consider flow f of value k.
(A) Can assume f is integral. Thus each edge has flow 1 or 0.
(B) Consider the set M of edges from L to R that have flow 1.
  (A) M has k edges because value of flow is equal to the number of non-zero flow edges crossing cut $(L \cup \{s\}, R \cup \{t\})$
  (B) Each vertex has at most one edge in M incident upon it. Why?

■

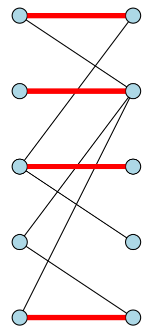


Figure 19.1: This graph does not have a perfect matching

19.2.9.4 Correctness of Reduction

Theorem 19.2.10. *The maximum flow value in G' = maximum cardinality of matching in G.*

Consequence Thus, to find maximum cardinality matching in G, we construct G' and find the maximum flow in G'. Note that the matching itself (not just the value) can be found efficiently from the flow.

19.2.9.5 Running Time

For graph G with n vertices and m edges G' has $O(n+m)$ edges, and $O(n)$ vertices.
(A) Generic Ford-Fulkerson: Running time is $O(mC) = O(nm)$ since $C = n$.
(B) Capacity scaling: Running time is $O(m^2 \log C) = O(m^2 \log n)$.
Better running time is known: $O(m\sqrt{n})$.

19.2.10 Perfect Matchings

19.2.10.1 Perfect Matchings

Definition 19.2.11. *A matching M is* **perfect** *if every vertex has one edge in M incident upon it.*

19.2.10.2 Characterizing Perfect Matchings

Problem When does a bipartite graph have a perfect matching?
(A) Clearly $|L| = |R|$
(B) Are there any necessary and sufficient conditions?

19.2.10.3 A Necessary Condition

Lemma 19.2.12. *If $G = (L \cup R, E)$ has a perfect matching then for any $X \subseteq L$, $|N(X)| \geq |X|$, where $N(X)$ is the set of neighbors of vertices in X.*

Proof: Since G has a perfect matching, every vertex of X is matched to a different neighbor, and so $|N(X)| \geq |X|$. ■

19.2.10.4 Hall's Theorem

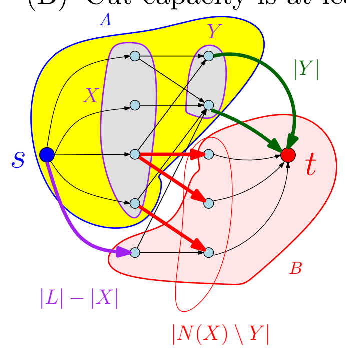
(A) Frobenius-Hall theorem:

> **Theorem 19.2.13 ().** *Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. G has a perfect matching if and only if for every $X \subseteq L$, $|N(X)| \geq |X|$.*

(B) One direction is the necessary condition.
(C) For the other direction we will show the following:
   (A) Create flow network G' from G.
   (B) If $|N(X)| \geq |X|$ for all X, show that minimum s-t cut in G' is of capacity $n = |L| = |R|$.
   (C) Implies that G has a perfect matching.

19.2.10.5 Proof of Sufficiency

(A) Assume $|N(X)| \geq |X|$ for any $X \subseteq L$. Then show that min s-t cut in G' is of capacity at least n.
(B) Let (A, B) be an *arbitrary* s-t cut in G'
   (A) Let $X = A \cap L$ and $Y = A \cap R$.
   (B) Cut capacity is at least $(|L| - |X|) + |Y| + |N(X) \setminus Y|$



Because there are...
(A) $|L| - |X|$ edges from s to $L \cap B$.
(B) $|Y|$ edges from Y to t.
(C) there are at least $|N(X) \setminus Y|$ edges from X to vertices on the right side that are not in Y.

19.2.11 Proof of Sufficiency

19.2.11.1 Continued...

(A) By the above, cut capacity is at least
$$\alpha = (|L| - |X|) + |Y| + |N(X) \setminus Y|.$$
(B) $|N(X) \setminus Y| \geq |N(X)| - |Y|$.
   (This holds for any two sets.)
(C) By assumption $|N(X)| \geq |X|$ and hence
$$|N(X) \setminus Y| \geq |N(X)| - |Y| \geq |X| - |Y|.$$
(D) Cut capacity is therefore at least
$$\begin{aligned}\alpha &= (|L| - |X|) + |Y| + |N(X) \setminus Y| \\ &\geq |L| - |X| + |Y| + |X| - |Y| \geq |L| = n.\end{aligned}$$

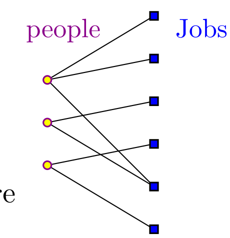
(E) Any s-t cut capacity is at least $n \implies$ max flow at least n units $\implies$ perfect matching.**QED**

19.2.11.2 Hall's Theorem: Generalization

Theorem 19.2.14 (Frobenius-Hall). *Let $G = (L \cup R, E)$ be a bipartite graph with $|L| \leq |R|$. G has a matching that matches all nodes in L if and only if for every $X \subseteq L$, $|N(X)| \geq |X|$.*

Proof is essentially the same as the previous one.

19.2.11.3 Problem: Assigning jobs to people



Problem:
(A) n jobs or tasks
(B) m people.
(C) for each job a set of people who can do that job.
(D) for each person j a limit on number of jobs k_j.
(E) **Goal:** find an assignment of jobs to people so that all jobs are assigned and no person is overloaded.

19.2.11.4 Application: Assigning jobs to people

(A) Reduce to max-flow similar to matching.
(B) Arises in many settings. Using *minimum-cost flows* can also handle the case when assigning a job i to person j costs c_{ij} and goal is assign all jobs but minimize cost of assignment.

19.2.12 Reduction to Maximum Flow

19.2.12.1 For assigning jobs to people

(A) Create directed graph $G = (V, E)$ as follows
 (A) $V = \{s, t\} \cup L \cup R$: L set of n jobs, R set of m people
 (B) add edges (s, i) for each job $i \in L$, capacity 1
 (C) add edges (j, t) for each person $j \in R$, capacity k_j
 (D) if job i can be done by person j add an edge (i, j), capacity 1
(B) Compute max s-t flow. There is an assignment if and only if flow value is n.

19.2.12.2 Matchings in General Graphs

(A) Matchings in general graphs more complicated.
(B) There is a polynomial time algorithm to compute a maximum matching in a general graph. Best known running time is $O(m\sqrt{n})$.

Chapter 20

More Network Flow Applications

OLD CS 473: Fundamental Algorithms, Spring 2015
April 4, 2015

20.1 Airline Scheduling

20.1.1 Airline Scheduling

20.1.1.1 Lower bounds

(A) The following example requires the ability to solve network flow with lower bounds on the edges.
(B) This can be reduced to regular network flow (we are not going to show the details – they are a bit tedious).
(C) The integrality property holds – if there is an integral solution our `network flow with lower bounds` solver would compute such a solution.

20.1.1.2 Airline Scheduling

Problem 20.1.1. *Given information about flights that an airline needs to provide, generate a profitable schedule.*

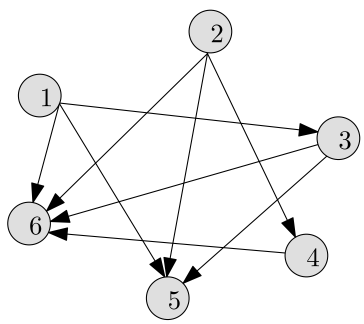
(A) Input: detailed information about "legs" of flight.
(B) $\mathcal{F}$: set of flights by
(C) Purpose: find minimum # airplanes needed.

20.1.2 Example

20.1.2.1 (i) a set $\mathcal{F}$ of flights that have to be served, and (ii) the corresponding graph G representing these flights.

1: Boston (depart 6 A.M.) - Washington DC (arrive 7 A.M,).
2: Urbana (depart 7 A.M.) - Champaign (arrive 8 A.M.)
3: Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)
4: Urbana (depart 11 A.M.) - San Francisco (arrive 2 P.M.)
5: San Francisco (depart 2:15 P.M.) - Seattle (arrive 3:15 P.M.)
6: Las Vegas (depart 5 P.M.) - Seattle (arrive 6 P.M.).

(i)



(ii)

20.1.2.2 Flight scheduling...

(A) Use same airplane for two segments i and j:
  (a) destination of i is the origin of the segment j,
  (b) there is enough time in between the two flights.
(B) Also, airplane can fly from dest(i) to origin(j) (assuming time constraints are satisfied).

Example 20.1.2. *As a concrete example, consider the flights:*

1. *Boston (depart 6 A.M.) - Washington D.C. (arrive 7 A.M,).*
2. *Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)*
3. *Las Vegas (depart 5 P.M.) - Seattle (arrive 6 P.M.)*

This schedule can be served by a single airplane by adding the leg "Los Angeles (depart 12 noon)-Las Vegas (1 P,M.)" to this schedule.

20.1.2.3 Modeling the problem

(A) model the feasibility constraints by a graph.
(B) G: directed graph over flight legs.
(C) For i and j (legs), $(i \rightarrow j) \in \mathsf{E}(\mathsf{G}) \iff$ same airplane can serve both i and j.
(D) G is acyclic.
(E) Q: Can required legs can be served using only k airplanes?

20.1.2.4 Solution

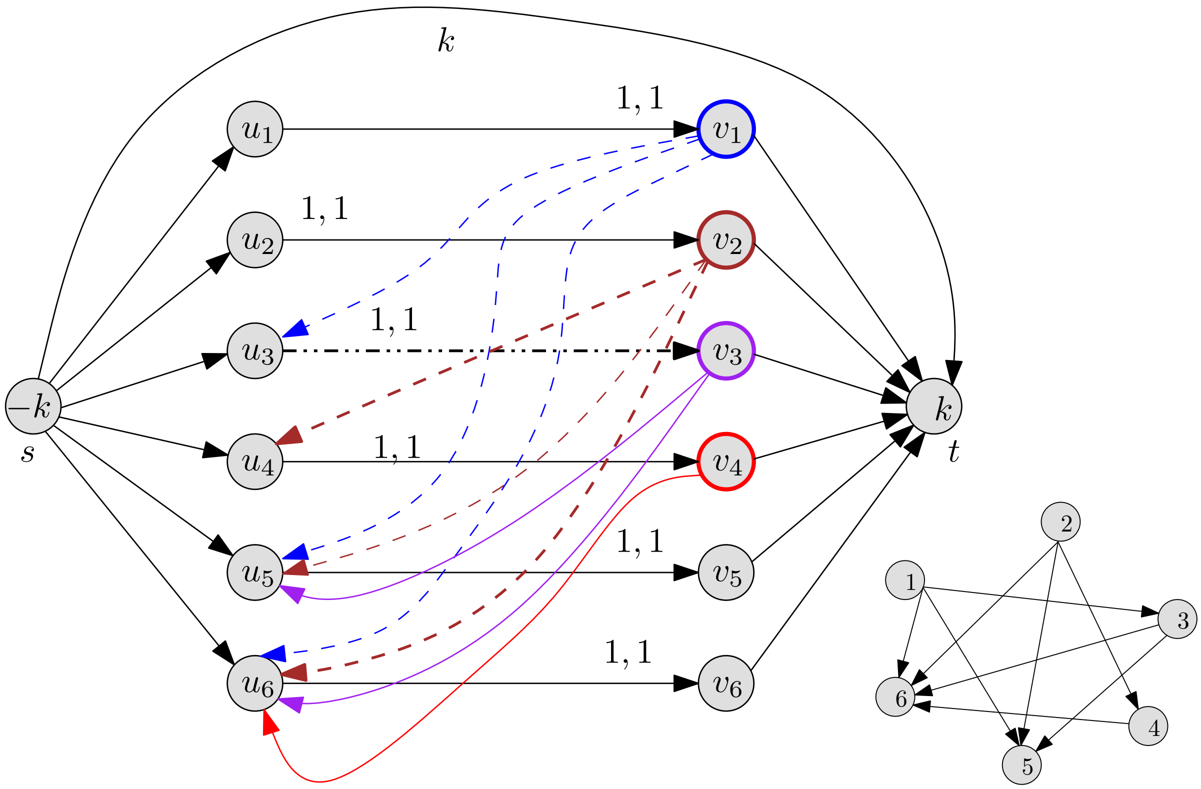
(A) Reduction to computation of circulation.
(B) Build graph H.
(C) $\forall$ leg i, two new vertices $u_i, v_i \in \mathsf{VH}$.
  s: source vertex. t: sink vertex.
(D) Set demand at t to k, Demand at s to be $-k$.
(E) Flight must be served: New edge $e_i = (u_i \rightarrow v_i)$, for leg i.
  Also $\ell(e_i) = 1$ and $c(e_i) = 1$.

(F) If same plane can so i and j (i.e., $(i \to j) \in \mathsf{E}(\mathsf{G})$) then add edge $(v_i \to u_j)$ with capacity 1 to H.
(G) Since any airplane can start the day with flight i: add an edge $(s \to u_i)$ with capacity 1 to H, $\forall i$.
(H) Add edge $(v_j \to t)$ with capacity 1 to G, $\forall j$.
(I) Overflow airplanes: "overflow" edge $(s \to t)$ with capacity k.

Let H denote the resulting graph.

20.1.3 Example of resulting graph

20.1.3.1 The resulting graph H for the instance of airline scheduling show before.



20.1.3.2 Lemma

Lemma 20.1.3. *$\exists$ way perform all flights of $\mathcal{F} \leq k$ planes $\iff$ $\exists$ circulation in H.*

Proof:
(A) Given feasible solution $\to$ translate into valid circulation.
(B) Given feasible circulation...
(C) ... extract paths from flow.
(D) ... every path is a plane.

■

20.1.3.3 Extensions and limitations

(A) a lot of other considerations:
  (i) airplanes have to undergo long term maintenance treatments every once in awhile,
  (ii) one needs to allocate crew to these flights,
  (iii) schedule differ between days, and
  (iv) ultimately we interested in maximizing revenue.
(B) Network flow is used in practice, real world problems are complicated, and network flow can capture only a few aspects.
(C) ... a good starting point.

20.1.4 Baseball Pennant Race

20.1.4.1 Pennant Race

TUESDAY, SEPTEMBER 10, 1996

San Francisco Chronicle

The Gate
Sports Online
http://www.sfgate.com

SPORTING G

49ers, Young Get Big Brea

Quarterback m

By Gary Swan
Chronicle Staff Writer

The bye week has come at a perfect time for the 49ers and quarterback Steve Young. If they had a game next Sunday, there's a good chance Young would not play.

But the pulled groin muscle on his up-

Giants Officially Leave the NL West Race

By Nancy Gay
Chronicle Staff Writer

With the smack of another National League West bat 500 miles away, the Giants' run at the division title ended last night, just as they were handing the visiting St. Louis Cardinals an even bigger lead in the NL Central.

CARDINALS 6
GIANTS 2

In San Diego, Greg Vaughn's three-run homer in the eighth pushed the Padres over the Pirates and officially shoved the rest of the Giants' season into the background. On the heels of their tedious 6-2 loss before an announced crowd of 10,307 at Candlestick Park, the Giants fell 19½ games off the lead.

As it is, the worst the Padres (80-65) can finish is 80-82. The Giants have fallen to 59-83 with 20 games left; they cannot win 80 games. Coming off a miserable 2-8 mark on a three-city road trip that saw their road record drop to 27-47, the Giants were hoping to get off on the right foot in their longest homestand of the year (15 games, 14 days).

Financing In Place
For Giants' New Stadium
SEE PAGE B1, MAIN NEWS

"Where we are, you're going to be eliminated sooner or later," Baker said quietly. "But it doesn't alter the fact that we've still got to play ball. You've still got to play hard, the fans come out to watch you play. You've got to play for the fact of loving to play, no matter where you are in the standings.

"You've got to play the role of spoiler, to not make it easier on

GIANTS: Page D5 Col. 3

20.1.4.2 Pennant Race: Example

Example 20.1.4.

Team	Won	Left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	89	2

Can Boston win the pennant?
No, because Boston can win at most 91 games.

20.1.4.3 Another Example

Can Boston win the pennant?
Not clear unless we know what the remaining games are!

Example 20.1.5.

Team	Won	Left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	90	2

20.1.4.4 Refining the Example

Example 20.1.6.

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

(A) Boston wins both its games to get 92 wins
(B) New York must lose both games; now both Baltimore and Toronto have at least 92
(C) Winner of Baltimore-Toronto game has 93 wins!

20.1.4.5 Abstracting the Problem

Given

(A) A set of teams S
(B) For each $x \in S$, the current number of wins w_x
(C) For any $x, y \in S$, the number of remaining games g_{xy} between x and y
(D) A team z

Can z win the pennant?

20.1.4.6 Towards a Reduction

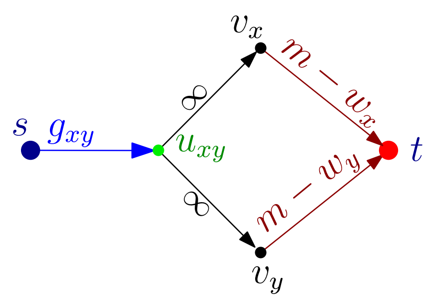
$\overline{z}$ can win the pennant if

(A) $\overline{z}$ wins at least m games
  (A) to maximize $\overline{z}$'s chances we make $\overline{z}$ win all its remaining games and hence $m = w_{\overline{z}} + \sum_{x \in S} g_{x\overline{z}}$
(B) no other team wins more than m games
  (A) for each $x, y \in S$ the g_{xy} games between them have to be *assigned* to either x or y.
  (B) each team $x \neq \overline{z}$ can win at most $m - w_x - g_{x\overline{z}}$ remaining games

Is there an assignment of remaining games to teams such that no team $x \neq \overline{z}$ wins more than $m - w_x$ games?

20.1.4.7 Flow Network: The basic gadget

(A) s: source
(B) t: sink
(C) x, y: two teams
(D) g_{xy}: number of games remaining between x and y.
(E) w_x: number of points x has.
(F) m: maximum number of points x can win before team of interest is eliminated.

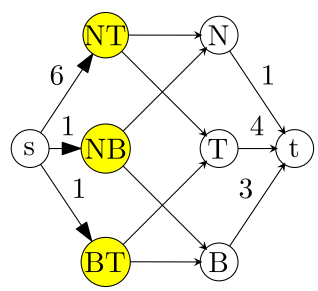


20.1.5 Flow Network: An Example

20.1.5.1 Can Boston win?

Team	Won	Left	NY	Bal	Tor	Bos
New York	90	11	–	1	6	4
Baltimore	88	6	1	–	1	4
Toronto	87	11	6	1	–	4
Boston	79	12	4	4	4	–

(A) $m = 79 + 12 = 91$: Boston can get at most 91 points.



20.1.5.2 Constructing Flow Network

Notations
(A) S: set of teams,
(B) w_x wins for each team, and
(C) g_{xy} games left between x and y.
(D) m be the maximum number of wins for $\overline{z}$,
(E) and $S' = S \setminus \{\overline{z}\}$.

Reduction Construct the flow network G as follows
(A) One vertex v_x for each team $x \in S'$, one vertex u_{xy} for each pair of teams x and y in S'
(B) A new source vertex s and sink t
(C) Edges (u_{xy}, v_x) and (u_{xy}, v_y) of capacity ∞
(D) Edges (s, u_{xy}) of capacity g_{xy}
(E) Edges (v_x, t) of capacity equal $m - w_x$

20.1.5.3 Correctness of reduction

Theorem 20.1.7. *G' has a maximum flow of value $g^* = \sum_{x,y \in S'} g_{xy}$ if and only if $\overline{z}$ can win the most number of games (including possibly tie with other teams).*

20.1.5.4 Proof of Correctness

Proof: Existence of g^* flow $\Rightarrow$ $\overline{z}$ wins pennant
(A) An integral flow saturating edges out of s, ensures that each remaining game between x and y is added to win total of either x or y

(B) Capacity on (v_x, t) edges ensures that no team wins more than m games

Conversely, $\overline{z}$ wins pennant $\Rightarrow$ flow of value g^*

(A) Scenario determines flow on edges; if x wins k of the games against y, then flow on (u_{xy}, v_x) edge is k and on (u_{xy}, v_y) edge is $g_{xy} - k$

∎

20.1.5.5 Proof that $\overline{z}$ cannot with the pennant

(A) Suppose $\overline{z}$ cannot win the pennant since $g^* < g$. How do we *prove* to some one *compactly* that $\overline{z}$ cannot win the pennant?

(B) Show them the min-cut in the reduction flow network!

(C) See text book for a natural interpretation of the min-cut as a certificate.

20.1.6 An Application of Min-Cut to Project Scheduling

20.1.6.1 Project Scheduling

Problem:

(A) n projects/tasks $1, 2, \ldots, n$

(B) *dependencies* between projects: i depends on j implies i cannot be done unless j is done. dependency graph is *acyclic*

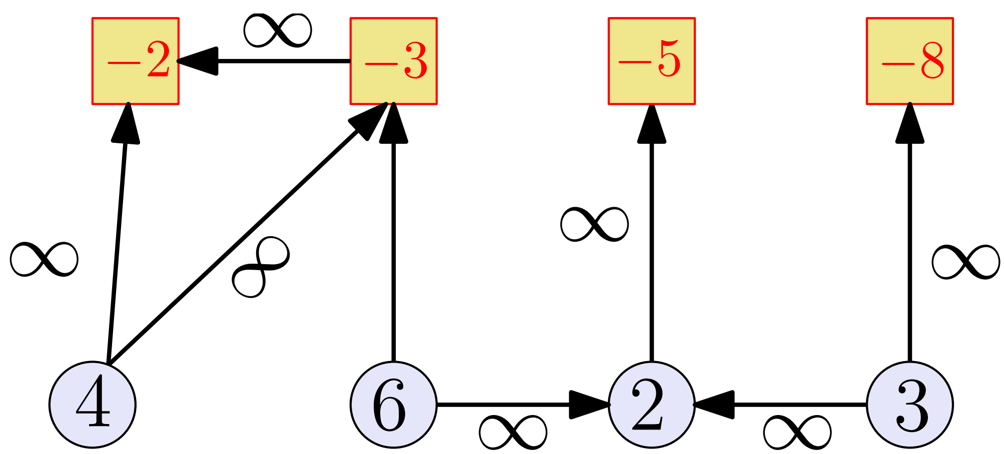
(C) each project i has a cost/profit p_i

  (A) $p_i < 0$ implies i requires a cost of $-p_i$ units

  (B) $p_i > 0$ implies that i generates p_i profit

Goal: Find projects to do so as to *maximize* profit.

20.1.6.2 Project selection example



20.1.6.3 Notation

For a set A of projects:

(A) A is a *valid* solution if A is *dependency closed*, that is for every $i \in A$, all projects that i depends on are also in A.

(B) $profit(A) = \sum_{i \in A} p_i$. Can be negative or positive.

Goal: find valid A to maximize $profit(A)$.

20.1.6.4 Idea: Reduction to Minimum-Cut

Finding a set of projects is partitioning the projects into two sets: those that are done and those that are not done.

Can we express this is a minimum cut problem?

Several issues:

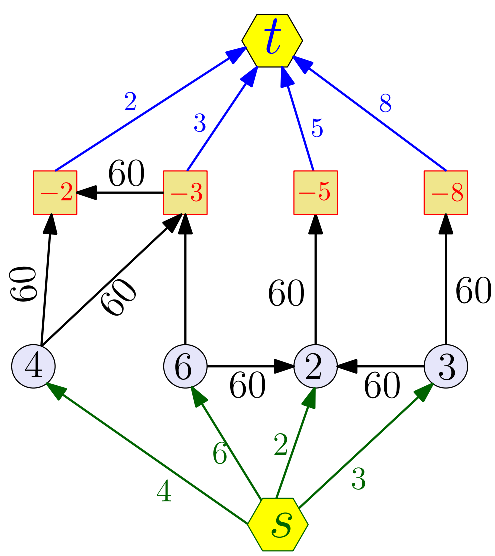
(A) We are interested in maximizing profit but we can solve minimum cuts.
(B) We need to convert negative profits into positive capacities.
(C) Need to ensure that chosen projects is a valid set.
(D) The cut value captures the profit of the chosen set of projects.

20.1.6.5 Reduction to Minimum-Cut

Note: We are reducing a *maximization* problem to a *minimization* problem.

(A) projects represented as nodes in a graph
(B) if i depends on j then (i, j) is an edge
(C) add source s and sink t
(D) for each i with $p_i > 0$ add edge (s, i) with capacity p_i
(E) for each i with $p_i < 0$ add edge (i, t) with capacity $-p_i$
(F) for each dependency edge (i, j) put capacity ∞ (more on this later)

20.1.6.6 Reduction: Flow Network Example



20.1.6.7 Reduction contd

Algorithm:

(A) form graph as in previous slide
(B) compute s-t minimum cut (A, B)
(C) output the projects in $A - \{s\}$

20.1.6.8 Understanding the Reduction

Let $C = \sum_{i: p_i > 0} p_i$: maximum possible profit.

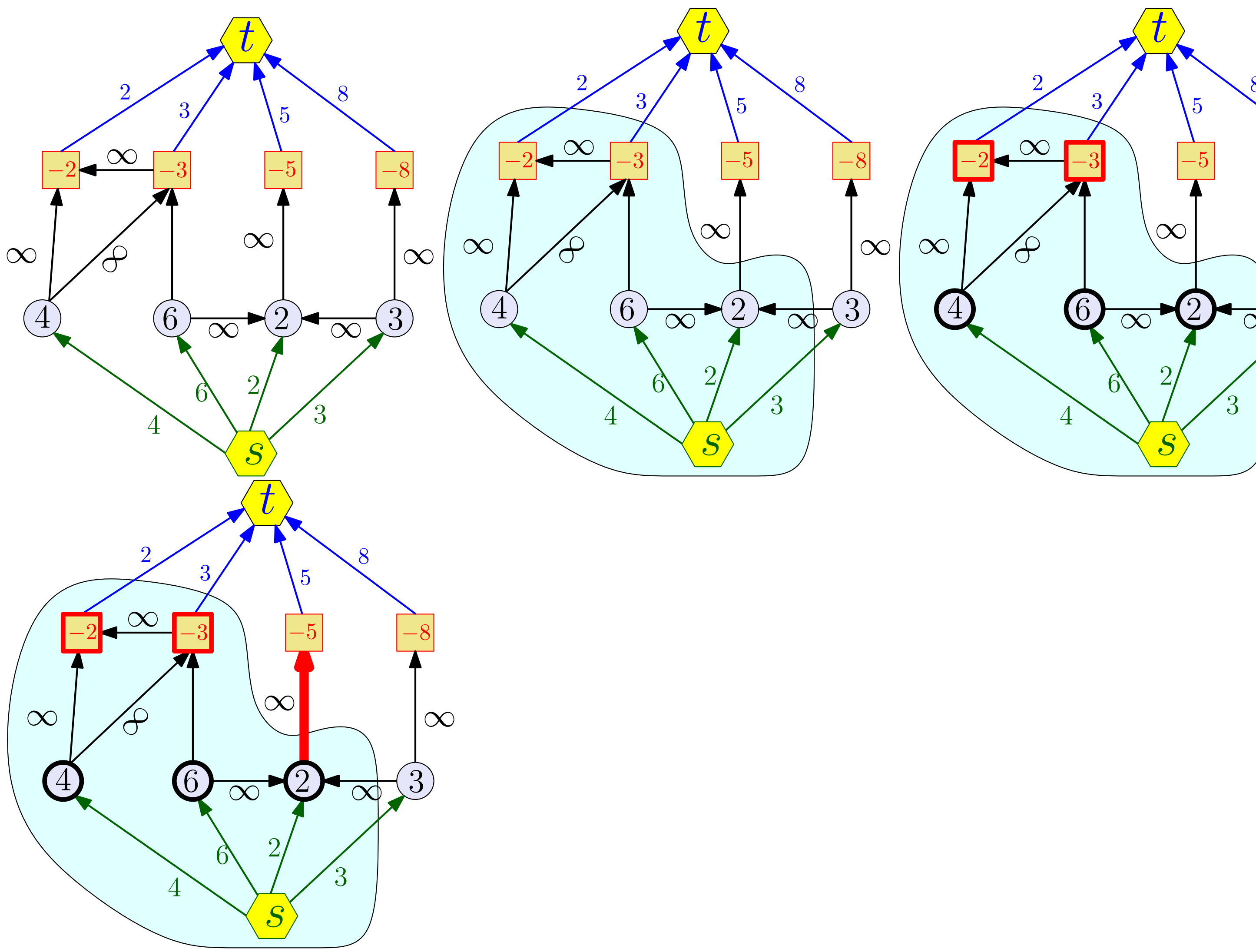
Observation: The minimum s-t cut value is $\leq C$. Why?

Lemma 20.1.8. *Suppose (A, B) is an s-t cut of finite capacity (no ∞) edges. Then projects in $A - \{s\}$ are a* valid *solution.*

Proof: If $A - \{s\}$ is not a valid solution then there is a project $i \in A$ and a project $j \notin A$ such that i depends on j
Since (i, j) capacity is ∞, implies (A, B) capacity is ∞, contradicting assumption. ∎

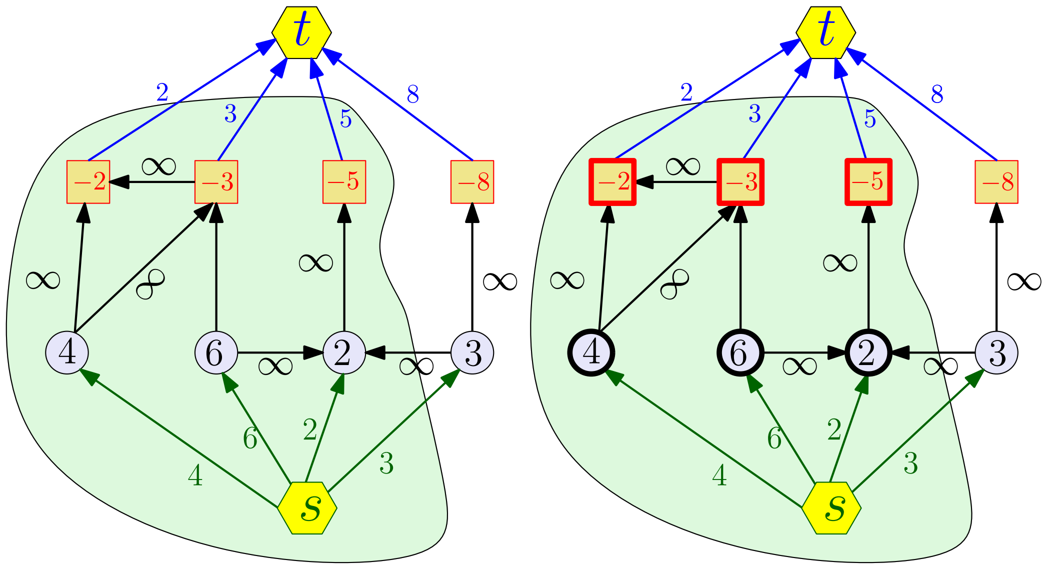
20.1.7 Reduction: Flow Network Example

20.1.7.1 Bad selection of projects



20.1.8 Reduction: Flow Network Example

20.1.8.1 Good selection of projects



20.1.8.2 Correctness of Reduction

Recall that for a set of projects X, $profit(X) = \sum_{i \in X} p_i$.

Lemma 20.1.9. *Suppose (A, B) is an s-t cut of finite capacity (no ∞) edges. Then $c(A, B) = C - profit(A - \{s\})$.*

Proof: Edges in (A, B):
(A) (s, i) for $i \in B$ and $p_i > 0$: capacity is p_i
(B) (i, t) for $i \in A$ and $p_i < 0$: capacity is $-p_i$
(C) cannot have ∞ edges

■

20.1.8.3 Proof contd

For project set A let
(A) $cost(A) = \sum_{i \in A: p_i < 0} -p_i$
(B) $benefit(A) = \sum_{i \in A: p_i > 0} p_i$
(C) $profit(A) = benefit(A) - cost(A)$.

Proof: Let $A' = A \cup \{s\}$.

$$\begin{aligned} c(A', B) &= cost(A) + benefit(B) \\ &= cost(A) - benefit(A) + benefit(A) + benefit(B) \\ &= -profit(A) + C \\ &= C - profit(A) \end{aligned}$$

■

20.1.8.4 Correctness of Reduction contd

We have shown that if (A, B) is an s-t cut in G with finite capacity then
(A) $A - \{s\}$ is a valid set of projects
(B) $c(A, B) = C - profit(A - \{s\})$
Therefore a *minimum* s-t cut (A^*, B^*) gives a *maximum* profit set of projects $A^* - \{s\}$ since C is fixed.
Question: How can we use ∞ in a real algorithm?
Set capacity of ∞ arcs to $C + 1$ instead. Why does this work?

20.1.9 Extensions to Maximum-Flow Problem

20.1.9.1 Lower Bounds and Costs

Two generalizations:
(A) flow satisfies $f(e) \le c(e)$ for all e. suppose we are given *lower bounds* $\ell(e)$ for each e. can we find a flow such that $\ell(e) \le f(e) \le c(e)$ for all e?
(B) suppose we are given a cost $w(e)$ for each edge. cost of routing flow $f(e)$ on edge e is $w(e)f(e)$. can we (efficiently) find a flow (of at least some given quantity) at minimum cost?
Many applications.

20.1.9.2 Flows with Lower Bounds

Definition 20.1.10. *A flow in a network $G = (V, E)$, is a function $f : E \to \mathbb{R}^{\ge 0}$ such that*
(A) ***Capacity Constraint:*** *For each edge e, $f(e) \le c(e)$*
(B) ***Lower Bound Constraint:*** *For each edge e, $f(e) \ge \ell(e)$*
(C) ***Conservation Constraint:*** *For each vertex v*

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

Question: Given G and $c(e)$ and $\ell(e)$ for each e, is there a flow?
As difficult as finding an s-t maximum-flow without lower bounds!

20.1.9.3 Regular flow via lower bounds

Given usual flow network G with source s and sink t, create lower-bound flow network G' as follows:
(A) set $\ell(e) = 0$ for each e in G
(B) add new edge (t, s) with lower bound v and upper bound ∞

Claim 20.1.11. *There exists a flow of value v from s to t in G if and only if there* exists *a feasible flow with lower bounds in G'.*

Above reduction show that lower bounds on flows are naturally related to **circulations**. With lower bounds, cannot guarantee acyclic flows from s to t.

20.1.9.4 Flows with Lower Bounds

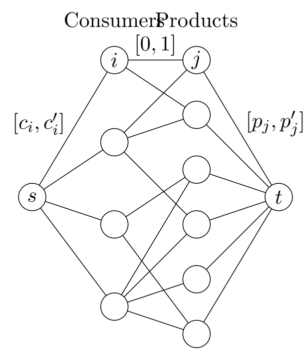
(A) Flows with lower bounds can be reduced to standard maximum flow problem. See text book. Reduction goes via circulations.
(B) If all bounds are integers then there is a flow that is integral. Useful in applications.

20.1.10 Survey Design

20.1.10.1 Application of Flows with Lower Bounds

(A) Design survey to find information about n_1 products from n_2 customers.
(B) Can ask customer questions only about products purchased in the past.
(C) Customer can only be asked about at most c'_i products and at least c_i products.
(D) For each product need to ask at east p_i consumers and at most p'_i consumers.

20.1.10.2 Reduction to Circulation



(A) include edge (i, j) is customer i has bought product j
(B) Add edge (t, s) with lower bound 0 and upper bound ∞.
    (A) Consumer i is asked about product j if the integral flow on edge (i, j) is 1

20.1.10.3 Minimum Cost Flows

(A) **Input:** Given a flow network G and also edge costs, $w(e)$ for edge e, and a flow requirement F.
(B) **Goal;** Find a *minimum cost* flow of value F from s to t
Given flow $f : E \to R^+$, cost of flow $= \sum_{e \in E} w(e)f(e)$.

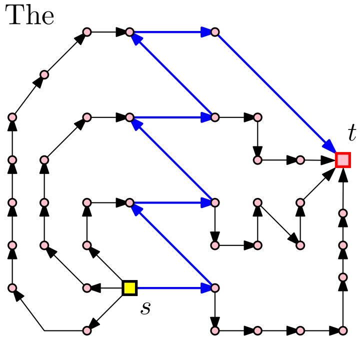
20.1.10.4 Minimum Cost Flow: Facts

(A) problem can be solved efficiently in polynomial time
    (A) $O(nm \log C \log(nW))$ time algorithm where C is maximum edge capacity and W is maximum edge cost
    (B) $O(m \log n(m + n \log n))$ time strongly polynomial time algorithm
(B) for integer capacities there is always an optimum solutions in which flow is integral

20.1.10.5 How much damage can a single path cause?

Consider the following network. All the edges have capacity 1. Clearly the maximum flow in this network has value 4.

The network



Why removing the shortest path might ruin everything

(A) However... The shortest path between s and t is the blue path.

(B) And if we remove the shortest path, s and t become disconnected, and the maximum flow drop to 0.

Chapter 21

Polynomial Time Reductions

OLD CS 473: Fundamental Algorithms, Spring 2015
April 14, 2015

21.0.11 Introduction to Reductions

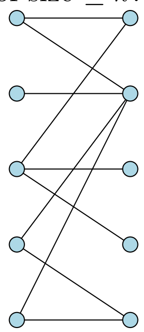
21.0.12 Overview

21.0.12.1 Reductions

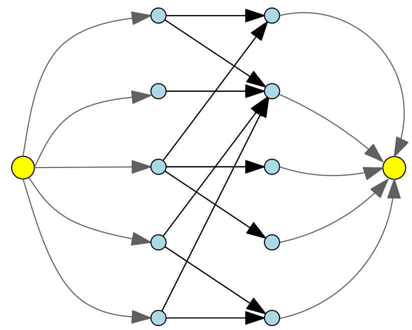
(A) Reduction from Problem X to Problem Y (informally): having algorithm for Y, then have algorithm for Problem X.
(B) We use reductions to find algorithms to solve problems.
(C) We also use reductions to show that we **can't** find algorithms for some problems. (We say that these problems are **hard**.)
(D) Also, the right reductions might win you a million dollars!

21.0.12.2 Example 1: Bipartite Matching and Flows

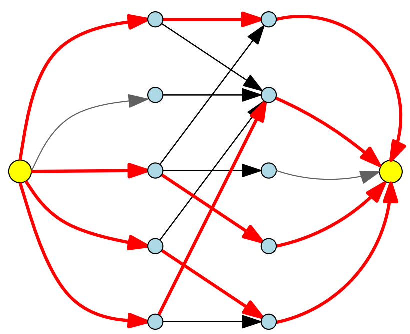
How do we solve the **Bipartite Matching** Problem? Given a bipartite graph $G = (U \cup V, E)$ and number k, does G have a matching of size $\geq k$?



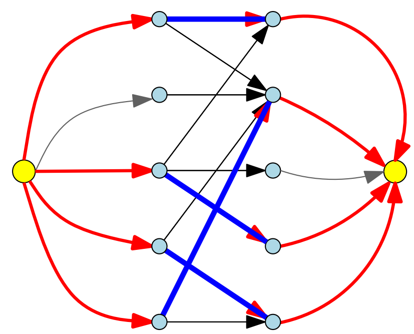
$\Longrightarrow$



$\Longrightarrow$



$\Longrightarrow$



¡4-¿Solution Reduce it to **Max-Flow**. G has a matching of size $\geq k \iff$ there is a flow from s to t of value $\geq k$.

21.0.13 Definitions

21.0.13.1 Types of Problems

Decision, Search, and Optimization

(A) ***Decision problem***. Example: given n, **is** n prime?.

(B) ***Search problem***. Example: given n, **find** a factor of n if it exists.

(C) ***Optimization problem***. Example: find the **smallest** prime factor of n.

21.0.14 Optimization and Decision problems

21.0.14.1 For max flow...

(A) Max-flow as optimization problem:

Problem 21.0.12 (Max-Flow optimization version). *Given an instance* G *of network flow, find the maximum flow between* s *and* t*.*

(B) Max-flow as decision problem:

Problem 21.0.13 (Max-Flow decision version). *Given an instance* G *of network flow and a parameter* K*, is there a flow in* G*, from* s *to* t*, of value at least* K*?*

(C) While using reductions and comparing problems, we typically work with the decision versions. Decision problems have **Yes/No** answers. This makes them easy to work with.

21.0.14.2 Problems vs Instances

(A) A **problem** Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an **instance**.
(B) The **size** of an instance I is the number of bits in its representation.
(C) For an instance I, $sol(I)$ is a set of **feasible solutions** to I.
(D) For optimization problems each solution $s \in sol(I)$ has an associated **value**.

21.0.14.3 Examples

(A) Instance **Bipartite Matching**: a bipartite graph, and integer k.
(B) Solution is "YES" if graph has matching size $\geq k$, else "NO".

(C) Instance **Max-Flow**: graph G with edge-capacities, two vertices s, t, and an integer k.
(D) Solution to instance is "YES" if there is a flow from s to t of value $\geq k$, else "NO".

(E) An algorithm for a decision Problem X?
(F) ***Decision algorithm***: Input an instance of X, and outputs either "YES" or "NO".

21.0.14.4 Encoding an instance into a string

(A) I; Instance of some problem.
(B) I can be fully and precisely described (say in a text file).
(C) Resulting text file is a binary string.
(D) $\implies$ Any input can be interpreted as a binary string S.
(E) ... Running time of algorithm: Function of length of S (i.e., n).

21.0.14.5 Decision Problems and Languages

(A) A finite **alphabet** Σ. Σ^* is set of all finite strings on Σ.
(B) A **language** L is simply a subset of Σ^*; a set of strings.
(C) Language $\equiv$ decision problem.
  (A) For any language $L \implies$ there is a decision problem Π_L.
  (B) For any decision problem $\Pi \implies$ an associated language L_Π.
(D) Given L, Π_L is the decision problem: Given $x \in \Sigma^*$, is $x \in L$? Each string in Σ^* is an instance of Π_L and L is the set of instances for which the answer is YES.
(E) Given Π the associated language is $L_\Pi = \left\{ I \,\middle|\, I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}$.
(F) Thus, decision problems and languages are used interchangeably.

21.0.14.6 Example

(A) The decision problem **Primality**, and the language

$$L = \left\{ \#p \,\middle|\, p \text{ is a prime number} \right\}.$$

Here $\#p$ is the string in base 10 representing p.
(B) **Bipartite** (is given graph is bipartite. The language is

$$L = \left\{ \mathcal{S}(\mathsf{G}) \,\middle|\, \mathsf{G} \text{ is a bipartite graph} \right\}.$$

Here $\mathcal{S}(\mathsf{G})$ is the string encoding the graph G.

21.0.14.7 Reductions, revised.

(A) For decision problems X, Y, a ***reduction from X to Y*** is:
  (A) An algorithm ...
  (B) Input: I_X, an instance of X.
  (C) Output: I_Y an instance of Y.
  (D) Such that:

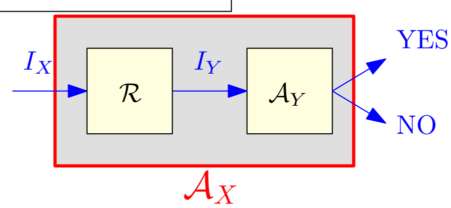
$$\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$$

(B) (Actually, this is only one type of reduction, but this is the one we'll use most often.)

21.0.14.8 Using reductions to solve problems

(A) $\mathcal{R}$: Reduction $X \to Y$
(B) $\mathcal{A}_Y$: algorithm for Y:
(C) $\implies$ New algorithm for X:

$\mathcal{A}_X(I_X)$:
    `// ` I_X`: instance of `X`.`
    $I_Y \Leftarrow \mathcal{R}(I_X)$
    return $\mathcal{A}_Y(I_Y)$



In particular, if $\mathcal{R}$ and $\mathcal{A}_Y$ are polynomial-time algorithms, $\mathcal{A}_X$ is also polynomial-time.

21.0.14.9 Comparing Problems

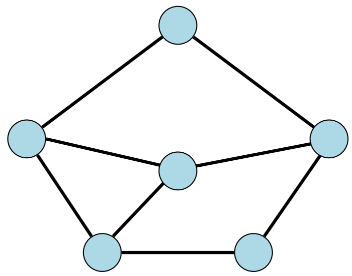
(A) Reductions allow us to formalize the notion of "Problem X is no harder to solve than Problem Y".
(B) If Problem X **reduces to** Problem Y (we write $X \le Y$), then X cannot be harder to solve than Y.
(C) **Bipartite Matching** $\le$ **Max-Flow**.
Therefore, **Bipartite Matching** cannot be harder than **Max-Flow**.
(D) Equivalently,
Max-Flow is **at least as hard as** **Bipartite Matching**.
(E) More generally, if $X \le Y$, we can say that X is no harder than Y, or Y is at least as hard as X.

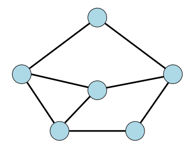
21.0.15 Examples of Reductions

21.0.16 Independent Set and Clique

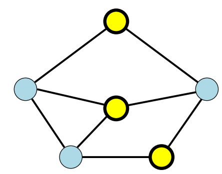
21.0.16.1 Independent Sets and Cliques

(A) Given a graph G.

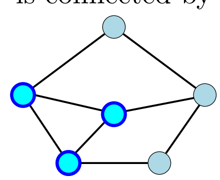




(B) A set of vertices V' is an ***independent set***: if no two vertices of V' are connected by an edge of G.



(C) ***clique***: *every* pair of vertices in V' is connected by an edge of G.



21.0.16.2 The **Independent Set** and **Clique** Problems

Problem: Independent Set

Instance: A graph G and an integer k. **Question:** Does G has an independent set of size $\geq k$?

Problem: Clique

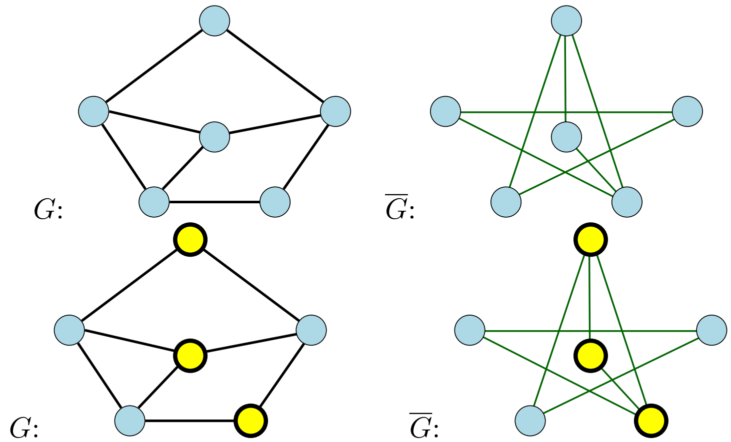
Instance: A graph G and an integer k. **Question:** Does G has a clique of size $\geq k$?

21.0.16.3 Recall

For decision problems X, Y, a reduction from X to Y is:

(A) An algorithm ...
(B) that takes I_X, an instance of X as input ...
(C) and returns I_Y, an instance of Y as output ...
(D) such that the solution (YES/NO) to I_Y is the same as the solution to I_X.

21.0.16.4 Reducing **Independent Set** to **Clique**



(A) An instance of **Independent Set** is a graph G and an integer k.
(B) Convert G to $\overline{G}$, in which (u, v) is an edge $\iff$ (u, v) is **not** an edge of G. ($\overline{G}$ is the *complement* of G.)
(C) ([)]$\overline{G}, k$: instance of **Clique**.

21.0.16.5 **Independent Set** and **Clique**

(A) **Independent Set** $\leq$ **Clique**.
What does this mean?
(B) If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
(C) **Clique** is *at least as hard as* **Independent Set**.
(D) Also... **Independent Set** is *at least as hard as* **Clique**.

21.0.17 NFAs/DFAs and Universality

21.0.17.1 DFAs and NFAs

(A) DFAs (Remember 373?) are determinstic automata that accept regular languages.
(B) NFAs are the same, except that non-deterministic.
(C) Every NFA can be converted to a DFA that accepts the same language using the **subset construction**.
(D) (How long does this take?)
(E) The smallest DFA equivalent to an NFA with n states may have $\approx 2^n$ states.

21.0.17.2 DFA Universality

(A) A DFA M is **universal** if it accepts every string.
(B) That is, $L(M) = \Sigma^*$, the set of all strings.
(C) DFA universality problem:

Problem 21.0.14 (DFA universality).
Input: *A* DFA M.
Goal: *Is* M *universal?*

(D) How do we solve **DFA Universality**?

(E) We check if M has *any* reachable non-final state.
(F) Alternatively, minimize M to obtain M' and see if M' has a single state which is an accepting state.

21.0.17.3 NFA Universality

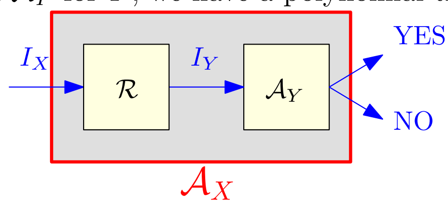
(A) An NFA N is **universal** if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.
(B) NFA universality problem:

Problem 21.0.15 (NFA universality).
Input: *A NFA M.*
Goal: *Is M universal?*

(C) How do we solve **NFA Universality**?
(D) Reduce it to **DFA Universality**...
(E) Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
(F) The reduction takes **exponential time**!

21.0.17.4 Polynomial-time reductions

(A) An algorithm is ***efficient*** if it runs in polynomial-time.
(B) To find efficient algorithms for problems, we are only interested in **polynomial-time** reductions. Reductions that take longer are not useful.
(C) If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.



21.0.17.5 Polynomial-time Reduction

(A) A polynomial time reduction from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ that has the following properties:
    (A) given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
    (B) $\mathcal{A}$ runs in time polynomial in $|I_X|$.
    (C) Answer to I_X YES $\iff$ answer to I_Y is YES.
(B) Polynomial transitivity:

Proposition 21.0.16. *If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.*

(C) Such a reduction is a ***Karp reduction***. Most reductions we will need are Karp reductions.

21.0.17.6 Polynomial-time reductions and hardness

(A) For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.

(B) If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?
(C) Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!
(D) If $X \leq_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

21.0.17.7 Polynomial-time reductions and instance sizes

Proposition 21.0.17. *Let $\mathcal{R}$ be a polynomial-time reduction from X to Y. Then for any instance I_X of X, the size of the instance I_Y of Y produced from I_X by $\mathcal{R}$ is polynomial in the size of I_X.*

Proof: $\mathcal{R}$ is a polynomial-time algorithm and hence on input I_X of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$.

I_Y is the output of $\mathcal{R}$ on input I_X.

$\mathcal{R}$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$. ■

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.

21.0.17.8 Polynomial-time Reduction

A polynomial time reduction from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ that has the following properties:
(A) Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
(B) $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of I_Y) is polynomial in $|I_X|$.
(C) Answer to I_X YES *iff* answer to I_Y is YES.

Proposition 21.0.18. *If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.*

Such a reduction is called a Karp reduction. Most reductions we will need are Karp reductions

21.0.17.9 Transitivity of Reductions

(A) Reductions are transitive:

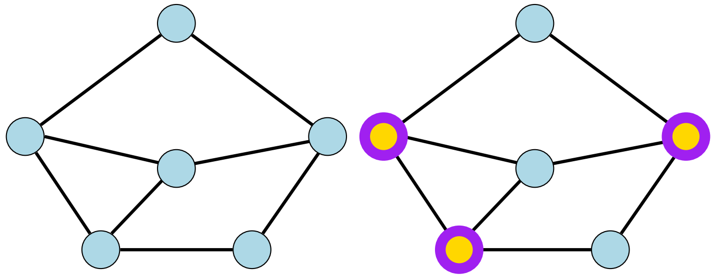
Proposition 21.0.19. *$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.*

(B) **Note:** $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.
(C) To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y.
(D) In other words show that an algorithm for Y implies an algorithm for X.

21.0.18 Independent Set and Vertex Cover

21.0.18.1 Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:
(A) A ***vertex cover*** if every $e \in E$ has at least one endpoint in S.



21.0.18.2 The **Vertex Cover** Problem

Problem 21.0.20 (Vertex Cover).
Input: *A graph* G *and integer* k*.*
Goal: *Is there a vertex cover of size* $\leq k$ *in* G*?*

Can we relate **Independent Set** and **Vertex Cover**?

21.0.19 Relationship between...

21.0.19.1 Vertex Cover and Independent Set

Proposition 21.0.21. *Let* $G = (V, E)$ *be a graph.* S *is an independent set if and only if* $V \setminus S$ *is a vertex cover.*

Proof:
($\Rightarrow$) Let S be an independent set
- (A) Consider any edge $uv \in E$.
- (B) Since S is an independent set, either $u \notin S$ or $v \notin S$.
- (C) Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
- (D) $V \setminus S$ is a vertex cover.

($\Leftarrow$) Let $V \setminus S$ be some vertex cover:
- (A) Consider $u, v \in S$
- (B) uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
- (C) $\implies$ S is thus an independent set.

∎

21.0.19.2 **Independent Set** $\leq_P$ **Vertex Cover**

(A) G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.
(B) G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$
(C) (G, k) is an instance of **Independent Set** , and $(G, n - k)$ is an instance of **Vertex Cover** with the same answer.
(D) Therefore, **Independent Set** $\leq_P$ **Vertex Cover**. Also **Vertex Cover** $\leq_P$ **Independent Set**.

21.0.20 Vertex Cover and Set Cover

21.0.20.1 A problem of Languages

(A) Suppose you work for the United Nations. Let U be the set of all **languages** spoken by people across the world. The United Nations also has a set of **translators**, all of whom speak English, and some other languages from U.
(B) Due to budget cuts, you can only afford to keep k translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in U?
(C) More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

21.0.20.2 The Set Cover Problem

Problem 21.0.22 (Set Cover).

Input: *Given a set U of n elements, a collection $S_1, S_2, \ldots S_m$ of subsets of U, and an integer k.*
Goal: *Is there a collection of at most k of these sets S_i whose union is equal to U?*

Example 21.0.23. *¡2-¿Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with*

$$\begin{array}{ll} S_1 = \{3,7\} & \text{¡}3- > S_2 = \{3,4,5\} \\ S_3 = \{1\} & S_4 = \{2,4\} \\ S_5 = \{5\} & \text{¡}3- > S_6 = \{1,2,6,7\} \end{array}$$

$\{S_2, S_6\}$ is a set cover

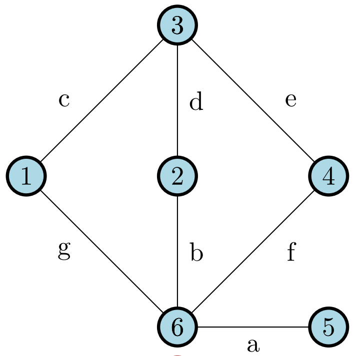
21.0.20.3 Vertex Cover $\leq_P$ Set Cover

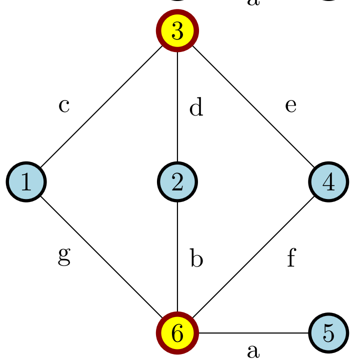
Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

(A) Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.
(B) $U = E$.
(C) We will have one set corresponding to each vertex; $S_v = \{e \mid e \text{ is incident on } v\}$.

Observe that G has vertex cover of size k if and only if $U, \{S_v\}_{v \in V}$ has a set cover of size k. (Exercise: Prove this.)

21.0.20.4 Vertex Cover $\leq_P$ Set Cover: Example





Let $U = \{a, b, c, d, e, f, g\}$, $k = 2$ with

$S_1 = \{c, g\}$	$S_2 = \{b, d\}$
¡3$- >$ $S_3 = \{c, d, e\}$	$S_4 = \{e, f\}$
$S_5 = \{a\}$	¡3$- >$ $S_6 = \{a, b, f, g\}$

$\{S_3, S_6\}$ is a set cover

$\{3, 6\}$ is a vertex cover

21.0.20.5 Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

(A) Transforms an instance I_X of X into an instance I_Y of Y.

(B) Satisfies the property that answer to I_X is YES iff I_Y is YES.

  (A) typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES

  (B) **typical difficult direction to prove**: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).

(C) Runs in *polynomial* time.

21.0.20.6 Example of incorrect reduction proof

Try proving **Matching** $\leq_P$ **Bipartite Matching** via following reduction:

(A) Given graph $G = (V, E)$ obtain a bipartite graph $G' = (V', E')$ as follows.

  (A) Let $V_1 = \{u_1 \mid u \in V\}$ and $V_2 = \{u_2 \mid u \in V\}$. We set $V' = V_1 \cup V_2$ (that is, we make two copies of V)

  (B) $E' = \left\{ u_1 v_2 \;\middle|\; u \neq v \text{ and } uv \in E \right\}$

(B) Given G and integer k the reduction outputs G' and k.

21.0.20.7 Example

21.0.20.8 "Proof"

Claim 21.0.24. *Reduction is a poly-time algorithm. If G has a matching of size k then G' has a matching of size k.*

Proof: Exercise. ■

Claim 21.0.25. *If G' has a matching of size k then G has a matching of size k.*

Incorrect! Why? Vertex $u \in V$ has two copies u_1 and u_2 in G'. A matching in G' may use both copies!

21.0.20.9 Summary

- (A) We looked at **polynomial-time reductions**.
- (B) Using polynomial-time reductions
  - (A) If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
  - (B) If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.
- (C) We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Chapter 22

Reductions and NP

OLD CS 473: Fundamental Algorithms, Spring 2015
April 16, 2015

22.1 Reductions Continued

22.1.1 Reductions

22.1.2 Polynomial Time Reduction

22.1.2.1 Karp reduction

A ***polynomial time reduction*** from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ that has the following properties:
(A) given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
(B) $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of I_Y) is polynomial in $|I_X|$
(C) Answer to I_X YES *iff* answer to I_Y is YES.
Notation: $X \leq_P Y$ if X reduces to Y

Proposition 22.1.1. *If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.*

Such a reduction is called a ***Karp reduction***. Most reductions we will need are Karp reductions.

22.1.3 A More General Reduction

22.1.3.1 Turing Reduction

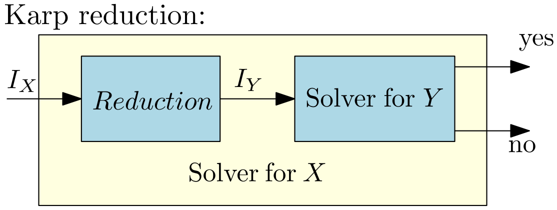
Definition 22.1.2 (Turing reduction.). *Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:*
(A) on any given instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
(B) a step is either a standard computation step, or

(C) a sub-routine call to an algorithm that solves Y.
This is a **Turing reduction**.

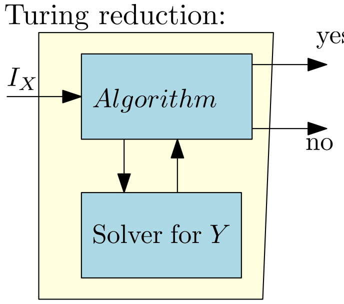
Note: In making sub-routine call to algorithm to solve Y, $\mathcal{A}$ can only ask questions of size polynomial in $|I_X|$. Why?

22.1.3.2 Comparing reductions

(A) Karp reduction:



(B) Turing reduction:



Turing reduction
(A) Algorithm to solve X can call solver for Y many times.
(B) Conceptually, every call to the solver of Y takes constant time.

22.1.3.3 Example of Turing Reduction

Problem 22.1.3 (Independent set in circular arcs graph.).
Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem 22.1.4 (Independent set of intervals.).
Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How? Used algorithm for interval problem multiple times.

22.1.3.4 Turing vs Karp Reductions

(A) Turing reductions more general than Karp reductions.
(B) Turing reduction useful in obtaining algorithms via reductions.
(C) Karp reduction is simpler and easier to use to prove hardness of problems.
(D) Perhaps surprisingly, Karp reductions, although limited, suffice for most known **NP-Complete**ness proofs.
(E) Karp reductions allow us to distinguish between **NP** and **co-NP** (more on this later).

22.1.4 The Satisfiability Problem (SAT)

22.1.4.1 Propositional Formulas

Definition 22.1.5. *Consider a set of boolean variables $x_1, x_2, \ldots x_n$.*

(A) A **literal** *is either a boolean variable x_i or its negation $\neg x_i$.*

(B) A **clause** *is a disjunction of literals.*
For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.

(C) A **formula in conjunctive normal form** *(CNF) is propositional formula which is a conjunction of clauses*
  (A) $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.

(D) A formula φ is a 3CNF:
A CNF formula such that every clause has ***exactly*** *3 literals.*
  (A) $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_1)$ is a 3CNF formula, but $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is not.

22.1.4.2 Satisfiability

Problem: SAT

Instance: A CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Problem: 3SAT

Instance: A 3CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

22.1.4.3 Satisfiability

SAT Given a CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

Example 22.1.6. *(A) $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true*

(B) $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is not satisfiable.

3SAT Given a 3CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

(More on **2SAT** in a bit...)

22.1.4.4 Importance of SAT and 3SAT

(A) **SAT** and **3SAT** are basic constraint satisfaction problems.
(B) Many different problems can reduced to them because of the simple yet powerful expressively of logical constraints.
(C) Arise naturally in many applications involving hardware and software verification and correctness.
(D) As we will see, it is a fundamental problem in theory of **NP-Complete**ness.

22.1.5 Converting a boolean formula with 3 variables to 3SAT

22.1.5.1 Converting $z = x \wedge y$ to 3SAT

z	x	y	$z = x \wedge y$	$z \vee \overline{x} \vee \overline{y}$	$\overline{z} \vee x \vee y$	$\overline{z} \vee x \vee \overline{y}$	$\overline{z} \vee \overline{x} \vee y$
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1
0	1	1	0	00	1	1	1
1	0	0	0	1	00	1	1
1	0	1	0	1	1	00	1
1	1	0	0	1	1	1	00
1	1	1	1	1	1	1	1

$$\Big(z = x \wedge y\Big)$$
$$\equiv$$
$$(z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y) \wedge (\overline{z} \vee x \vee \overline{y}) \wedge (\overline{z} \vee \overline{x} \vee y)$$

22.1.5.2 Converting $z = x \wedge y$ to 3SAT

z	x	y	$z = x \wedge y$	clauses
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	00	$z \vee \overline{x} \vee \overline{y}$
1	0	0	00	$\overline{z} \vee x \vee y$
1	0	1	00	$\overline{z} \vee x \vee y$
1	1	0	00	$\overline{z} \vee x \vee y$
1	1	1	1	

$$\Big(z = x \wedge y\Big)$$
$$\equiv$$
$$(z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y) \wedge (\overline{z} \vee x \vee \overline{y}) \wedge (\overline{z} \vee \overline{x} \vee y)$$

22.1.6 Converting $z = x \vee y$ to 3SAT

22.1.6.1 Simplify further if you want to

(A) Using that $(x \vee y) \wedge (x \vee \overline{y}) = x$, we have that:
  (A) $\big(\overline{z} \vee x \vee u\big) \wedge \big(\overline{z} \vee x \vee \overline{y}\big) = \big(\overline{z} \vee x\big)$

(B) $\left(\overline{z} \vee x \vee y\right) \wedge \left(\overline{z} \vee \overline{x} \vee y\right) = \left(\overline{z} \vee y\right)$

(B) Using the above two observation, we have that our formula $\psi \equiv \left(z \vee \overline{x} \vee \overline{y}\right) \wedge \left(\overline{z} \vee x \vee y\right) \wedge \left(\overline{z} \vee x \vee \overline{y}\right) \wedge \left(\overline{z} \vee \overline{x} \vee y\right)$

is equivalent to $\psi \equiv \left(z \vee \overline{x} \vee \overline{y}\right) \wedge \left(\overline{z} \vee x\right) \wedge \left(\overline{z} \vee y\right)$

Lemma 22.1.7. $\left(z = x \wedge y\right) \quad \equiv \quad \left(z \vee \overline{x} \vee \overline{y}\right) \wedge \left(\overline{z} \vee x\right) \wedge \left(\overline{z} \vee y\right)$

22.1.6.2 Converting $z = x \vee y$ to 3SAT

z	x	y	$z = x \vee y$	clauses
0	0	0	1	
0	0	1	00	$z \vee x \vee \overline{y}$
0	1	0	00	$z \vee \overline{x} \vee y$
0	1	1	00	$z \vee \overline{x} \vee \overline{y}$
1	0	0	00	$\overline{z} \vee x \vee y$
1	0	1	1	
1	1	0	1	
1	1	1	1	

$$\left(z = x \vee y\right)$$
$$\equiv$$
$$(z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y)$$

22.1.7 Converting $z = x \vee y$ to 3SAT

22.1.7.1 Simplify further if you want to

$\left(z = x \vee y\right) \equiv (z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y)$

(A) Using that $(x \vee y) \wedge (x \vee \overline{y}) = x$, we have that:

(A) $(z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee \overline{y}) = z \vee \overline{y}$.

(B) $(z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) = z \vee \overline{x}$

(B) Using the above two observation, we have the following.

Lemma 22.1.8. *The formula $z = x \vee y$ is equivalent to the CNF formula* $\left(z = x \vee y\right) \equiv (z \vee \overline{y}) \wedge (z \vee \overline{x}) \wedge (\overline{z} \vee x \vee y)$

22.1.7.2 Converting $z = \overline{x}$ to CNF

Lemma 22.1.9. $z = \overline{x} \quad \equiv \quad (z \vee x) \wedge (\overline{z} \vee \overline{x})$.

22.1.7.3 Converting into CNF- summary

Lemma 22.1.10.

$$
\begin{aligned}
z = \overline{x} &\equiv (z \vee x) \wedge (\overline{z} \vee \overline{x}). \\
z = x \vee y &\equiv (z \vee \overline{y}) \wedge (z \vee \overline{x}) \wedge (\overline{z} \vee x \vee y) \\
z = x \wedge y &\equiv (z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x) \wedge (\overline{z} \vee y)
\end{aligned}
$$

22.1.8 SAT and 3SAT

22.1.8.1 SAT$\leq_P$ 3SAT

How **SAT** is different from **3SAT**?In **SAT** clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

$$\Big(x \vee y \vee z \vee w \vee u\Big) \wedge \Big(\neg x \vee \neg y \vee \neg z \vee w \vee u\Big) \wedge \Big(\neg x\Big)$$

In **3SAT** every clause must have ***exactly*** 3 different literals.

To reduce from an instance of **SAT** to an instance of **3SAT**, we must make all clauses to have exactly 3 variables...

Basic idea

(A) Pad short clauses so they have 3 literals.
(B) Break long clauses into shorter clauses.
(C) Repeat the above till we have a 3CNF.

22.1.8.2 3SAT$\leq_P$ SAT

(A) **3SAT** $\leq_P$ **SAT**.
(B) Because...
A **3SAT** instance is also an instance of **SAT**.

22.1.8.3 SAT$\leq_P$ 3SAT

Claim 22.1.11. *SAT $\leq_P$ 3SAT.*

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

(A) φ is satisfiable iff φ' is satisfiable.
(B) φ' can be constructed from φ in time polynomial in $|\varphi|$.

Idea: if a clause of φ is not of length 3, replace it with several clauses of length exactly 3.

22.1.9 SAT$\leq_P$ 3SAT

22.1.9.1 A clause with a single literal

Reduction Ideas **Challenge:** Some of the clauses in φ may have less or more than 3 literals. For each clause with < 3 or > 3 literals, we will construct a set of logically equivalent clauses.

(A) **Case clause with one literal:** Let c be a clause with a single literal (i.e., $c = \ell$). Let u, v be new variables. Consider

$$
\begin{aligned}
c' = \; & \Big(\ell \vee u \vee v\Big) \wedge \Big(\ell \vee u \vee \neg v\Big) \\
& \wedge \Big(\ell \vee \neg u \vee v\Big) \wedge \Big(\ell \vee \neg u \vee \neg v\Big).
\end{aligned}
$$

Observe that c' is satisfiable iff c is satisfiable

22.1.10 SAT$\leq_P$ 3SAT

22.1.10.1 A clause with two literals

Reduction Ideas: 2 and more literals

(A) **Case clause with 2 literals:** Let $c = \ell_1 \vee \ell_2$. Let u be a new variable. Consider

$$c' = \Big(\ell_1 \vee \ell_2 \vee u\Big) \wedge \Big(\ell_1 \vee \ell_2 \vee \neg u\Big).$$

Again c is satisfiable iff c' is satisfiable

22.1.10.2 Breaking a clause

Lemma 22.1.12. *For any boolean formulas X and Y and z a new boolean variable. Then*

$$X \vee Y \textit{ is satisfiable}$$

if and only if, z can be assigned a value such that

$$\Big(X \vee z\Big) \wedge \Big(Y \vee \neg z\Big) \textit{ is satisfiable}$$

(with the same assignment to the variables appearing in X and Y).

22.1.11 SAT$\leq_P$ 3SAT (contd)

22.1.11.1 Clauses with more than 3 literals

Let $c = \ell_1 \vee \cdots \vee \ell_k$. Let $u_1, \ldots u_{k-3}$ be new variables. Consider

$$\begin{aligned} c' = &\Big(\ell_1 \vee \ell_2 \vee u_1\Big) \wedge \Big(\ell_3 \vee \neg u_1 \vee u_2\Big) \\ &\wedge \Big(\ell_4 \vee \neg u_2 \vee u_3\Big) \wedge \\ &\cdots \wedge \Big(\ell_{k-2} \vee \neg u_{k-4} \vee u_{k-3}\Big) \wedge \Big(\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}\Big). \end{aligned}$$

Claim 22.1.13. *c is satisfiable iff c' is satisfiable.*

Another way to see it — reduce size of clause by one:

$$c' = \Big(\ell_1 \vee \ell_2 \ldots \vee \ell_{k-2} \vee u_{k-3}\Big) \wedge \Big(\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}\Big).$$

22.1.11.2 An Example

Example 22.1.14.

$$\varphi = \Big(\neg x_1 \vee \neg x_4\Big) \wedge \Big(x_1 \vee \neg x_2 \vee \neg x_3\Big) \\ \wedge \Big(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\Big) \wedge \Big(x_1\Big).$$

Equivalent form:

$$\begin{aligned} \psi = & (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z) \\ & \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1) \\ & \wedge (x_1 \vee u \vee v) \wedge (x_1 \vee u \vee \neg v) \\ & \wedge (x_1 \vee \neg u \vee v) \wedge (x_1 \vee \neg u \vee \neg v). \end{aligned}$$

22.1.12 Overall Reduction Algorithm

22.1.12.1 Reduction from SAT to 3SAT

```
ReduceSATTo3SAT(φ):
    // φ: CNF formula.
    for each clause c of φ do
        if c does not have exactly 3 literals then
            construct c' as before
        else
            c' = c
    ψ is conjunction of all c' constructed in loop
    return Solver3SAT(ψ)
```

Correctness (informal) φ is satisfiable iff ψ is satisfiable because for each clause c, the new 3CNF formula c' is logically equivalent to c.

22.1.12.2 What about 2SAT?

2SAT can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from **SAT** (or **3SAT**) to **2SAT**. If there was, then **SAT** and **3SAT** would be solvable in polynomial time.

Why the reduction from **3SAT** to **2SAT** fails?

Consider a clause $(x \vee y \vee z)$. We need to reduce it to a collection of 2CNF clauses. Introduce a face variable α, and rewrite this as

$$\begin{array}{lll} & (x \vee y \vee \alpha) \wedge (\neg \alpha \vee z) & \text{(bad! clause with 3 vars)} \\ \text{or} & (x \vee \alpha) \wedge (\neg \alpha \vee y \vee z) & \text{(bad! clause with 3 vars).} \end{array}$$

(In animal farm language: **2SAT** good, **3SAT** bad.)

22.1.12.3 What about 2SAT?

A challenging exercise: Given a **2SAT** formula show to compute its satisfying assignment...

(Hint: Create a graph with two vertices for each variable (for a variable x there would be two vertices with labels $x = 0$ and $x = 1$). For ever 2CNF clause add two directed edges in the graph. The edges are implication edges: They state that if you decide to assign a certain value to a variable, then you must assign a certain value to some other variable.

Now compute the strong connected components in this graph, and continue from there...)

22.1.13 3SAT and Independent Set

22.1.13.1 Independent Set

Problem: Independent Set

Instance: A graph G, integer k.
Question: Is there an independent set in G of size k?

22.1.13.2 3SAT$\leq_P$ Independent Set

The reduction **3SAT$\leq_P$ Independent Set Input:** Given a 3CNF formula φ

Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

G_φ should be constructable in time polynomial in size of φ

(A) **Importance of reduction:** Although **3SAT** is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.
(B) **Notice:** We handle only 3CNF formulas – reduction would not work for other kinds of boolean formulas.

22.1.13.3 Interpreting 3SAT

There are two ways to think about **3SAT**

(A) Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
(B) Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in **conflict**, i.e., you pick x_i and $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

22.1.13.4 The Reduction

(A) G_φ will have one vertex for each literal in a clause
(B) Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
(C) Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
(D) Take k to be the number of clauses

22.1.13.5 Correctness

Proposition 22.1.15. *φ is satisfiable iff G_φ has an independent set of size k (= number of clauses in φ).*

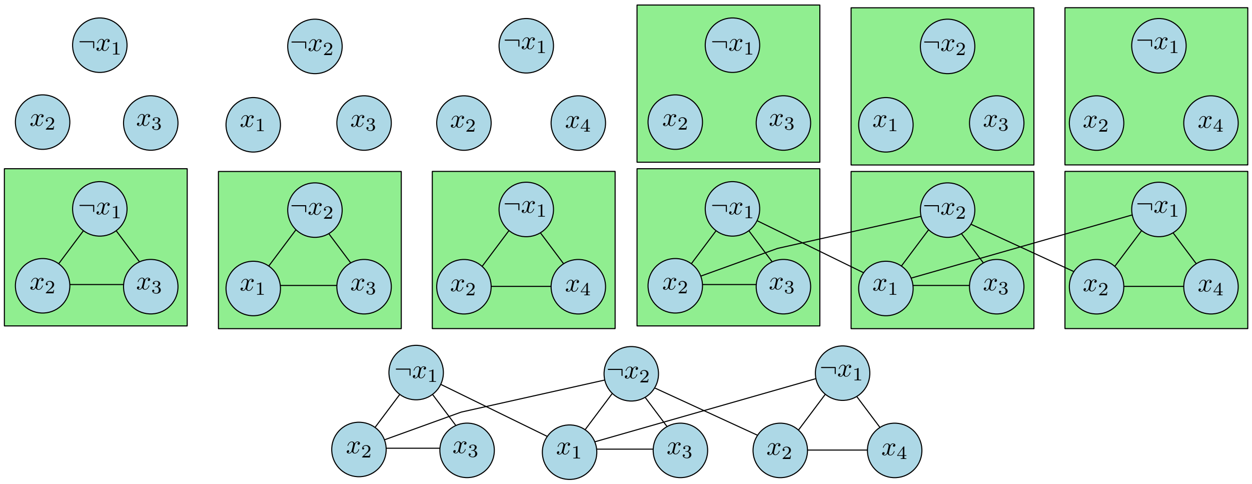


Figure 22.1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Proof:

$\Rightarrow$ Let a be the truth assignment satisfying φ

(A) Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size

■

22.1.13.6 Correctness (contd)

Proposition 22.1.16. *φ is satisfiable iff G_φ has an independent set of size k (= number of clauses in φ).*

Proof:

$\Leftarrow$ Let S be an independent set of size k

(A) S must contain exactly one vertex from each clause

(B) S cannot contain vertices labeled by conflicting clauses

(C) Thus, it is possible to obtain a truth assignment that makes in the literals in S true; such an assignment satisfies one literal in every clause

■

22.1.13.7 Transitivity of Reductions

Lemma 22.1.17. *$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.*

Note: $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.

To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y
In other words show that an algorithm for Y implies an algorithm for X.

22.2 Definition of NP

22.2.0.8 Recap ...

Problems

(A) **Independent Set**
(B) **Vertex Cover**
(C) **Set Cover**
(D) **SAT**
(E) **3SAT**

Relationship

$$\textbf{3SAT} \le_P \textbf{Independent Set} \overset{\le_P}{\ge_P} \textbf{Vertex Cover} \le_P \textbf{Set Cover}$$
$$\textbf{3SAT} \le_P \textbf{SAT} \le_P \textbf{3SAT}$$

22.2.1 Preliminaries

22.2.2 Problems and Algorithms

22.2.2.1 Problems and Algorithms: Formal Approach

Decision Problems

(A) **Problem Instance:** Binary string s, with size $|s|$
(B) **Problem:** A set X of strings on which the answer should be "yes"; we call these YES instances of X. Strings not in X are NO instances of X.

Definition 22.2.1. *(A) A is an **algorithm for problem** X if $A(s) =$ "yes" iff $s \in X$.*
*(B) A is said to have a **polynomial running time** if there is a polynomial $p(\cdot)$ such that for every string s, $A(s)$ terminates in at most $O(p(|s|))$ steps.*

22.2.2.2 Polynomial Time

Definition 22.2.2. Polynomial time *(denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.*

Example 22.2.3. *Problems in **P** include*
(A) Is there a shortest path from s to t of length $\le k$ in G?
(B) Is there a flow of value $\ge k$ in network G?
(C) Is there an assignment to variables to satisfy given linear constraints?

22.2.2.3 Efficiency Hypothesis

*A problem X has an efficient algorithm iff $X \in$ **P**, that is X has a polynomial time algorithm.*

Justifications:

(A) Robustness of definition to variations in machines.
(B) A sound theoretical definition.
(C) Most known polynomial time algorithms for "natural" problems have small polynomial running times.

22.2.2.4 Problems with no known polynomial time algorithms

Problems
(A) **Independent Set**
(B) **Vertex Cover**
(C) **Set Cover**
(D) **SAT**
(E) **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.
Question: What is common to above problems?

22.2.2.5 Efficient Checkability

Above problems share the following feature:

Checkability *For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.*

Examples:
(A) **SAT** formula φ: proof is a satisfying assignment.
(B) **Independent Set** in graph G and k: a subset S of vertices.

22.2.3 Certifiers/Verifiers

22.2.3.1 Certifiers

Definition 22.2.4. *An algorithm $C(\cdot,\cdot)$ is a* **certifier** *for problem X if for every $s \in X$ there is some string t such that $C(s,t) =$ "yes", and conversely, if for some s and t, $C(s,t) =$ "yes" then $s \in X$.*

The string t is called a ***certificate*** *or* ***proof*** *for s.*

Definition 22.2.5 (Efficient Certifier.). *A certifier C is an* **efficient certifier** *for problem X if there is a polynomial $p(\cdot)$ such that for every string s, we have that*
- ⋆ *$s \in X$ if and only if*
- ⋆ *there is a string t:*
  - *(A) $|t| \le p(|s|)$,*
  - *(B) $C(s,t) =$ "yes",*
  - *(C) and C runs in polynomial time.*

22.2.3.2 Example: Independent Set

(A) **Problem:** Does $G = (V, E)$ have an independent set of size $\ge k$?
  (A) **Certificate:** Set $S \subseteq V$.
  (B) **Certifier:** Check $|S| \ge k$ and no pair of vertices in S is connected by an edge.

22.2.4 Examples

22.2.4.1 Example: Vertex Cover

(A) **Problem:** Does G have a vertex cover of size $\le k$?

  (A) **Certificate:** $S \subseteq V$.
  (B) **Certifier:** Check $|S| \leq k$ and that for every edge at least one endpoint is in S.

22.2.4.2 Example: SAT

(A) **Problem:** Does formula φ have a satisfying truth assignment?
  (A) **Certificate:** Assignment a of 0/1 values to each variable.
  (B) **Certifier:** Check each clause under a and say "yes" if all clauses are true.

22.2.4.3 Example:Composites

Problem: Composite

Instance: A number s. **Question:** Is the number s a composite?

(A) **Problem: Composite**.
  (A) **Certificate:** A factor $t \leq s$ such that $t \neq 1$ and $t \neq s$.
  (B) **Certifier:** Check that t divides s.

22.2.5 NP

22.2.6 Definition

22.2.6.1 Nondeterministic Polynomial Time

Definition 22.2.6. ***Nondeterministic Polynomial Time*** *(denoted by* **NP***) is the class of all problems that have efficient certifiers.*

Example 22.2.7. **Independent Set**, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, *and* **Composite** *are all examples of problems in* **NP**.

22.2.7 Why is it called...

22.2.7.1 Nondeterministic Polynomial Time

A certifier is an algorithm $C(I, c)$ with two inputs:
(A) I: instance.
(B) c: proof/certificate that the instance is indeed a YES instance of the given problem.

One can think about C as an algorithm for the original problem, if:
(A) Given I, the algorithm guess (non-deterministically, and who knows how) the certificate c.
(B) The algorithm now verifies the certificate c for the instance I.

Usually **NP** is described using Turing machines (gag).

22.2.7.2 Asymmetry in Definition of NP

Note that only YES instances have a short proof/certificate. NO instances need not have a short certificate.

Example 22.2.8. **SAT** *formula φ. No easy way to prove that φ is NOT satisfiable!*

More on this and **co-NP** later on.

22.2.8 Intractability

22.2.8.1 P versus NP

Proposition 22.2.9. $\mathbf{P} \subseteq \mathbf{NP}$.

For a problem in **P** no need for a certificate!

Proof: Consider problem $X \in \mathbf{P}$ with algorithm A. Need to demonstrate that X has an efficient certifier:

(A) Certifier C on input s, t, runs $A(s)$ and returns the answer.
(B) C runs in polynomial time.
(C) If $s \in X$, then for every t, $C(s,t) =$ "yes".
(D) If $s \notin X$, then for every t, $C(s,t) =$ "no".

■

22.2.8.2 Exponential Time

Definition 22.2.10. **Exponential Time** *(denoted* **EXP***) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e.,* $O(2^{\mathrm{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O(2^{n^3})$, ...

22.2.8.3 NP versus EXP

Proposition 22.2.11. $\mathbf{NP} \subseteq \mathbf{EXP}$.

Proof: Let $X \in \mathbf{NP}$ with certifier C. Need to design an exponential time algorithm for X.

(A) For every t, with $|t| \leq p(|s|)$ run $C(s,t)$; answer "yes" if any one of these calls returns "yes".
(B) The above algorithm correctly solves X (exercise).
(C) Algorithm runs in $O(q(|s| + |p(s)|)2^{p(|s|)})$, where q is the running time of C.

■

22.2.8.4 Examples

(A) **SAT**: try all possible truth assignment to variables.
(B) **Independent Set**: try all possible subsets of vertices.
(C) **Vertex Cover**: try all possible subsets of vertices.

22.2.8.5 Is NP efficiently solvable?

We know $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Big Question Is there are problem in **NP** that **does not** belong to **P**? Is $\mathbf{P} = \mathbf{NP}$?

22.2.9 If $P = NP \ldots$

22.2.9.1 Or: If pigs could fly then life would be sweet.

(A) Many important optimization problems can be solved efficiently.
(B) The RSA cryptosystem can be broken.
(C) No security on the web.
(D) No e-commerce ...
(E) Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

22.2.9.2 P versus NP

Status Relationship between **P** and **NP** remains one of the most important open problems in mathematics/computer science.

Consensus: Most people feel/believe $P \neq NP$.

Resolving **P** versus **NP** is a Clay Millennium Prize Problem. You can win a million dollars in addition to a Turing award and major fame!

22.3 Not for lecture: Converting any boolean formula into CNF

22.3.0.3 The dark art of formula conversion into CNF

Consider an arbitrary boolean formula ϕ defined over k variables. To keep the discussion concrete, consider the formula $\phi \equiv x_k = x_i \wedge x_j$. We would like to convert this formula into an equivalent CNF formula.

22.3.1 Formula conversion into CNF

22.3.1.1 Step 1

Build a truth table for the boolean formula.

x_k	x_i	x_j	value of $x_k = x_i \wedge x_j$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

22.3.2 Formula conversion into CNF

22.3.2.1 Step 1.5 - understand what a single CNF clause represents

Given an assignment, say, $x_k = 1$, $k_i = 1$ and $k_j = 0$, consider the CNF clause $x_k \vee x_i \vee \overline{x_j}$ (you negate a variable if it is assigned zero). Its truth table is

x_k	x_i	x_j	$x_k \vee x_i \vee \overline{x_j}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Observe that a single clause assigns zero to one row, and one everywhere else. An conjunction of several such clauses, as such, would result in a formula that is 0 in all the rows that corresponds to these clauses, and one everywhere else.

22.3.3 Formula conversion into CNF

22.3.3.1 Step 2

Write down the CNF clause for every row in the table that is zero.

x_k	x_i	x_j	$x_k = x_i \wedge x_j$	CNF clause
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	0	$x_k \vee \overline{x_i} \vee \overline{x_j}$
1	0	0	0	$\overline{x_k} \vee x_i \vee x_j$
1	0	1	0	$\overline{x_k} \vee x_i \vee \overline{x_j}$
1	1	0	0	$\overline{x_k} \vee \overline{x_i} \vee x_j$
1	1	1	1	

The conjunction (i.e., and) of all these clauses is clearly equivalent to the original formula. In this case $\psi \equiv \left(x_k \vee \overline{x_i} \vee \overline{x_j}\right) \wedge \left(\overline{x_k} \vee x_i \vee x_j\right) \wedge \left(\overline{x_k} \vee x_i \vee \overline{x_j}\right) \wedge \left(\overline{x_k} \vee \overline{x_i} \vee x_j\right)$

Chapter 23

NP Completeness and Cook-Levin Theorem

OLD CS 473: Fundamental Algorithms, Spring 2015
April 21, 2015

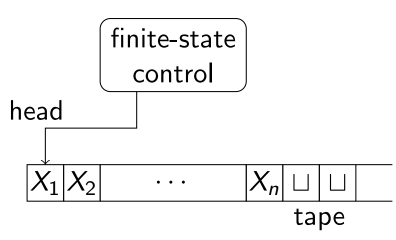
23.0.4 NP

23.0.4.1 P and NP and Turing Machines

(A) Polynomial vs. polynomial time verifiable...
  (A) **P**: set of decision problems that have polynomial time algorithms.
  (B) **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
(B) **Question:** What is an algorithm? Depends on the model of computation!
(C) What is our model of computation?
(D) Formally speaking our model of computation is Turing Machines.

23.0.5 Turing machines

23.0.5.1 Turing Machines: Recap



(A) Infinite tape.
(B) Finite state control.
(C) Input at beginning of tape.
(D) Special tape letter "blank" $\sqcup$.
(E) Head can move only one cell to left or right.

23.0.5.2 Turing Machines: Formally

(A) A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
   (A) Q is set of states in finite control
   (B) q_0 start state, q_{accept} is accept state, q_{reject} is reject state
   (C) Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
   (D) $\delta : Q \times \Gamma \to \{L, R\} \times \Gamma \times Q$ is transition function
      (A) $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
(B) $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
   (A) TM is started in state q_0.
   (B) Initially, the tape head is located at the first cell.
   (C) The tape contain s on the tape followed by blanks.
   (D) The TM halts in the state q_{accept}.

23.0.5.3 P via TMs

(A) Polynomial time Turing machine.

   Definition 23.0.1. *M is a polynomial time TM if there is some polynomial $p(\cdot)$ such that on all inputs w, M halts in $p(|w|)$ steps.*

(B) Polynomial time language.

   Definition 23.0.2. *L is a language in **P** iff there is a polynomial time TM M such that $L = L(M)$.*

23.0.5.4 NP via TMs

(A) **NP** language...

   Definition 23.0.3. *L is an **NP** language iff there is a* non-deterministic *polynomial time TM M such that $L = L(M)$.*

(B) ***Non-deterministic TM***: each step has a choice of moves
   (A) $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
      (A) Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that M can non-deterministically choose one of the three possible moves from (q, a).
   (B) $L(M)$: set of all strings s on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

23.0.5.5 Non-deterministic TMs vs certifiers

(A) Two definition of **NP**:
   (A) L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   (B) L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
(B) Equivalence...

   Claim 23.0.4. *Two definitions are equivalent.*

(C) Why?

(D) Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
(E) In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

23.0.5.6 Non-determinism, guessing and verification

(A) A non-deterministic machine has choices at each step and accepts a string if there *exists* a set of choices which lead to a final state.
(B) Equivalently the choices can be thought of as *guessing* a solution and then *verifying* that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The "guess" is the "proof" and the "verifier" is the "certifier".
(C) Note: Symmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.

23.0.5.7 Algorithms: TMs vs RAM Model

(A) Why do we use TMs some times and RAM Model other times?
(B) TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
    (A) Simplicity is useful in proofs.
    (B) The "right" formal bare-bones model when dealing with subtleties.
(C) RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
    (A) Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space

23.0.6 Cook-Levin Theorem

23.0.7 Completeness

23.0.7.1 "Hardest" Problems

(A) Question What is the hardest problem in **NP**? How do we define it?
(B) Towards a definition
    (A) Hardest problem must be in **NP**.
    (B) Hardest problem must be at least as "difficult" as every other problem in **NP**.

23.0.7.2 NP-Complete Problems

Definition 23.0.5. *A problem X is said to be* **NP-Complete** *if*
(A) $X \in$ **NP**, *and*
(B) ***(Hardness)*** *For any $Y \in$* **NP**, *$Y \leq_P X$.*

23.0.7.3 Solving NP-Complete Problems

Proposition 23.0.6. *Suppose X is* **NP-Complete**. *Then X can be solved in polynomial time if and only if* **P** = **NP**.

Proof:

⇒ Suppose X can be solved in polynomial time

(A) Let $Y \in$ **NP**. We know $\mathsf{Y} \leq_P \mathsf{X}$.

(B) We showed that if $\mathsf{Y} \leq_P \mathsf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.

(C) Thus, every problem $Y \in$ **NP** is such that $Y \in P$; $NP \subseteq P$.

(D) Since **P** $\subseteq$ **NP**, we have **P** $=$ **NP**.

⇐ Since **P** $=$ **NP**, and $X \in$ **NP**, we have a polynomial time algorithm for X.

■

23.0.7.4 NP-Hard Problems

(A) **NP-Hard** problems:

Definition 23.0.7. *A problem X is said to be* **NP-Hard** *if*

(A) ***(Hardness)*** *For any $Y \in$* **NP***, we have that $\mathsf{Y} \leq_P \mathsf{X}$.*

(B) An **NP-Hard** problem need not be in **NP**!

(C) **Example:** Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

23.0.7.5 Consequences of proving NP-Completeness

(A) If X is **NP-Complete**

(A) Since we believe **P** $\neq$ **NP**,

(B) and solving X implies **P** $=$ **NP**.

(B) $\implies$ X is **unlikely** to be efficiently solvable.

(C) $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.

(D) (This is proof by mob opinion — take with a grain of salt.)

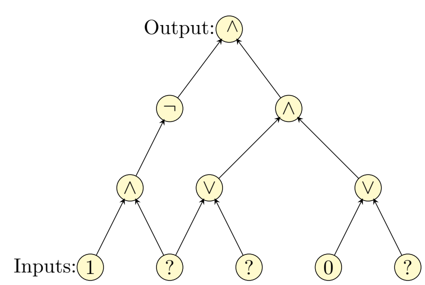
23.0.8 Preliminaries

23.0.8.1 NP-Complete Problems

Question Are there any problems that are **NP-Complete**? Answer Yes! Many, many problems are **NP-Complete**.

23.0.8.2 Circuits

Definition 23.0.8. *A circuit is a directed* acyclic *graph with*



(A)

(B)

(C)

23.0.9 Cook-Levin Theorem

23.0.9.1 Cook-Levin Theorem

Definition 23.0.9 (Circuit Satisfaction (CSAT).). *Given a circuit as input, is there an assignment to the input variables that causes the output to get value 1?*

Theorem 23.0.10 (Cook-Levin). **CSAT** *is* **NP-Complete**.

Need to show

(A) **CSAT** is in **NP**.
(B) *every* **NP** problem X reduces to **CSAT**.

23.0.9.2 CSAT: Circuit Satisfaction

Claim 23.0.11. **CSAT** *is in* **NP**.

(A) **Certificate:** Assignment to input variables.
(B) **Certifier:** Evaluate the value of each gate in a topological sort of DAG and check the output gate value.

23.0.9.3 CSAT is NP-hard: Idea

(A) Need to show that *every* **NP** problem X reduces to **CSAT**.
(B) What does it mean that $X \in$ **NP**?
(C) $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
  (A) If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s,t)$ says YES.
  (B) If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s,t)$ says NO.
  (C) $C(s,t)$ runs in time $q(|s|+|t|)$ time (hence polynomial time).

23.0.9.4 Reducing X to CSAT

(A) X is in **NP** means we have access to $p(), q(), C(\cdot,\cdot)$.
(B) What is $C(\cdot,\cdot)$? It is a program or equivalently a Turing Machine!
(C) How are $p()$ and $q()$ given? As numbers (coefficients and powers).
(D) Example: if 3 is given then $p(n) = n^3$.
(E) Thus an **NP** problem is essentially a three tuple $\langle p, q, C\rangle$ where C is either a program or a TM.

23.0.9.5 Reducing X to CSAT

(A) Thus an **NP** problem is essentially a three tuple $\langle p, q, C\rangle$ where C is either a program or TM.
(B) **Problem X:** Given string s, is $s \in X$?
(C) Same as the following: is there a proof t of length $p(|s|)$ such that $C(s,t)$ says YES.
(D) How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
  (A) takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
  (B) G is satisfiable if and only if there is a proof t such that $C(s,t)$ says YES.

23.0.9.6 Reducing X to CSAT

(A) How do we reduce X to **CSAT**?
(B) Need an algorithm $\mathcal{A}$ that
   (A) takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
   (B) G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
(C) **Simple but Big Idea:** Programs are essentially the same as Circuits!
   (A) Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   (B) We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   (C) Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

23.0.9.7 Example: Independent Set

(A) **Problem:** Does $G = (V, E)$ have an **Independent Set** of size $\geq k$?
   (A) **Certificate:** Set $S \subseteq V$.
   (B) **Certifier:** Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.
(B) Formally, why is **Independent Set** in **NP**?

23.0.9.8 Example: Independent Set

Formally why is **Independent Set** in **NP**?

(A) Input: $< n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
   (A) n is number of vertices in G
   (B) $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
   (C) k is size of independent set.
(B) Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

23.0.9.9 Certifier for Independent Set

Certifier $C(s, t)$ for **Independent Set**:

```
if (t_1 + t_2 + ... + t_n < k) then
    return NO
else
    for each (i,j) do
        if (t_i ∧ t_j ∧ y_{i,j}) then
            return NO

return YES
```

23.0.10 Example: Independent Set

23.0.10.1 A certifier circuit for Independent Set

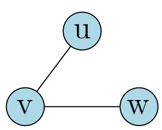
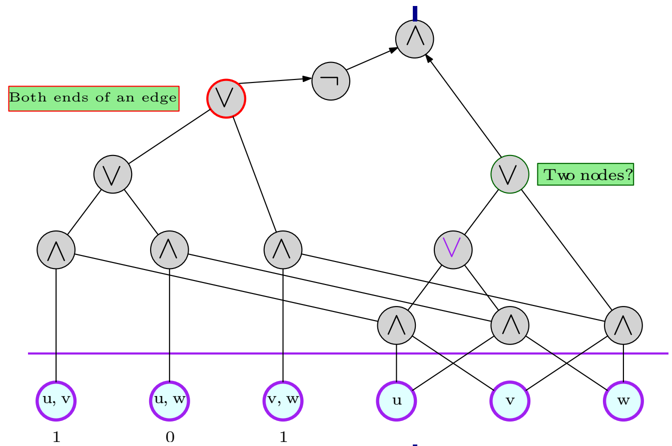
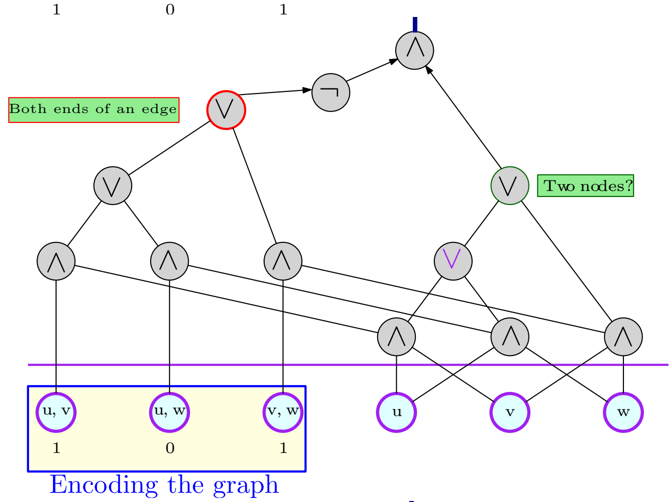
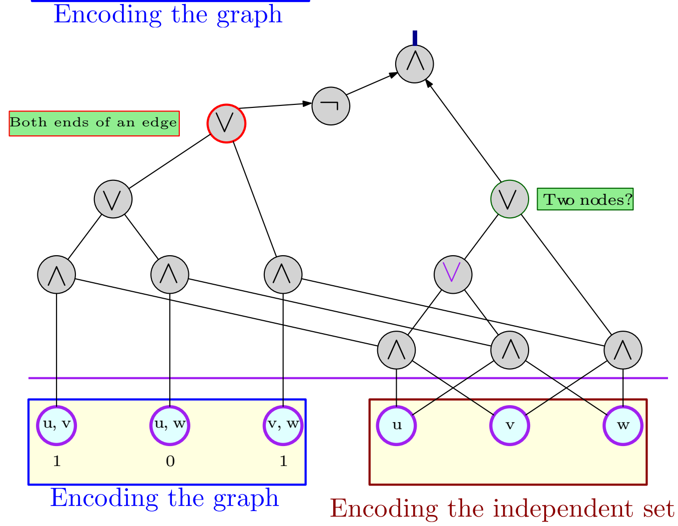


Figure 23.1: Graph G with $k = 2$







23.0.10.2 Programs, Turing Machines and Circuits

(A) Consider "program" A that takes $f(|s|)$ steps on input string s.

(B) **Question:** What computer is the program running on and what does *step* mean?
(C) Real computers difficult to reason with mathematically because
   (A) instruction set is too rich
   (B) pointers and control flow jumps in one step
   (C) assumption that pointer to code fits in one word
(D) Turing Machines
   (A) simpler model of computation to reason with
   (B) can simulate real computers with *polynomial* slow down
   (C) all moves are *local* (head moves only one cell)

23.0.10.3 Certifiers that at TMs

(A) Assume $C(\cdot,\cdot)$ is a (deterministic) Turing Machine M
(B) **Problem:** Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
(C) There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
   (A) $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
   (B) Knows that M can use at most $q(|s|)$ memory/tape cells
   (C) Knows that M can run for at most $q(|s|)$ time
   (D) Simulates the evolution of the state of M and memory over time using a big circuit.

23.0.10.4 Simulation of Computation via Circuit

(A) Think of M's state at time ℓ as a string $x^\ell = x_1 x_2 \dots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
(B) At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
(C) At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
(D) We write a circuit C_ℓ which captures the transition of M from time ℓ to time $\ell + 1$.
(E) Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
(F) The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

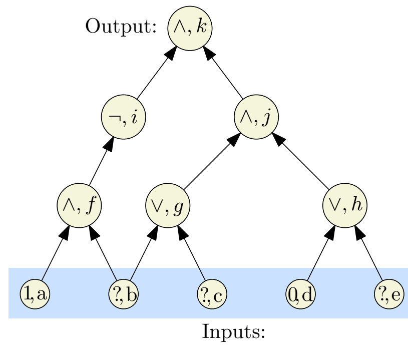
23.0.10.5 NP-Hardness of Circuit Satisfaction

(A) Key Ideas in reduction:
   (A) Use TMs as the code for certifier for simplicity
   (B) Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
   (C) Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time
(B) **Note:** Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

23.0.11 Other NP Complete Problems

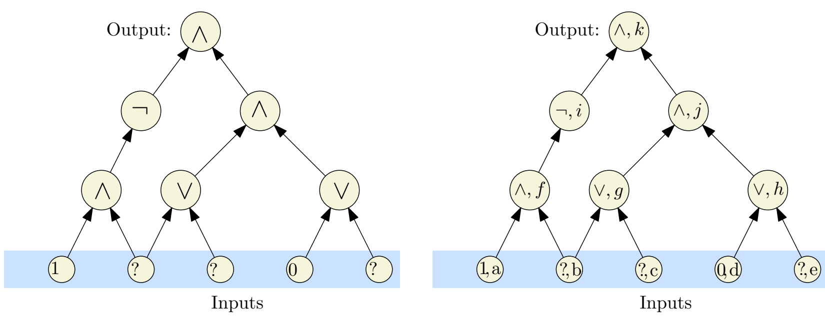
23.0.11.1 SAT is NP-Complete

(A) We have seen that **SAT** $\in$ NP
(B) To show **NP-Hardness**, we will reduce Circuit Satisfiability (**CSAT**) to **SAT**
   Instance of **CSAT** (we label each node):



23.0.12 Converting a circuit into a CNF formula

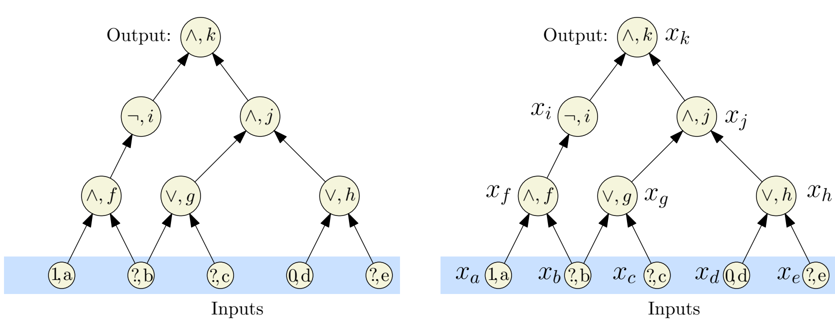
23.0.12.1 Label the nodes



(A) Input circuit (B) Label the nodes.

23.0.13 Converting a circuit into a CNF formula

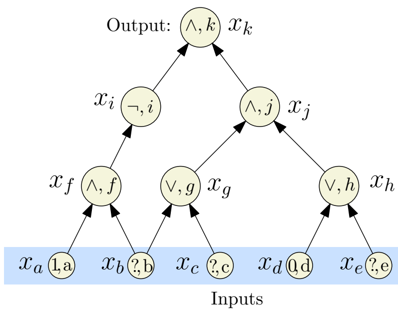
23.0.13.1 Introduce a variable for each node



(B) Label the nodes. (C) Introduce var for each node.

23.0.14 Converting a circuit into a CNF formula

23.0.14.1 Write a sub-formula for each variable that is true if the var is computed correctly.



(C) Introduce var for each node.

x_k (Demand a sat' assignment!)
$x_k = x_i \wedge x_k$
$x_j = x_g \wedge x_h$
$x_i = \neg x_f$
$x_h = x_d \vee x_e$
$x_g = x_b \vee x_c$
$x_f = x_a \wedge x_b$
$x_d = 0$
$x_a = 1$

(D) Write a sub-formula for each variable that is true if the var is computed correctly.

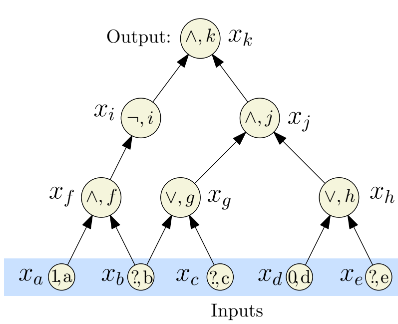
23.0.15 Converting a circuit into a CNF formula

23.0.15.1 Convert each sub-formula to an equivalent CNF formula

x_k	x_k
$x_k = x_i \wedge x_j$	$(\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j) \wedge (x_k \vee \neg x_i \vee \neg x_j)$
$x_j = x_g \wedge x_h$	$(\neg x_j \vee x_g) \wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h)$
$x_i = \neg x_f$	$(x_i \vee x_f) \wedge (\neg x_i \vee x_f)$
$x_h = x_d \vee x_e$	$(x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e) \wedge (\neg x_h \vee x_d \vee x_e)$
$x_g = x_b \vee x_c$	$(x_g \vee \neg x_b) \wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c)$
$x_f = x_a \wedge x_b$	$(\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b) \wedge (x_f \vee \neg x_a \vee \neg x_b)$
$x_d = 0$	$\neg x_d$
$x_a = 1$	x_a

23.0.16 Converting a circuit into a CNF formula

23.0.16.1 Take the conjunction of all the CNF sub-formulas



$$
\begin{aligned}
& x_k \wedge (\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j) \\
& \wedge (x_k \vee \neg x_i \vee \neg x_j) \wedge (\neg x_j \vee x_g) \\
& \wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h) \\
& \wedge (x_i \vee x_f) \wedge (\neg x_i \vee x_f) \\
& \wedge (x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e) \\
& \wedge (\neg x_h \vee x_d \vee x_e) \wedge (x_g \vee \neg x_b) \\
& \wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c) \\
& \wedge (\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b) \\
& \wedge (x_f \vee \neg x_a \vee \neg x_b) \wedge ([)] \neg x_d \wedge x_a
\end{aligned}
$$

We got a CNF formula that is satisfiable if and only if the original circuit is satisfiable.

23.0.16.2 Reduction: **CSAT** $\leq_P$ **SAT**

(A) For each gate (vertex) v in the circuit, create a variable x_v

(B) **Case $\neg$:** v is labeled $\neg$ and has one incoming edge from u (so $x_v = \neg x_u$). In **SAT** formula generate, add clauses $(x_u \vee x_v)$, $(\neg x_u \vee \neg x_v)$. Observe that

$$x_v = \neg x_u \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} (x_u \vee x_v) \\ (\neg x_u \vee \neg x_v) \end{array} \text{ both true.}$$

23.0.17 Reduction: **CSAT** $\leq_P$ **SAT**

23.0.17.1 Continued...

(A) **Case $\vee$:** So $x_v = x_u \vee x_w$. In **SAT** formula generated, add clauses $(x_v \vee \neg x_u)$, $(x_v \vee \neg x_w)$, and $(\neg x_v \vee x_u \vee x_w)$. Again, observe that

$$\Big(x_v = x_u \vee x_w\Big) \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} (x_v \vee \neg x_u), \\ (x_v \vee \neg x_w), \\ (\neg x_v \vee x_u \vee x_w) \end{array} \quad \text{all true.}$$

23.0.18 Reduction: **CSAT** $\leq_P$ **SAT**

23.0.18.1 Continued...

(A) **Case $\wedge$:** So $x_v = x_u \wedge x_w$. In **SAT** formula generated, add clauses $(\neg x_v \vee x_u)$, $(\neg x_v \vee x_w)$, and $(x_v \vee \neg x_u \vee \neg x_w)$. Again observe that

$$x_v = x_u \wedge x_w \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} (\neg x_v \vee x_u), \\ (\neg x_v \vee x_w), \\ (x_v \vee \neg x_u \vee \neg x_w) \end{array} \quad \text{all true.}$$

23.0.19 Reduction: **CSAT** $\leq_P$ **SAT**

23.0.19.1 Continued...

(A) If v is an input gate with a fixed value then we do the following. If $x_v = 1$ add clause x_v. If $x_v = 0$ add clause $\neg x_v$

(B) Add the clause x_v where v is the variable for the output gate

23.0.19.2 Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

- $\Rightarrow$ Consider a satisfying assignment a for C
  - (A) Find values of all gates in C under a
  - (B) Give value of gate v to variable x_v; call this assignment a'
  - (C) a' satisfies φ_C (exercise)
- $\Leftarrow$ Consider a satisfying assignment a for φ_C
  - (A) Let a' be the restriction of a to only the input variables
  - (B) Value of gate v under a' is the same as value of x_v in a
  - (C) Thus, a' satisfies C

23.0.19.3 Showed that...

Theorem 23.0.12. **SAT** *is* **NP-Complete***.*

23.0.19.4 Proving that a problem X is NP-Complete

- (A) To prove X is **NP-Complete**, show
  - (A) Show X is in **NP**.
    - (A) certificate/proof of polynomial size in input
    - (B) polynomial time certifier $C(s,t)$
  - (B) Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
- (B) SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
- (C) Transitivity of reductions:
- (D) $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

23.0.19.5 NP-Completeness via Reductions

- (A) What we know so far:
  - (A) **CSAT** is **NP-Complete**.
  - (B) **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
  - (C) **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
  - (D) **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
  - (E) **Vertex Cover** is **NP-Complete**.
  - (F) **Clique** is **NP-Complete**.
- (B) Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
- (C) A surprisingly frequent phenomenon!

Chapter 24

More NP-Complete Problems

OLD CS 473: Fundamental Algorithms, Spring 2015
April 23, 2015

24.0.19.6 Recap

(A) **NP**: languages that have polynomial time certifiers/verifiers
(B) A language L is **NP-Complete** iff
  - L is in **NP**
  - for every L' in **NP**, $L' \leq_P L$

(C) L is **NP-Hard** if for every L' in **NP**, $L' \leq_P L$.
(D) Cook-Levin theorem...

Theorem 24.0.13 (Cook-Levin). **Circuit-SAT** *and* **SAT** *are* **NP-Complete**.

24.0.19.7 Recap contd

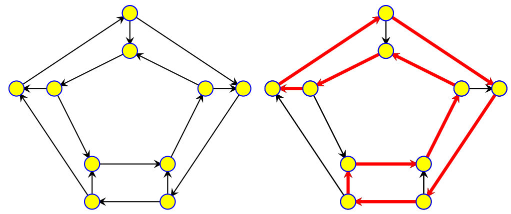
Theorem 24.0.14 (Cook-Levin). **Circuit-SAT** *and* **SAT** *are* **NP-Complete**.

(A) Establish **NP-Complete**ness via reductions:
(B) **SAT** $\leq_P$ **3SAT** and hence **3SAT** is **NP**-complete
(C) **3SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence...
(D) **Independent Set** is **NP-Complete**.
(E) **Vertex Cover** is **NP-Complete**
(F) **Clique** is **NP-Complete**.
(G) **Set Cover** is **NP-Complete**.

24.0.19.8 Today

Prove

- **Hamiltonian Cycle** Problem is **NP-Complete**
- **3-Coloring** is **NP-Complete**



24.0.20 NP-Completeness of Hamiltonian Cycle

24.0.21 Reduction from 3SAT to Hamiltonian Cycle

24.0.21.1 Directed Hamiltonian Cycle

Input Given a directed graph $G = (V, E)$ with n vertices

Goal Does G have a **Hamiltonian cycle**?

- A Hamiltonian cycle is a cycle in the graph that visits every vertex in G exactly once

24.0.21.2 Directed Hamiltonian Cycle is NP-Complete

- Directed Hamiltonian Cycle is in NP
  - **Certificate:** Sequence of vertices
  - **Certifier:** Check if every vertex (except the first) appears exactly once, and that consecutive vertices are connected by a directed edge
- **Hardness:** We will show 3-SAT $\leq_P$ Directed Hamiltonian Cycle

24.0.21.3 Reduction

(A) Given **3SAT** formula φ create a graph G_φ such that

- G_φ has a Hamiltonian cycle if and only if φ is satisfiable
- G_φ should be constructible from φ by a polynomial time algorithm $\mathcal{A}$

(B) **Notation:** φ has n variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$.

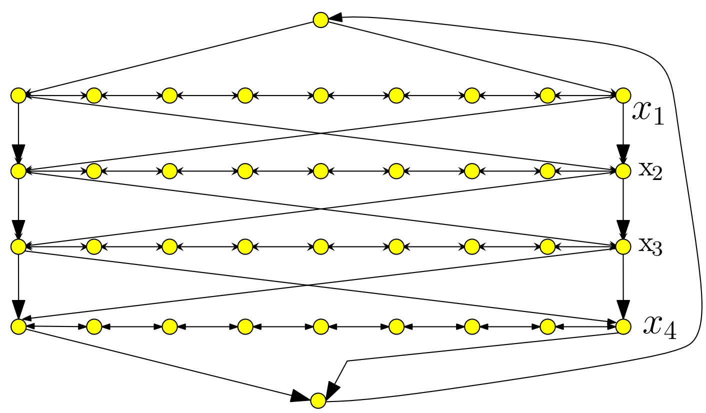
24.0.21.4 Reduction: First Ideas

(A) Viewing **SAT**: Assign values to n variables, and each clauses has 3 ways in which it can be satisfied.
(B) Construct graph with 2^n Hamiltonian cycles, where each cycle corresponds to some boolean assignment.
(C) Then add more graph structure to encode constraints on assignments imposed by the clauses.

24.0.21.5 The Reduction: Phase I

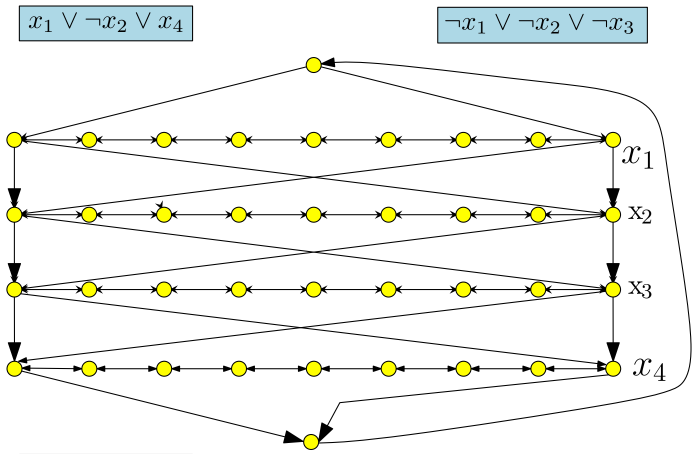
- Traverse path i from left to right iff x_i is set to true

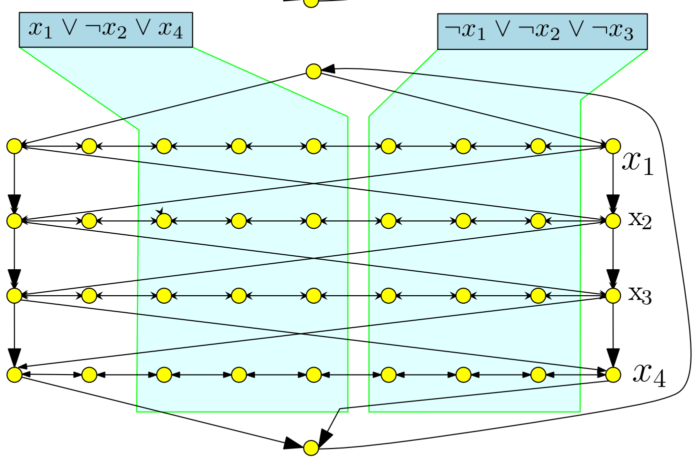
- Each path has $3(m+1)$ nodes where m is number of clauses in φ; nodes numbered from left to right (1 to $3m+3$)

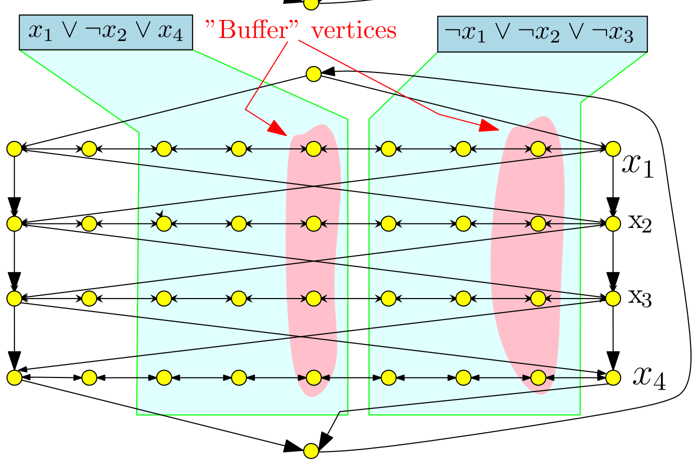


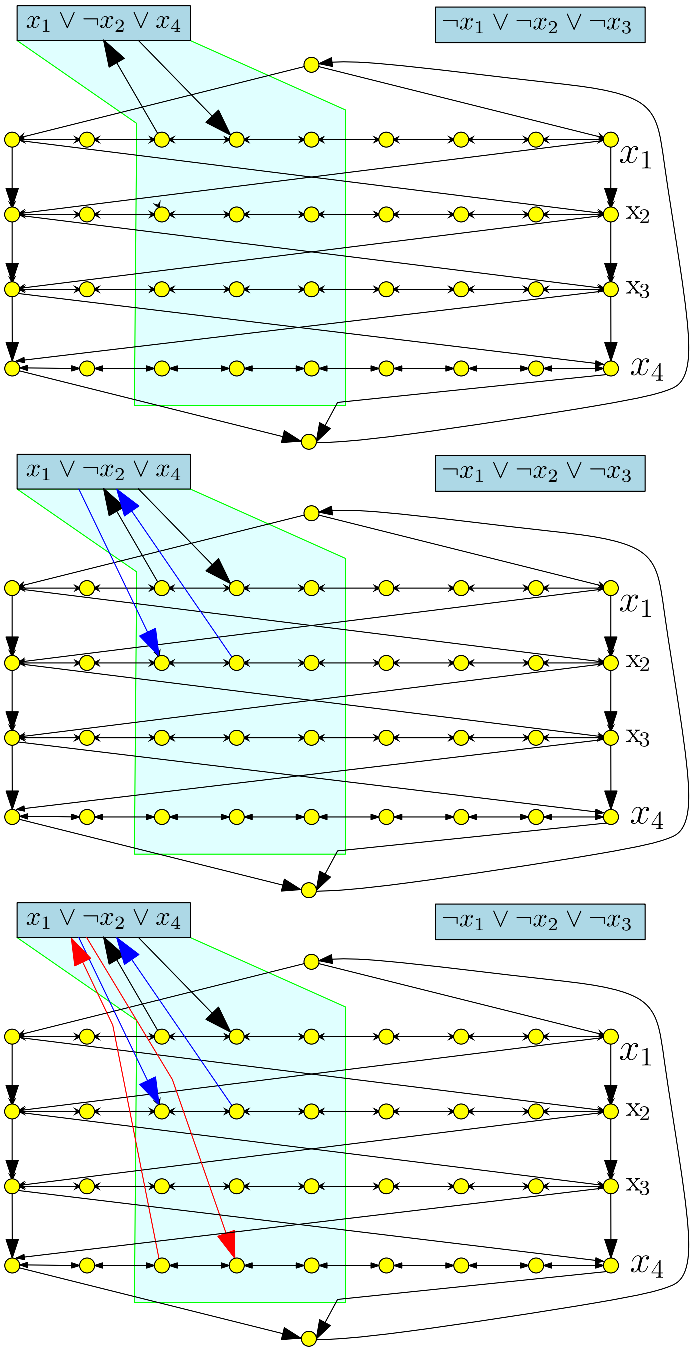
24.0.21.6 The Reduction: Phase II

- Add vertex c_j for clause C_j. c_j has edge *from* vertex $3j$ and *to* vertex $3j+1$ on path i if x_i appears in clause C_j, and has edge *from* vertex $3j+1$ and *to* vertex $3j$ if $\neg x_i$ appears in C_j.

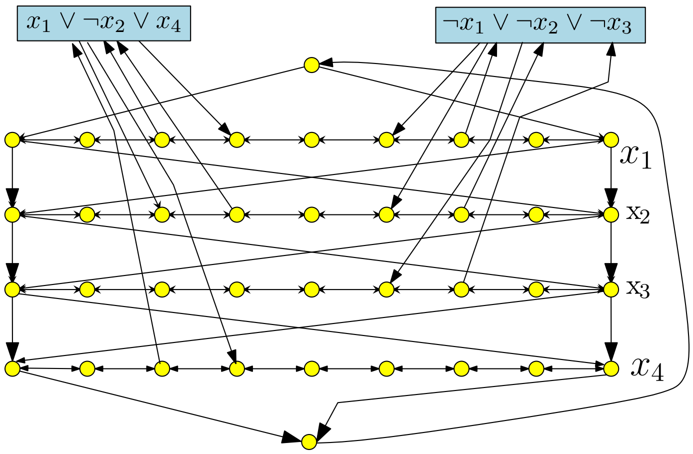

$x_1 \vee \neg x_2 \vee x_4$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x$_2$
x$_3$
x_4


$x_1 \vee \neg x_2 \vee x_4$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x$_2$
x$_3$
x_4


$x_1 \vee \neg x_2 \vee x_4$
"Buffer" vertices
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x$_2$
x$_3$
x_4



$x_1 \vee \neg x_2 \vee x_4$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x2
x3
x_4
$x_1 \vee \neg x_2 \vee x_4$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x2
x3
x_4
$x_1 \vee \neg x_2 \vee x_4$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x2
x3
x_4



24.0.21.7 Correctness Proof

Proposition 24.0.15. *φ has a satisfying assignment iff G_φ has a Hamiltonian cycle.*

Proof:

$\Rightarrow$ Let a be the satisfying assignment for φ. Define Hamiltonian cycle as follows

- If $a(x_i) = 1$ then traverse path i from left to right
- If $a(x_i) = 0$ then traverse path i from right to left
- For each clause, path of at least one variable is in the "right" direction to splice in the node corresponding to clause

■

24.0.21.8 Hamiltonian Cycle $\Rightarrow$ Satisfying assignment

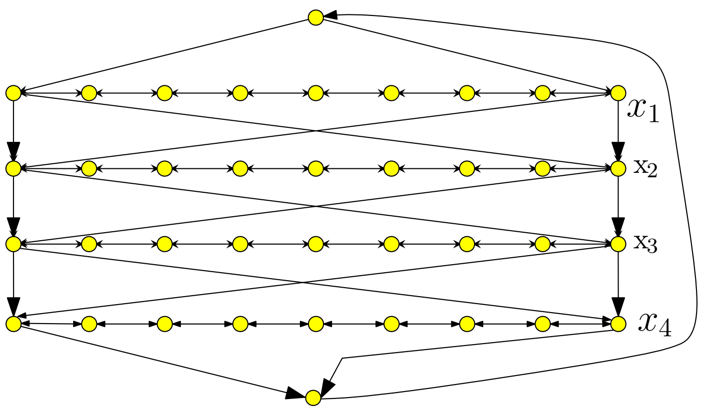
Suppose Π is a Hamiltonian cycle in G_φ

- If Π enters c_j (vertex for clause C_j) from vertex $3j$ on path i then it must leave the clause vertex on edge to $3j+1$ on the *same path* i
  - If not, then only unvisited neighbor of $3j+1$ on path i is $3j+2$
  - Thus, we don't have two unvisited neighbors (one to enter from, and the other to leave) to have a Hamiltonian Cycle
- Similarly, if Π enters c_j from vertex $3j+1$ on path i then it must leave the clause vertex c_j on edge to $3j$ on path i

24.0.21.9 Example

24.0.21.10 Hamiltonian Cycle $\implies$ Satisfying assignment (contd)

- Thus, vertices visited immediately before and after C_i are connected by an edge
- We can remove c_j from cycle, and get Hamiltonian cycle in $G - c_j$
- Consider Hamiltonian cycle in $G - \{c_1, \ldots c_m\}$; it traverses each path in only one direction, which determines the truth assignment



24.0.22 Hamiltonian cycle in undirected graph

24.0.22.1 Hamiltonian Cycle

Problem 24.0.16. Input *Given **undirected** graph $G = (V, E)$*

Goal *Does G have a Hamiltonian cycle? That is, is there a cycle that visits every vertex exactly one (except start and end vertex)?*

24.0.22.2 NP-Completeness

Theorem 24.0.17. **Hamiltonian cycle** *problem for* undirected *graphs is* **NP-Complete**.

Proof:

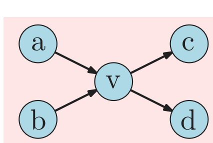
- The problem is in **NP**; proof left as exercise.
- Hardness proved by reducing Directed Hamiltonian Cycle to this problem

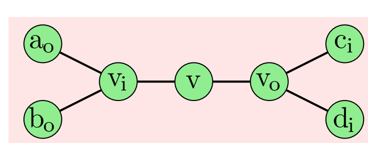
∎

24.0.22.3 Reduction Sketch

Goal: Given directed graph G, need to construct undirected graph G' such that G has Hamiltonian Path iff G' has Hamiltonian path Reduction

- Replace each vertex v by 3 vertices: v_{in}, v, and v_{out}
- A directed edge (a, b) is replaced by edge (a_{out}, b_{in})





24.0.22.4 Reduction: Wrapup

- The reduction is polynomial time (exercise)
- The reduction is correct (exercise)

24.0.23 NP-Completeness of Graph Coloring

24.0.23.1 Graph Coloring

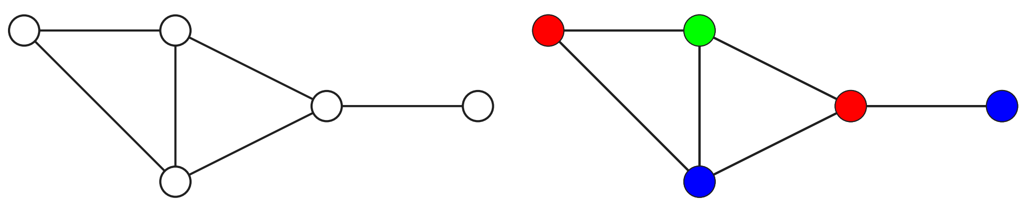
Problem: Graph Coloring

> **Instance:** $G = (V, E)$: Undirected graph, integer k.
> **Question:** Can the vertices of the graph be colored using k colors so that vertices connected by an edge do not get the same color?

24.0.23.2 Graph 3-Coloring

Problem: 3 Coloring

> **Instance:** $G = (V, E)$: Undirected graph.
> **Question:** Can the vertices of the graph be colored using 3 colors so that vertices connected by an edge do not get the same color?



24.0.23.3 Graph Coloring

(A) **Observation:** If G is colored with k colors then each color class (nodes of same color) form an independent set in G.
(B) G can be partitioned into k independent sets iff G is k-colorable.
(C) Graph 2-Coloring can be decided in polynomial time.
(D) G is 2-colorable iff G is bipartite!
(E) There is a linear time algorithm to check if G is bipartite using **BFS** (we saw this earlier).

24.0.24 Problems related to graph coloring

24.0.24.1 Graph Coloring and Register Allocation

Register Allocation Assign variables to (at most) k registers such that variables needed at the same time are not assigned to the same register Interference Graph Vertices are variables, and there is an edge between two vertices, if the two variables are "live" at the same time. Observations

- **[Chaitin]** Register allocation problem is equivalent to coloring the interference graph with k colors
- Moreover, 3-COLOR $\leq_P$ k-Register Allocation, for any $k \geq 3$

24.0.24.2 Class Room Scheduling

(A) Given n classes and their meeting times, are k rooms sufficient?
(B) Reduce to Graph k-Coloring problem
(C) Create graph G

- a node v_i for each class i
- an edge between v_i and v_j if classes i and j *conflict*

(D) Exercise: G is k-colorable iff k rooms are sufficient.

24.0.24.3 Frequency Assignments in Cellular Networks

(A) Cellular telephone systems that use Frequency Division Multiple Access (FDMA) (example: GSM in Europe and Asia and AT&T in USA)

- Breakup a frequency range $[a, b]$ into disjoint *bands* of frequencies $[a_0, b_0], [a_1, b_1], \ldots, [a_k, b_k]$
- Each cell phone tower (simplifying) gets one band
- Constraint: nearby towers cannot be assigned same band, otherwise signals will interference

(B) **Problem:** given k bands and some region with n towers, is there a way to assign the bands to avoid interference?
(C) Can reduce to k-coloring by creating intereference/conflict graph on towers.

24.0.25 Showing hardness of 3 COLORING

24.0.25.1 3-Coloring is NP-Complete

- 3-Coloring is in NP.
  - **Certificate:** for each node a color from $\{1, 2, 3\}$.
  - **Certifier:** Check if for each edge (u, v), the color of u is different from that of v.
- **Hardness:** We will show 3-SAT $\leq_P$ 3-Coloring.

24.0.25.2 Reduction Idea

(A) φ: Given **3SAT** formula (i.e., 3CNF formula).
(B) φ: variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$.
(C) Create graph G_φ s.t. G_φ 3-colorable $\iff$ φ satisfiable.

¡+-¿ encode assignment $x_1, \ldots, x_n$ in colors assigned nodes of G_φ.

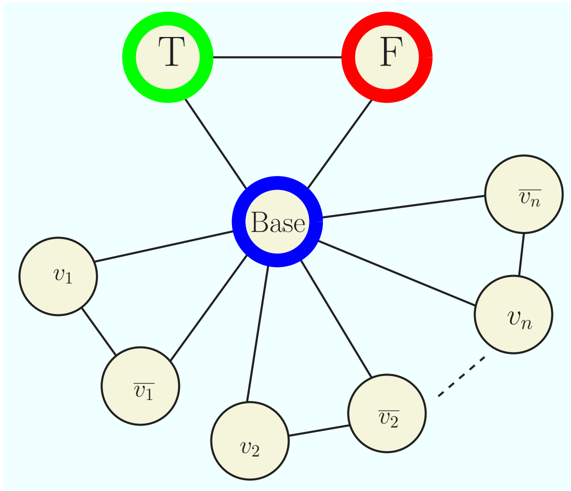
¡+-¿ create triangle with node True, False, Base

¡+-¿ for each variable x_i two nodes v_i and $\bar{v}_i$ connected in a triangle with common Base

¡+-¿ If graph is 3-colored, either v_i or $\bar{v}_i$ gets the same color as True. Interpret this as a truth assignment to v_i

¡+-¿ Need to add constraints to ensure clauses are satisfied (next phase)

24.0.25.3 Figure

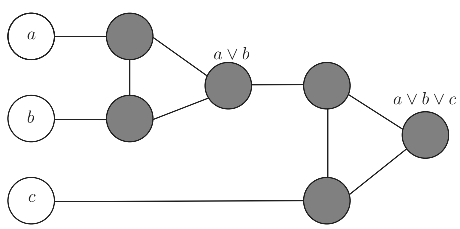


24.0.25.4 Clause Satisfiability Gadget

(A) For each clause $C_j = (a \vee b \vee c)$, create a small gadget graph

- gadget graph connects to nodes corresponding to a, b, c
- needs to implement OR

(B) OR-gadget-graph:

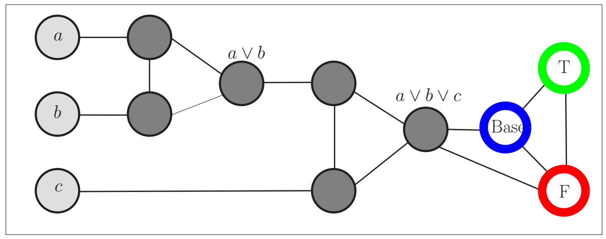


24.0.25.5 OR-Gadget Graph

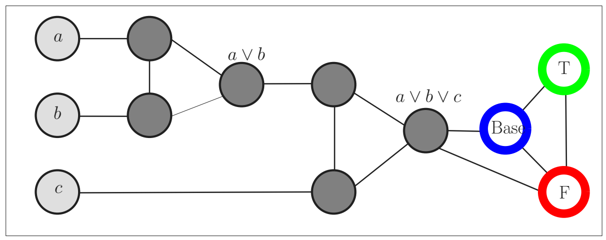
Property: if a, b, c are colored False in a 3-coloring then output node of OR-gadget has to be colored False.
Property: if one of a, b, c is colored True then OR-gadget can be 3-colored such that output node of OR-gadget is colored True.

24.0.25.6 Reduction

- create triangle with nodes True, False, Base
- for each variable x_i two nodes v_i and $\bar{v}_i$ connected in a triangle with common Base
- for each clause $C_j = (a \vee b \vee c)$, add OR-gadget graph with input nodes a, b, c and connect output node of gadget to both False and Base

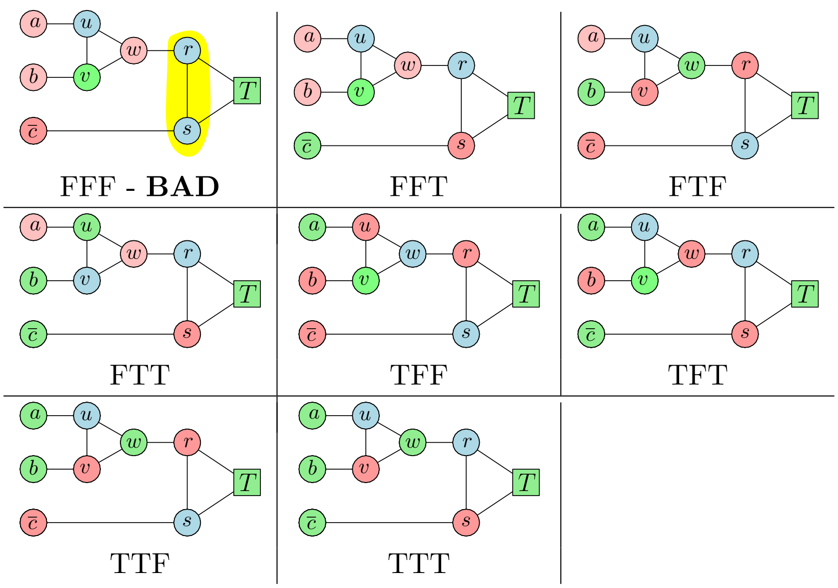


24.0.25.7 Reduction



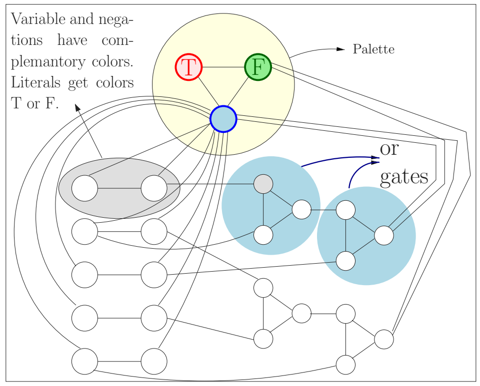
Claim 24.0.18. *No legal* 3*-coloring of above graph (with coloring of nodes* T, F, B *fixed) in which* a, b, c *are colored False. If any of* a, b, c *are colored True then there is a legal* 3*-coloring of above graph.*

24.0.25.8 3 coloring of the clause gadget



24.0.25.9 Reduction Outline

Example 24.0.19. $\varphi = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$



24.0.25.10 Correctness of Reduction

φ is satisfiable implies G_φ is 3-colorable

¡+-¿ if x_i is assigned True, color v_i True and $\bar{v_i}$ False

¡+-¿ for each clause $C_j = (a \vee b \vee c)$ at least one of a, b, c is colored True. OR-gadget for C_j can be 3-colored such that output is True.

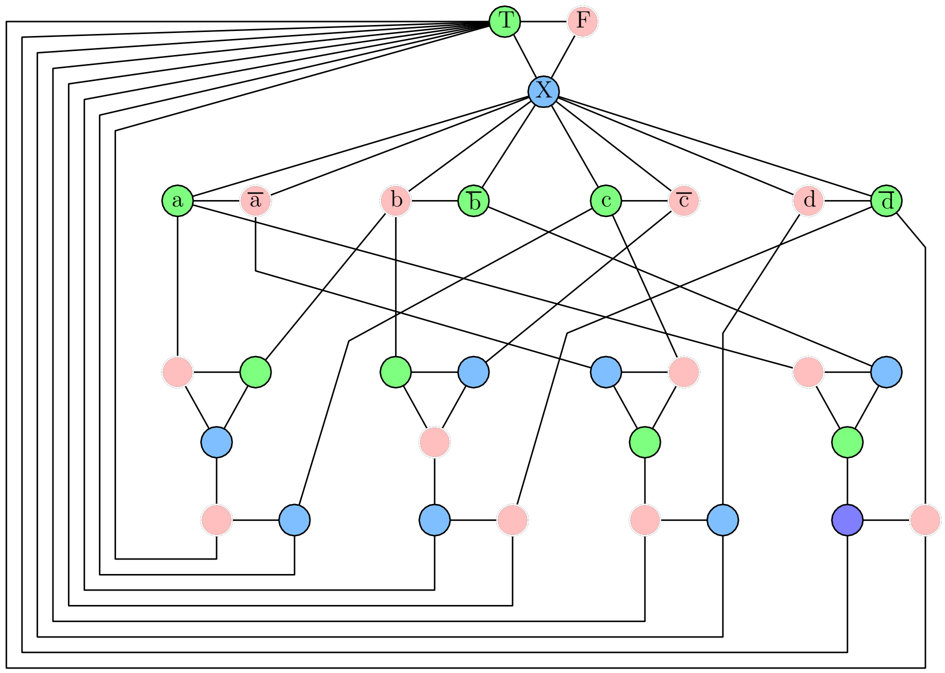
G_φ is 3-colorable implies φ is satisfiable

¡+-¿ if v_i is colored True then set x_i to be True, this is a legal truth assignment

¡+-¿ consider any clause $C_j = (a \vee b \vee c)$. it cannot be that all a, b, c are False. If so, output of OR-gadget for C_j has to be colored False but output is connected to Base and False!

24.0.26 Graph generated in reduction...

24.0.26.1 ... from 3SAT to 3COLOR



24.0.27 Hardness of **Subset Sum**

24.0.27.1 Subset Sum

Problem: Subset Sum

> **Instance:** S - set of positive integers, t: - an integer number (Target)
> **Question:** Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Claim 24.0.20. **Subset Sum** *is* **NP-Complete**.

24.0.27.2 Vec Subset Sum

We will prove following problem is **NP-Complete**...

Problem: Vec Subset Sum

> **Instance:** S - set of n vectors of dimension k, each vector has non-negative numbers for its coordinates, and a target vector $\overrightarrow{t}$.
> **Question:** Is there a subset $X \subseteq S$ such that $\sum_{\overrightarrow{x} \in X} \overrightarrow{x} = \overrightarrow{t}$?

Reduction from **3SAT**.

24.0.28 Vec Subset Sum

24.0.28.1 Handling a single clause

Think about vectors as being lines in a table.

First gadget

Selecting between two lines.

Target	??	??	01	???
a_1	??	??	01	??
a_2	??	??	01	??

Two rows for every variable x: selecting either $x = 0$ or $x = 1$.

24.0.28.2 Handling a clause...

We will have a column for every clause...

numbers	...	$C \equiv a \vee b \vee \overline{c}$	...
a	...	01	...
$\overline{a}$	...	00	...
b	...	01	...
$\overline{b}$	...	00	...
c	...	00	...
$\overline{c}$	...	01	...
C fix-up 1	000	07	000
C fix-up 2	000	08	000
C fix-up 3	000	09	000
TARGET		10	

24.0.28.3 3SAT to Vec Subset Sum

numbers	$a \vee \overline{a}$	$b \vee \overline{b}$	$c \vee \overline{c}$	$d \vee \overline{d}$	$D \equiv \overline{b} \vee c \vee \overline{d}$	$C \equiv a \vee b \vee \overline{c}$
a	1	0	0	0	00	01
$\overline{a}$	1	0	0	0	00	00
b	0	1	0	0	00	01
$\overline{b}$	0	1	0	0	01	00
c	0	0	1	0	01	00
$\overline{c}$	0	0	1	0	00	01
d	0	0	0	1	00	00
$\overline{d}$	0	0	0	1	01	01
C fix-up 1	0	0	0	0	00	07
C fix-up 2	0	0	0	0	00	08
C fix-up 3	0	0	0	0	00	09
D fix-up 1	0	0	0	0	07	00
D fix-up 2	0	0	0	0	08	00
D fix-up 3	0	0	0	0	09	00
TARGET	1	1	1	1	10	10

24.0.28.4 **Vec Subset Sum** to **Subset Sum**

numbers
010000000001
010000000000
000100000001
000100000100
000001000100
000001000001
000000010000
000000010101
000000000007
000000000008
000000000009
000000000700
000000000800
000000000900
010101011010

24.0.28.5 Other NP-Complete Problems

- 3-Dimensional Matching
- Subset Sum

Read book.

24.0.28.6 Need to Know NP-Complete Problems

- **3SAT**.
- **Circuit-SAT**.
- **Independent Set**.
- **Vertex Cover**.
- **Clique**.
- **Set Cover**.
- **Hamiltonian Cycle** (in Directed/Undirected Graphs).
- **3Coloring**.
- **3-D Matching**.
- **Subset Sum**.

24.0.28.7 Subset Sum and Knapsack

(A) **Subset Sum Problem:** Given n integers $a_1, a_2, \ldots, a_n$ and a target B, is there a subset of S of $\{a_1, \ldots, a_n\}$ such that the numbers in S add up *precisely* to B?
(B) Subset Sum is **NP-Complete**— see book.
(C) **Knapsack:** Given n items with item i having size s_i and profit p_i, a knapsack of capacity B, and a target profit P, is there a subset S of items that can be packed in the knapsack and the profit of S is at least P?
(D) Show Knapsack problem is **NP-Complete** via reduction from Subset Sum (exercise).

24.0.28.8 Subset Sum and Knapsack

(A) Subset Sum can be solved in $O(nB)$ time using dynamic programming (exercise).
(B) Implies that problem is hard only when numbers $a_1, a_2, \ldots, a_n$ are exponentially large compared to n. That is, each a_i requires polynomial in n bits.
(C) *Number problems* of the above type are said to be ***weakly*** **NPComplete**.

Chapter 25

Introduction to Linear Programming

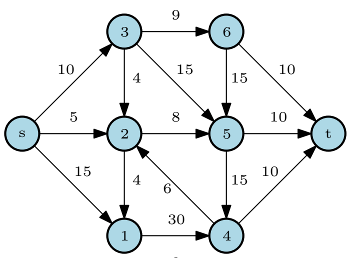
OLD CS 473: Fundamental Algorithms, Spring 2015
April 28, 2015

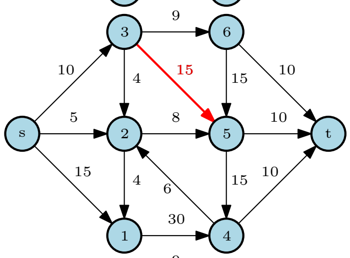
25.1 Introduction to Linear Programming

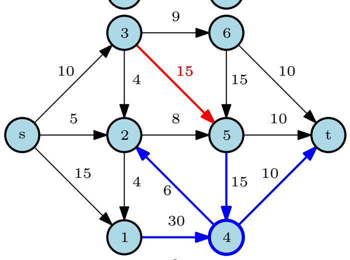
25.1.1 Introduction

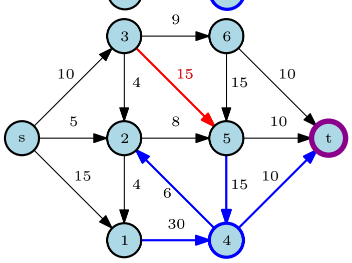
25.1.2 Examples

25.1.2.1 Maximum Flow in Network









Need to compute values $f_{s1}, f_{s2}, \ldots f_{25}, \ldots f_{5t}, f_{6t}$ such that

$$
\begin{array}{lll}
f_{s1} \leq 15 & f_{s2} \leq 5 & f_{s3} \leq 10 \\
f_{14} \leq 30 & f_{21} \leq 4 & f_{25} \leq 8 \\
f_{32} \leq 4 & f_{35} \leq 15 f_{35} \leq 15 & f_{36} \leq 9 \\
f_{42} \leq 6 & f_{4t} \leq 10 & f_{54} \leq 15 \\
f_{5t} \leq 10 & f_{65} \leq 15 & f_{6t} \leq 10
\end{array}
$$

and

$$
\begin{array}{lll}
f_{s1} + f_{21} = f_{14} & f_{s2} + f_{32} = f_{21} + f_{25} & f_{s3} = f_{32} + f_{35} + f_{36} \\
f_{14} + f_{54} = f_{42} + f_{4t} \boxed{f_{14} + f_{54} = f_{42} + f_{4t}} & f_{25} + f_{35} + f_{65} = f_{54} + f_{5t} & f_{36} = f_{65} + f_{6t} \\
f_{s1} \geq 0 \qquad f_{s2} \geq 0 & f_{s3} \geq 0 \qquad \cdots \qquad f_{4t} \geq 0 & f_{5t} \geq 0 \qquad f_{6t} \geq 0
\end{array}
$$

and $f_{s1} + f_{s2} + f_{s3}$ is maximized.

25.1.2.2 Maximum Flow as a Linear Program

For a general flow network $G = (V, E)$ with capacities c_e on edge $e \in E$, we have variables f_e indicating flow on edge e

$$
\begin{array}{lll}
\text{Maximize} & \displaystyle\sum_{e \text{ out of } s} f_e & \\
\text{subject to} & f_e \leq c_e & \text{for each } e \in E \\
& \displaystyle\sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e = 0 & \forall v \in V \setminus \{s, t\} \\
& f_e \geq 0 & \text{for each } e \in E.
\end{array}
$$

Number of variables: m, one for each edge.
Number of constraints: $m + n - 2 + m$.

25.1.3 Minimum Cost Flow with Lower Bounds

25.1.3.1 ... as a Linear Program

For a general flow network $G = (V, E)$ with capacities c_e, lower bounds ℓ_e, and costs w_e, we have variables f_e indicating flow on edge e. Suppose we want a min-cost flow of value at least v.

$$
\begin{array}{lll}
\text{Minimize} & \displaystyle\sum_{e \in E} w_e f_e & \\
\text{subject to} & \displaystyle\sum_{e \text{ out of } s} f_e \geq v & \\
& f_e \leq c_e \qquad f_e \geq \ell_e & \text{for each } e \in E \\
& \displaystyle\sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e = 0 & \text{for each } v \in V - \{s, t\} \\
& f_e \geq 0 & \text{for each } e \in E.
\end{array}
$$

Number of variables: m, one for each edge
Number of constraints: $1 + m + m + n - 2 + m = 3m + n - 1$.

25.1.4 General Form

25.1.4.1 Linear Programs

Problem Find a vector $x \in \mathbb{R}^d$ that

$$
\begin{array}{lll}
\text{maximize/minimize} & \sum_{j=1}^{d} c_j x_j & \\
\text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i & \text{for } i = p+1 \ldots q \\
& \sum_{j=1}^{d} a_{ij} x_j \geq b_i & \text{for } i = q+1 \ldots n
\end{array}
$$

Input is matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $b = (b_i) \in \mathbb{R}^n$, and row vector $c = (c_j) \in \mathbb{R}^d$

25.1.5 Canonical Forms

25.1.5.1 Canonical Form of Linear Programs

Canonical Form A linear program is in **canonical form** if it has the following structure

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots n \\ & x_j \geq 0 & \text{for } j = 1 \ldots d \end{array}$$

¡2-¿Conversion to Canonical Form

(A) Replace each variable x_j by $x_j^+ - x_j^-$ and inequalities $x_j^+ \geq 0$ and $x_j^- \geq 0$
(B) Replace $\sum_j a_{ij} x_j = b_i$ by $\sum_j a_{ij} x_j \leq b_i$ and $-\sum_j a_{ij} x_j \leq -b_i$
(C) Replace $\sum_j a_{ij} x_j \geq b_i$ by $-\sum_j a_{ij} x_j \leq -b_i$

25.1.5.2 Matrix Representation of Linear Programs

A linear program in canonical form can be written as

$$\begin{array}{ll} \text{maximize} & c \cdot x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \end{array}$$

where $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $b = (b_i) \in \mathbb{R}^n$, row vector $c = (c_j) \in \mathbb{R}^d$, and column vector $x = (x_j) \in \mathbb{R}^d$

(A) Number of variable is d
(B) Number of constraints is $n + d$

25.1.5.3 Other Standard Forms for Linear Programs

$$\begin{array}{ll} \text{maximize} & c \cdot x \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

$$\begin{array}{ll} \text{minimize} & c \cdot x \\ \text{subject to} & Ax \geq b \\ & x \geq 0 \end{array}$$

25.1.6 History

25.1.6.1 Linear Programming: A History

(A) First formalized applied to problems in economics by Leonid Kantorovich in the 1930s
    (A) However, work was ignored behind the Iron Curtain and unknown in the West
(B) Rediscovered by Tjalling Koopmans in the 1940s, along with applications to economics
(C) First algorithm (Simplex) to solve linear programs by George Dantzig in 1947
(D) Kantorovich and Koopmans receive Nobel Prize for economics in 1975 ; Dantzig, however, was ignored
    (A) Koopmans contemplated refusing the Nobel Prize to protest Dantzig's exclusion, but Kantorovich saw it as a vindication for using mathematics in economics, which had been written off as "a means for apologists of capitalism"

25.1.7 Shortest path as a linear program

25.1.8 Solving Linear Programs

25.1.9 Algorithm for 2 Dimensions

25.1.9.1 A Factory Example

Problem Suppose a factory produces two products I and II. Each requires three resources A, B, C.

(A) Producing one unit of Product I requires 1 unit each of resources A and C.
(B) One unit of Product II requires 1 unit of resource B and 1 units of resource C.
(C) We have 200 units of A, 300 units of B, and 400 units of C.
(D) Product I can be sold for \$1 and product II for \$6.
How many units of product I and product II should the factory manufacture to maximize profit?
Solution: Formulate as a linear program.

25.1.9.2 A Factory Example

Problem Suppose a factory produces two products I and II. Each requires three resources A, B, C.
(A) Producing unit I: Req. 1 unit of A, C.
(B) Producing unit II: Requ. 1 unit of B, C.
(C) Have A: 200, B: 300 , and C: 400.
(D) Price I: \$1, and II: \$6.
How many units of I and II to manufacture to max profit?

25.1.9.3 A Factory Example

Problem Suppose a factory produces two products I and II. Each requires three resources A, B, C.
(A) Producing unit I: Req. 1 unit of A, C.
(B) Producing unit II: Requ. 1 unit of B, C.
(C) Have A: 200, B: 300 , and C: 400.
(D) Price I: \$1, and II: \$6.
How many units of I and II to manufacture to max profit?

$$
\begin{array}{lll}
\max & x_I + 6x_{II} & \\
\text{s.t.} & x_I \leq 200 & (A) \\
& x_{II} \leq 300 & (B) \\
& x_I + x_{II} \leq 400 & (C) \\
& x_I \geq 0 & \\
& x_{II} \geq 0 &
\end{array}
$$

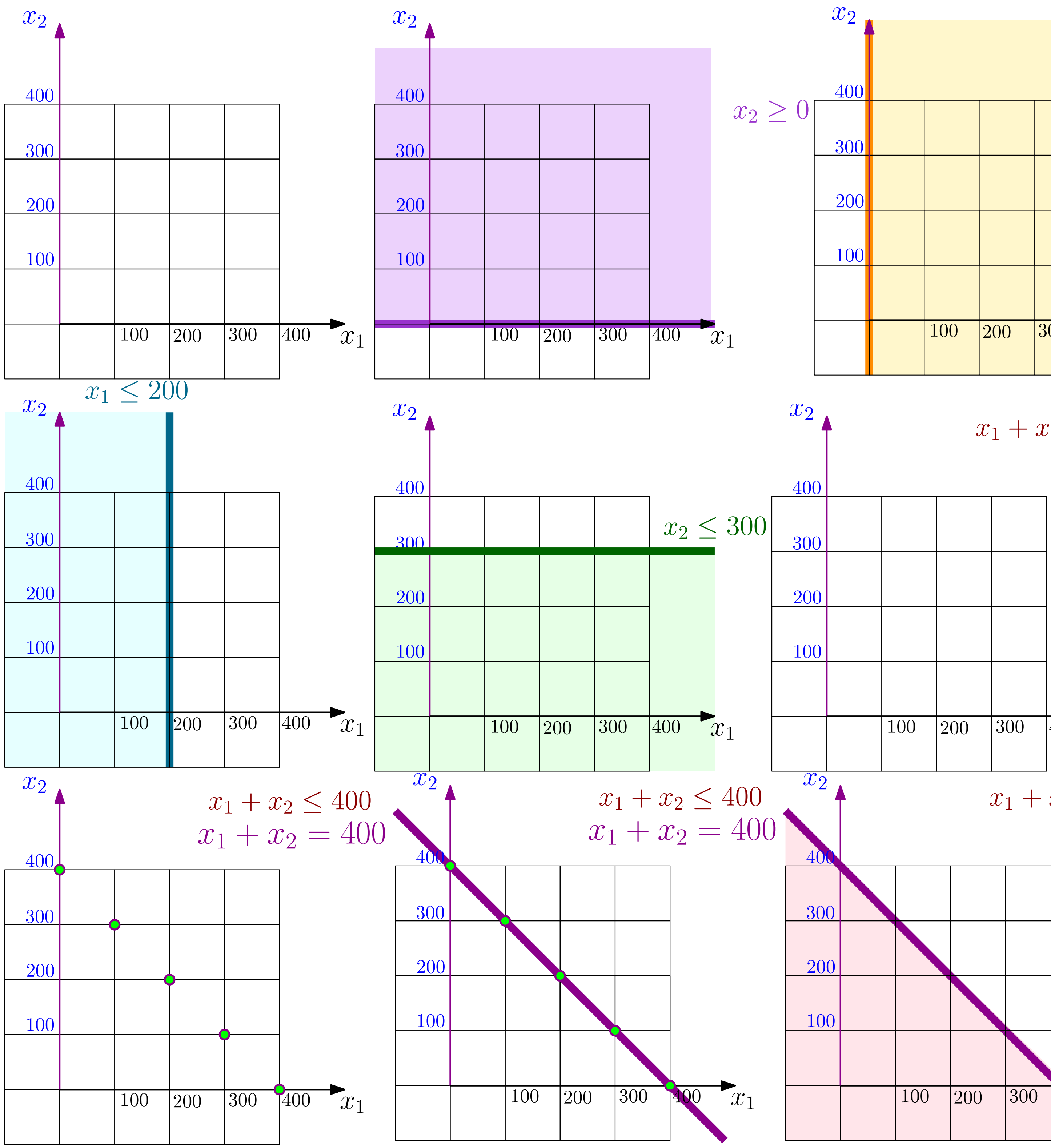
25.1.9.4 Linear Programming Formulation

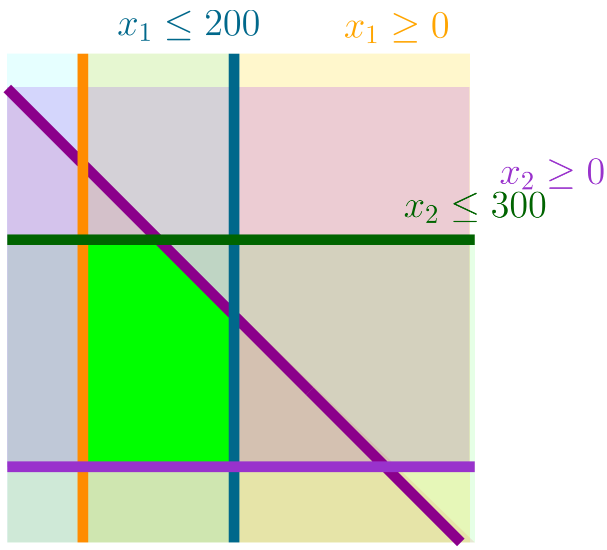
Let us produce x_1 units of product I and x_2 units of product II. Our profit can be computed by solving

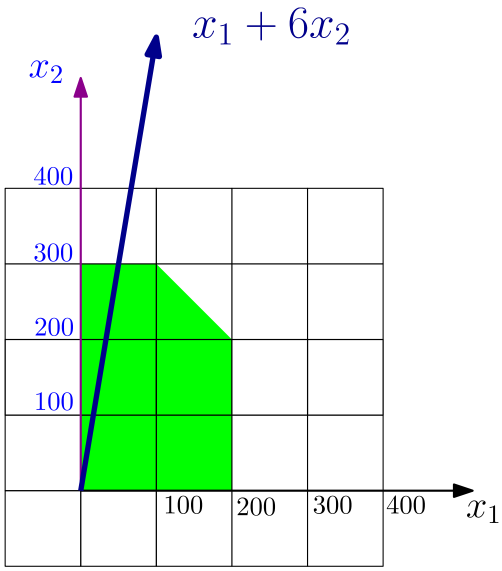
$$
\begin{array}{ll}
\text{maximize} & x_1 + 6x_2 \\
\text{subject to} & x_1 \leq 200 \\
& x_2 \leq 300 \\
& x_1 + x_2 \leq 400 \\
& x_1, x_2 \geq 0
\end{array}
$$

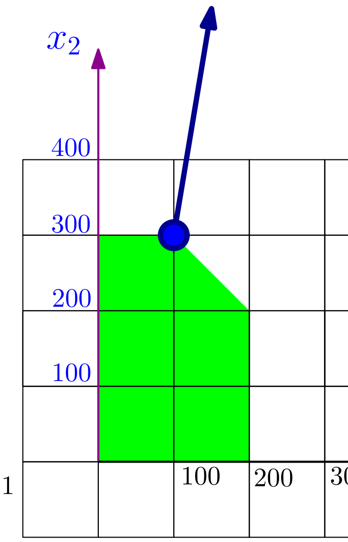
What is the solution?

25.1.9.5 Graphical interpretation of LP

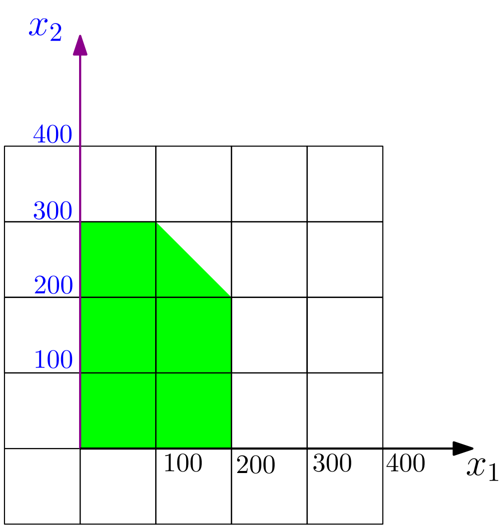






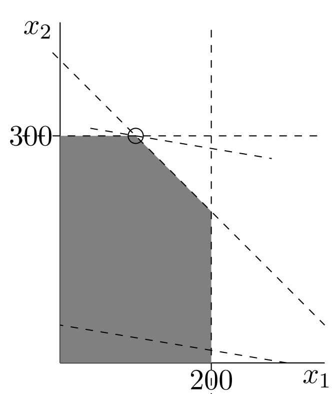


25.1.9.6 Solving the Factory Example



$$
\begin{array}{ll}
\text{maximize} & x_1 + 6x_2 \\
\text{subject to} & x_1 \leq 200 \\
& x_2 \leq 300 \\
& x_1 + x_2 \leq 400 \\
& x_1, x_2 \geq 0
\end{array}
$$

25.1.9.7 Solving the Factory Example



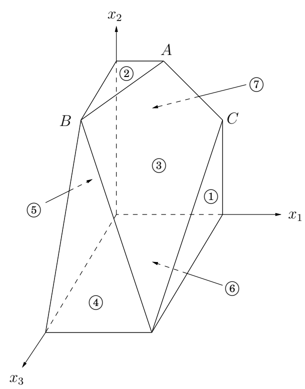
(A) Feasible values of x_1 and x_2 are shaded region.

(B) Objective function is a direction — the line represents all points with same value of the function; moving the line until it just leaves the feasible region, gives optimal values.

25.1.9.8 Linear Programming in 2-d

(A) Each constraint a half plane
(B) Feasible region is intersection of finitely many half planes — it forms a polygon
(C) For a fixed value of objective function, we get a line. Parallel lines correspond to different values for objective function.
(D) Optimum achieved when objective function line just leaves the feasible region

25.1.9.9 An Example in 3-d



$$
\begin{array}{rll}
\max & x_1 + 6x_2 + 13x_3 & \\
 & x_1 \leq 200 & ① \\
 & x_2 \leq 300 & ② \\
 & x_1 + x_2 + x_3 \leq 400 & ③ \\
 & x_2 + 3x_3 \leq 600 & ④ \\
 & x_1 \geq 0 & ⑤ \\
 & x_2 \geq 0 & ⑥ \\
 & x_3 \geq 0 & ⑦
\end{array}
$$

Figure from Dasgupta etal book.

25.1.10 Simplex in 2 Dimensions

25.1.10.1 Factory Example: Alternate View

Original Problem Recall we have,

$$
\begin{array}{ll}
\text{maximize} & x_1 + 6x_2 \\
\text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 \quad x_1 + x_2 \leq 400 \\
 & x_1, x_2 \geq 0
\end{array}
$$

¡2-¿Transformation Consider new variable x'_1 and x'_2, such that $x_1 = -6x'_1 + x'_2$ and $x_2 = x'_1 + 6x'_2$. Then in terms of the new variables we have

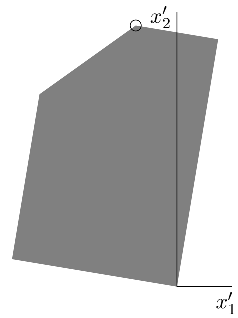
$$
\begin{array}{ll}
\text{maximize} & 37x'_2 \\
\text{subject to} & -6x'_1 + x'_2 \leq 200 \quad x'_1 + 6x'_2 \leq 300 \quad -5x'_1 + 7x_2 \leq 400 \\
 & -6x'_1 + x'_2 \geq 0 \qquad x'_1 + 6x'_2 \geq 0
\end{array}
$$

25.1.10.2 Transformed Picture

Feasible region rotated, and optimal value at the highest point on polygon

25.1.10.3 Observations about the Transformation

Observations
(A) Linear program can always be transformed to get a linear program where the optimal value is achieved at the point in the feasible region with highest y-coordinate



(B) Optimum value attained at a vertex of the polygon
(C) Since feasible region is convex, every local optimum is a global optimum

25.1.10.4 A Simple Algorithm in 2-d

(A) optimum solution is at a vertex of the feasible region
(B) a vertex is defined by the intersection of two lines (constraints)

Algorithm:

(A) find all intersections between the n lines — n^2 points
(B) for each intersection point $p = (p_1, p_2)$
  (A) check if p is in feasible region (how?)
  (B) if p is feasible evaluate objective function at p: $val(p) = c_1 p_1 + c_2 p_2$
(C) Output the feasible point with the largest value

Running time: $O(n^3)$.

25.1.10.5 Simple Algorithm in General Case

Real problem: d-dimensions
(A) optimum solution is at a vertex of the feasible region
(B) a vertex is defined by the intersection of d hyperplanes
(C) number of vertices can be $\Omega(n^d)$

Running time: $O(n^{d+1})$ which is not polynomial since problem size is at least nd. Also not practical.

How do we find the intersection point of d hyperplanes in $\mathbb{R}^d$? Using Gaussian elimination to solve $Ax = b$ where A is a $d \times d$ matrix and b is a $d \times 1$ matrix.

25.1.10.6 Simplex in 2-d

Simplex Algorithm

1. Start from some vertex of the feasible polygon
2. Compare value of objective function at current vertex with the value at "neighboring" vertices of polygon
3. If neighboring vertex improves objective function, move to this vertex, and repeat step 2
4. If current vertex is local optimum, then stop.

25.1.11 Simplex in Higher Dimensions

25.1.11.1 Linear Programming in d-dimensions

(A) Each linear constraint defines a ***halfspace***.
(B) Feasible region, which is an intersection of halfspaces, is a convex ***polyhedron***.
(C) Optimal value attained at a vertex of the polyhedron.
(D) Every local optimum is a global optimum.

25.1.11.2 Simplex in Higher Dimensions

1. Start at a vertex of the polytope.

2. Compare value of objective function at each of the d "neighbors".

3. Move to neighbor that improves objective function, and repeat step 2.

4. If local optimum, then stop

Simplex is a **greedy local-improvement** algorithm! Works because a local optimum is also a global optimum — convexity of polyhedra.

25.1.11.3 Solving Linear Programming in Practice

(A) Naïve implementation of Simplex algorithm can be very inefficient
  (A) Choosing which neighbor to move to can significantly affect running time
  (B) Very efficient Simplex-based algorithms exist
  (C) Simplex algorithm takes exponential time in the worst case but works extremely well in practice with many improvements over the years
(B) Non Simplex based methods like interior point methods work well for large problems.

25.1.11.4 Polynomial time Algorithm for Linear Programming

Major open problem for many years: is there a polynomial time algorithm for linear programming?
Leonid Khachiyan in 1979 gave the first polynomial time algorithm using the **Ellipsoid method**.

(A) major theoretical advance
(B) highly impractical algorithm, not used at all in practice
(C) routinely used in theoretical proofs.

Narendra Karmarkar in 1984 developed another polynomial time algorithm, the **interior point method**.
(A) very practical for some large problems and beats simplex
(B) also revolutionized theory of interior point methods

Following interior point method success, Simplex has been improved enormously and is the method of choice.

25.1.11.5 Degeneracy

(A) The linear program could be **infeasible**: No points satisfy the constraints.
(B) The linear program could be **unbounded**: Polygon unbounded in the direction of the objective function.

25.1.11.6 Infeasibility: Example

$$\begin{array}{lc} \text{maximize} & x_1 + 6x_2 \\ \text{subject to} & x_1 \le 2 \quad x_2 \le 1 \quad x_1 + x_2 \ge 4 \\ & x_1, x_2 \ge 0 \end{array}$$

Infeasibility has to do only with constraints.

25.1.11.7 Unboundedness: Example

$$\begin{array}{rcl} \text{maximize } x_2 & & \\ x_1 + x_2 & \ge & 2 \\ x_1, x_2 & \ge & 0 \end{array}$$

Unboundedness depends on both constraints and the objective function.

25.1.12 Duality

25.1.13 Lower Bounds and Upper Bounds

25.1.13.1 Feasible Solutions and Lower Bounds

Consider the program

$$\begin{array}{lrrrl} \text{maximize} & 4x_1+ & x_2+ & 3x_3 & \\ \text{subject to} & x_1+ & 4x_2 & & \le 1 \\ & 3x_1- & x_2+ & x_3 & \le 3 \\ & & & x_1, x_2, x_3 & \ge 0 \end{array}$$

(A) $(1, 0, 0)$ satisfies all the constraints and gives value 4 for the objective function.
(B) Thus, optimal value σ^* is at least 4.
(C) $(0, 0, 3)$ also feasible, and gives a better bound of 9.
(D) How good is 9 when compared with σ^*?

25.1.13.2 Obtaining Upper Bounds

(A) Let us multiply the first constraint by 2 and the second by 3 and add the result

$$\begin{array}{rrrrl} 2(& x_1+ & 4x_2 & &) \leq 2(1) \\ +3(& 3x_1- & x_2+ & x_3 &) \leq 3(3) \\ \hline & 11x_1+ & 5x_2+ & 3x_3 & \leq 11 \end{array}$$

(B) Since x_is are positive, compared to objective function $4x_1 + x_2 + 3x_3$, we have

$$4x_1 + x_2 + 3x_3 \leq 11x_1 + 5x_2 + 3x_3 \leq 11$$

(C) Thus, 11 is an upper bound on the optimum value!

25.1.13.3 Generalizing ...

(A) Multiply first equation by y_1 and second by y_2 (both y_1, y_2 being positive) and add

$$\begin{array}{rrrrl} y_1(& x_1+ & 4x_2 & &) \leq y_1(1) \\ +y_2(& 3x_1- & x_2+ & x_3 &) \leq y_2(3) \\ \hline & (y_1+3y_2)x_1+ & (4y_1-y_2)x_2+ & (y_2)x_3 & \leq y_1 + 3y_2 \end{array}$$

(B) $y_1 + 3y_2$ is an upper bound, provided coefficients of x_i are as large as in the objective function, i.e.,

$$y_1 + 3y_2 \geq 4 \quad 4y_1 - y_2 \geq 1 \quad y_2 \geq 3$$

(C) The best upper bound is when $y_1 + 3y_2$ is minimized!

25.1.14 Dual Linear Programs

25.1.14.1 Dual LP: Example

Thus, the optimum value of program

$$\begin{array}{lr} \text{maximize} & 4x_1 + x_2 + 3x_3 \\ \text{subject to} & x_1 + 4x_2 \leq 1 \\ & 3x_1 - x_2 + x_3 \leq 3 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

is upper bounded by the optimal value of the program

$$\begin{array}{lr} \text{minimize} & y_1 + 3y_2 \\ \text{subject to} & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{array}$$

25.1.14.2 Dual Linear Program

Given a linear program Π in canonical form

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \le b_i & i = 1, 2, \ldots n \\ & x_j \ge 0 & j = 1, 2, \ldots d \end{array}$$

the dual Dual(Π) is given by

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} b_i y_i & \\ \text{subject to} & \sum_{i=1}^{n} y_i a_{ij} \ge c_j & j = 1, 2, \ldots d \\ & y_i \ge 0 & i = 1, 2, \ldots n \end{array}$$

Proposition 25.1.1. *¡2-¿* Dual(Dual(Π)) *is equivalent to* Π

25.1.15 Duality Theorems

25.1.15.1 Duality Theorem

Theorem 25.1.2 (Weak Duality). *If x is a feasible solution to Π and y is a feasible solution to* Dual(Π) *then $c \cdot x \le y \cdot b$.*

Theorem 25.1.3 (Strong Duality). *If x^* is an optimal solution to Π and y^* is an optimal solution to* Dual(Π) *then $c \cdot x^* = y^* \cdot b$.*

Many applications! Maxflow-Mincut theorem can be deduced from duality.

25.1.15.2 Maximum Flow Revisited

For a general flow network $G = (V, E)$ with capacities c_e on edge $e \in E$, we have variables f_e indicating flow on edge e

$$\begin{array}{ll} \text{Maximize } \sum_{e \text{ out of } s} f_e & \text{subject to} \\ f_e \le c_e & \text{for each } e \in E \\ \sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e = 0 & \text{for each } v \in V - \{s, t\} \\ f_e \ge 0 & \text{for each } e \in E \end{array}$$

Number of variables: m, one for each edge
Number of constraints: $m + n - 2 + m$

Maximum flow can be reduced to Linear Programming.

25.1.16 Integer Linear Programming

25.1.16.1 Integer Linear Programming

Problem Find a vector $x \in Z^d$ (integer values) that

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \le b_i & \text{for } i = 1 \ldots n \end{array}$$

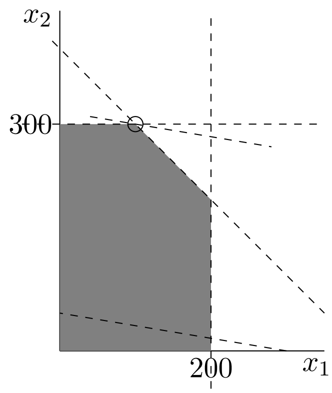
Input is matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $b = (b_i) \in \mathbb{R}^n$, and row vector $c = (c_j) \in \mathbb{R}^d$

25.1.16.2 Factory Example

$$\begin{array}{ll} \text{maximize} & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 \quad x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{array}$$

Suppose we want x_1, x_2 to be integer valued.

25.1.16.3 Factory Example Figure



(A) Feasible values of x_1 and x_2 are integer points in shaded region
(B) Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

25.1.16.4 Integer Programming

Can model many difficult discrete optimization problems as integer programs!

Therefore integer programming is a hard problem. NP-hard.

Can relax integer program to linear program and *approximate*.

Practice: integer programs are solved by a variety of methods

(A) branch and bound
(B) branch and cut
(C) adding cutting planes
(D) linear programming plays a fundamental role

25.1.16.5 Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.
Then solving linear program is same as solving integer program. We know how to solve linear programs efficiently (polynomial time) and hence we get an integer solution for free!

Luck or *Structure*:

(A) Linear program for flows with integer capacities have integer vertices
(B) Linear program for matchings in bipartite graphs have integer vertices
(C) A complicated linear program for matchings in general graphs have integer vertices.

All of above problems can hence be solved efficiently.

25.1.16.6 Linear Programs with Integer Vertices

Meta Theorem: A combinatorial optimization problem can be solved efficiently if and only if there is a linear program for problem with integer vertices.

Consequence of the Ellipsoid method for solving linear programming.

In a sense linear programming and other geometric generalizations such as convex programming are the most general problems that we can solve efficiently.

25.1.16.7 Summary

(A) Linear Programming is a useful and powerful (modeling) problem.
(B) Can be solved in polynomial time. Practical solvers available commercially as well as in open source. Whether there is a strongly polynomial time algorithm is a major open problem.
(C) Geometry and linear algebra are important to understand the structure of LP and in algorithm design. Vertex solutions imply that LPs have poly-sized optimum solutions. This implies that LP is in **NP**.
(D) Duality is a critical tool in the theory of linear programming. Duality implies the Linear Programming is in **co-NP**. Do you see why?
(E) Integer Programming in **NP-Complete**. LP-based techniques critical in heuristically solving integer programs.

Chapter 26

Approximation Algorithms using Linear Programming

OLD CS 473: Fundamental Algorithms, Spring 2015
April 30, 2015

26.0.17 Weighted vertex cover

26.0.18 Weighted vertex cover

26.0.18.1 Weighted vertex cover

Weighted Vertex Cover problem $G = (V, E)$.

Each vertex $v \in V$: cost c_v.

Compute a vertex cover of minimum cost.

(A) vertex cover: subset of vertices V so each edge is covered.
(B) **NP-Hard**
(C) ...unweighted **Vertex Cover** problem.
(D) ... write as an integer program (IP):
(E) $\forall v \in V$: $x_v = 1 \iff$ v in the vertex cover.
(F) $\forall vu \in E$: covered. $\implies x_v \vee x_u$ true. $\implies x_v + x_u \geq 1$.
(G) minimize total cost: $\min \sum_{v \in V} x_v c_v$.

26.0.19 Weighted vertex cover

26.0.19.1 State as IP $\implies$ Relax $\implies$ LP

$$
\begin{aligned}
\min \quad & \sum_{v \in V} c_v x_v, & & \\
\text{such that} \quad & x_v \in \{0, 1\} & \forall v \in V & \qquad (26.1)\\
& x_v + x_u \geq 1 & \forall vu \in E. &
\end{aligned}
$$

(A) ... **NP-Hard**.
(B) relax the integer program.
(C) allow x_v get values $\in [0,1]$.
(D) $x_v \in \{0,1\}$ replaced by $0 \le x_v \le 1$. The resulting LP is

$$\begin{aligned} \min \quad & \sum_{v \in V} c_v x_v, & \\ \text{s.t.} \quad & 0 \le x_v & \forall v \in V, \\ & x_v \le 1 & \forall v \in V, \\ & x_v + x_u \ge 1 & \forall vu \in E. \end{aligned}$$

26.0.19.2 Weighted vertex cover – rounding the LP

(A) Optimal solution to this LP: $\widehat{x_v}$ value of var X_v, $\forall v \in V$.
(B) optimal value of LP solution is $\widehat{\alpha} = \sum_{v \in V} c_v \widehat{x_v}$.
(C) optimal integer solution: x_v^I, $\forall v \in V$ and α^I.
(D) **Any valid solution to IP is valid solution for LP!**
(E) $\widehat{\alpha} \le \alpha^I$.
Integral solution not better than LP.
(F) Got fractional solution (i.e., values of $\widehat{x_v}$).
(G) Fractional solution is better than the optimal cost.
(H) Q: How to turn fractional solution into a (valid!) integer solution?
(I) Using ***rounding***.

26.0.19.3 How to round?

(A) consider vertex v and fractional value $\widehat{x_v}$.
(B) If $\widehat{x_v} = 1$ then include in solution!
(C) If $\widehat{x_v} = 0$ then do **not**not include in solution.
(D) if $\widehat{x_v} = 0.9 \implies$ LP considers v as being 0.9 useful.
(E) The LP puts its money where its belief is...
(F) ...$\widehat{\alpha}$ value is a function of this "belief" generated by the LP.
(G) **Big idea**: Trust LP values as guidance to usefulness of vertices.

26.0.19.4 II: How to round?

$$\begin{aligned} \min \quad & \sum_{v \in V} c_v x_v, & \\ \text{s.t.} \quad & 0 \le x_v & \forall v \in V \\ & x_v \le 1 & \forall v \in V \\ & x_v + x_u \ge 1 & \forall vu \in E \end{aligned}$$

(A) Pick all vertices $\ge$ threshold of usefulness according to LP.
(B) $S = \left\{ v \;\middle|\; \widehat{x_v} \ge 1/2 \right\}$.
(C) **Claim**: S a valid vertex cover, and cost is low.

(A) Indeed, edge cover as: $\forall vu \in E$ have $\widehat{x_v} + \widehat{x_u} \ge 1$.
(B) $\widehat{x_v}, \widehat{x_u} \in (0,1)$
$\implies \widehat{x_v} \ge 1/2$ or $\widehat{x_u} \ge 1/2$.
$\implies v \in S$ or $u \in S$ (or both).
$\implies S$ covers all the edges of G.

26.0.19.5 Cost of solution

Cost of S:

$$\mathsf{c}_S = \sum_{\mathsf{v}\in S} \mathsf{c}_\mathsf{v} = \sum_{\mathsf{v}\in S} 1\cdot \mathsf{c}_\mathsf{v} \le \sum_{\mathsf{v}\in S} 2\widehat{x_\mathsf{v}}\cdot \mathsf{c}_\mathsf{v} \le 2\sum_{\mathsf{v}\in \mathsf{V}} \widehat{x_\mathsf{v}}\mathsf{c}_\mathsf{v} = 2\widehat{\alpha} \le 2\alpha^I,$$

since $\widehat{x_\mathsf{v}} \ge 1/2$ as $\mathsf{v}\in S$.

α^I is cost of the optimal solution $\implies$

Theorem 26.0.4. *The* **Weighted Vertex Cover** *problem can be* 2*-approximated by solving a single* LP*. Assuming computing the* LP *takes polynomial time, the resulting approximation algorithm takes polynomial time.*

26.0.20 The lessons we can take away

26.0.20.1 Or not - boring, boring, boring.

(A) Weighted vertex cover is simple, but resulting approximation algorithm is non-trivial.
(B) Not aware of any other 2-approximation algorithm does not use LP. (For the weighted case!)
(C) Solving a ***relaxation*** of an optimization problem into a LP provides us with insight.
(D) But... have to be creative in the rounding.

26.0.21 Revisiting Set Cover

26.0.21.1 Revisiting Set Cover

(A) Purpose: See new technique for an approximation algorithm.
(B) Not better than greedy algorithm already seen $O(\log n)$ approximation.

Problem: Set Cover

Instance: $(S, \mathcal{F})$
S - a set of n elements
$\mathcal{F}$ - a family of subsets of S, s.t. $\bigcup_{X\in\mathcal{F}} X = S$.
Question: The set $\mathcal{X} \subseteq F$ such that $\mathcal{X}$ contains as few sets as possible, and $\mathcal{X}$ covers S.

26.0.21.2 Set Cover – IP & LP

$$\begin{aligned}
\min \quad & \alpha = \sum_{U\in\mathfrak{F}} x_U, & \\
\text{s.t.} \quad & x_U \in \{0,1\} & \forall U \in \mathfrak{F}, \\
& \sum_{U\in\mathfrak{F}, s\in U} x_U \ge 1 & \forall s \in S.
\end{aligned}$$

Next, we relax this IP into the following LP.

$$
\begin{aligned}
\min \qquad & \alpha = \sum_{U \in \mathcal{F}} x_U, & \\
& 0 \le x_U \le 1 & \forall U \in \mathcal{F}, \\
& \sum_{U \in \mathcal{F}, s \in U} x_U \ge 1 & \forall s \in S.
\end{aligned}
$$

26.0.21.3 Set Cover – IP & LP

(A) LP solution: $\forall U \in \mathcal{F}$, $\widehat{x_U}$, and $\widehat{\alpha}$.
(B) Opt IP solution: $\forall U \in \mathcal{F}$, x_U^I, and α^I.
(C) Use LP solution to guide in rounding process.
(D) If $\widehat{x_U}$ is close to 1 then pick U to cover.
(E) If $\widehat{x_U}$ close to 0 do not.
(F) **Idea:** Pick $U \in \mathcal{F}$: randomly choose U with ***probability*** $\widehat{x_U}$.
(G) Resulting family of sets $\mathcal{G}$.
(H) Z_S: indicator variable. 1 if $S \in \mathcal{G}$.
(I) Cost of $\mathcal{G}$ is $\sum_{S \in \mathcal{F}} Z_S$, and the expected cost is $\mathbf{E}\big[\text{cost of } \mathcal{G}\big] = \mathbf{E}\big[\sum_{S \in \mathcal{F}} Z_S\big] = \sum_{S \in \mathcal{F}} \mathbf{E}\big[Z_S\big] = \sum_{S \in \mathcal{F}} \mathbf{Pr}\big[S \in \mathcal{G}\big] = \sum_{S \in \mathcal{F}} \widehat{x_S} = \widehat{\alpha} \le \alpha^I$.
(J) In expectation, $\mathcal{G}$ is not too expensive.
(K) Bigus problumos: $\mathcal{G}$ might fail to cover some element $s \in S$.

26.0.21.4 Set Cover – Rounding continued

(A) **Solution**: Repeat rounding stage $m = 10\lceil \lg n \rceil = O(\log n)$ times.
(B) $n = |S|$.
(C) $\mathcal{G}_i$: random cover computed in ith iteration.
(D) $\mathcal{H} = \cup_i \mathcal{G}_i$. Return $\mathcal{H}$ as the required cover.

26.0.21.5 The set $\mathcal{H}$ covers S

(A) For an element $s \in S$, we have that

$$
\sum_{U \in \mathcal{F}, s \in U} \widehat{x_U} \ge 1, \tag{26.2}
$$

(B) probability s not covered by $\mathcal{G}_i$ (ith iteration set).

$$
\begin{aligned}
&\mathbf{Pr}\Big[s \text{ not covered by } \mathcal{G}_i\Big] \\
&= \mathbf{Pr}\big[\text{ no } U \in \mathcal{F}, \text{ s.t. } s \in U \text{ picked into } \mathcal{G}_i\big] \\
&= \textstyle\prod_{U \in \mathcal{F}, s \in U} \mathbf{Pr}\big[U \text{ was not picked into } \mathcal{G}_i\big] \\
&= \prod_{U \in \mathcal{F}, s \in U} (1 - \widehat{x_U}) \le \prod_{U \in \mathcal{F}, s \in U} \exp(-\widehat{x_U}) \\
&= \exp\Big(-\textstyle\sum_{U \in \mathcal{F}, s \in U} \widehat{x_U}\Big) \le \exp(-1) \le \frac{1}{2}, \le \frac{1}{2}
\end{aligned}
$$

26.0.22 The set $\mathcal{H}$ covers S

26.0.22.1 Probability of a single item to be covered

(A) $\mathbf{Pr}\Big[s \text{ not covered by } \mathcal{G}_i\Big] \leq 1/2$.
(B) Number of iterations of rounding $m = O(\log n)$.
(C) Covering with sets in $\mathcal{G}_1, \ldots, \mathcal{G}_m$.
(D) probability s is not covered in all m iterations

$$
\begin{aligned}
P_s &= \mathbf{Pr}\Big[s \text{ not covered by } \mathcal{G}_1, \ldots, \mathcal{F}_m\Big] \\
&\leq \mathbf{Pr}\Big[(s \notin \mathcal{F}_1) \cap (s \notin \mathcal{F}_2) \cap \ldots \cap (s \notin \mathcal{F}_m)\Big] \\
&\leq \mathbf{Pr}\Big[s \notin \mathcal{F}_1\Big]\, \mathbf{Pr}\Big[s \notin \mathcal{F}_2\Big] \cdots \mathbf{Pr}\Big[s \notin \mathcal{F}_m\Big] \\
&= \frac{1}{2} \times \frac{1}{2} \times \cdots \times \frac{1}{2} = \left(\frac{1}{2}\right)^m < \frac{1}{n^{10}},
\end{aligned}
$$

26.0.23 The set $\mathcal{H}$ covers S

26.0.23.1 Probability of all items to be covered

(A) $n = |S|$,
(B) Probability of $s \in S$, not to be in $\mathcal{G}_1 \cup \ldots \cup \mathcal{F}_m$ is

$$
P_s < \frac{1}{n^{10}}.
$$

(C) probability one of n elements of S is not covered by $\mathcal{H}$ is

$$
\sum_{s \in S} \mathbf{Pr}[s \notin \mathcal{G}_1 \cup \ldots \cup \mathcal{F}_m] = \sum_{s \in S} P_s < n(1/n^{10}) = 1/n^9.
$$

XXX

26.0.23.2 Reminder: LP for Set Cover

$$
\begin{aligned}
\min \qquad & \alpha = \sum_{U \in \mathcal{F}} x_U, & \\
& 0 \leq x_U \leq 1 & \forall U \in \mathcal{F}, \\
& \sum_{U \in \mathcal{F}, s \in U} x_U \geq 1 & \forall s \in S.
\end{aligned}
$$

(A) Solve the LP.
(B) $\widehat{x_U}$: Value of x_u in the optimal LP solution.
(C) Fractional solution: $\widehat{\alpha} = \sum_{U \in \mathcal{F}} \widehat{x_U}$.
(D) Integral solution (what we want): $\alpha^I \geq \widehat{\alpha}$.

26.0.23.3 Cost of solution

(A) $(S, \mathcal{F})$: Given instance of **Set Cover**.
(B) For $U \in \mathcal{F}$, $\widehat{x_U}$: LP value for set U in optimal solution.
(C) For $\mathcal{G}_i$: Indicator variable $Z_u = 1 \iff U \in \mathcal{G}_i$.
(D) Expected number of sets in the ith sample:
$\mathbf{E}[|\mathcal{G}_i|] = \mathbf{E}\left[\sum_{U \in \mathcal{F}} Z_U\right] = \sum_{U \in \mathcal{F}} \mathbf{E}[Z_U] = \sum_{U \in \mathcal{F}} \widehat{x_U}$
$= \widehat{\alpha} \leq \alpha^I$.
(E) $\implies$ Each iteration expected cost of cover $\leq$ cost of optimal solution (i.e., α^I). XXX
(F) Expected size of the solution is

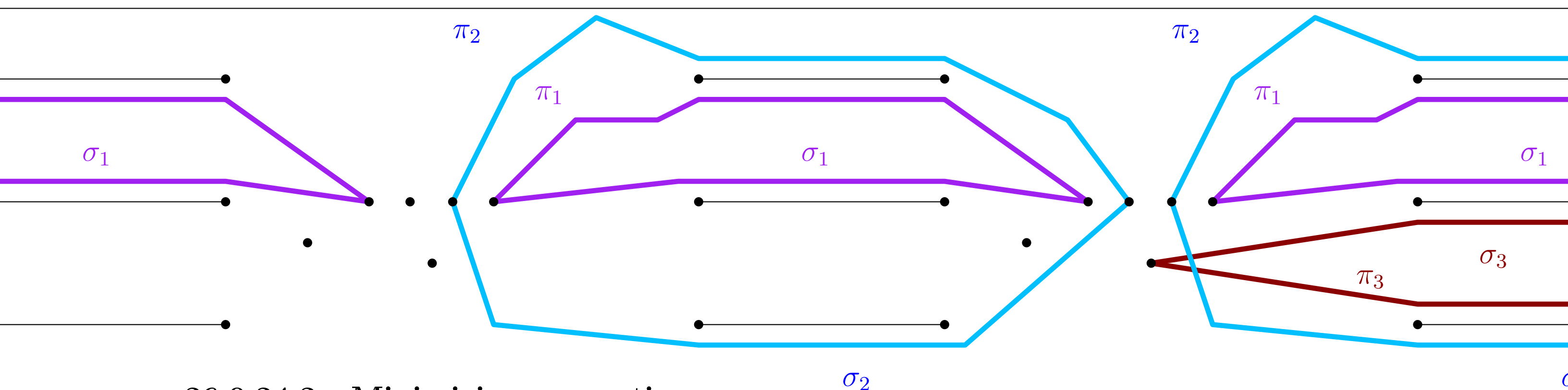
$$\mathbf{E}[|\mathcal{H}|] = \mathbf{E}[|\cup_i \mathcal{G}_i|] \leq \mathbf{E}\left[\sum_i |\mathcal{G}_i|\right] \leq m\alpha^I = O(\alpha^I \log n).$$

26.0.23.4 The result

Theorem 26.0.5. *By solving an LP one can get an $O(\log n)$-approximation to* **Set Cover** *by a randomized algorithm. The algorithm succeeds with high probability.*

26.0.24 Minimizing congestion

26.0.24.1 Minimizing congestion by example



26.0.24.2 Minimizing congestion

(A) G: graph. n vertices.
(B) π_i, σ_i paths with the same endpoints $\mathsf{v}_i, \mathsf{u}_i \in \mathsf{V}(\mathsf{G})$, for $i = 1, \ldots, t$.
(C) Rule I: Send one unit of flow from v_i to u_i.
(D) Rule II: Choose whether to use π_i or σ_i.
(E) Target: No edge in G is being used too much.

Definition 26.0.6. *Given a set X of paths in a graph G, the* **congestion** *of X is the maximum number of paths in X that use the same edge.*

26.0.24.3 Minimizing congestion

(A) IP $\implies$ LP:

$$
\begin{aligned}
\min \quad & w \\
\text{s.t.} \quad & x_i \geq 0 & i = 1, \ldots, t, \\
& x_i \leq 1 & i = 1, \ldots, t, \\
& \sum_{\mathsf{e} \in \pi_i} x_i + \sum_{\mathsf{e} \in \sigma_i} (1 - x_i) \leq w & \forall \mathsf{e} \in E.
\end{aligned}
$$

(B) $\widehat{x_i}$: value of x_i in the optimal LP solution.
(C) $\widehat{w}$: value of w in LP solution.
(D) Optimal congestion must be bigger than $\widehat{w}$.
(E) X_i: random variable one with probability $\widehat{x_i}$, and zero otherwise.
(F) If $X_i = 1$ then use π to route from v_i to u_i.
(G) Otherwise use σ_i.

26.0.24.4 Minimizing congestion

(A) Congestion of e is $Y_{\mathsf{e}} = \sum_{\mathsf{e} \in \pi_i} X_i + \sum_{\mathsf{e} \in \sigma_i} (1 - X_i)$.
(B) And in expectation

$$
\begin{aligned}
\alpha_{\mathsf{e}} = \mathbf{E}\Big[Y_{\mathsf{e}}\Big] &= \mathbf{E}\left[\sum_{\mathsf{e} \in \pi_i} X_i + \sum_{\mathsf{e} \in \sigma_i} (1 - X_i)\right] \\
&= \sum_{\mathsf{e} \in \pi_i} \mathbf{E}\Big[X_i\Big] + \sum_{\mathsf{e} \in \sigma_i} \mathbf{E}\Big[(1 - X_i)\Big] \\
&= \sum_{\mathsf{e} \in \pi_i} \widehat{x_i} + \sum_{\mathsf{e} \in \sigma_i} (1 - \widehat{x_i}) \leq \widehat{w}.
\end{aligned}
$$

(C) $\widehat{w}$: Fractional congestion (from LP solution).

26.0.24.5 Minimizing congestion - continued

(A) $Y_{\mathsf{e}} = \sum_{\mathsf{e} \in \pi_i} X_i + \sum_{\mathsf{e} \in \sigma_i} (1 - X_i)$.
(B) Y_{e} is just a sum of independent $0/1$ random variables!
(C) Chernoff inequality tells us sum can not be too far from expectation!

26.0.24.6 Minimizing congestion - continued

(A) By Chernoff inequality:

$$
\mathbf{Pr}\big[Y_{\mathsf{e}} \geq (1 + \delta)\alpha_{\mathsf{e}}\big] \leq \exp\left(-\frac{\alpha_{\mathsf{e}} \delta^2}{4}\right) \leq \exp\left(-\frac{\widehat{w}\delta^2}{4}\right).
$$

(B) Let $\delta = \sqrt{\dfrac{400}{\widehat{w}} \ln t}$. We have that

$$
\mathbf{Pr}\Big[Y_{\mathsf{e}} \geq (1 + \delta)\alpha_{\mathsf{e}}\Big] \leq \exp\left(-\frac{\delta^2 \widehat{w}}{4}\right) \leq \frac{1}{t^{100}},
$$

(C) If $t \geq n^{1/50} \implies \forall$ edges in graph congestion $\leq (1 + \delta)\widehat{w}$.
(D) t: Number of pairs, n: Number of vertices in G.

26.0.24.7 Minimizing congestion - continued

(A) Got: For $\delta = \sqrt{\frac{400}{\widehat{w}} \ln t}$. We have

$$\mathbf{Pr}\Big[Y_{\mathsf{e}} \geq (1+\delta)\alpha_{\mathsf{e}}\Big] \leq \exp\left(-\frac{\delta^2 \widehat{w}}{4}\right) \leq \frac{1}{t^{100}},$$

(B) Play with the numbers. If $t = n$, and $\widehat{w} \geq \sqrt{n}$. Then, the solution has congestion larger than the optimal solution by a factor of

$$1 + \delta = 1 + \sqrt{\frac{20}{\widehat{w}} \ln t} \leq 1 + \frac{\sqrt{20 \ln n}}{n^{1/4}},$$

which is of course extremely close to 1, if n is sufficiently large.

26.0.24.8 Minimizing congestion: result

Theorem 26.0.7. *(A)* G*: Graph* n *vertices.*
(B) $(s_1, t_1), \ldots, (s_t, t_t)$*: pairs o vertices*
(C) π_i, σ_i*: two different paths connecting* s_i *to* t_i
(D) $\widehat{w}$*: Fractional congestion at least* $n^{1/2}$*.*
(E) opt*: Congestion of optimal solution.*
(F) $\implies$ *In polynomial time (*LP *solving time) choose paths*
  (A) congestion $\forall$ *edges:* $\leq (1+\delta)\mathrm{opt}$
  (B) $\delta = \sqrt{\frac{20}{\widehat{w}} \ln t}$.

26.0.24.9 When the congestion is low

(A) Assume $\widehat{w}$ is a constant.
(B) Can get a better bound by using the Chernoff inequality in its more general form.
(C) set $\delta = c \ln t / \ln \ln t$, where c is a constant. For $\mu = \alpha_{\mathsf{e}}$, we have that

$$\begin{aligned}\mathbf{Pr}\Big[Y_{\mathsf{e}} \geq (1+\delta)\mu\Big] &\leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\mu} \\ &= \exp\Big(\mu\big(\delta - (1+\delta)\ln(1+\delta)\big)\Big) \\ &= \exp\Big(-\mu c' \ln t\Big) \leq \frac{1}{t^{O(1)}},\end{aligned}$$

where c' is a constant that depends on c and grows if c grows.

26.0.24.10 When the congestion is low

(A) Just proved that...
(B) if the optimal congestion is $O(1)$, then...
(C) algorithm outputs a solution with congestion $O(\log t / \log \log t)$, and this holds with high probability.

26.0.25 Reminder about Chernoff inequality

26.0.25.1 The Chernoff Bound — General Case

26.0.25.2 Chernoff inequality

Problem 26.0.8. *Let $X_1, \ldots X_n$ be n independent Bernoulli trials, where*

$$\mathbf{Pr}\left[X_i = 1\right] = p_i, \qquad \mathbf{Pr}\left[X_i = 0\right] = 1 - p_i,$$
$$Y = \sum_i X_i, \qquad \textit{and} \qquad \mu = \mathbf{E}\left[Y\right].$$

We are interested in bounding the probability that $Y \geq (1+\delta)\mu$.

26.0.25.3 Chernoff inequality

Theorem 26.0.9 (Chernoff inequality). *For any $\delta > 0$,*

$$\mathbf{Pr}\left[Y > (1+\delta)\mu\right] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

Or in a more simplified form, for any $\delta \leq 2e - 1$,

$$\mathbf{Pr}\left[Y > (1+\delta)\mu\right] < \exp\left(-\mu\delta^2/4\right),$$

and

$$\mathbf{Pr}\left[Y > (1+\delta)\mu\right] < 2^{-\mu(1+\delta)},$$

for $\delta \geq 2e - 1$.

26.0.25.4 More Chernoff...

Theorem 26.0.10. *Under the same assumptions as the theorem above, we have*

$$\mathbf{Pr}\left[Y < (1-\delta)\mu\right] \leq \exp\left(-\mu\frac{\delta^2}{2}\right).$$

Chapter 27

Final review...

OLD CS 473: Fundamental Algorithms, Spring 2015
May 5, 2015

27.0.26 Multiple choice

27.0.26.1 Part I

For each of the questions below choose the most appropriate answer.

(A) Given a graph G. Deciding if there is an independent set X in G, such that $\mathsf{G} \setminus X$ (i.e., the graph G after we remove the vertices of X from it) is bipartite can be solved in polynomial time.

False: ☐ **True**: ☐ **Answer depends on whether P = NP**: ☐

27.0.27 Multiple choice

27.0.27.1 Part II

(B) Consider any two problems X and Y both of them in **NPC**. There always exists a polynomial time reduction from X to Y.

False: ☐ **True**: ☐ **Answer depends on whether P = NP**: ☐

27.0.28 Multiple choice

27.0.28.1 Part III

(C) Given a graph represented using adjacency lists, it can be converted into matrix representation in linear time in the size of the graph (i.e., linear in the number of vertices and edges of the graph).
False: ☐ **True**: ☐ **Answer depends on whether P = NP**: ☐

27.0.29 Multiple choice

27.0.29.1 Part IV

(D) Given a **2SAT** formula F, there is always an assignment to its variables that satisfies at least $(7/8)m$ of its clauses. **False**: ☐ **True**: ☐ **Answer depends on whether P = NP**: ☐

27.0.30 Multiple choice

27.0.30.1 Part V

(E) Given a graph G, deciding if contains a clique made out of 165 vertices is **NP-Complete**.
False: ☐ **True**: ☐ **Answer depends on whether P = NP**: ☐

27.0.31 Multiple choice

27.0.31.1 Part VI

(F) Given a network flow G with lower bounds and capacities on the edges (say all numbers are integers that are smaller than n). Assume f and g are two different maximum flows in G that complies with the lower bounds and capacity constraints. Then, the flow $0.7f + 0.3g$ is always a valid maximum flow in G. **False**: ☐ **True**: ☐

27.0.32 Multiple choice

27.0.32.1 Part VII

(G) Given a directed graph G with positive weights on the edges, and a number k, finding if there is simple path in G from s to t (two given vertices of G) with weight $\geq k$, can be done in polynomial time.
False: ☐ **True**: ☐ **Answer depends on whether P = NP**: ☐

27.0.33 Multiple choice

27.0.33.1 Part VIII

(H) Given a directed graph G with (positive or negative) weights on its edges, computing the shortest walk from s to t in G can be done in polynomial time. **False**: $\square$ **True**: $\square$ **Answer depends on whether P = NP**: $\square$

27.0.34 2: Short Questions.

27.0.34.1 Part (A)

(A) Give a tight asymptotic bound for each of the following recurrences.

(I) $A(n) = A(n - 3\lceil \log n \rceil) + A(\lceil \log n \rceil) + \log n$, for $n > 2$ and $A(1) = A(2) = 1$.

27.0.35 2: Short Questions.

27.0.35.1 Part (A) II

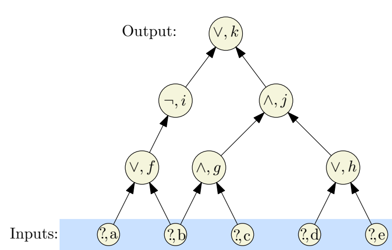
(A) ...

(II) $B(n) = 12B(\lfloor n/4 \rfloor) + B(\lfloor n/2 \rfloor) + n^2$, for $n > 10$ and $B(i) = 1$ for $1 \leq i \leq 10$.

27.0.36 2: Short Questions.

27.0.36.1 Part (B)

(B) Convert the following boolean circuit (i.e., an instance of **Circuit-SAT**) into a CNF formula (i.e., an instance of SAT) such that the resulting formula is satisfiable if and only if the circuit sat instance is satisfiable. Use $x_a, x_b, x_c, x_d, \ldots$ as the variable names for the corresponding gates in the drawing. (You may need additional variables.) Note, that a node $(\wedge, g)$ in the figure below denotes an `and` gate, where g is its label.



27.0.36.2 Scratch slide

27.0.36.3 3: Balancing vampires.

There are n vampires $p_1, \ldots, p_n$ in Champaign. ith vampire has integer score $w_i \geq 0$ of how well it can climb mountains. Task: Divide the vampires into two teams, and you want the division of teams to be as fair as possible. The score of a team is the sum of the scores of all the vampires in that team. Minimize the differences of the scores of the two teams. Assume that for all i, $w_i \leq W$.

(A) Given integers $\alpha, \beta \geq 0$, and T_α, T_β, such that $\alpha + \beta = n$, describe an algorithm, as fast as possible, to compute the partition into two teams, such that the first team has α players of total score T_α, and the second team has β players with total score T_β. What is the running time of your algorithm? (For any credit, it has to be polynomial in n and W.)
(To simplify things, you can solve the decision version problem first, and describe shortly how to modify it to yield the desired partition.)

27.0.36.4 Scratch slide

27.0.36.5 3 (B): Balancing vampires.

(B) Describe an algorithm, as fast as possible, to compute the scores of the two teams in an optimal division that is as balanced as possible, when requiring that the two teams have exactly the same number of players (assume n is even). What is the running time of your algorithm?

27.0.36.6 3 (C): Balancing vampires.

(C) State **formally** the decision version of the problem in (B), and prove that it is **NP-Complete**. (There are several possible solutions for this part – pick the one you find most natural. Note, that the teams must have the same number of players.)

27.0.36.7 4: MAX Cut and MAX 2SAT.

Problem: MAX Cut

Instance: Undirected graph G with n vertices and m edges, and an integer k. **Question:** Is there an undirected cut in G that cuts at least k edges?

Problem: MAX 2SAT

Instance: A 2CNF formula F, and an integer k. **Question:** Is there a truth assignment in F that satisfies at least k clauses.

You are given that **MAX Cut** is **NP-Complete**. Prove that **MAX 2SAT** is **NP-Complete** by a reduction to/from **MAX Cut** (be sure to do the reduction in the right direction!).

Hint: Think about how to encode a cut, by associating a boolean variable with each vertex of the graph. It might be a good idea to verify your answer by considering a graph with two vertices and a single edge between them and checking all possibilities for this case.

27.0.36.8 Scratch slide

27.0.36.9 5: Billboards are forever.

Consider a stretch of Interstate-57 that is m miles long. We are given an ordered list of mile markers, $x_1, x_2, \ldots, x_n$ in the range 0 to m, at each of which we are allowed to construct billboards (suppose they are given as an array $X[1 \ldots n]$). Suppose we can construct billboards for free, and that we are given an array $R[1 \ldots n]$, where $R[i]$ is the revenue we would receive by constructing a billboard at location $X[i]$. Given that state law requires billboards to be at least 5 miles apart, describe an algorithm, as fast as possible, to compute the maximum revenue we can acquire by constructing billboards.

What is the running time of your algorithm? Your algorithm has to be as fast as possible.

27.0.36.10 Scratch slide
27.0.36.11 6: Best edge ever.

You are given a directed graph G with n vertices and m edges. For every edge $e \in \mathsf{E}(\mathsf{G})$, there is an associated weight $w(e) \in \mathbb{R}$. For a path (not necessarily simple) π in G, its ***quality*** is $W(\pi) = \max_{e \in \pi} w(e)$. We are interested in computing the highest quality walk in G between two given vertices (say s and t). Either **prove** that computing such a walk is **NP-Hard**, or alternatively, provide an algorithm (and **prove** its correctness) for this problem (the algorithm has to be as fast as possible – what is the running time of your algorithm?).

27.0.36.12 Scratch slide
27.0.36.13 7: Dominate this.

You are given a set of intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$ on the real line (assume all with distinct endpoints) – they are given in arbitrary order (i.e., you can not assume anything on the ordering). Consider the problem of finding a set of intervals $\mathcal{K} \subseteq \mathcal{I}$, as small as possible, that dominates all the other intervals. Formally, $\mathcal{K}$ ***dominates*** $\mathcal{I}$, if for every interval $I \in \mathcal{I}$, there is an interval $K \in \mathcal{K}$, such that I intersects K.

Describe an algorithm (as fast as possible) that computes such a minimal dominating set of $\mathcal{I}$. What is the running time of your algorithm? Prove the correctness of your algorithm.

27.0.36.14 Scratch slide
27.0.36.15 8: Network flow.

You are given a network flow G (with integer capacities on the edges, and a source s and a sink t), and a maximum flow f on it (you can assume f is integral). You want increase the maximum flow in G by one unit by applying a single augmenting path to f. Naturally, to be able to do that, you must increase the capacity of some of the edges of G. In particular, for every edge $e \in E(G)$, there is an associated cost $\mathrm{cost}(e)$ of increasing its capacity by one unit. Describe an algorithm, that computes (as fast as possible), the cheapest collection of edges of G, such that if we increase the capacity on each of these edges by 1, then one can find an augmenting path to f that increases its flow by one unit. How fast is your algorithm?

Provide an argument that explains why your algorithm is correct.

Bibliography

[Chazelle, 2000] Chazelle, B. (2000). A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. Assoc. Comput. Mach.*, 47(6):1028–1047.

[Dinic, 1970] Dinic, E. A. (1970). Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280.

[Edmonds and Karp, 1972] Edmonds, J. and Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19(2):248–264.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34(3):596–615.

[Fredman and Willard, 1994] Fredman, M. L. and Willard, D. E. (1994). Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Sys. Sci.*, 48(3):533–551.

[Hoare, 1962] Hoare, C. A. R. (1962). Quicksort. *Comput. J.*, 5(1):10–15.

[Karger et al., 1995] Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *J. Assoc. Comput. Mach.*, 42(2):321–328.

[Menger, 1927] Menger, K. (1927). Zur allgemeinen kruventheorie. *Fund. Math.*, 10:96–115.

Index