CSE 548: (*Design and*) Analysis of Algorithms

Divide-and-conquer Algorithms

R. Sekar

Divide-and-Conquer: A versatile strategy

Steps

- Break a problem into subproblems that are smaller instances of the same problem
- Recursively solve these subproblems
- Combine these answers to obtain the solution to the problem

Benefits

Conceptual simplification

Speed up:

- rapidly (exponentially) reduce problem space
- exploit commonalities in subproblem solutions

Parallelism: Divide-and-conquer algorithms are amenable to parallelization

Locality: Their depth-first nature increases locality, extremely important for today's processors.

Topics

Binary Search

Problem: Find a key k in an ordered collection

Examples: Sorted array $A[n]$: Compare k with $A[n/2]$, then recursively search in $A[0 \cdots (n/2-1)]$ (if $k < A[n/2]$) or $A[n/2 \cdots n]$ (otherwise)

Binary search tree T: Compare k with $root(T)$, based on the result, recursively search left or right subtree of root.

B-Tree: Hybrid of the above two. Root stores an array M of m keys, and has $m+1$ children. Use binary search on M to identify which child can contain k, recursively search that subtree.

Exponentiation

- How many multiplications are required to compute x^n?
- Can we use a divide-and-conquer approach to make it faster?

ExpBySquaring(n, x)

if $n > 1$
  $y = ExpBySquaring(\lfloor n/2 \rfloor, x^2)$
  if $odd(n)$ $y = x * y$
  return y
else return x

Merge Sort

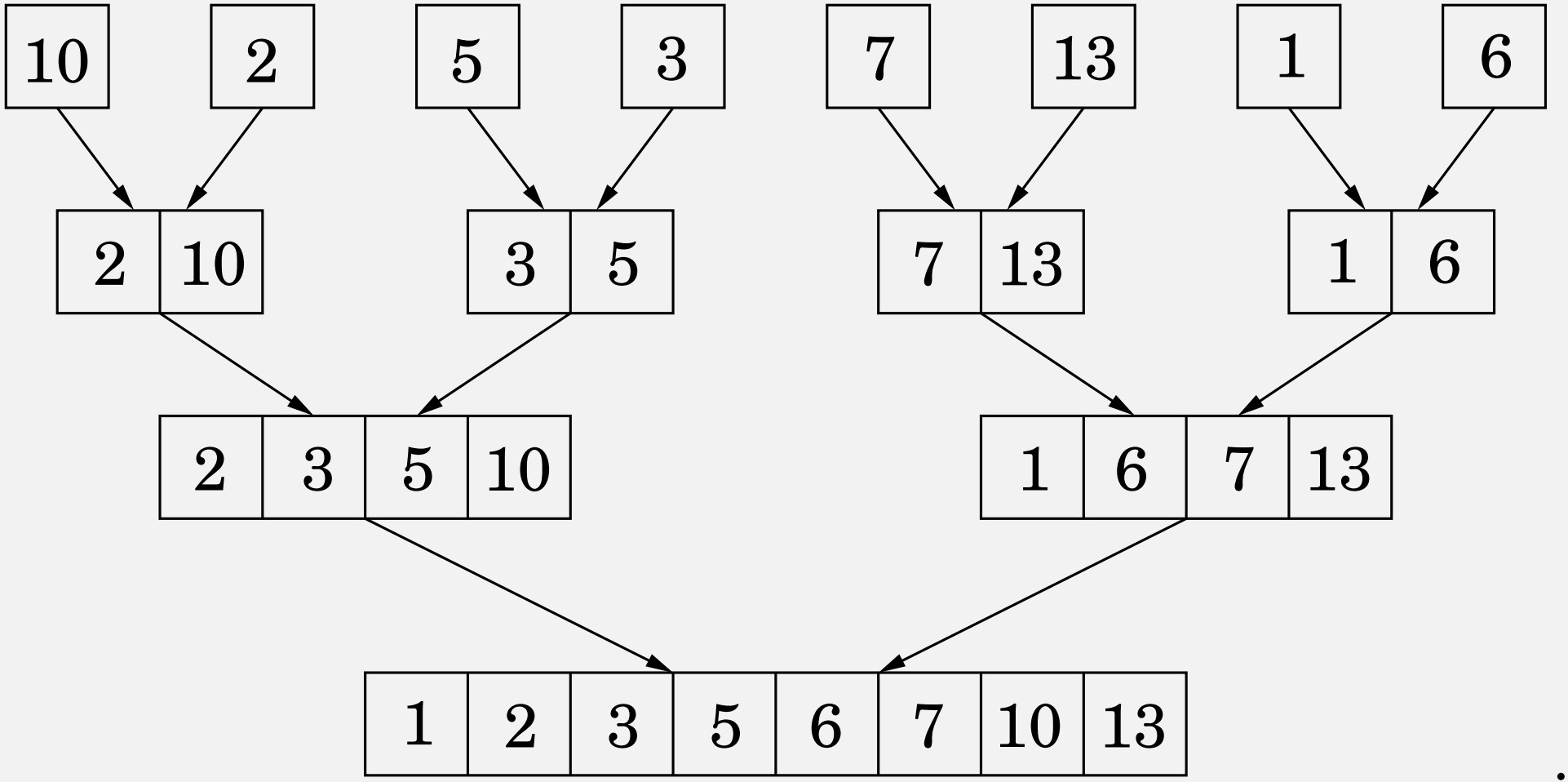
```
function mergesort(a[1...n])
Input:  An array of numbers a[1...n]
Output:  A sorted version of this array

if n > 1:
  return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋ + 1...n]))
else:
  return a
```

Merge Sort (Continued)

```
function merge(x[1...k], y[1...l])
if k = 0:   return y[1...l]
if l = 0:   return x[1...k]
if x[1] ≤ y[1]:
  return x[1] ∘ merge(x[2...k], y[1...l])
else:
  return y[1] ∘ merge(x[1...k], y[2...l])
```

Merge Sort Illustration



Merge sort time complexity

- *mergesort*(A) makes two recursive invocations of itself, each with an array half the size of A
- *merge*(A, B) takes time that is linear in $|A| + |B|$
- Thus, the runtime is given by the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- In divide-and-conquer algorithms, we often encounter recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

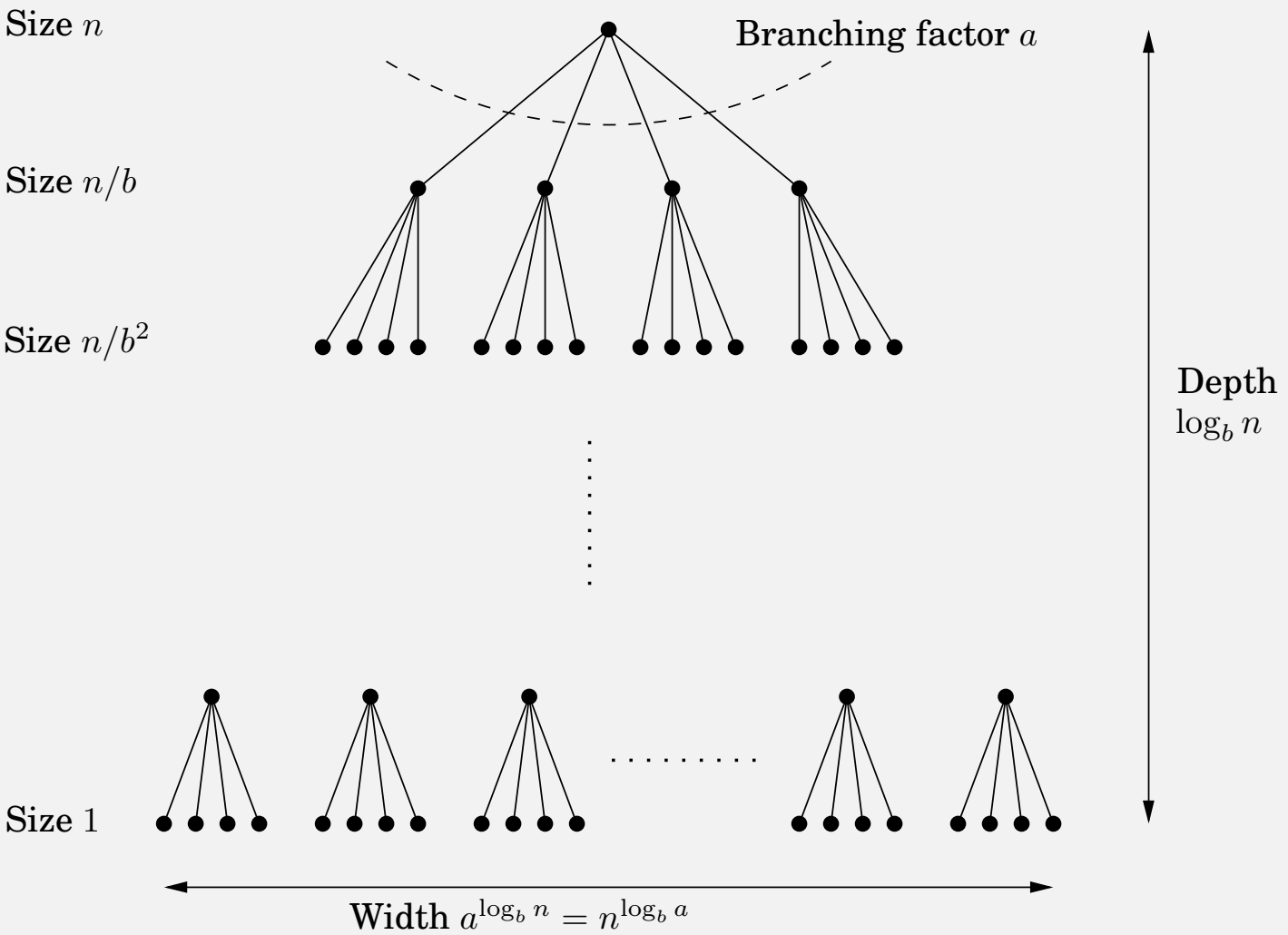
Can we solve them once for all?

Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for constants $a > 0, b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof of Master Theorem



Can be proved by induction, or by summing up the series where each term represents the work done at one level of this tree.

What if Master Theorem can't be appplied?

- Guess and check (prove by induction)
  - expand recursion for a few steps to make a guess
  - in principle, can be applied to any recurrence
- Akra-Bazzi method (not covered in class)
  - recurrences can be much more complex than that of Master theorem

More on time complexity: Fibonacci Numbers

function *fib1*(**int** n)

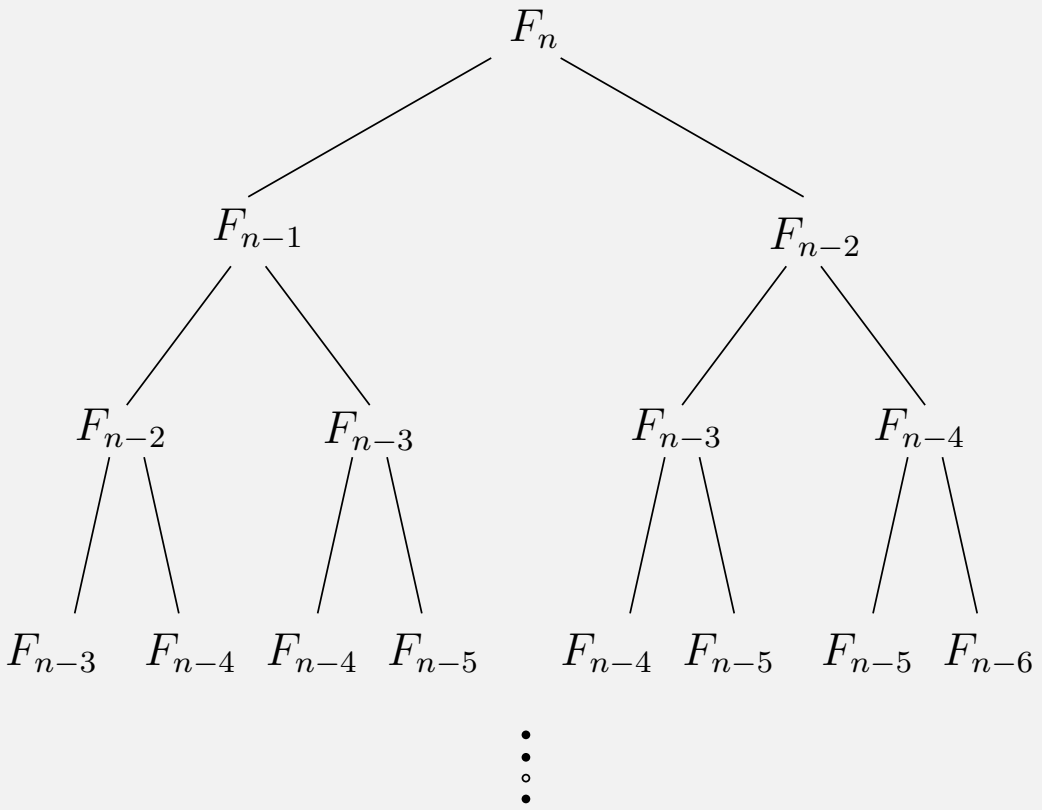
if $n = 0$ **return** 0;

if $n = 1$ **return** 1;

return $fib1(n-1) + fib1(n-2)$

- Is this algorithm correct? Yes: follows the defintion of Fibonacci
- What is its runtime?
  - $T(n) = T(n-1) + T(n-2) + 3$, with $T(k) \leq 2$ for $k < 2$
  - Solution is an exponential function ...
    - Prove this by induction!
- Can we do better?

Structure of calls to *fib*1



- Complete binary tree of depth n, contains 2^n calls to *fib*1
- But there are only n distinct Fibonacci numbers being computed!
  - Each Fibonacci number computed an exponential number of times!

Improved Algorithm for Fibonacci

function *fib2*(n)

int $f[max(2, n+1)]$;
$f[0] = 0; f[1] = 1;$
for $(i = 2; i \leq n; i{+}{+})$
    $f[i] = f[i-1] + f[i-2];$
return $f[n]$

- Linear-time algorithm!
- But wait! We are operating on very large numbers
  - n^{th} Fibonacci number requires approx. $0.694n$ bits
    - Prove *this* by induction!
  - Operation on k-bit numbers require k operations
  - i.e., Computing F_n requires $0.694n \log n$ operations

Quicksort

qs(A, l, h) /*sorts $A[l \ldots h]$*/

if $l >= h$ **return**;
$(h_1, l_2) =$
  partition(A, l, h);
qs(A, l, h_1);
qs(A, l_2, h)

partition(A, l, h)

$k =$ *selectPivot*(A, l, h); $p = A[k]$;
swap(A, h, k);
$i = l - 1$; $j = h$;
while *true* **do**
  do $i{+}{+}$ **while** $A[i] < p$;
  do $j{-}{-}$ **while** $A[j] > p$;
  if $i \geq j$ **break**;
  swap(A, i, j);
swap(A, i, h)
return $(j, i + 1)$

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$

where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Worst case: $n_1 = 1$, $n_2 = n - 1$. By master theorem, $T(n) = O(n^2)$

- *A fixed choice of pivot index, say, h, leads to worst-case behavior in common cases, e.g., input is sorted.*

Lucky/unlucky split: Alternate between best- and worst-case splits.

$$T(n) = n + T(1) + \boxed{\text{T(n-1)}} + n \quad \text{(worst case split)}$$

$$= n + 1 + \boxed{\text{(n-1) + 2T((n-1)/2)}} = 2n + 2T((n-1)/2)$$

which has an $O(n \log n)$ solution.

Three-fourths split:

$$T(n) = n + T(0.25n) + T(0.75n) \leq n + 2T(0.75n) = O(n \log n)$$

Average case analysis of *qs*

Define input distribution: All permutations equally likely

Simplifying assumption: all elements are distinct. (Nonessential assumption)

Set up the recurrence: When all permutations are qually likely, the selected pivot has an equal chance of ending up at the i^{th} position in the sorted order, for all $1 \leq i \leq n$. Thus, we have the following recurrence for the average case:

$$T(n) = n + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) + T(n-i))$$

Solve recurrence: Cannot apply the master theorem, but since it seems that we get an $O(n \log n)$ bound even in seemingly bad cases, we can try to establish a $cn \log n$ bound via induction.

Establishing average case of *qs*

- Establish base case. (Trivial.)
- Induction step involves summation of the form $\sum_{i=1}^{n-1} i \log i$.

  Attempt 1: bound $\log i$ above by $\log n$. (Induction fails.)

  Attempt 2: split the sum into two parts:

$$\sum_{i=1}^{n/2} i \log i + \sum_{i=n/2+1}^{n-1} i \log i$$

  and apply the approx. to each half. (Succeeds with $c \geq 4$.)

  Attempt 3: replace the summation with the upper bound

$$\int_{x=1}^{n} x \log x = \frac{x^2}{2}\left(\log x - \frac{1}{2}\right)\Bigg|_{x=1}^{n}$$

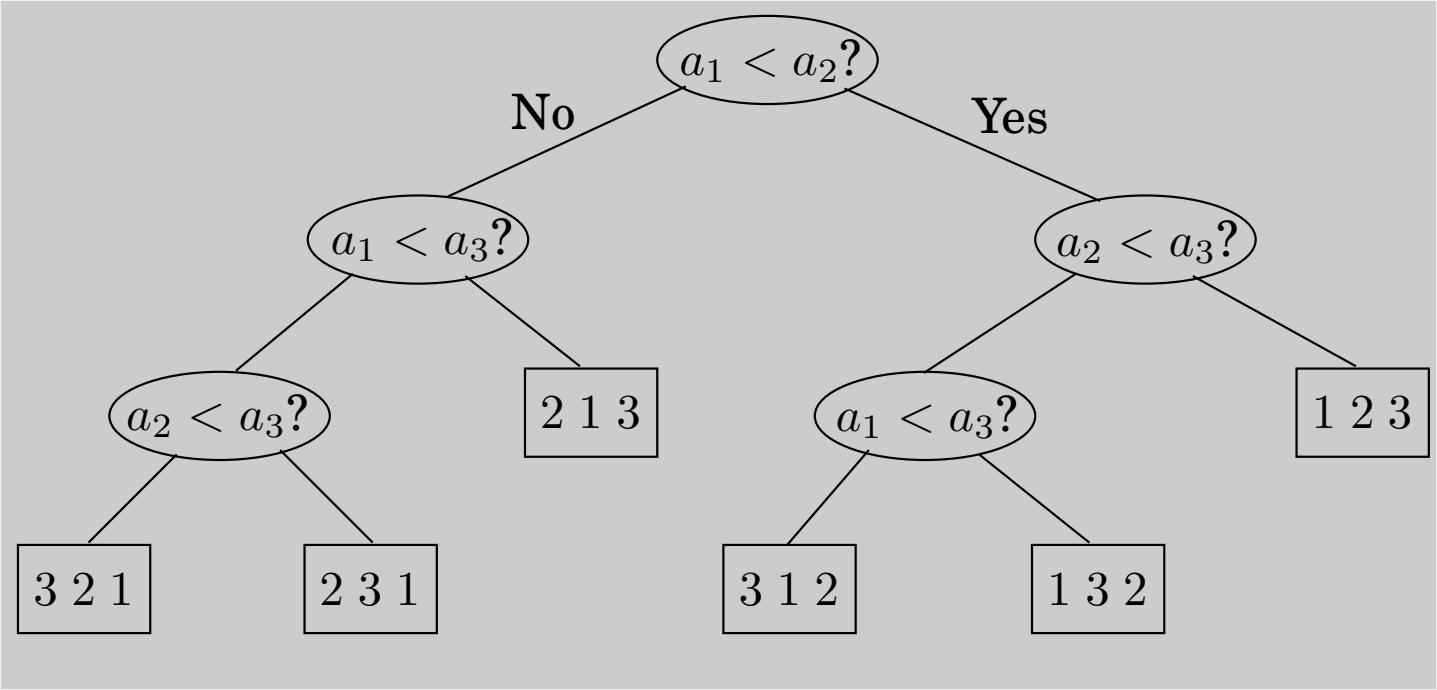
  (Succeeds with the constraint $c \geq 2$.)

Randomized Quicksort

- Picks a pivot at random
- What is its complexity?
  - For randomized algorithms, we talk about *expected complexity*, which is an average over all possible values of the random variable.
- If pivot index is picked uniformly at random over the interval $[l, h]$, then:
  - every array element is equally likely to be selected as the pivot
  - every partition is equally likely
  - thus, *expected* complexity of *randomized* quicksort is given by the same recurrence as the *average* case of *qs*.

Lower bounds for comparison-based sorting

- Sorting algorithms can be depicted as trees: each leaf identifies the input permutation that yields a sorted order.



- The tree has $n!$ leaves, and hence a height of $\log n!$. By Stirling's approximation, $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, so, $\log n! = O(n \log n)$
- No *comparison-based* sorting algorithm can do better!

Bucket sort

Overview

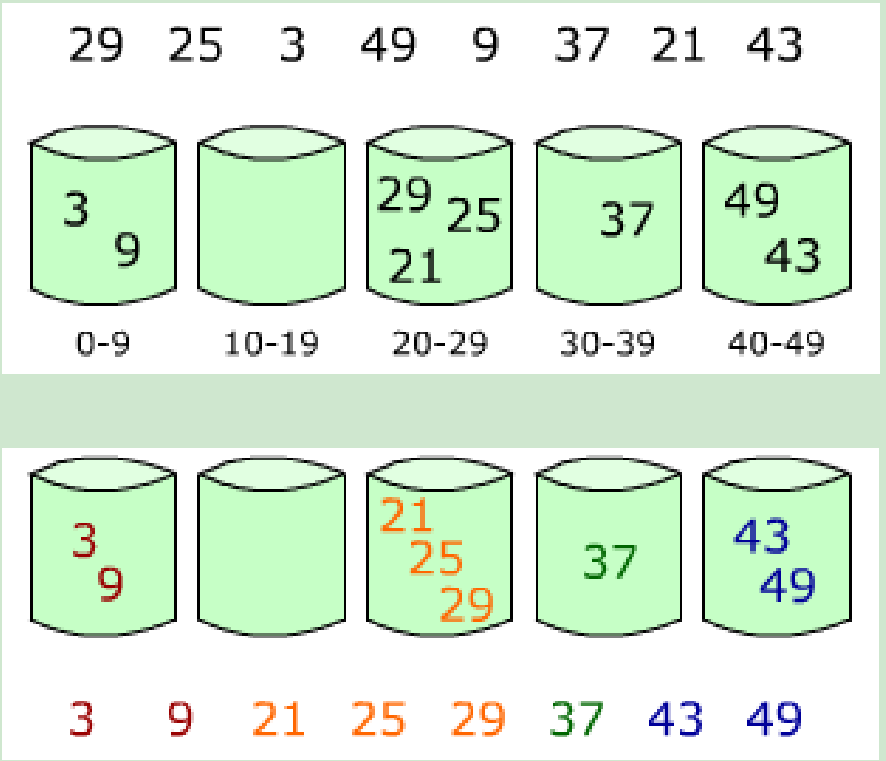
Divide: Partition input into intervals (buckets), based on key values

- Linear scan of input, drop into appropriate bucket

Recurse: Sort each bucket

Combine: Concatenate bin contents

Example



Images from Wikipedia commons

Bucket sort (Continued)

- Bucket sort generalizes quicksort to multiple partitions
  - Combination = concatenation
  - Worst case quadratic bound applies
  - But performance can be much better if input distribution is uniform.

    Exercise: What is the runtime in this case?
- Used by letter sorting machines in post offices

Counting Sort

Special case of bucket sort where each bin corresponds to an interval of size 1.

- No need to recurse. Divide = conquered!
- Makes sense only if range of key values is small (usually constant)
- Thus, counting sort can be done in $O(n)$ time!
  - *Hmm. How did we beat the $O(n \log n)$ lower bound?*

Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort

  LSD sorting: Sort first on least significant digit, and most significant digit last. After each round of counting sort, results can be simply concatenated, and given as input to the next stage.

  MSD sorting: Sort first on most significant digit, and least significant digit last. Unlike LSD sorting, we cannot concatenate after each stage.
- *Note:* Radix sort does not divide inputs into smaller subsets

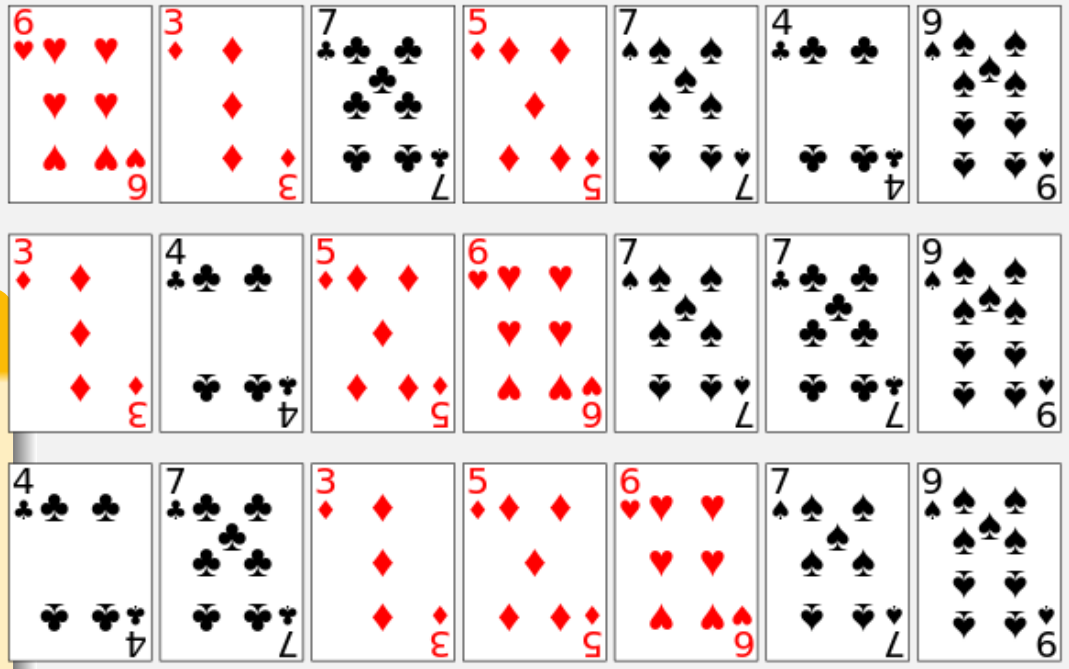
  If you think of input as multi-dimensional data, then we break down the problem to each dimension.

Stable sorting algorithms

- *Stable sorting algorithms:* don't change order of equal elements.
- Merge sort and LSD sort are stable. Quicksort is not stable.

Why is stability important?

- Effect of sorting on attribute A and then B is the same as sorting on $\langle B, A \rangle$
- LSD sort won't work without this property!
- Other examples: sorting spread sheets or tables on web pages



Images from Wikipedia Commons

Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
  - Requires a bit more care with variable-length strings.
    Starting point: use a special terminator character $t < a$ for all valid characters a.
- Easy to devise an $O(nl)$ algorithm, where n is the number of strings and l is the maximum size of any string.
  - But such an algorithm is *not* linear in input size.
- *Exercise:* Devise a linear-time string algorithm.

  Given a set $\mathcal{S}$ of strings, your algorithm should sort in $O(|\mathcal{S}|)$ time, where

$$|\mathcal{S}| = \sum_{s \in \mathcal{S}} |s|$$

Select k^{th} largest element

Obvious approach: Sort, pick k^{th} element — wasteful, $O(n \log n)$

Better approach: Recursive partitioning, search only on one side

$qsel(A, l, h, k)$

if $l = h$ **return** $A[l]$;
$(h_1, l_2) = partition(A, l, h)$;
if $k \leq h_1$
  return $qsel(A, l, h_1, k)$
else return $qsel(A, l_2, h, k)$

Complexity

Best case: Splits are even:
$T(n) = n + T(n/2)$, which has an $O(n)$ solution.

Skewed 10%/90% $T(n) \leq n + T(0.9n)$ — still linear

Worst case: $T(n) = n + T(n-1)$ — *quadratic!*

Worst-case $O(n)$ Selection

Intuition: Spend a bit more time to select a pivot that ensures reasonably balanced partitions

MoM Algorithm [Blum, Floyd, Pratt, Rivest and Tarjan 1973]

Time Bounds for Selection

by .

Manuel Blum, Robert W. Floyd, Vaughan Pratt,
Ronald L. Rivest, and Robert E. Tarjan

Abstract

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than $5.4305\dot{5}$ n comparisons are ever required. This bound is improved for

$O(n)$ Selection: MoM Algorithm

- Quick select (*qsel*) takes no time to pick a pivot, but then spends $O(n)$ to partition.
- Can we spend more time upfront to make a better selection of the pivot, so that we can avoid highly skewed splits?

Key Idea

- Use the selection algorithm itself to choose the pivot.
  - Divide into sets of 5 elements
  - Compute median of each set ($O(5)$, i.e., constant time)
  - Use selection recursively on these $n/5$ elements to pick their median
    - i.e., choose the median of medians (MoM) as the pivot
- Partition using MoM, and recurse to find kth largest element.

$O(n)$ Selection: MoM Algorithm

Theorem: MoM-based split won't be worse than 30%/70%

Result: Guaranteed linear-time algorithm!

Caveat: The constant factor is non-negligible; use as fall-back if random selection repeatedly yields unbalanced splits.

Selecting maximum element: Priority Queues

Heap

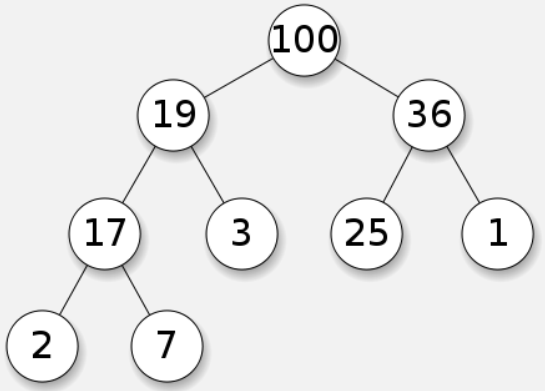
- A tree-based data structure for priority queues

  Heap property: H is a heap if for every subtree h of H

  $$\forall k \in keys(h) \quad root(h) \geq k$$

  where $keys(h)$ includes all keys appearing within h

  Note: No ordering of siblings or cousins
- Supports *insert*, *deleteMax* and *max*.
- Typically implemented using arrays, i.e., without an explicit tree data structure



Task of maintaining max is distributed to subsets of the entire set; alternatively, it can be thought of as maintaining several parallel queues with a single head.

Images from Wikimedia Commons

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

DeleteMax: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Insert: Append element to the end of array, fix up heap

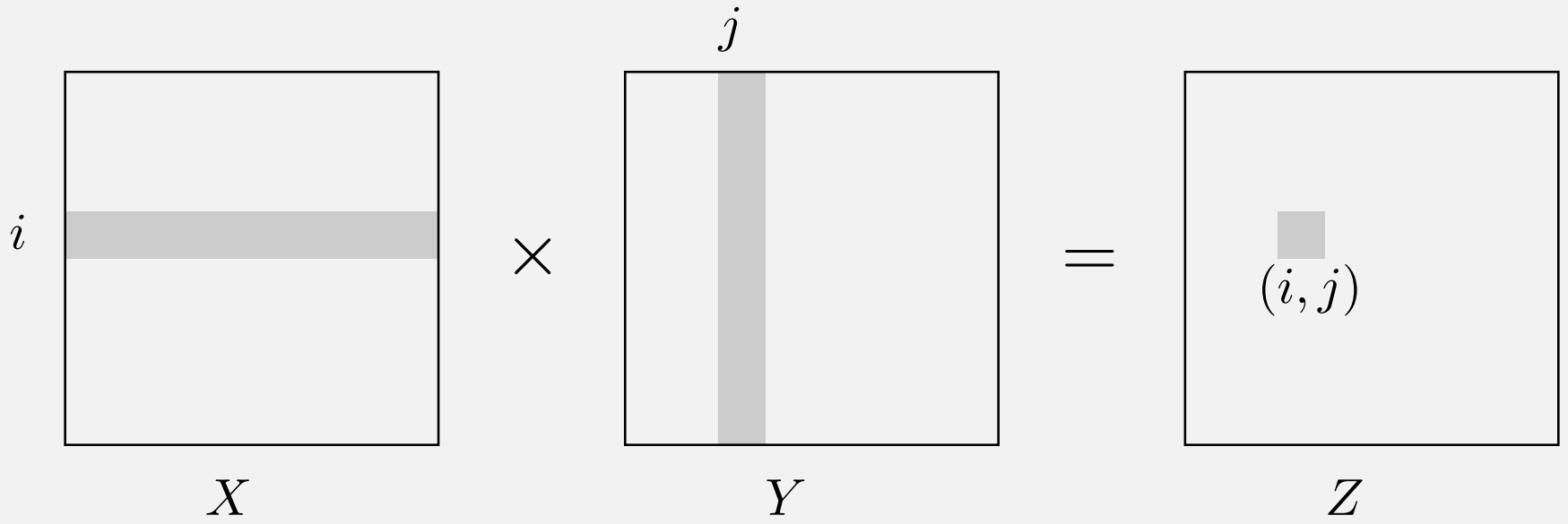
MkHeap: Fix up the entire heap. Takes $O(n)$ time.

Heapsort: $O(n \log n)$ algorithm, *MkHeap* followed by n calls to *DeleteMax*

Matrix Multiplication

The product Z of two $n \times n$ matrices X and Y is given by

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$ — leads to an $O(n^3)$ algorithm.



Divide-and-conquer Matrix Multiplication

Divide X and Y into four $n/2 \times n/2$ submatrices

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Recursively invoke matrix multiplication on these submatrices:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Divided, but did not conquer! $T(n) = 8T(n/2) + O(n^2)$, which is still $O(n^3)$

Strassen's Matrix Multiplication

Strassen showed that 7 multiplications are enough:

$$XY = \begin{bmatrix} P_6 + P_5 + P_4 - P_2 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{bmatrix} \quad \text{where}$$

$$P_1 = A(F - H) \qquad P_5 = (A + D)(E + H)$$
$$P_2 = (A + B)H \qquad P_6 = (B - D)(G + H)$$
$$P_3 = (C + D)E \qquad P_7 = (A - C)(E + F)$$
$$P_4 = D(G - E)$$

Now, the recurrence $T(n) = 7T(n/2) + O(n^2)$ has $O(n^{\log_2 7} = n^{2.81})$ solution!

Best-to-date complexity is about $O(n^{2.4})$, but this algorithm is not very practical.

Karatsuba's Algorithm

Same high-level strategy as Strassen — but predates Strassen.

Divide: n-digit numbers into halves, each with $n/2$-digits:

$$\begin{aligned} a &= \boxed{a_1}\boxed{a_0} = 2^{n/2}a_1 + a_0 \\ b &= \boxed{b_1}\boxed{b_0} = 2^{n/2}b_1 + b_0 \\ ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + b_1 a_0) + a_0 b_0 \end{aligned}$$

Key point — Instead of 4 multiplications, we can get by with 3 since:

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0$$

Recursively compute $a_1 b_1$, $a_0 b_0$ and $(a_1 + a_0)(b_1 + b_0)$.

Recurrence $T(n) = 3T(n/2) + O(n)$ has an $O(n^{\log_2 3} = n^{1.59})$ solution!

Note: This trick for using 3 (not 4) multiplications noted by Gauss (1777-1855) in the context of complex numbers.

Toom-Cook Multiplication

- Generalize Karatsuba
  - Divide into $n > 2$ parts
- Can be more easily understood when integer multiplication is viewed as a polynomial multiplication.

Integer Multiplication Revisited

- An integer represented using digits

  $$a_{n-1} \dots a_0$$

  over a base d (i.e., $0 \leq a_i < d$) is very similar to the polynomial

  $$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

  Specifically, the value of the integer is $A(d)$.
- Integer multiplication follows the same steps as polynomial multiplication:

  $$a_{n-1} \dots a_0 \times b_{n-1} \dots b_0 = (A(x) \times B(x))(d)$$

Polynomials: Basic Properties

Horner's rule

An n^{th} degree polynomial $\sum_{i=0}^{n} a_i x^i$ can be evaluated in $O(n)$ time:

$$((\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0)$$

Roots and Interpolation

- An n^{th} degree polynomial $A(x)$ has exactly n roots $r_1, \ldots, r_n$. In general, r_i's are complex and need not be distinct.
- It can be represented as a product of sums using these roots:

$$A(x) = \sum_{i=1}^{n} a_i x^i = \prod_{i=0}^{n} (x_i - r_i)$$

- Alternatively, $A(x)$ can be specified uniquely by specifying $n+1$ points (x_i, y_i) on it, i.e., $A(x_i) = y_i$.

Operations on Polynomials

Representation	Add	Mult
Coefficients	$O(n)$	$O(n^2)$
Roots	?	$O(n)$
Points	$O(n)$	$O(n)$

Note: Point representation is the best for computation! But usually, only the coefficients are given

Solution: Convert to point form by *evaluating* $A(x)$ at selected points.

But conversion defeats the purpose: requires $O(n)$ evaluations, each taking $O(n)$ time, thus we are back to $O(n^2)$ total time.

Toom (and FFT) Idea: Choose evaluation points judiciously to speed up evaluation

Matrix representation of Polynomial Evaluation

Given a polynomial

$$A(x) = \sum_{t=0}^{n-1} a_t x^n$$

choose m points $x_0, \ldots, x_m$ for its evaluation.

Evaluation can be expressed using matrix multiplication:

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Multiplication using Point Representation

- Let $A(x)$ and $B(x)$ be polynomials representing two numbers
- Evaluate both polynomials at chosen points $x_0, \ldots x_m$

$$P = \mathbf{X}A \qquad\qquad Q = \mathbf{X}B$$

where $P, \mathbf{X}, A, Q$ and B denote matrices as in last page

- Compute point-wise product

$$\begin{bmatrix} r_0 \\ \vdots \\ r_m \end{bmatrix} = \begin{bmatrix} p_0 * q_0 \\ \vdots \\ p_m * q_m \end{bmatrix}$$

- Compute polynomial C corresponding to R

$$R = \mathbf{X}C \Rightarrow C = \mathbf{X}^{-1}R$$

- To avoid overflow, m should be $degree(A) + degree(B) + 1$ for R

Improving complexity ...

- Key problem: Complexity of computing $\mathbf{X}$ and its inverse $\mathbf{X}^{-1}$
- Toom strategy:
  - Use low-degree polynomials e.g., Toom-2 = Karatsuba uses degree 1.
    - represents an n-bit number as a 2-digit number over a large base $d = 2^{n/2}$
  - Fix evaluation points for a given degree polynomial so that $\mathbf{X}$ and $\mathbf{X}^{-1}$ can be precomputed
    - For Toom-2, $x_0 = 0, x_1 = 1, x_2 = \infty$. (Define $A(\infty) = a_{n-1}$.)
  - Choose points so that expensive multiplications can be avoided while computing $P = \mathbf{X}A$, $Q = \mathbf{X}B$ and $C = \mathbf{X}^{-1}R$
- Toom-N on n-digit numbers needs $2N-1$ multiplications on n/N digit numbers:

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

which, by Master theorem, has a solution $O(n^{\log_N(2N-1)})$ solution

Karatsuba revisited as Toom-2

Given evaluation points $x_0 = 0, x_1 = 1, x_2 = \infty$,

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{X}A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_0 + a_1 \\ a_1 \end{bmatrix}$$

Similarly

$$\mathbf{X}B = \begin{bmatrix} b_0 \\ b_0 + b_1 \\ b_1 \end{bmatrix}$$

Point-wise multiplication yields:

$$R = \begin{bmatrix} a_0 b_0 \\ (a_0 + a_1)(b_0 + b_1) \\ a_1 b_1 \end{bmatrix}$$

and so on ...

Limitations of Toom

- In principle, complexity can be reduced to $n^{1+\epsilon}$ for arbitrarily small positive ϵ by increasing N
- In reality, the algorithm itself depends on the choice of N. Specifically, constant factors involved increase rapidly with N.
- As a practical matter, $N = 4$ or 5 is where we stop.
- Question: Can we go farther?

FFT and Schonhage-Strassen

- Key idea: evaluate polynomial on the complex plane
- Choose powers of Nth complex root of unity as the points for evaluation
- Enables sharing of operations in computing $\mathbf{X}A$ so that it can be done in $O(N \log N)$ time, rather than $O(N^2)$ time needed for the naive matrix-multiplication based approach

FFT to the Rescue!

Matrix form of DFT and interpretation as polynomial evaluation:

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{N-1} \end{bmatrix}$$

- *Voila!* FFT computes $A(x)$ at N points ($x_i = \omega^i$) in $O(N \log N)$ time!
- $O(N \log N)$ integer multiplication

Convert to point representation using FFT	$O(N \log N)$
Multiply on point representation	$O(N)$
Convert back to coefficients using FFT^{-1}	$O(N \log N)$

FFT to the Rescue!

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

FFT can be thought of as a clever way to choose points:

- Evaluations at many distinct points "collapse" together
- This is why we are left with $2T(n/2)$ work after division, instead of $4T(n/2)$ for a naive choice of points.

FFT-based multiplication: More careful analysis ...

- Computations on complex or real numbers can lose precision.
  - For integer operations, we should work in some other ring — usually, we choose a ring based on modulo arithmetic.
  - Ex: in mod 33 arithmetic, 2 is the 10^{th} root of 1, i.e., $2^{10} \equiv 1 \mod 33$

    More generally, 2 is the nth root of unity modulo $(2^{n/2}+1)$
- Point-wise additions and multiplications are not $O(1)$.
  - We are adding up to n numbers ("digits") — we need $\Omega(\log n)$ bits
  - So, total cost increases by at least $\log n$, i.e., $O(n \log^2 n)$.
- [Schonhage-Strassen '71] developed $O(n \log n \log\log n)$ algorithm: recursively apply their technique for "inner" operations.

Integer Multiplication Summary

- Algorithms implemented in libraries for arbitrary precision arithmetic, with applications in public key cryptography, computer algebra systems, etc.
- GNU MP is a popular library, uses various algorithms based on input size: naive, Karatsuba, Toom-3, Toom-4, or Schonhage-Strassen (at about 50K digits).
- Karatsuba is Toom-2. Toom-N is based on
  - Evaluating a polynomial at $2N$ points,
  - performing point-wise multiplication, and
  - interpolating to get back the polynomial, while
  - minimizing the operations needed for interpolation

Fast Fourier Transformation

One of the most widely used algorithms — yet most people are unaware of its use!

Solving differential equations: Applied to many computational problems in engineering, e.g., heat transfer

Audio: MP3, digital audio processors, music/speech synthesizers, speech recognition, ...

Image and video: JPEG, MPEG, vision, ...

Communication: modulation, filtering, radars, software-defined radios, H.264, ...

Medical diagnostics: MRI, PET, ultrasound, ...

Quantum computing: See text Ch. 10

Other: Optics, data compression, seismology, ...

CSE 548: (*Design and*) Analysis of Algorithms

Graphs

R. Sekar

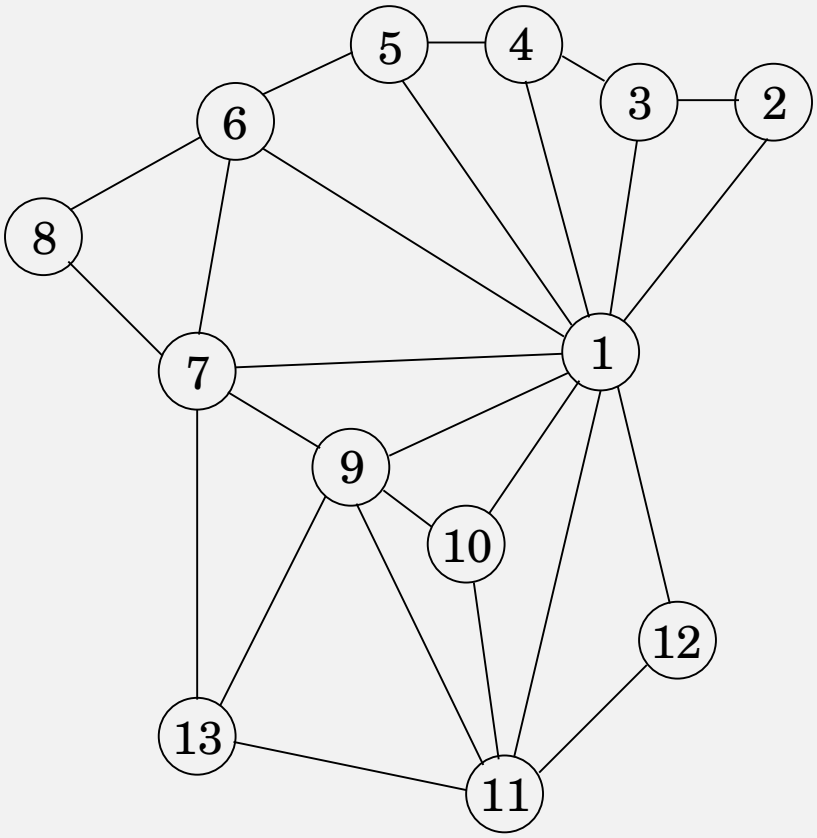
Overview

- Graphs provide a concise representation of a range problems

  Map coloring – more generally, resource contention problems

  Networks — communication, traffic, social, biological, ...

(b)



Definition and Representations

A *graph* $G = (V, E)$, where V is a set of vertices, and E a set of edges. An edge e of the form (v_1, v_2) is said to span vertices v_1 and v_2. The edges in a *directed graph* are directed.

A $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and E' includes every edge in E between vertices in V'.

Adjacency matrix

A graph $(V = \{v_1, \ldots, v_n\}, E)$ can be represented by an $n \times n$ matrix a, where $a_{ij} = 1$ iff $(v_i, v_j) \in E$

Adjacency list

Each vertex v is associated with a linked list consisting of all vertices u such that $(v, u) \in E$.

Note that adjacency matrix uses $O(n^2)$ storage, while adjacency list uses $O(|V| + |E|)$ storage. Both can represent directed as well as undirected graphs.

Depth-First Search (DFS)

- A technique for traversing all vertices in the graph
- Very versatile, forms the linchpin of many graph algorithms

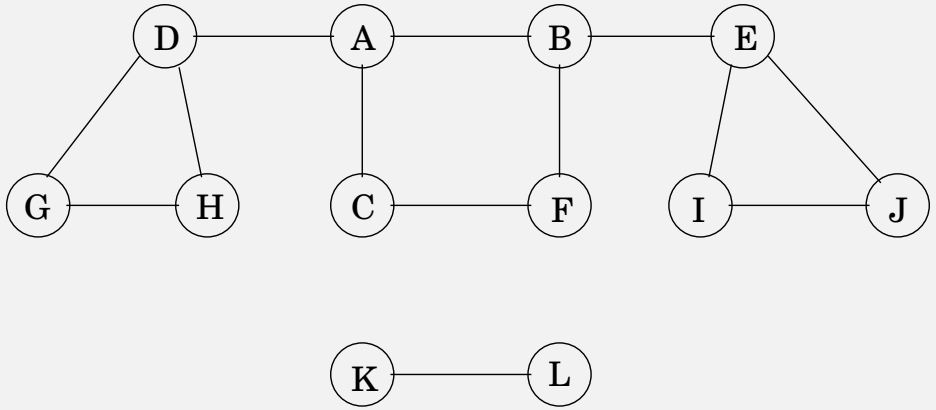
dfs(V, E)

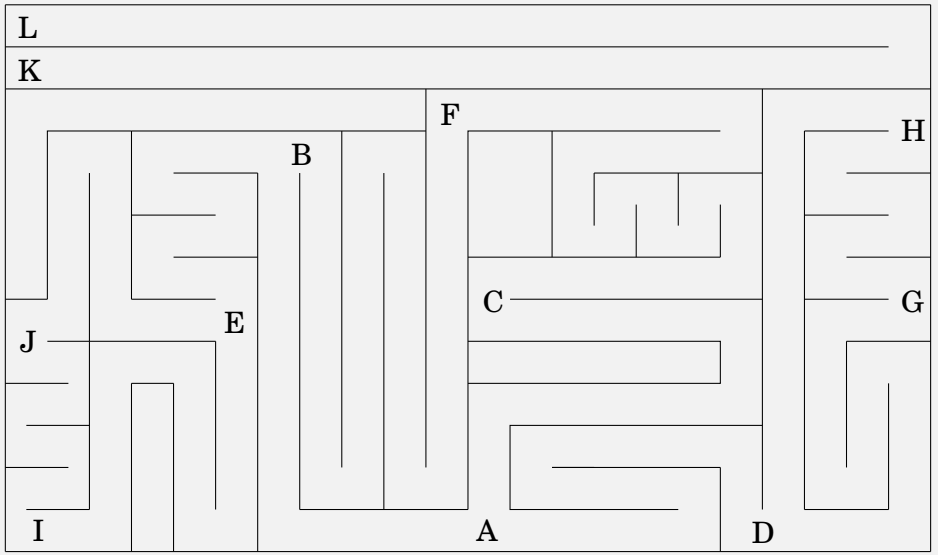
foreach $v \in V$ **do** $visited[v] = false$
foreach $v \in V$ **do**
  if not $visited[v]$ **then** *explore*(V, E, v)

explore(V, E, v)

$visited[v] = true$
previsit(v) /*A placeholder for now*/
foreach $(v, u) \in E$ **do**
  if not $visited[u]$ **then** *explore*(G, V, u)
postvisit(v) /*Another placeholder*/

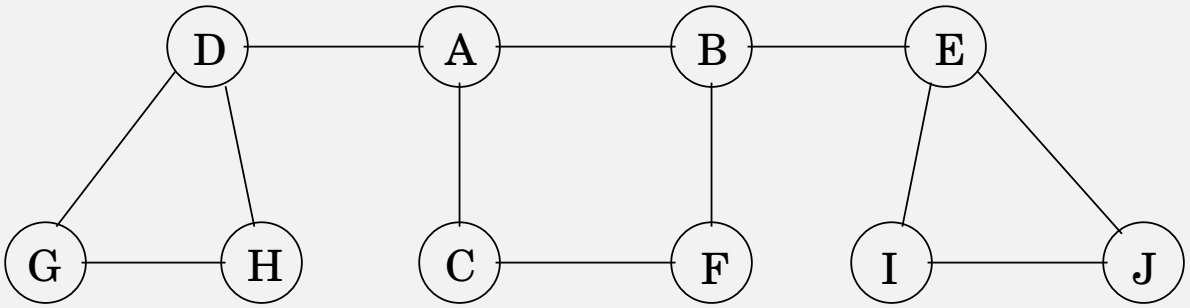
Graphs, Mazes and DFS

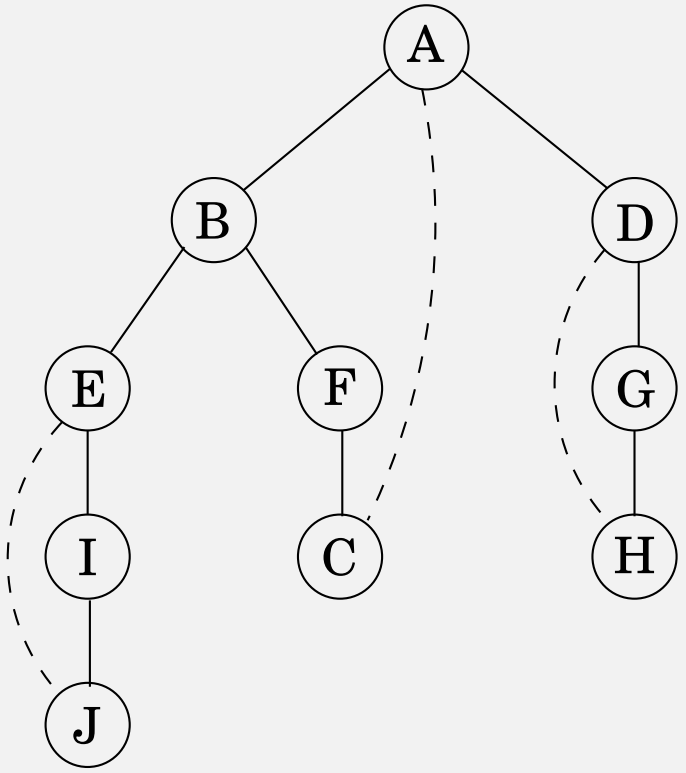




If a maze is represented as a graph, then DFS of the graph amounts to an exploration and mapping of the maze.

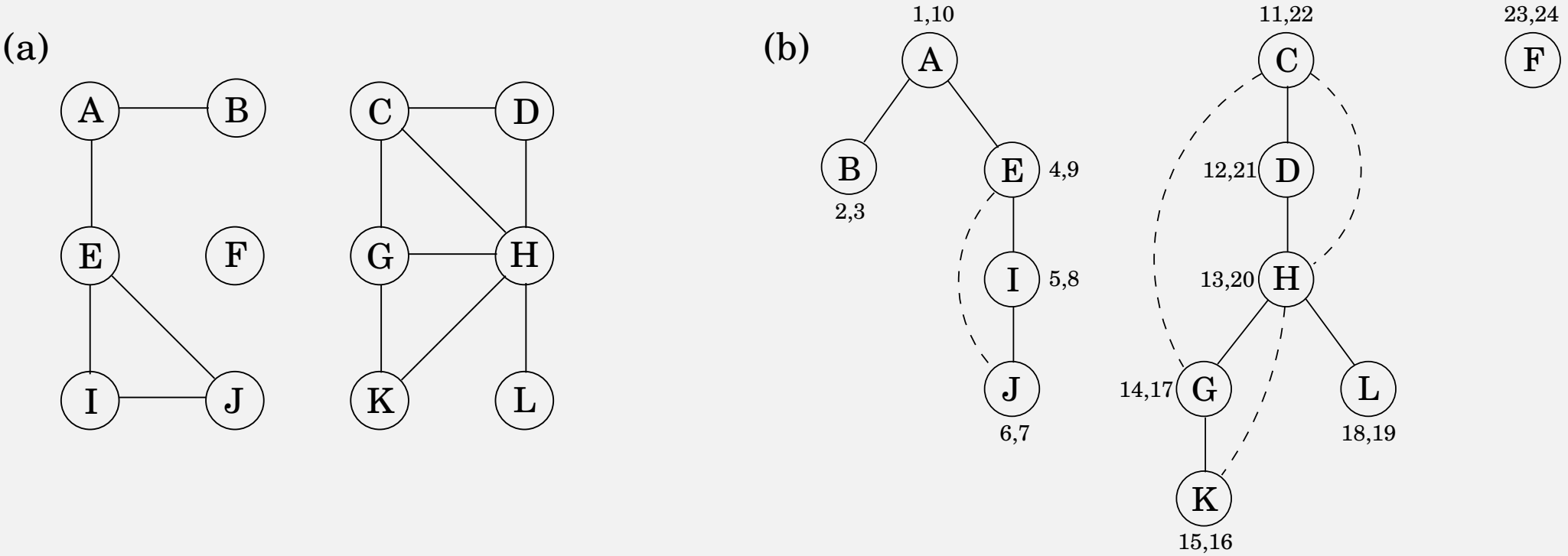
A graph and its DFS tree





DFS uses $O(|V|)$ space and $O(|E| + |V|)$ time.

DFS and Connected Components



A *connected component* of a graph is a maximal subgraph where there is path between any two vertices in the subgraph, i.e., it is a maximal *connected subgraph*.

DFS Numbering

Associate post and pre numbers with each visited node by defining *previsit* and *postvisit*

previsit(v)

$pre[v] = clock$

$clock{+}{+}$

postvisit(v)

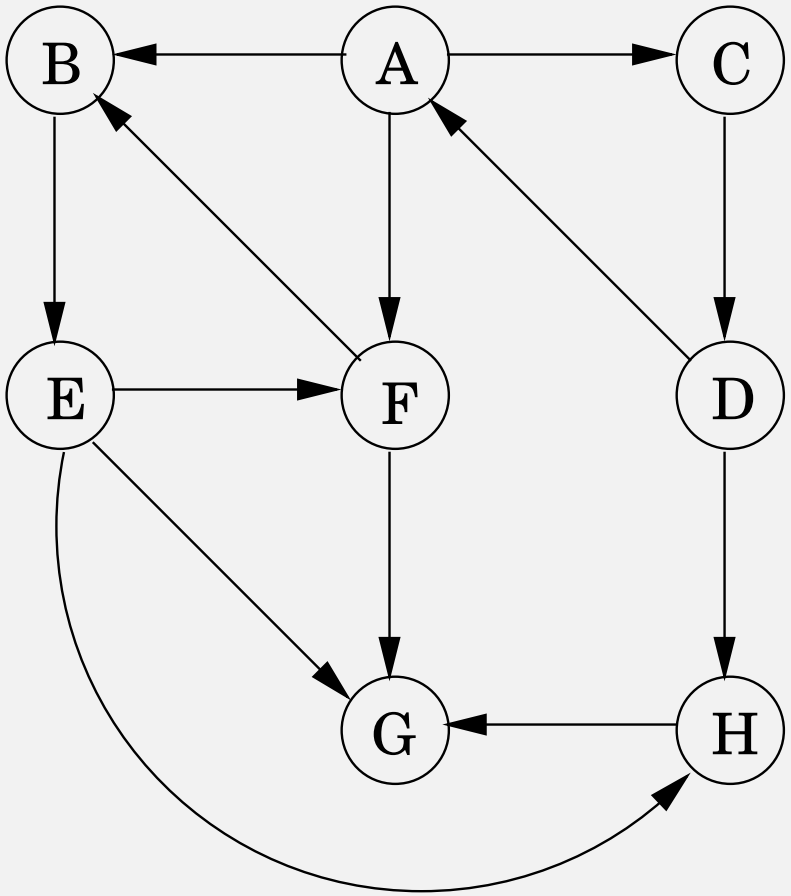
$post[v] = clock$

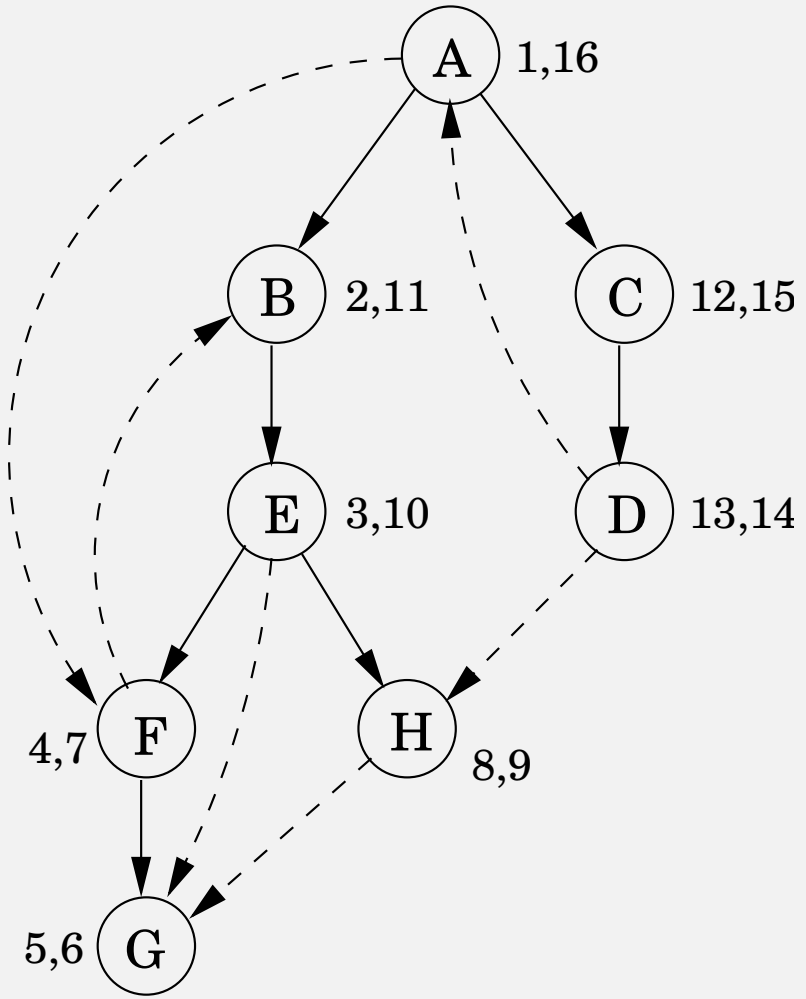
$clock{+}{+}$

Property

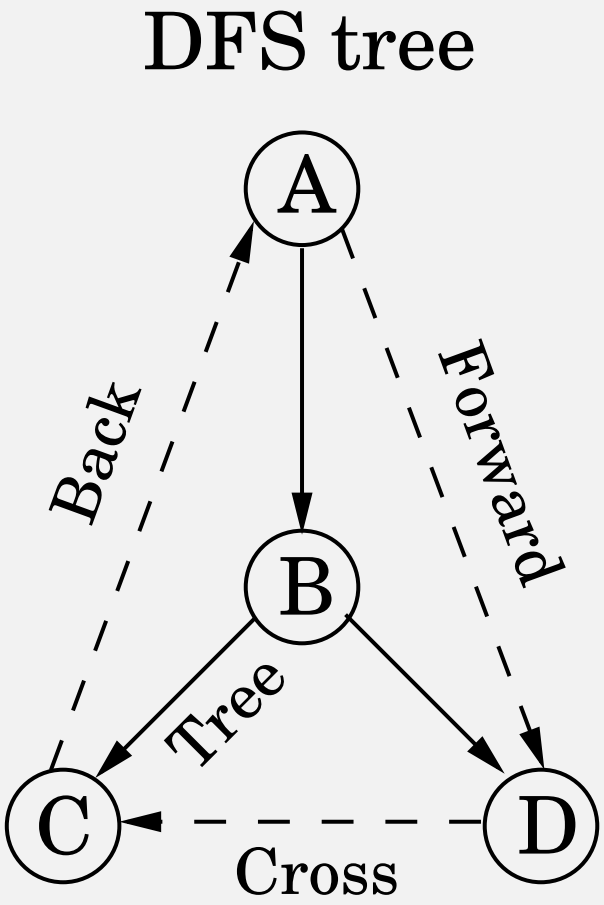
For any two vertices u and v, the intervals $[pre[u], post[u]]$ and $[pre[v], post[v]]$ are either disjoint, or one is contained entirely within another.

DFS of Directed Graph





DFS and Edge Types



`pre/post` *ordering for* (u, v)	*Edge type*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

No cross edges in undirected graphs!

Back and forward edges merge

Directed Acyclic Graphs (DAGs)

A directed graph that contains no cycles.

Often used to represent (acyclic) dependencies, partial orders,...

Property (DAGs and DFS)

- *A directed graph has a cycle iff its DFS reveals a back edge.*
- *In a dag, every edge leads to a vertex with lower post number.*
- *Every dag has at least one source and one sink.*

Strongly Connected Components (SCC)

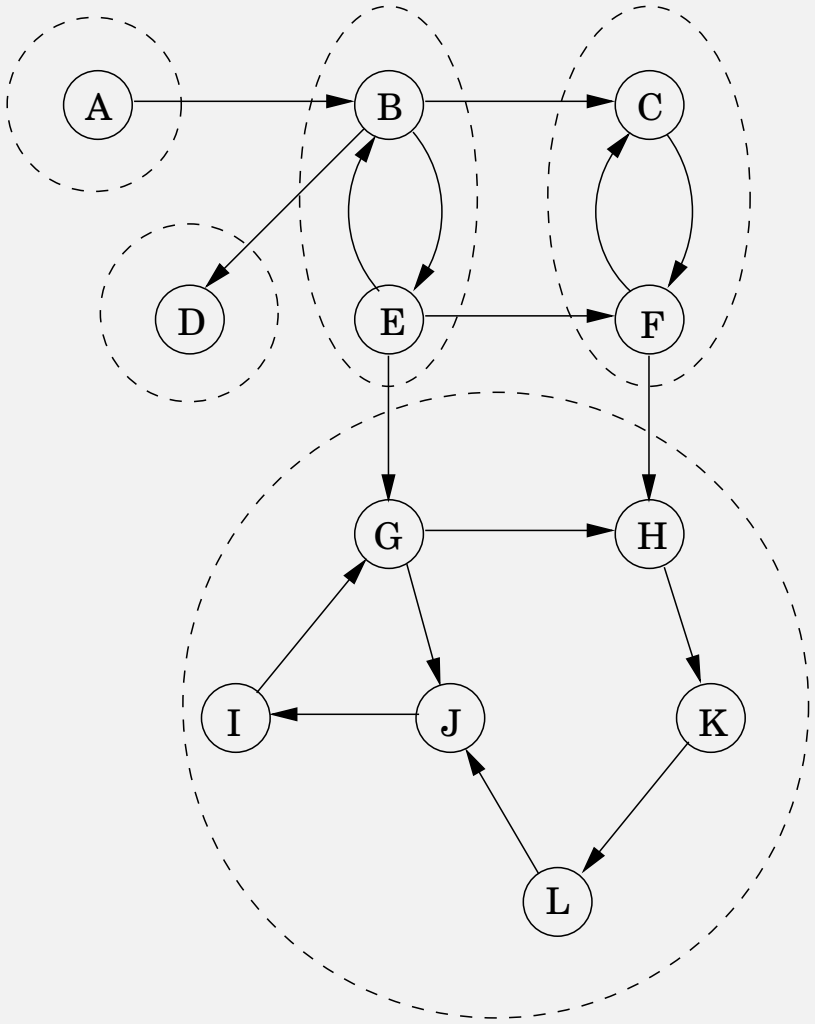
Analog of connected components for undirected graphs

Definition (SCC)

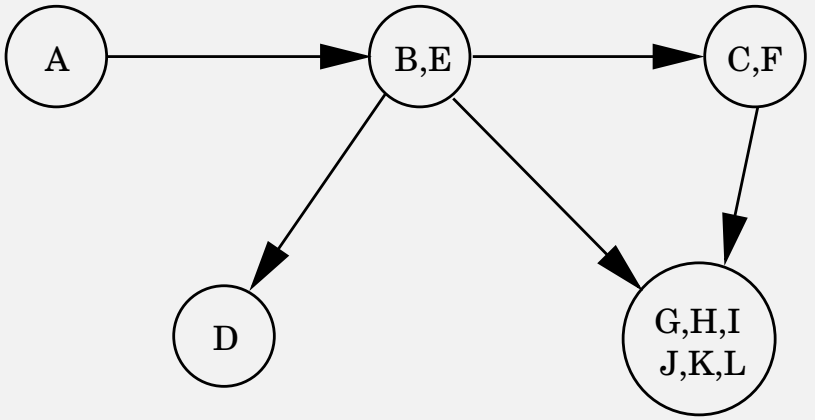
- *Two vertices u and v in a directed graph are* connected *if there is a path from u to v and vice-versa.*
- *A directed graph is* strongly connected *if any pair of vertices in the graph are connected.*
- *A subgraph of a directed graph is said to be an SCC if it is a maximal subgraph that is strongly connected.*

SCCs are also similar to biconnected components!

SCC Example



(b)



The textbook describes an algorithm for computing SCC in linear-time using DFS.

Breadth-first Search (BFS)

- Traverse the graph by "levels"
  - $BFS(v)$ visits v first
  - Then it visits all immediate children of v
  - then it visits children of children of v, and so on.
- As compared to DFS, BFS uses a queue (rather than a stack) to remember vertices that still need to be explored

BFS Algorithm

bfs(V, E, s)

foreach $u \in V$ **do** $visited[u] = false$

$q = \{s\}$; $visited[s] = true$

while q is nonempty **do**

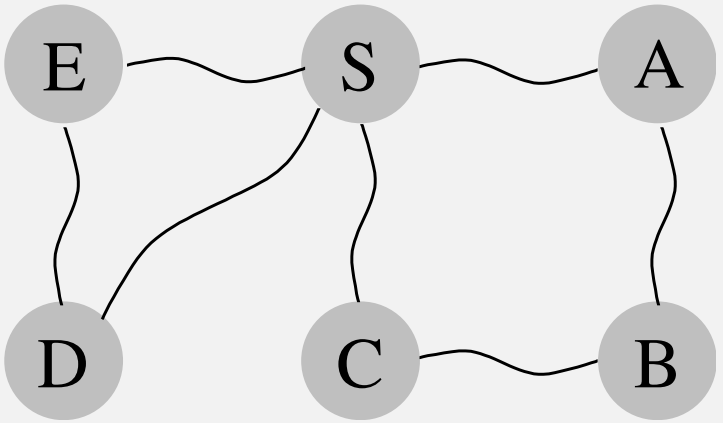
  $u = deque(q)$

  foreach edge $(u, v) \in E$ **do**

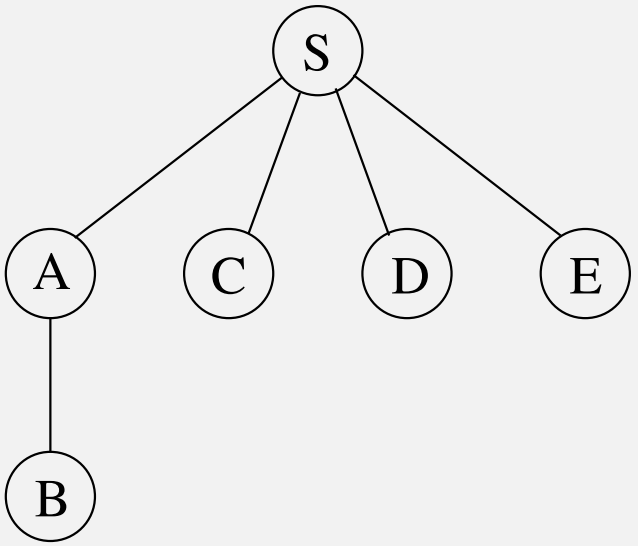
    if not $visited[v]$ **then**

      $queue(q, v)$; $visited[v] = true$

BFS Algorithm Illustration



Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



Shortest Paths and BFS

BFS automatically computes shortest paths!

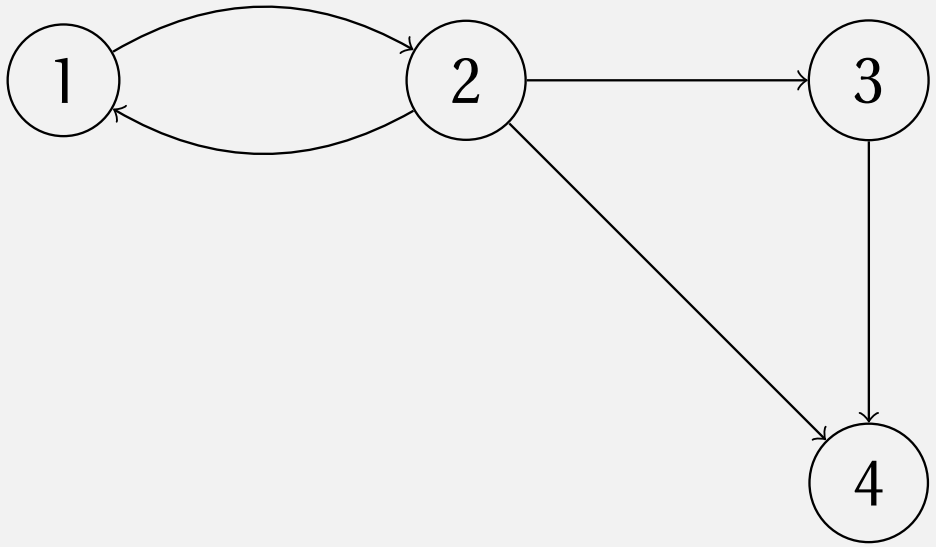
bfs(V, E, s)

foreach $u \in V$ **do** $dist[u] = \infty$
$q = \{s\}$; $dist[s] = 0$
while q is nonempty **do**
  $u = deque(q)$
  foreach edge $(u, v) \in E$ **do**
    if $dist[v] = \infty$ **then**
      $queue(q, v)$; $dist[v] = dist[u] + 1$

But not all paths are created equal! We would like to compute shortest weighted path — a topic of future lecture.

Graph paths and Boolean Matrices

A graph and its boolean matrix representation



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Graph paths and Boolean Matrices

- Let A be the adjacency matrix for a graph G, and $B = A \times A$. Now, $B_{ij} = 1$ iff there is path in the graph of length 2 from v_i to v_j
- Let $C = A + B$. Then $C_{ij} = 1$ iff there is path of length ≤ 2 between v_i and v_j
- Define $A^* = A^0 + A^1 + A^2 + \cdots$. If $D = A^*$ then $D_{ij} = 1$ iff v_j is reachable from v_i.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Shortest paths and Matrix Operations

- Redefine operations on matrix elements so that $+$ becomes *min*, and $*$ becomes integer addition.
- $D = A^*$ then $D_{ij} = k$ iff the shortest path from v_j to v_i is of length k

CSE 548: (*Design and*) Analysis of Algorithms

Greedy Algorithms

R. Sekar

Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost
- *Greedy strategy:* make a locally optimal choice, or simply, what appears best at the moment
- Often, *locally optimality* $\not\Rightarrow$ *global optimality*
- So, use with a great deal of care
  - *Always need to prove optimality*
- If it is unpredictable, why use it?
  - *It simplifies the task!*

Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for x cents ($0 < x < 100$) using *minimum number of coins.*

Greedy solution

makeChange(x)

if ($x = 0$) **return**

Let y be the largest denomination that satisfies $y \leq x$

Issue $\lfloor x/y \rfloor$ coins of denomination y

makeChange($x \mod y$)

- Show that it is optimal
- Is it optimal for arbitrary denominations?

When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

What does this mean for the coin change problem?

Optimal substructure

The optimal solution contains optimal solutions to subproblems.

Implies that a greedy algorithm can invoke itself recursively after making a greedy choice.

Knapsack Problem

- A sack that can hold a maximum of x lbs
- You have a choice of items you can pack in the sack
- Maximize the combined "value" of items in the sack

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable

Greedy choice: pick item that maximizes calories/lb

Will a greedy algorithm work, with $x = 5$?

Fractional Knapsack

Greedy choice property

Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

Optimal substructure

After taking as much of the item with jth maximal value/weight, suppose that the knapsack can hold y more lbs.
Then the optimal solution for the problem includes the optimal choice of how to fill a knapsack of size y with the remaining items.

Does not work for 0-1 knapsack because greedy choice property does not hold.

0-1 knapsack is NP-hard, but a pseudo-polynomial algorithm is

Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices V in the graph
- A subset of E such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

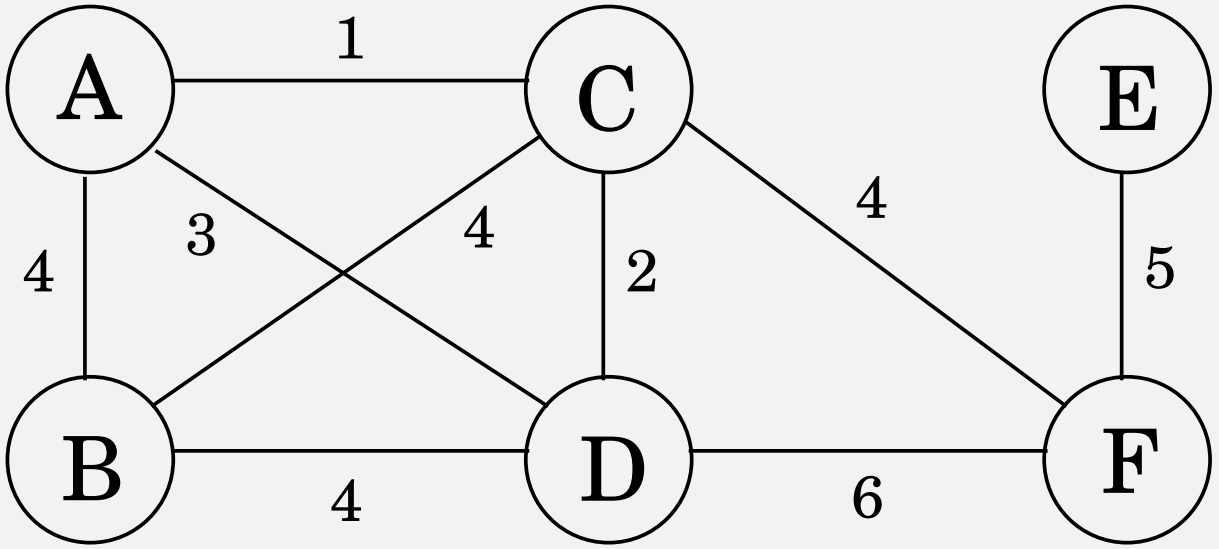
- A maximal subset of E such that the subgraph has no cycles
- A subset of E with $|V| - 1$ edges such that the subgraph is connected
- A subset of E such that there is a unique path between any two vertices in the subgraph

Minimal Spanning Tree (MST)

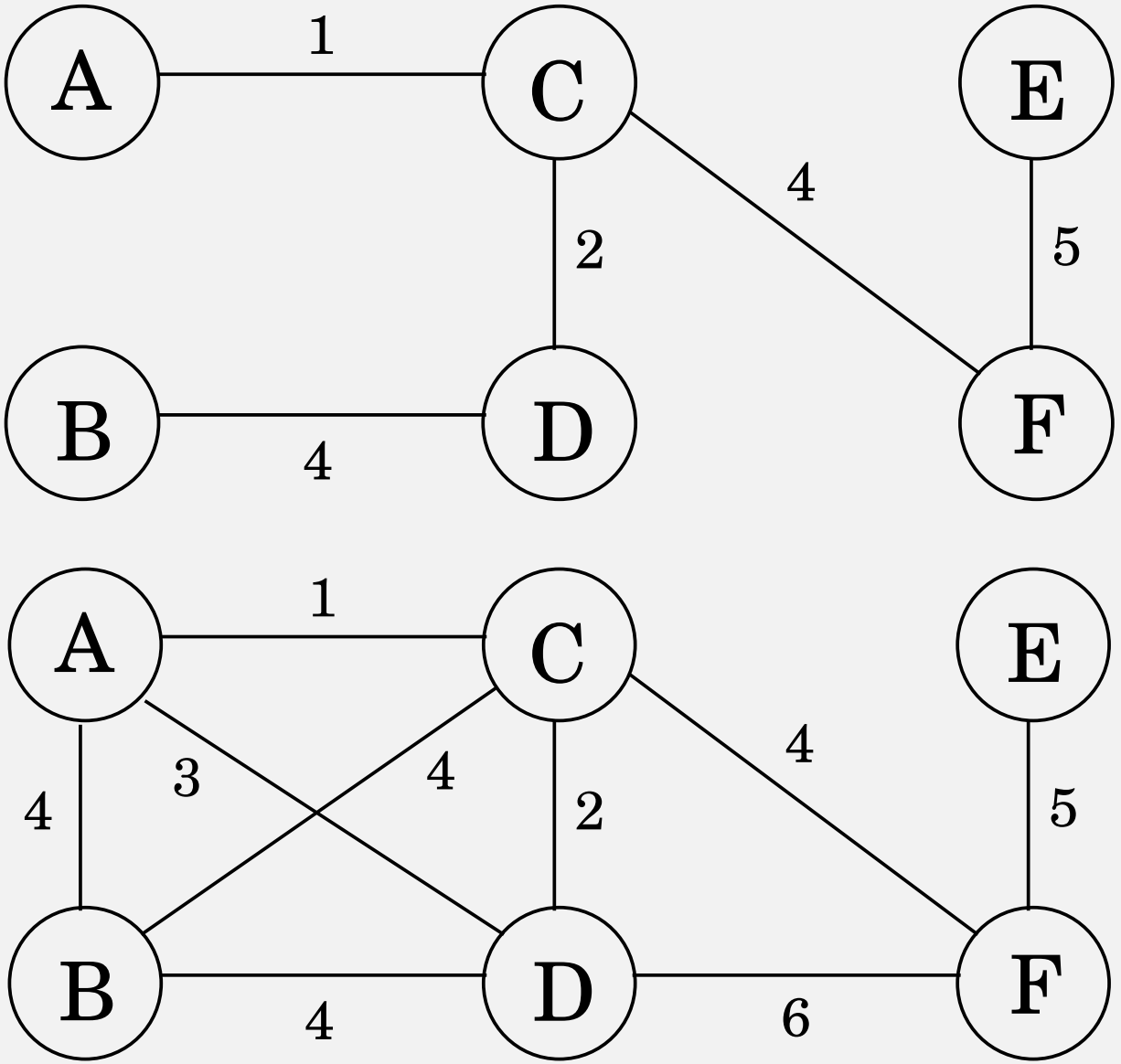
A spanning tree with *minimal cost.* Formally:

Input: An undirected graph $G = (V, E)$, a cost function $w : E \to \mathbb{R}$.

Output: A tree $T = (V, E')$ such that $E' \subseteq E$ that minimizes $\sum_{e \in E'} w(e)$



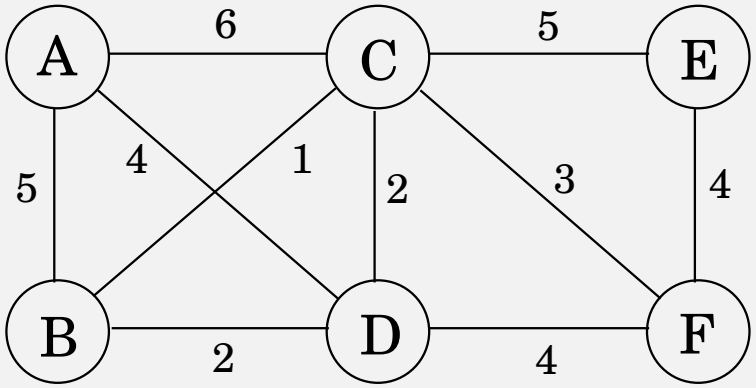
Minimal Spanning Tree (MST)

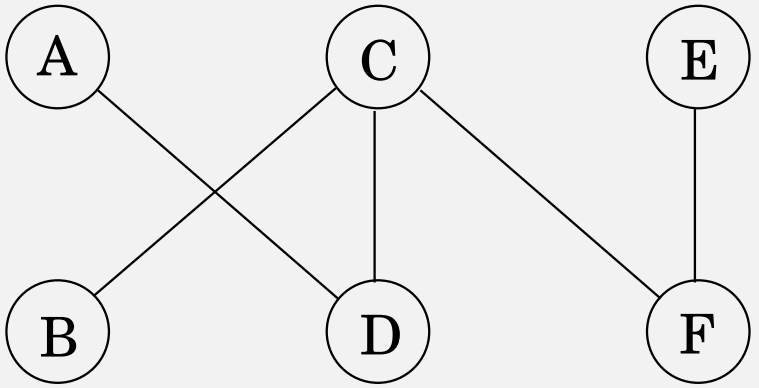


Kruskal's algorithm

- Start with the empty set of edges
- Repeat: add lightest edge that doesn't create a cycle

Adds edges B—C, C—D, C—F, A—D, E—F





Kruskal's algorithm

$MST(V, E, w)$

$X = \phi$

$Q = priorityQueue(E)$ // from min to max weight

while Q is nonempty

  $e = deleteMin(Q)$

  if e connects two disconnected components in (V, X)

    $X = X \cup \{e\}$

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's algorithm are included in some *minimal* spanning tree T

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that i+1th edge chosen by Kruskal's is in the MST T from induction hypothesis, i.e., prove greedy choice property.

- Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal's.
- T is a spanning tree: must include a unique path from v to w
- At least one edge e' on this path is not in X, the set of edges chosen in the first i steps by Kruskal's. (Otherwise, v and w will already be connected in X and so e won't be chosen by Kruskal's.)
- Since neither e nor e' are in X, and Kruskal's chose e, $w(e') \geq w(e)$.
- Replace e' by e in T to get another spanning tree T'. Either $w(T') < w(T)$, a contradiction to the assumption T is minimal; or $w(T') = w(T)$, and we have another MST T' consistent with $X \cup \{e\}$. In both cases, we have completed the induction step. ■

Kruskal's: Runtime complexity

$MST(V, E, w)$

$X = \phi$

$Q = priorityQueue(E, w)$ // from min to max weight

while Q is nonempty

  $e = deleteMin(Q)$

  if e connects two disconnected components in (V, X)

    $X = X \cup \{e\}$

- Priority queue: $O(\log |E|) = O(\log V)$ per operation
- Connectivity test: $O(\log V)$ per check using a disjoint set data structure

Thus, for $|E|$ iterations, we have a runtime of $O(|E| \log |V|)$

MST: Applications

Network design: Communication networks, transportation networks, electrical grid, oil/water pipelines, ...

Clustering: Application of minimum spanning forest (stop when $|X| = |V| - k$ to get k clusters

Broadcasting: Spanning tree protocol in Ethernets

Shortest Paths

Input: A directed graph $G = (V, E)$, a cost function $l : E \to \mathbb{R}$ assigning non-negative costs, source and destination vertices s and t

Output: The shortest cost path from s to t in G.

Note:

- Single source shortest paths: find shortest paths from s to all every vertex. Can be solved using the same algorithm, with the same complexity!
- This algorithm constructs a *spanning tree* called *shortest path tree (SPT)*

Applications: Routing protocols (OSPF, BGP, RIP, ...), Map routing (flights, cars, mass transit), ...

Dijkstra's Algorithm: Outline

Base case: Start with $explored = \{s\}$

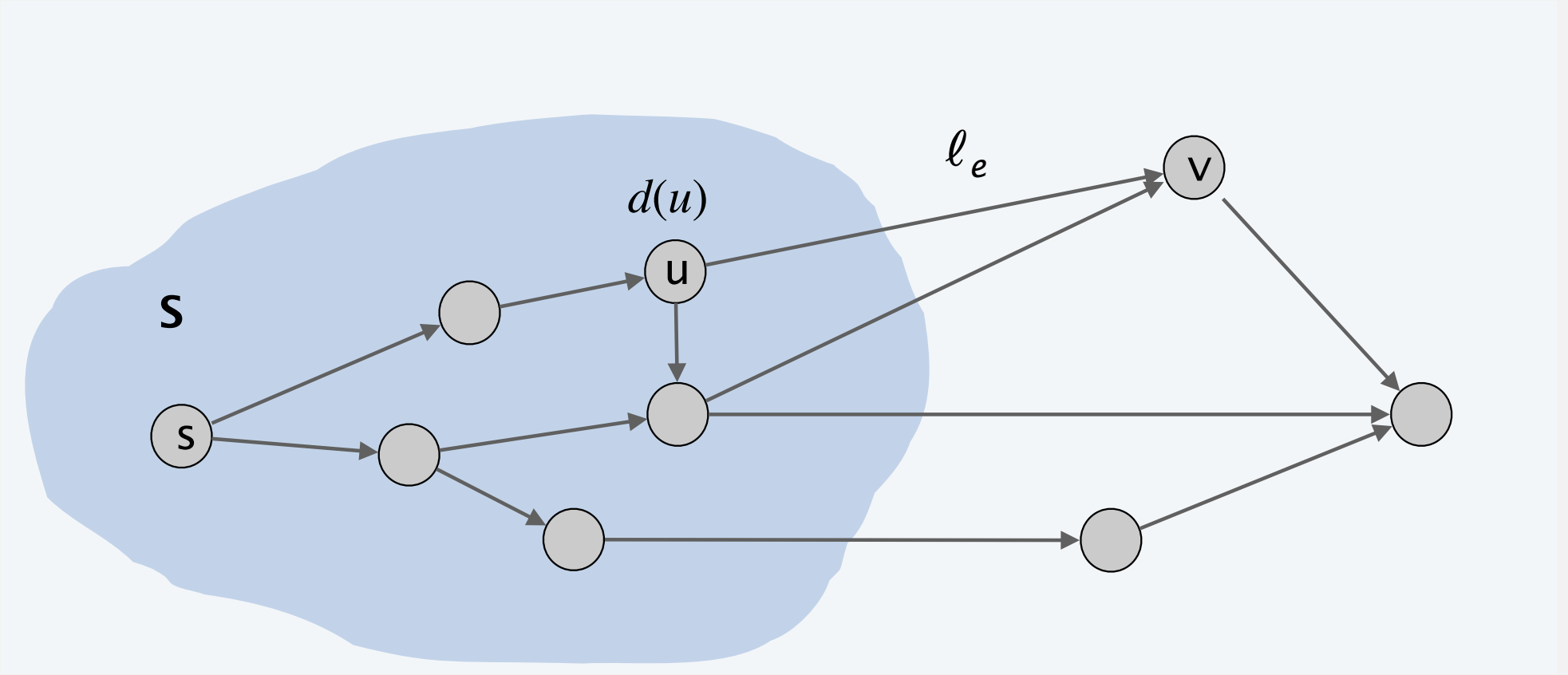
Inductive step:

Optimal substructure: After having computed the shortest path to all vertices in *explored*,

Greedy choice: extend *explored* with a v that can be reached using one edge e from some $u \in explored$ such that $dist(u) + l(e)$ is minimized

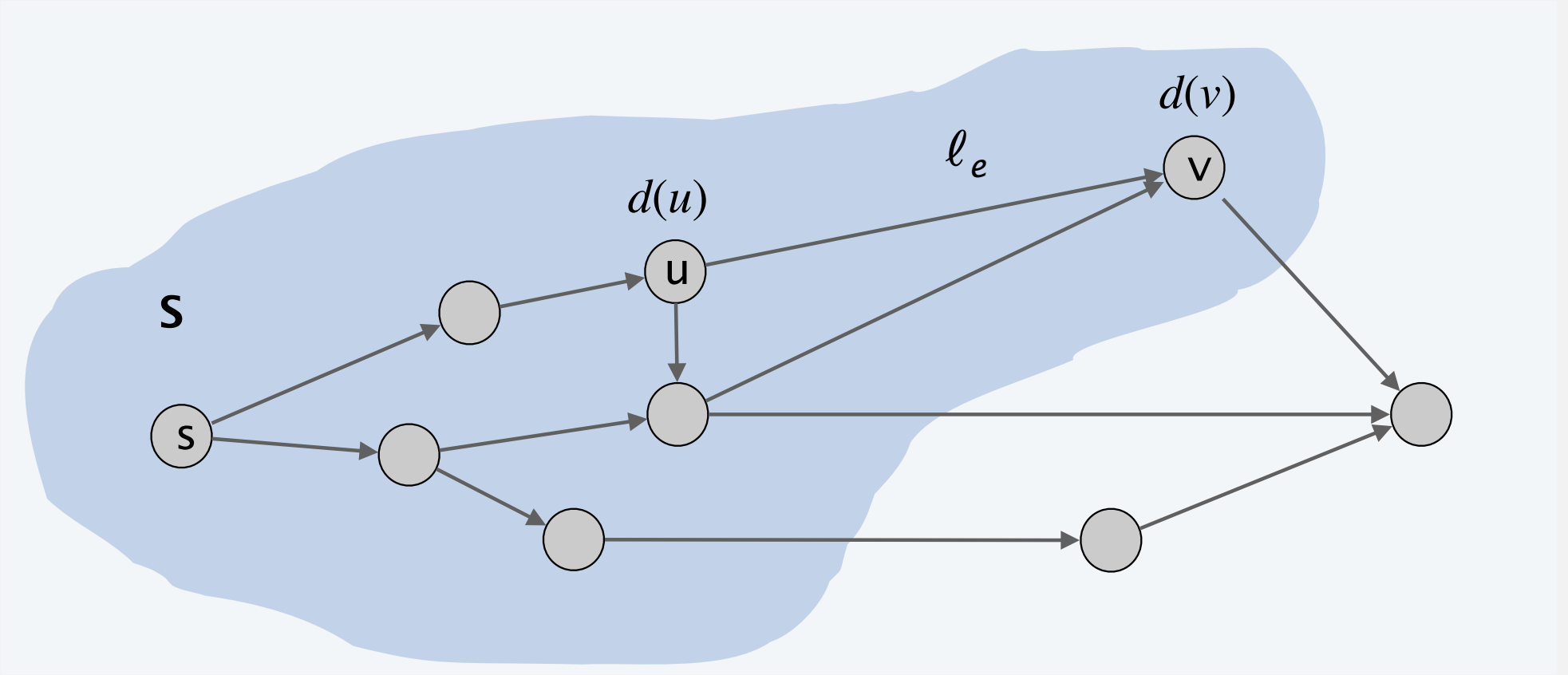
Finish: when $explored = V$

Dijkstra's: High-level intuition



Blue-colored region represents *explored*, i.e., we have already computed shortest paths to these vertices.

Dijkstra's: High-level intuition



In each iteration, we extend *explored* to include the vertex v that is the closest to any vertex in *explored*

Dijkstra's Algorithm

ShortestPathTree(V, E, l, s)

for v in V **do**

  $dist(v) = \infty$, $prev(v) = nil$

dist(s) = 0

$H = priorityQueue(V, dist)$

while H is nonempty

  $v = deleteMin(H)$ // Note: $explored = V - H$

  for $\langle v, w \rangle \in E$ **do**

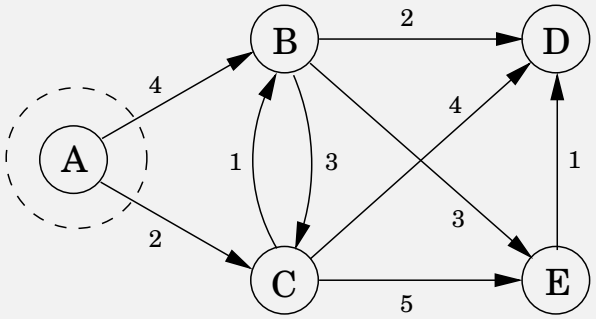
    if $dist(w) > dist(v) + l(\langle v, w \rangle)$

      $dist(w) = dist(v) + l(\langle v, w \rangle)$

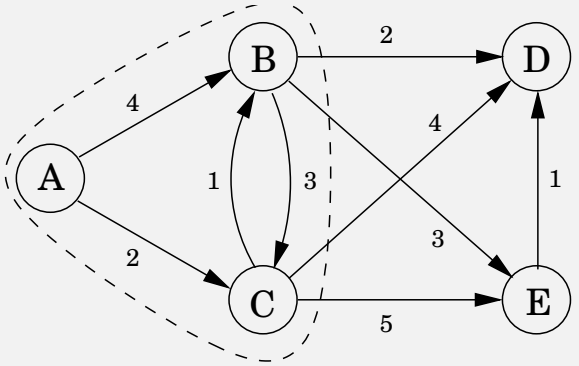
      $prev(w) = v$

      $decreaseKey(H, w)$

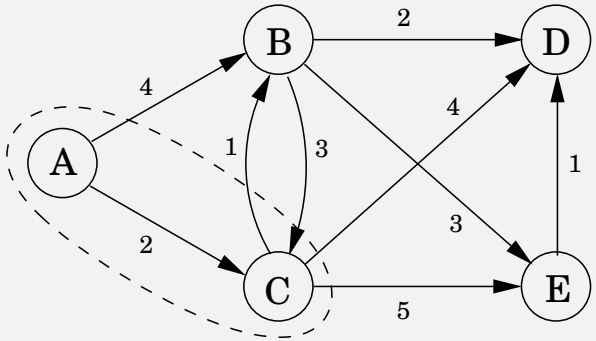
Dijkstra's Algorithm: Illustration



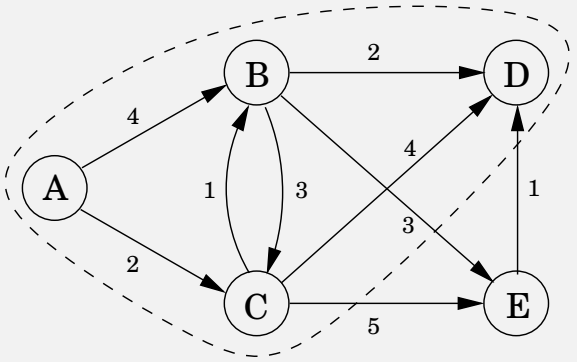
A: 0	D: ∞
B: 4	E: ∞
C: 2	



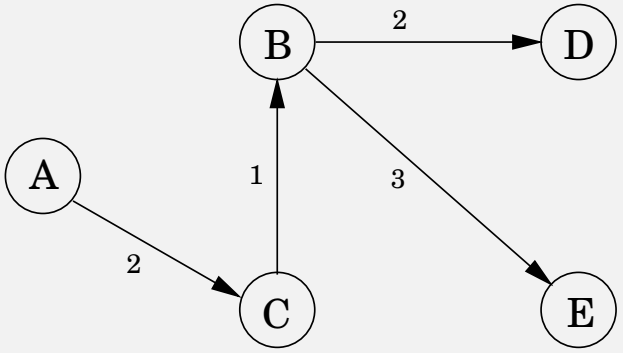
A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



Dijkstra's Algorithm: Correctness

Base case: Start with $explored = \emptyset$, so holds vacuously

Induction hypothesis: Tree T_i constructed so far (after i steps of Dijkstra's) is a subtree of an SPT T (*Optimal substructure*)

Induction step: By contradiction — similar to MST

Dijkstra's Algorithm: Correctness (2)

- Let $V_i = V - H$, and $E_i = \{prev(v) | v \in V_i\}$. Note that $T_i = (V_i, E_i)$
- Note that $v \in H$ chosen to be added to *explored* has the lowest *dist* in H. This means its *dist* must have been updated previously, and must have $prev(v)$ set to some $u \in explored$.
- Note $T_{i+1} = (V_i \cup \{v\}, E_i \cup (u, v))$. Need to show $(u, v) \in T$.
- Since T is a tree, it must have a unique path P from s to v
- P must have an edge $(u' \in V_i, v' \in H)$ that bridges V_i and H.
- If $v' = v$ and $u' = u$ we are done. Otherwise:
  - if $v' \neq v$ then note that $dist(v') \geq dist(v)$ (by how v was selected) and hence the so-called shortest path in T to v is longer than that in T_{i+1} — a contradiction. (Assuming $l(x, y) > 0 \forall x, y \in V$.)
  - if $u' \neq u$, then there is still a contradiction if $dist(u') + l(u', v) > dist(u) + l(u, v)$. Otherwise, the two sides should be equal, in which case we can obtain another SPT T' from T by replacing (u', v) by (u, v). This completes the induction step, as we have constructed an SPT consistent with T_{i+1}

■

Dijkstra's Algorithm: Runtime

while H is nonempty
  $v = deleteMin(H)$
  for $\langle v, w \rangle \in E$ **do**
    if $dist(w) > dist(v) + l(\langle v, w \rangle)$
      $dist(w) = dist(v) + l(\langle v, w \rangle)$
      $prev(w) = v$
      $decreasKey(H, w)$

- $O(|V|)$ iterations of *deleteMin*: $O(|V| \log |V|)$
- Inner loop executes $O(|E|)$ times, each iteration takes $O(\log V)$ time
- So, total time is $O((|E| + |V|) \log |V|)$

Information Theory and Coding

Information content

For an event e that occurs with probability p, its information content is given by $I(e) = -\log p$

- "surprise factor" — low probability event conveys more information; an event that is almost always likely ($p \approx 1$) conveys no information.
- Information content adds up: for two events e_1 and e_2, their combined information content is $-(\log p_1 + \log p_2)$

Information theory: Entropy

Information entropy

For a discrete random variable X that can take a value x_i with probability p_i, its entropy is defined as the *expectation* ("weighted average") over the information content of x_i:

$$H(X) = E[I(X)] = -\sum_{i=1}^{n} p_i \log p_i$$

- Entropy is a measure of uncertainty
- Plays a fundamental role in many areas, including coding theory and machine learning.

Optimal code length

Shannon's source coding theorem

A random variable X denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

- The first part of this theorem sets a lower bound, regardless of how clever the encoding is.
- Surprisingly simple proof for such a fundamental theorem! (See Wikipedia.)
- Huffman coding: an algorithm that achieves this bound

Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities $0.55, 0.02, 0.15, 0.28$.

- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code
  - Let us use as many bits as the *information content* of a character
  - A uses 1 bit, B uses 6 bits, C uses 3 bits, and D uses 2 bits.
  - You get an average saving of 15%

    $0.55 * 1 + 0.02 * 6 + 0.15 * 3 + 0.28 * 2 = 1.68$ bits
  - Lower bound (entropy)

    $-(.5 \log_2 .5 + .02 \log_2 .02 + .14 \log_2 .14 + .27 \log_2 .27) = 1.51$ bits

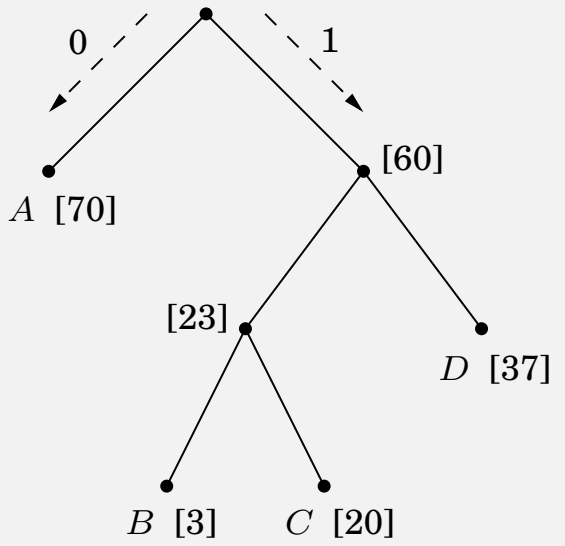
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.

- Let us try fixing the codes, not just their lengths:

  $A = 0, D = 11, C = 101, B = 100.$

- Note: enough to assign 3 bits to B, not 6. So, average coding size reduces to 1.62.

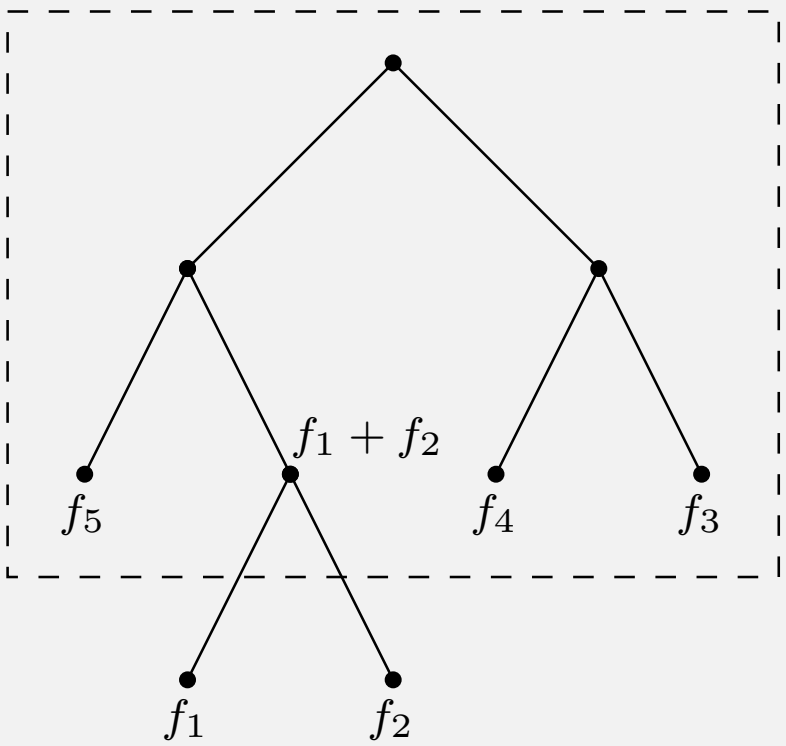


Prefix encoding

- No code is a prefix of another.
- Necessary property to enable decoding.
- Every such encoding can be represented using a full binary tree (either 0 or 2 children for every node)

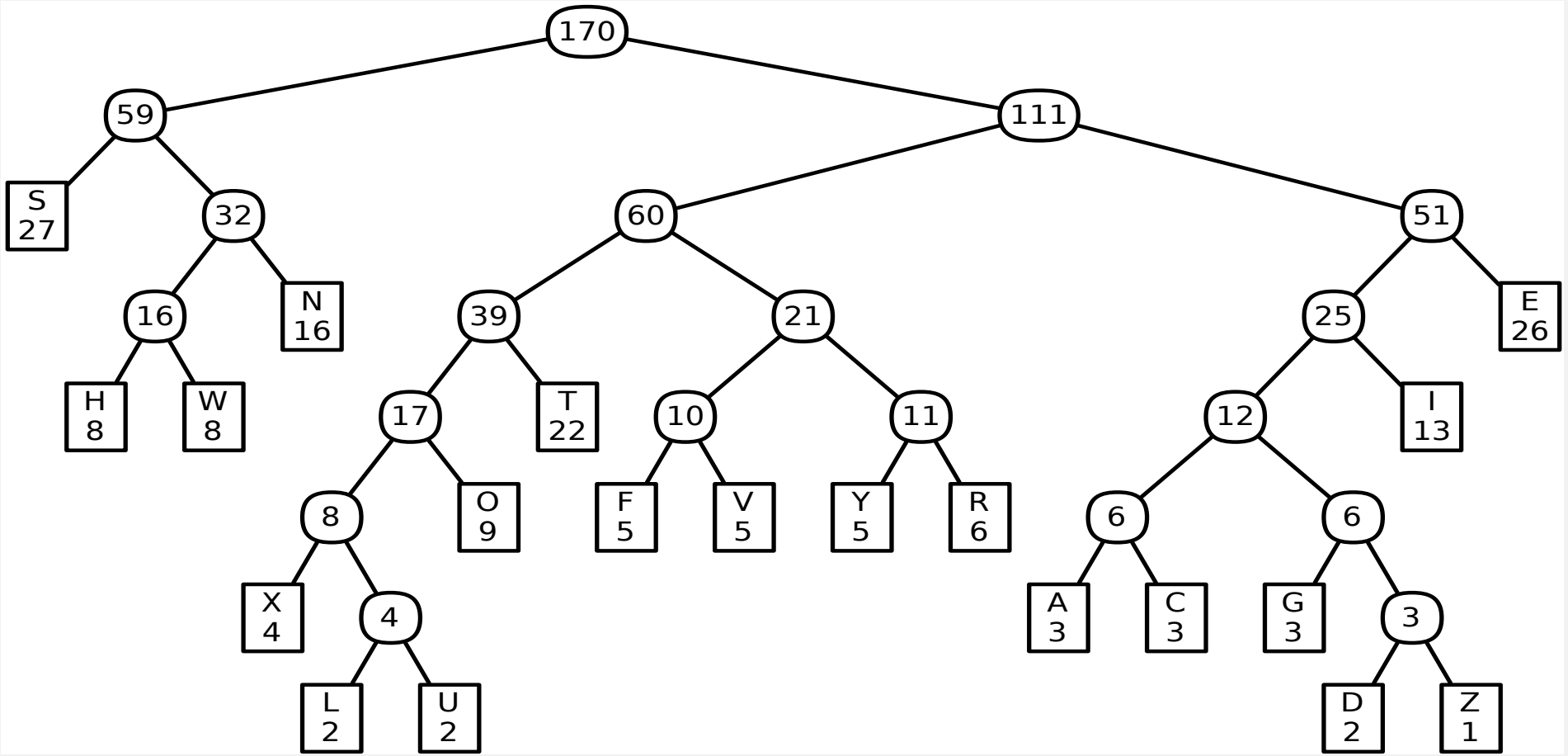
Huffman encoding

- Build the prefix tree bottom-up
- Start with a node whose children are codewords c_1 and c_2 that occur least often
- Remove c_1 and c_2 from alphabet, replace with c' that occurs with frequency $f_1 + f_2$
- Recurse



- How to make this algorithm fast?
- What is its complexity?

Huffman encoding: Example



This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

Images from Jeff Erickson's "Algorithms"

Uses about 650 bits, vs 850 for fixed-length (5-bit) code.

Huffman encoding: Optimality

- Crux of the proof: *Greedy choice property*
- Familiar exchange argument
  - Suppose the optimal prefix tree does not use longest path for two least frequent codewords c_1 and c_2
  - Show that by exchanging c_1 with the codeword using the longest path in the optimal tree, you can reduce the cost of the "optimal code" — a contradiction
  - Same argument holds for c_2

Huffman Coding: Applications

- Document compression
- Signal encoding
- As part of other compression algorithms (MP3, gzip, PKZIP, JPEG, ...)

Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
    - If they are English characters, effect is relatively small
    - if they are English words, or better, sentences, then much higher compression is possible
- To use words/sentences as codewords, we probably need to construct document-specific codebook
  - Larger alphabet size implies larger codebooks!
  - Need to consider the combined size of codebook plus the encoded document
- Can the codebook be constructed on-the-fly?
  - Lempel-Ziv compression algorithms (gzip)

`gzip` Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding W-bytes as the codebook ("sliding window", up to 32KB in `gzip`)

Encoding:

- Strings previously seen in the window are replaced by the pair $(\mathit{offset}, \mathit{length})$
  - Need to find the longest match for the current string
  - Matches should have a minimum length, or else they will be emitted as literals
  - Encode offset and length using Huffman encoding

Decoding: Interpret $(\mathit{offset}, \mathit{length})$ using the same window of W-bytes of preceding text. (Much faster than encoding.)

Greedy Algorithms: Summary

- One of the strategies used to solve *optimization problems*
- Frequently, locally optimal choices are *NOT* globally optimal, so use with a great deal of care.
  - *Always need to prove optimality.* Proof typically relies on *greedy choice property*, usually established by an "exchange" argument, and *optimal substructure*.
- Examples
  - MST and clustering
  - Shortest path
  - Huffman encoding

CSE 548: (*Design and*) Analysis of Algorithms

Dynamic Programming

R. Sekar

Overview

- Another approach for *optimization problems,* more general and versatile than greedy algorithms.
- *Optimal substructure* The optimal solution contains optimal solutions to subproblems.
- *Overlapping subproblems.* Typically, the same subproblems are solved repeatedly.
- Solve subproblems in *a certain order*, and *remember solutions* for later reuse.

Topics

Topological Sort

A way to linearize DAGs while ensuring that for every vertex, all its ancestors appear before itself.

Applications: Instruction scheduling, spreadsheet recomputation of formulas, Make (and other compile/build systems) and Task scheduling/project management.

Captures dependencies, and hence arises frequently in all types of programming tasks, and of course, dynamic programming!

Topological Sort

topoSort(V, E)

while $|V| \neq 0$

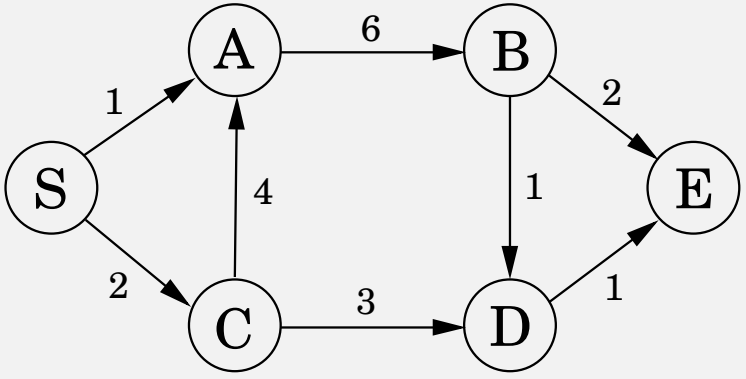
if there is a vertex v in V with in-degree of 0

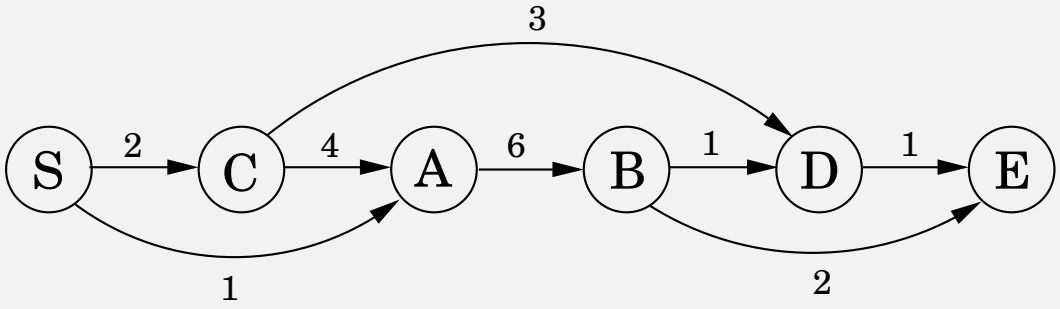
output v

$V = V - \{v\}$; $E = E - \{e \in E | e \text{ is incident on } v\}$)

else output "graph is cyclic"; **break**

return





Topological Sort

topoSort(V, E)

while $|V| \neq 0$

if there is a vertex v in V with in-degree of 0

output v

$V = V - \{v\}$; $E = E - \{e \in E | e \text{ is incident on } v\}$)

else output "graph is cyclic"; **break**

return

Correctness:

- If there is no vertex with in-degree 0, it is not a DAG
- When the algorithm outputs v, it has already output v's ancestors

Performance: What is the runtime? Can it be improved?

Shortest paths in DAGs

SSPDag(V, E, w, s)

for u in V **do**

  $dist(u) = \infty$

dist(s) = 0

for $v \in V - \{s\}$ in topological order **do**

  $dist(v) = min_{(u,v)\in E}(dist(u) + w(u, v))$

Thats all!

DAGs and Dynamic Programming

- *Canonical way to represent dynamic programming*

  Nodes in the DAG represent subproblems

  Edges capture dependencies between subproblems

  Topological sorting solves subproblems in the right order

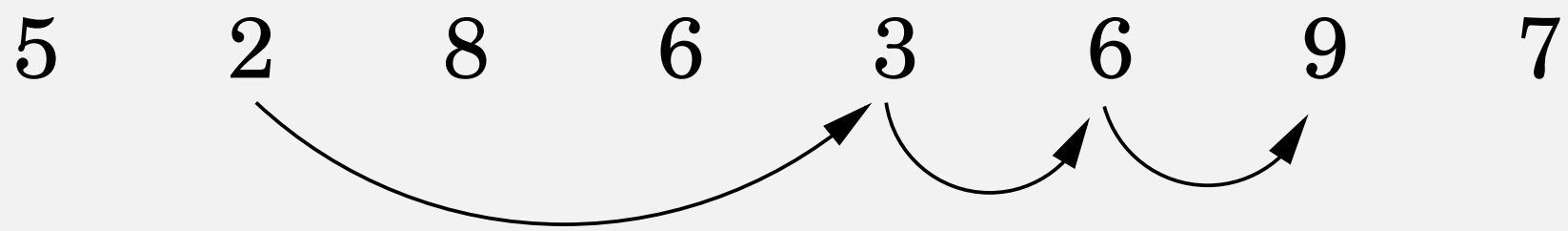
  Remember subproblem solutions to avoid recomputation

- Many bottom-up computations on trees/dags *are* instances of dynamic programming
  - applies to trees of recursive calls (w/ duplication), e.g., Fib
- For problems in other domains, DAGs are implicit, and topological sort is also done implicitly
  - Can you think of a way to do topological sorting implicitly, without modifying the dag at all?

Longest Increasing Subsequence

Definition

Given a sequence $a_1, a_2, \ldots, a_n$, its LIS is a sequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ that maximizes k subject to $i_j < i_{j+1}$ and $a_{i_j} \leq a_{i_{j+1}}$.



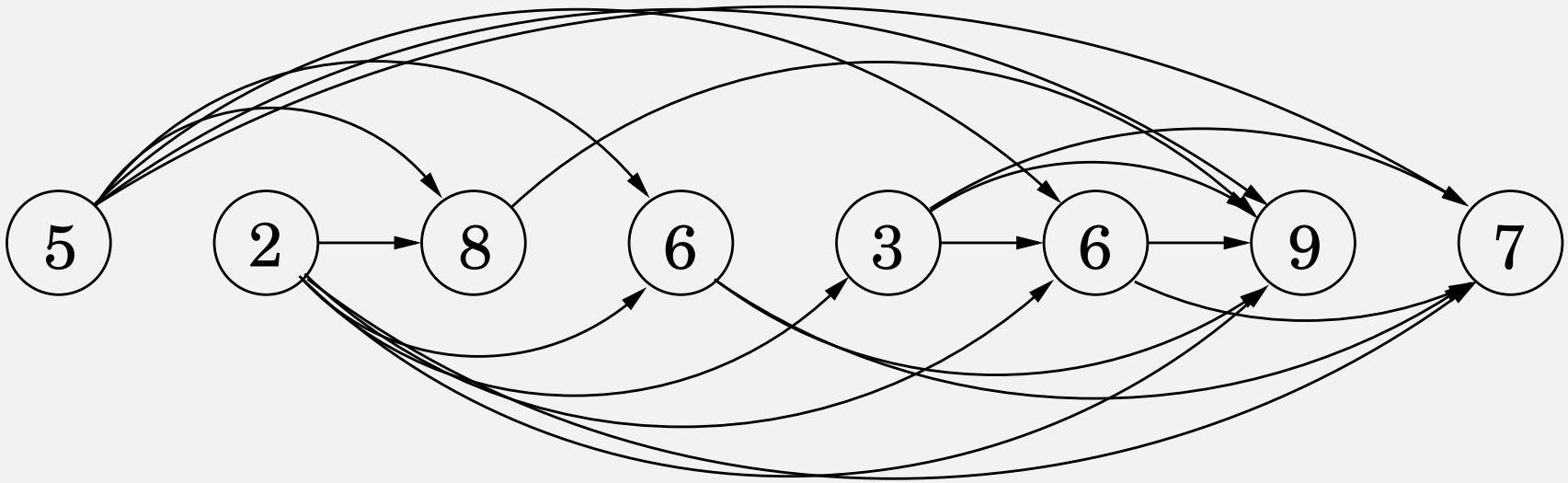
Casting LIS problem using a DAG

Nodes: represent elements in the sequence

Edges: connect an element to all followers that are larger

Topological sorting: sequence already topologically sorted

Remember: Using an array $L[1..n]$



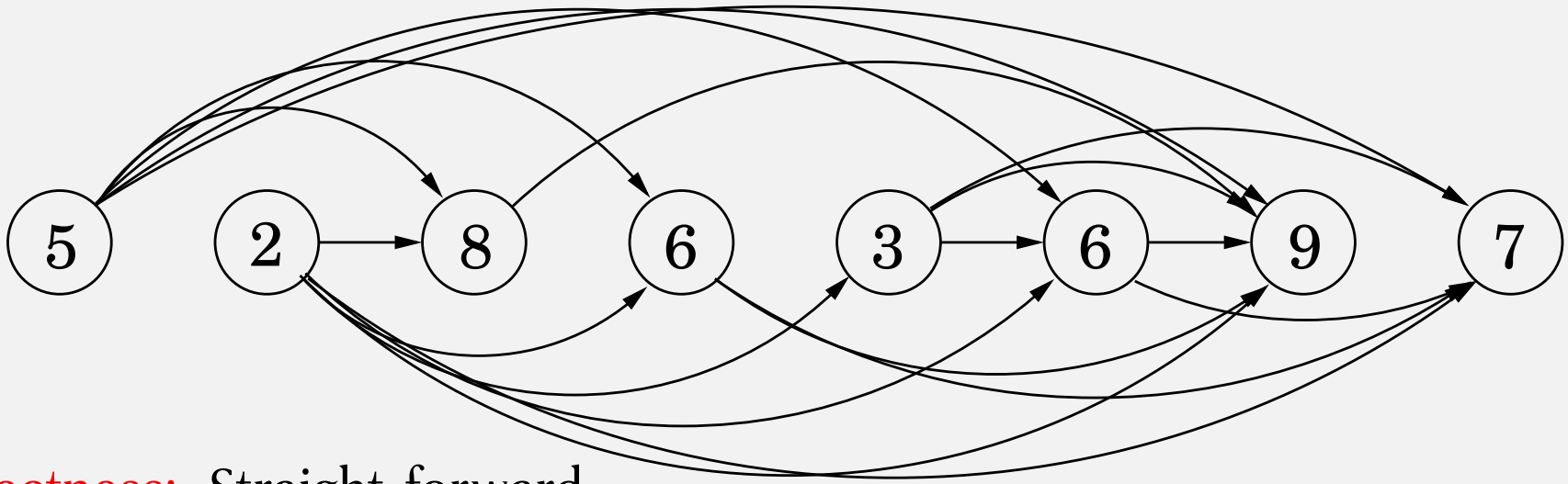
Algorithm for LIS

$LIS(E)$

for $j = 1$ **to** n **do**

$L[j] = 1 + max_{(i,j)\in E} L[i]$

return $max_{j=1}^{n} L[j]$



Correctness: Straight-forward

Complexity: What is it? Can it be improved?

Key step in Dyn. Prog.: Identifying subproblems

i. **The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.**

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. **The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.**

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

The number of subproblems is $O(mn)$.

iii. **The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.**

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. **The input is a rooted tree. A subproblem is a rooted subtree.**

Subsequence

Definition

A sequence $a[1..m]$ is a subsequence of $b[1..n]$ occurring at position r if there exist $i_1, \ldots, i_k$ such that $a[r..(r+l-1)] = b[i_1]b[i_2]\cdots b[i_l]$, where $i_j < i_{j+1}$

The relative order of elements is preserved in a subsequence, but unlike a substring, the elements needs not be contiguous.

Example: *BDEFHJ* is a subsequence of *ABCDEFGHIJK*

Longest Common Subsequence

Definition (LCS)

The LCS of two sequences $x[1..m]$ and $y[1..n]$ is the longest sequence $z[1..k]$ that is a subsequence of both x and y.

Example: $BEHJ$ is a common subsequence of $A\underline{B}CD\underline{E}FG\underline{H}I\underline{J}KLM$ and $AA\underline{B}BX\underline{E}J\underline{HJ}Z$

By aligning elements of z with the corresponding elements of x and y, we can compare x and y

$$\begin{array}{lccccccccc} x: & P & R & O & F & - & E & S & S & O & R \\ z: & P & R & O & F & - & E & S & - & - & R \\ y: & P & R & O & F & F_{ins} & E & S & -_{del} & U_{sub} & R \end{array}$$

to identify the *edit* operations (insert, delete, substitute) operations needed to map x to y

Edit (Levenshtein) distance

Definition (ED)

Given sequences x and y and functions I, D and S that associate costs with each insert, delete and substitute operations, what is the minimum cost of any the edit sequence that transforms x into y.

Applications

- Spell correction (Levenshtein automata)
- `diff`
- In the context of version control, reconcile/merge concurrent updates by different users.
- DNA sequence alignment, evolutionary trees and other applications in computational biology

Towards a dynamic programming solution (I)

What subproblems to consider?

- Just like the LIS problem, we proceed from left to right, i.e., compute $L[j]$ as j goes from1 to n
- But there are two strings x and y for LCS, so the subproblems correspond to prefixes of both x and y — there are $O(mn)$ such prefixes.

E X P O N E N T I A L

P O L Y N O M I A L

The subproblem above can be represented as $E[7, 5]$.

$E[i, j]$ represents the edit distance of $x[1..i]$ and $y[1..j]$

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

- $x[k] = y[l]$: in this case, $E[k, l] = E[k-1, l-1]$ — the edit distance has not increased as we extend the string by one character, since these characters match
- $x[k] \neq y[l]$: Three possibilities
  - extend $E[k-1, l]$ by deleting $x[k]$: $E[k, l] = E[k-1, l] + DC(x[k])$
  - extend $E[k, l-1]$ by inserting $y[l]$: $E[k, l] = E[k, l-1] + IC(y[l])$
  - extend $E[k-1, l-1]$ by substituting $x[k]$ with $y[l]$: $E[k, l] = E[k-1, l-1] + SC(x[k], y[l])$

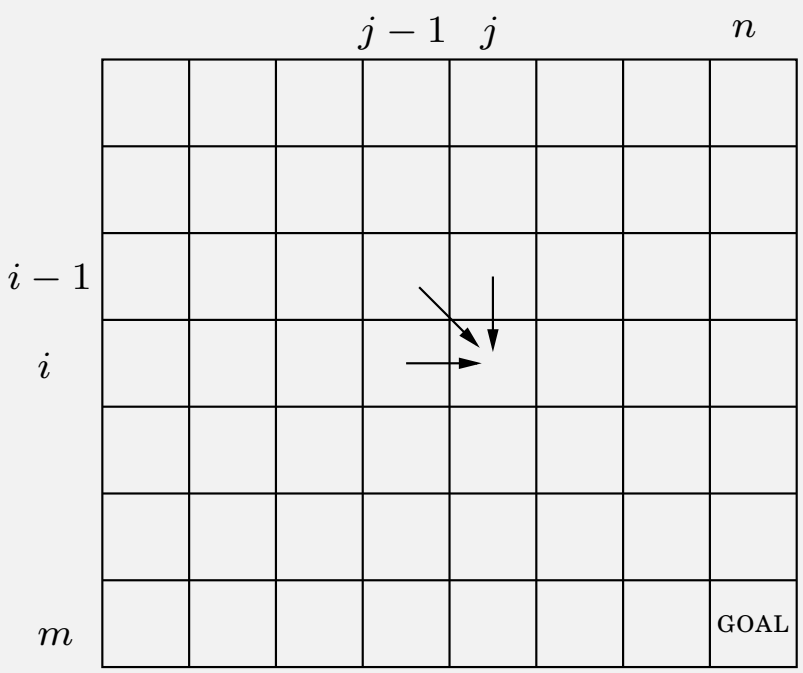
Towards a dynamic programming solution (3)

$$
\begin{aligned}
E[k,l] = \quad & min(\quad E[k-1,l] + DC(x[k]), && // \downarrow \\
& \qquad\quad E[k,l-1] + IC(y[l]), && // \rightarrow \\
& \qquad\quad E[k-1,l-1] + SC(x[k],y[l])) && // \searrow \\
E[0,l] = \quad & \sum_{i=1}^{l} IC(y[i]) \\
E[k,0] = \quad & \sum_{i=1}^{k} DC(x[i])
\end{aligned}
$$

Edit distance $= E[m,n]$

(Recall: m and n are lengths of strings x and y)

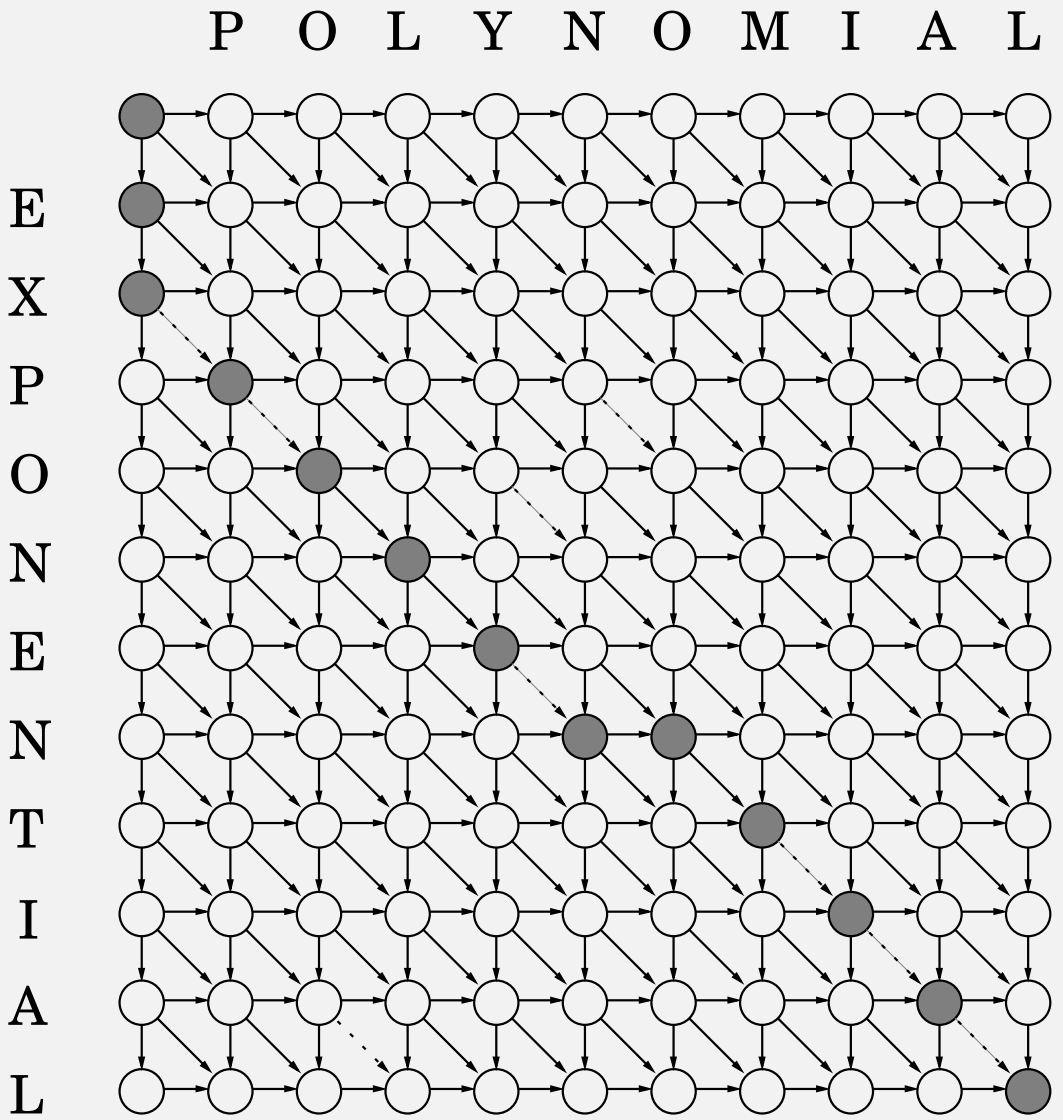
Towards a dynamic programming solution (4)



		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

$$
\begin{aligned}
E[k,l] = \quad & min(E[k-1,l] + DC(x[k]), && // \downarrow \\
& E[k,l-1] + IC(y[l]), && // \rightarrow \\
& E[k-1,l-1] + SC(x[k],y[l])) && // \searrow
\end{aligned}
$$

Towards a dynamic programming solution (5)



$$E[k,l] = min(E[k-1,l]+DC(x[k]), E[k,l-1]+IC(y[l]), E[k-1,l-1]+SC(x[k],y[l]))$$

Variations

Approximate prefix:

Is y approx. prefix of x? Decide based on

$$max_{1 \leq k \leq m} E[k, n]$$

Approximate suffix:

Initialize $E[k, 0] = 0$, use $E[m, n]$ to determine if y is an approximate suffix of x

Approximate substring:

Initialize $E[k, 0] = 0$, use $max_{1 \leq k \leq m} E[k, n]$ to decide if y is an approximate substring of x.

More variations

Supporting transpositions:

Use a fourth term within *min*:

$$E[k-2, l-2] + TC(x[k-1]x[k], y[l-1]y[l])$$

where TC is a small value for transposed characters, and ∞ otherwise.

Similarity Vs Edit-distance

Edit-distance cannot be interpreted on its own, and needs to take into account the lengths of strings involved.

Similarity can stand on its own.

$$
\begin{aligned}
S[k,l] = \quad & max(S[k-1,l] - DC(x[k]), && // \downarrow \\
& \quad S[k,l-1] - IC(y[l]), && // \rightarrow \\
& \quad S[k-1,l-1] - SC(x[k],y[l])) && // \searrow \\
S[0,l] = \quad & -\sum_{i=1}^{l} IC(y[i]) \\
S[k,0] = \quad & -\sum_{i=1}^{k} DC(x[i])
\end{aligned}
$$

- $SC(r,r)$ should be negative, while IC and DC should be positive.
- Formulations in biology are usually based on similarity

LCS application: UNIX `diff`

Each line is considered a "character:"

- Number of lines far smaller than number of characters
- Difference at the level of lines is easy to convey to users
- Much higher degree of confidence when things line up. Leads to better results on programs.

  But does not work that well on document types where line breaks are not meaningful, e.g., text files where each paragraph is a line.

Aligns lines that are preserved.

- The edits are then printed in the familiar "diff" format.

LCS applications: version control, patch,...

Software patches often distributed as "diffs." Programs such as `patch` can apply these patches to source code or any other file.

Concurrent updates in version control systems are resolved using LCS.

- Let x be the version in the repository
- Suppose that user A checks it out, edits it to get version y
- Meanwhile, B also checks out x, edits it to z.
- If $x \longrightarrow y$ edits target a disjoint set of locations from those targeted by the $x \longrightarrow z$ edits, both edits can be committed; otherwise a conflict is reported.

Recursive formulation of Dynamic programming

- Recursive formulation can often simplify algorithm presentation, avoiding need for explicit scheduling
  - Dependencies between subproblems can be left implicit an equation such as $K[w] = K[w - w[j]] + v[j]$
  - A call to compute $K[w]$ will automatically result in a call to compute $K[w - w[j]]$ because of dependency
  - *Can avoid solving (some) unneeded subproblems*
- *Memoization:* Remember solutions to function calls so that repeat invocations can use previously returned solutions

Recursive 0-1 Knapsack Algorithm

*BestVal*01(u, j)

if $u = 0$ **or** $j = 0$ **return** 0

if $w[j] > u$ **return** *BestVal*01$(u, j-1)$

else return $max($*BestVal*01$(u, j-1), v[j] +$ *BestVal*01$(u-w[j], j-1))$

- Much simpler in structure than iterative version
- Unneeded entries are not computed, e.g. *BestVal*01$(3, _)$ when all weights involved are even
- *Exercise:* Write a recursive version of ChainMM.

Note: m_i's give us the dimension of matrices, specifically, M_i is an $m_{i-1} \times m_i$ matrix

Complexity: $O(n^3)$

Dyn. Prog. and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

  The equation implies dependencies on subproblem solutions.
- *Dynamic programming algorithm:* finds a schedule that respects these dependencies
- *Typically, dependencies form a DAG:* its topological sort yields the right schedule
- *Cyclic dependencies:* What if dependencies don't form a DAG, but is a general graph.
- *Key Idea:* Use iterative techniques to solve (recursive) equations

Fixpoints

- A fixpoint is a solution to an equation:
- Substitute the solution on the rhs, it yields the lhs.
- *Example 1:* $y = y^2 - 12$.
  - A fixpoint is $y = 4$, another is $y = -3$.

$$rhs\big|_{y=4} = 4^2 - 12 = 4 = lhs\big|_{y=4} \quad \text{— a fixpoint}$$

Fixpoints (2)

- A fixpoint is a solution to an equation:
  - *Example 2:* $7x = 2y - 4, y = x^2 + y/x + 0.5$.
    - One fixpoint is $x = 2, y = 9$.

$$rhs_1\,|_{x=2,y=9} = 14 = lhs_1|_{x=2,y=9}$$

$$rhs_2\,|_{x=2,y=9} = 9 = lhs_2|_{x=2,y=9}$$

- The term "fixpoint" emphasizes an iterative strategy.
- *Example techniques:* Gauss-Seidel method (linear system of equations), Newton's method (finding roots), ...

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

    Well-founded order: An order that has no infinite ascending chain (i.e., sequence of elements $a_0 < a_1 < a_2 < \cdots$ where there is no maximum)

    Monotonicity: Successive iterations produce larger values with respect to the order, i.e., $rhs|_{sol_i} \geq sol_i$

    Result: Start with an initial guess S^0, note $S^i = rhs|_{S^{i-1}}$.
    - Due to monotonicity, $S^i \geq S^{i-1}$, and
    - by well-foundedness, the chain $S^0, S^1, \ldots$ can't go on forever.
    - Hence iteration must converge, i.e., $\exists k \, \forall i > k \;\; S^i = S^k$

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.
- Drawback of *explicit fixpoint iteration:* hard to analyze the number of iterations, and hence the runtime complexity
- So, algorithms tend to rely on inductive, bottom-up constructions with enough detail to reason about runtime.
- Fixpoint iteration thus serves two main purposes:
  - When it is possible to bound its complexity in advance, e.g., non-recursive definitions
  - As an intermediate step that can be manually analyzed to uncover inductive structure explicitly.

Shortest Path Problems

Graphs with cycles: Natural example where the optimal substructure equations are recursive.

Single source: $d_v = min_{u|(u,v)\in E}\,(d_u + l_{uv})$

All pairs: $d_{uv} = min_{w|(w,v)\in E}\,(d_{uw} + l_{wv})$

or, alternatively, $d_{uv} = min_{w\in V}\,(d_{uw} + d_{wv})$

Our study of shortest path algorithms is based on fixpoint formulation

- Shows how different shortest path algorithms can be derived from this perspective
- Highlights the similarities between these algorithms, making them easier to understand/remember

Single-source shortest paths

For the source vertex s, $d_s = 0$. For $v \neq s$, we have the following equation that captures the optimal substructure of the problem. We use the convention $l_{uu} = 0$ for all u, as it simplifies the equation:

$$d_v = min_{u|(u,v)\in E} \left(d_u + l_{uv}\right)$$

Expressing edge lengths as a matrix, this equation becomes:

$$\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_j \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 1_{12} & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \vdots & \vdots \\ l_{1j} & l_{2j} & \cdots & l_{jn} \\ \vdots & \vdots & \vdots & \vdots \\ l_{1n} & l_{2n} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_j \\ \vdots \\ d_n \end{bmatrix}$$

Matches the form of linear simultaneous equations, except that point-wise multiplication and addition become the integer "+" and *min* operations respectively.

Single-source shortest paths

SSP, written as a recursive matrix equation is:

$$D = \mathbf{L}D$$

Now, solve this equation iteratively:

$$\begin{aligned} D^0 &= Z \qquad (Z \text{ is the column matrix consisting of all } \infty \text{ except } d_s = 0) \\ D^1 &= \mathbf{L}Z \\ D^2 &= \mathbf{L}D^1 = \mathbf{L}(\mathbf{L}Z) = \mathbf{L}^2 Z \end{aligned}$$

Or, more generally, $D^i = \mathbf{L}^i Z$

- $\mathbf{L}$ is the generalized adjacency matrix, with entries being edge weights (aka edge lengths) rather than booleans.
- Side note: In this domain, multiplicative identity $\mathbf{I}$ is a matrix with zeroes on the main diagonal, and ∞ in all other places.
  - So, $\mathbf{L} = \mathbf{I} + \mathbf{L}$, and hence $\mathbf{L}^* = \lim_{r \to \infty} \mathbf{L}^r$

Single-source shortest paths

- Recall the connection between paths and the entries in $\mathbf{L}^i$.
- Thus, D^i represents the shortest path using i or fewer edges!
- Unless there are cycles with negative cost in the graph, all shortest paths must have a length less than n, so:

- D^n contains all of the shortest paths from the source vertex s
- d_i^n is the shortest path length from s to the vertex i.

Computing $\mathbf{L} \times \mathbf{L}$ takes $O(n^3)$, so overall SSP cost is $O(n^4)$.

SSP: Improving Efficiency of Matrix Formulation

- Compute the product from right: $(\mathbf{L} \times (\mathbf{L} \times \cdots (\mathbf{L} \times Z) \cdots)$
  - Each multiplication involves $n \times n$ and $1 \times n$ matrix, so takes $O(n^2)$ instead of $O(n^3)$ time.
  - Overall time reduced to $O(n^3)$.
- To compute $\mathbf{L} \times d_j$, enough to consider neighbors of j, and not all n vertices

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj})$$

  - Computes each matrix multiplication in $O(|E|)$ time, so we have an overall $O(|E||V|)$ algorithm.
- *We have stumbled onto the Bellman-Ford algorithm!*

Further Optimization on Iteration

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj})$$

- *Optimization 1:* If none of the d_k's on the rhs changed in the previous iteration, then d_j^i will be the same as d_j^{i-1}, so we can skip recomputing it in this iteration.
- Can be an useful improvement in practice, but asymptotic complexity unchanged from $O(|V||E|)$

Optimizing Iteration

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj}))$$

Optimization 2: Wait to update d_j on account of d_k on the rhs *until* d_k*'s cost stabilizes*

- Avoids repeated propagation of min cost from k to j — instead propagation takes place just once per edge, i.e., $O(|E|)$ times
- If all weights are non-negative, we can determine when costs have stabilized for a vertex k
  - There must be at least r vertices whose shortest path from the source s uses r or fewer edges.
  - In other words, if d_k^i has the rth lowest value, then d_k^i has stabilized if $r \leq i$

Voila! We have Dijkstra's Algorithm!

All pairs Shortest Path (I)

$$d^i_{uv} = min_{w|(w,v)\in E}(d^{i-1}_{uw} + l_{wv})$$

- Note that d_{uv} depends on d_{uw}, but not on any d_{xy}, where $x \neq u$.
- So, solutions for d_{xy} don't affect d_{uv}.
- i.e., we can solve a separate SSP, each with one of the vertices as source
- i.e., we run Dijkstra's $|V|$ times, overall complexity $O(|E||V| \log |V|)$

All pairs Shortest Path (II)

$$d^i_{uv} = min_{w \in E} \left(d^{i-1}_{uw} + d^{i-1}_{wv} \right)$$

Matrix formulation:

$$\mathbf{D} = \mathbf{D} \times \mathbf{D}$$

with $\mathbf{D}^0 = \mathbf{L}$.

Iterative formulation of the above equation yields

$$\mathbf{D}^i = \mathbf{L}^{2^i}$$

We need only consider paths of length $\leq n$, so stop at $i = \log n$.

Thus, overall complexity is $O(n^3 \log n)$, as each step requires $O(n^3)$ multiplication.

We have just uncovered a variant of Floyd-Warshall algorithm!

- Typically used with matrix-multiplication based formulation.

Matches ASP I complexity for dense graphs ($|E| = \Theta(|V|^2)$)

Further Improving ASP II

Each step has $O(n^3)$ complexity as it considers all (u, w, v) combinations
Note: Blind fixpoint iteration "breaks" recursion by limiting path length.

- Converts d_{uv} into d^i_{uv} where i is the path length
- Worked well for SSP & ASP I, not so well for ASP II

Can we break cycles by limiting something else, say, vertices on the path?
Floyd-Warshall: Define d^k_{uv} as the shortest path from u to v that only uses intermediate vertices 1 to k.

$$d^k_{uv} = min(d^{k-1}_{uv}, d^{k-1}_{uk} + d^{k-1}_{kv})$$

Complexity: Need n iterations to consider $k = 1, \ldots, n$ but each iteration considers only n^2 pairs, so overall runtime becomes $O(n^3)$

Summary

- A versatile, robust technique to solve optimization problems
- *Key step:* Identify *optimal substructure* in the form of an equation for optimal cost
- If equations are non-recursive, then either
  - identify underlying DAG, compute costs in topological order, or,
  - write down a memoized recursive procedure
- For recursive equations, "break" recursion by introducing additional parameters.
  - A fixpoint iteration can help expose such parameters.
- Remember the choices made while computing the optimal cost, use these to construct optimal solution.

CSE 548: (*Design and*) Analysis of Algorithms

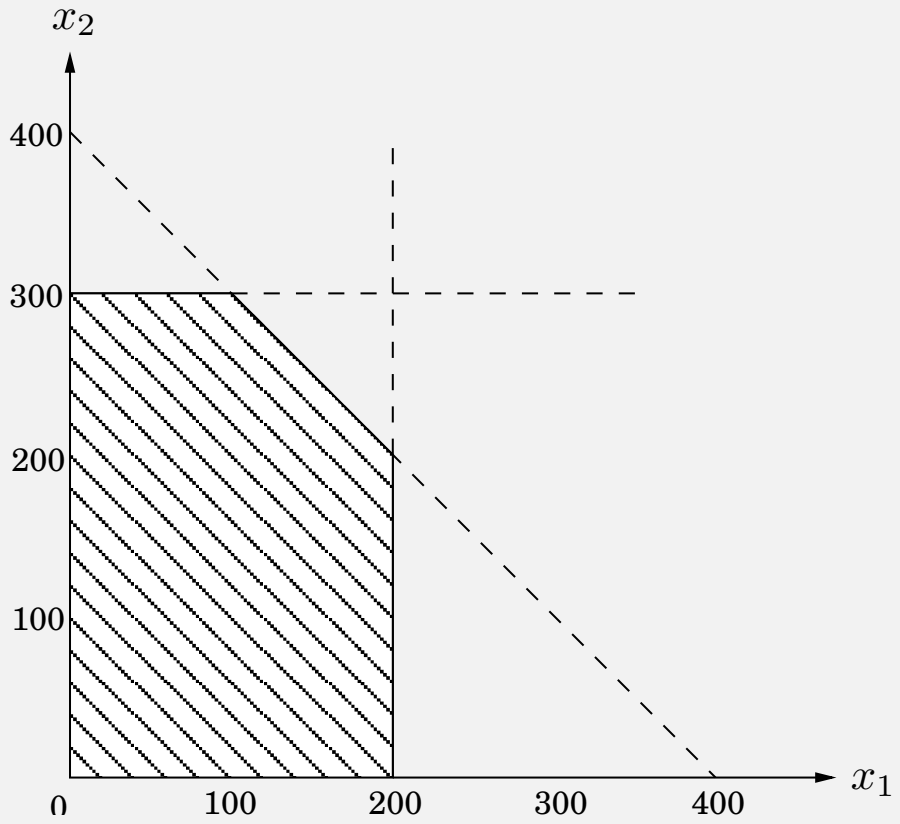
Linear Programming

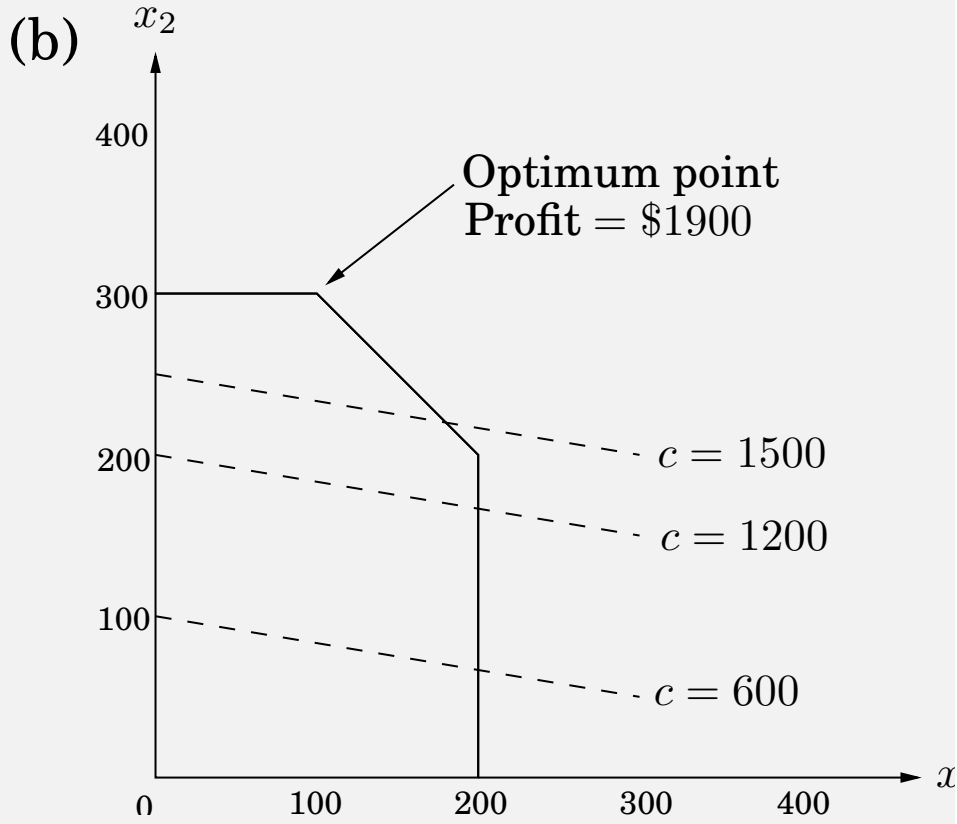
R. Sekar

Overview

- A technique for modeling a diverse range of optimization problems
  - LP is more of a modeling technique: You are not being asked to develop new "LP algorithms," but to model existing problems using LP.
  - Existing solvers can solve these problems
- We cover the intuition behind the solver, but not in great depth.

Example 1: Profit Maximization

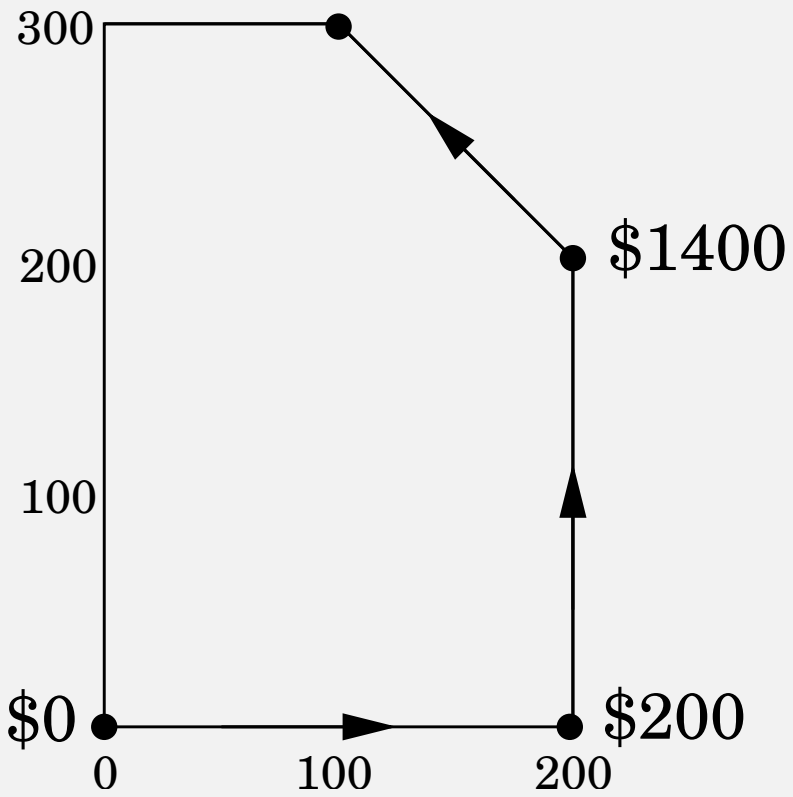




- Product P_1 generates \$1/unit, P_2 generates \$6/unit — Max $x_1 + 6x_2$
- Max 200 units of P_1 and 300 of P_2 can be sold — $x_1 \leq 200, x_2 \leq 300$
- Company can produce a total of 400 units — $x_1 + x_2 \leq 400$
- (Cannot produce negative number of units!) — $x_1, x_2 \geq 0$

Note: It is easy to see that a maximum should be at a vertex

Simplex Method



- Applicable to *convex problems,* i.e., conjunctions, and *linear constraints,* i.e., no squaring/multiplication of variables.
- Feasible regions are *convex polygons*

Simplex

- Start at the origin
- Switch to neighboring vertex if objective function $f(\bar{x})$ is higher
- Repeat until you reach a local maxima
  - which *will* be a global maxima
  - Consider the line $f(\bar{x}) = c$ passing through the vertex. Rest of the polygon must be below this line.

Simplex Algorithm

"Pebble falling down:"

- If you rotate the axes so that the normal to the hyperplane represented by the objective function faces down,
- then simplex operation resembles that of a pebble starting from one vertex, sliding down to the next vertex down and the next vertex down,
- until it reaches the minimum.

For simplicity, we consider only those cases where there is a unique solution, i.e., ignore degenerate cases.

Simplex Algorithm

- What is the space of feasible solutions?
  - A convex polyhedron in n-dimensions (n = number of variables)
- What is a vertex?
  - A point of intersection of n inequalities ("hyperplanes")
- What is a neighboring vertex?
  - Two vertices are neighbors if they share $n-1$ inequalities.
  - Vertex found by solving n simultaneous equations
- How many times can it fall?
  - There are m inequalities and n variables, so $\binom{m+n}{n}$ vertices can be there.
  - This is an exponential number, but simplex works exceptionally well in practice.

History and Main LP Algorithms

Fourier (1800s) Informal/implicit use

Kantorovich (1930) Applications to problems in Economics

Koopmans (1940) Application to shipping problems

Dantzig (1947) Simplex method.

Nobel Prize (1975) Kantorovich and Koopmans, not Dantzig

Khachiyan (1979) Ellipsoid algorithm, polynomial time but not competitive in practice.

Karmarkar (1984) Interior point method, polynomial time, good practical performance.

CSE 548: (*Design and*) Analysis of Algorithms

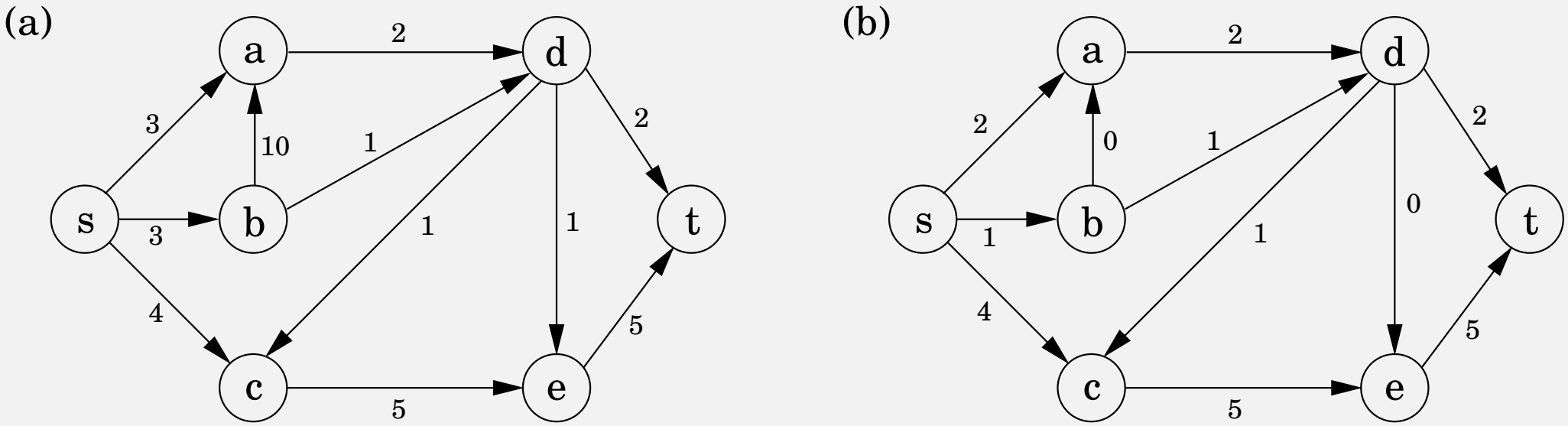
Flows in Networks

R. Sekar

Overview

- Network flows model important real-world problems
  - Oil pipelines, water and sewage networks, ...
  - Electricity grids
  - Communication networks
- In addition, several graph problems can be solved using maxflow algorithms
  - Bipartite matching, weighted bipartite matching, assignment problems,...
- Can be solved using linear programming
  - But we will study more efficient algorithms

Example 1: Maximizing Oil Flow



A pipeline network (a) and an assignment of flows (b)

- Edge capacities cannot be exceeded: $0 \leq f_e \leq c_e$
- Except for the source and sink nodes, incoming oil = outgoing oil:

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}$$

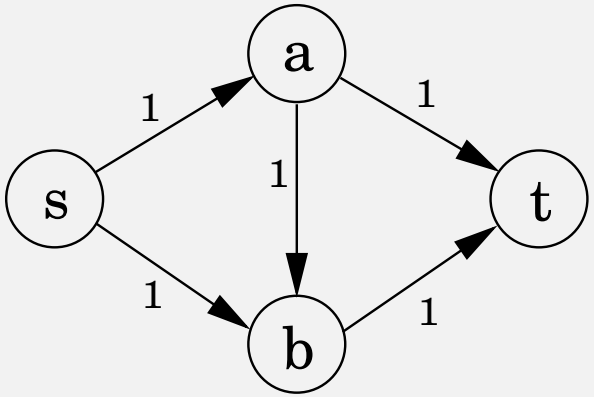
- *Maximize flow from s to t subject to these constraints.*

Solving Oil Flow

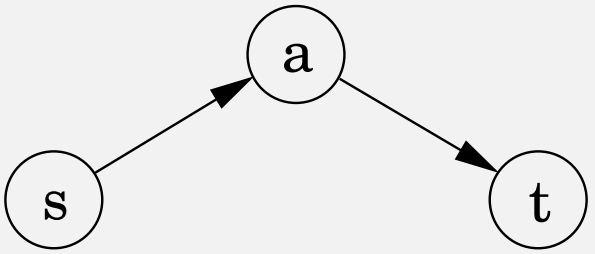
- *Can be posed as an LP problem:*
  - Objective: maximize the sum of flows on edges out of s
  - One variable per edge, with capacity constraint
  - Conservation conditions become equality constraints
- *Advantage of studying a powerful technique:*
  - Even in situations where it may not most efficient, we can use it to solve many problems
  - By studying this solution, we can gain insight that enable us to develop a direct algorithm that is more efficient.
- *So, how does Simplex solve flow problems?*
  - Start at the origin, i.e., zero flow
  - move to next corner: push max flow through one s—t path
  - repeat until no more paths can be added.

Simplex in Action

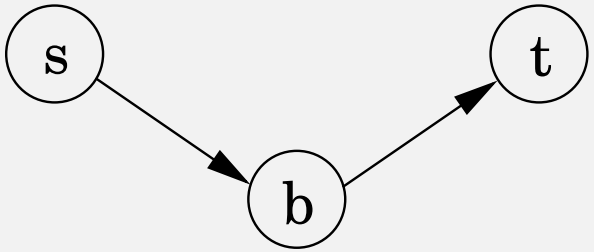
(a)



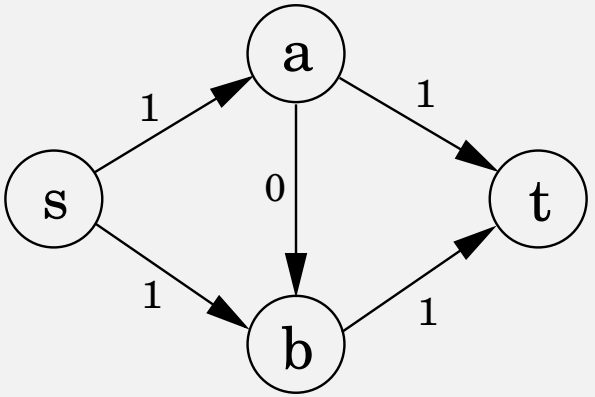
(b)



(c)

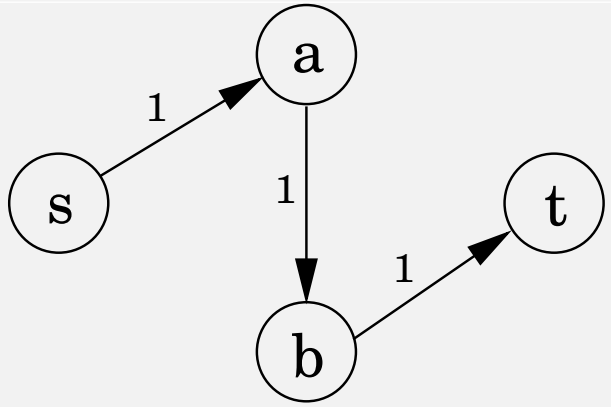


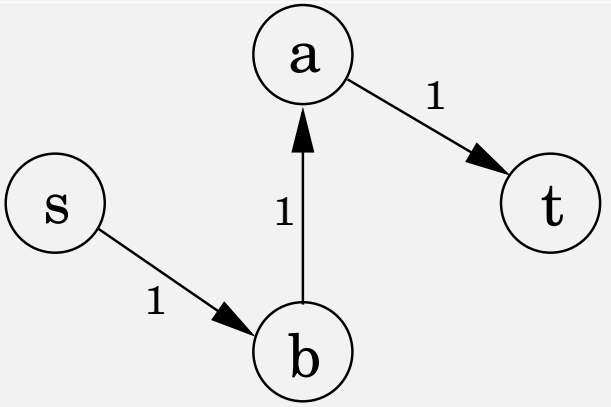
(d)



A pipeline network (a), steps taken by Simplex (b), (c), and the final assignment of flows (d)

But what happens if you pick the wrong path?





Incorrect path selected: left or right

- It seems we are stuck! What does Simplex do?
  - Simplex can increase a variable, but decrease later, so not stuck!
  - Will pick (left) and then (right), thus getting to maxflow
  - Flows in opposite directions in the middle edge cancel out
- Can we model this directly in a graph algorithm?
  - Construct a *residual graph*, with edges representing *positive or negative changes* that can be made to the current assignment.

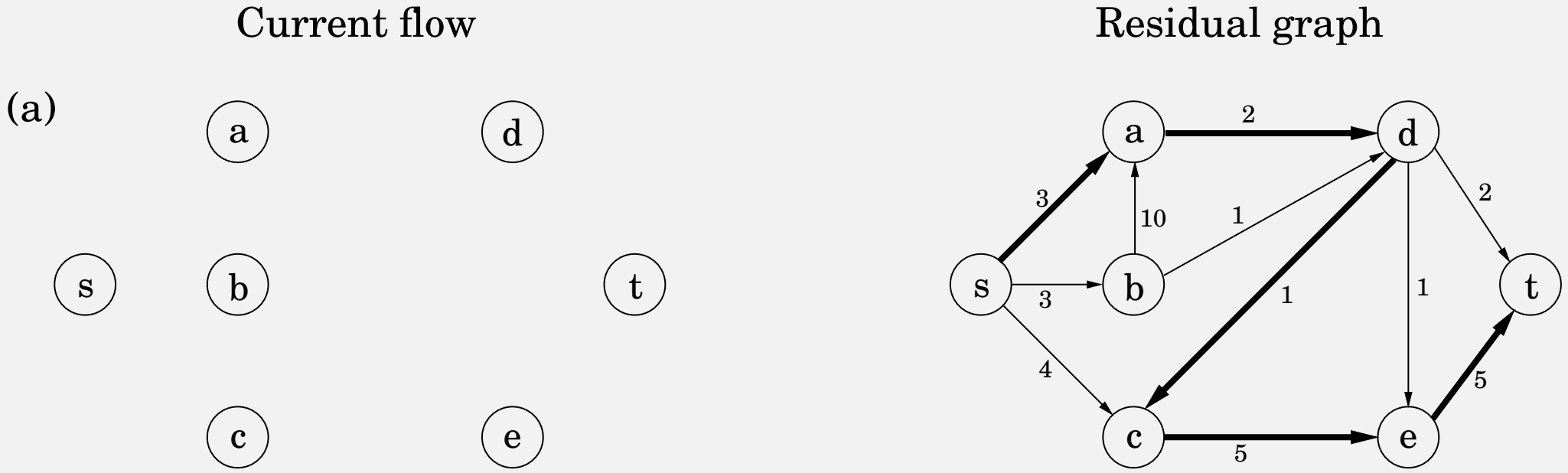
Augmented Graph G_f

Residual vertices: Same as G

Residual Edges: Edges representing left over capacities c^f

- If an edge e is not at full capacity in G, then $c_f = c_e - f_e$
- There is also an edge in opposite direction to each edge with a capacity f_e
  - Represents the fact we can cut back current flow to zero.

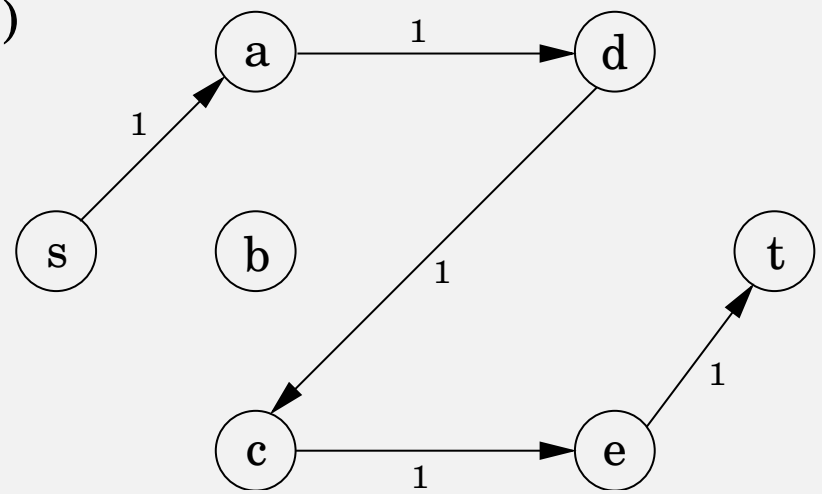
Maxflow Algorithm Illustration (1)

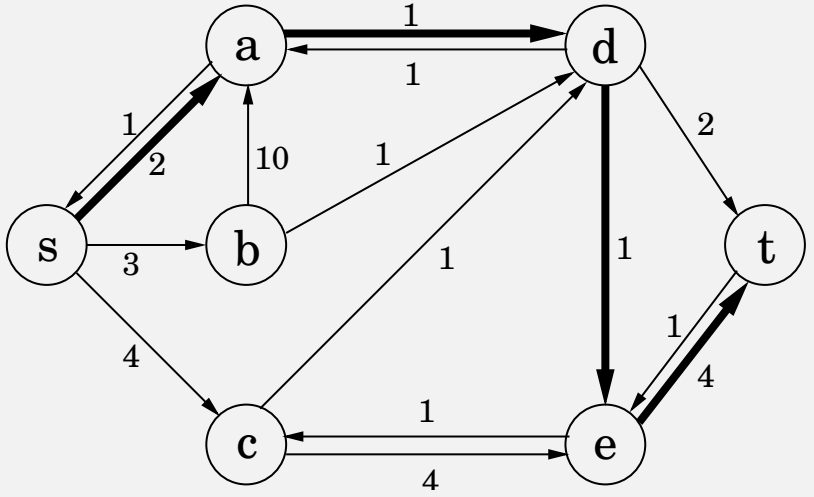


- Initial assignment is zero flows on all edges
- So, the residual graph G_f is exactly the same as G
- Thick edges show a possible new path P for additional flow
  - The algorithm sends a flow of $min_{e \in P}(c_e^f)$ on this path

Maxflow Algorithm Illustration (2)

(b)

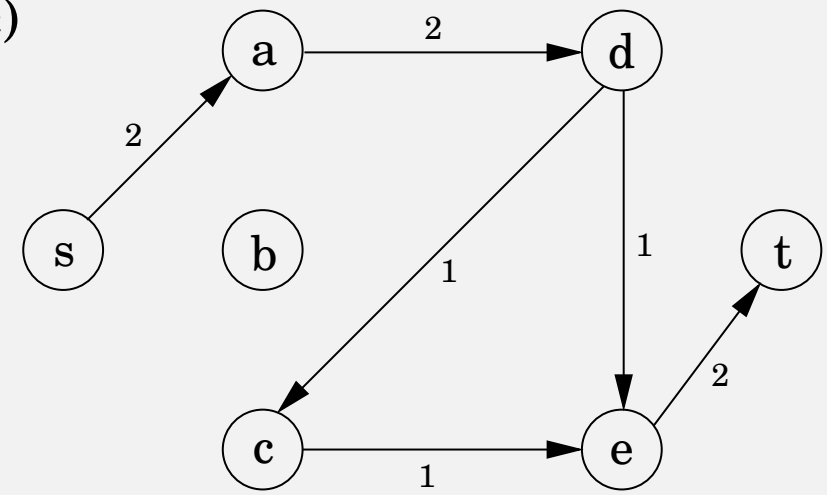


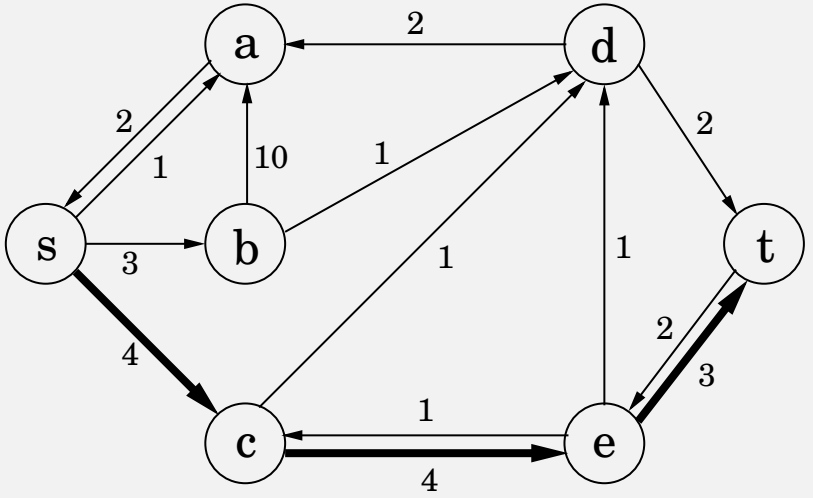


- Note addition of back edges in G_f on the right for each forward edge given a flow (see left)
- Capacity of a forward edge shrunk by amount of current flow
  - Full forward edges disappear, e.g., (d, c)
- Thick edges show the next possible path P for additional flow
  - The algorithm sends a flow of $min_{e \in P}(c_e^f)$ on this path

Maxflow Algorithm Illustration (3)

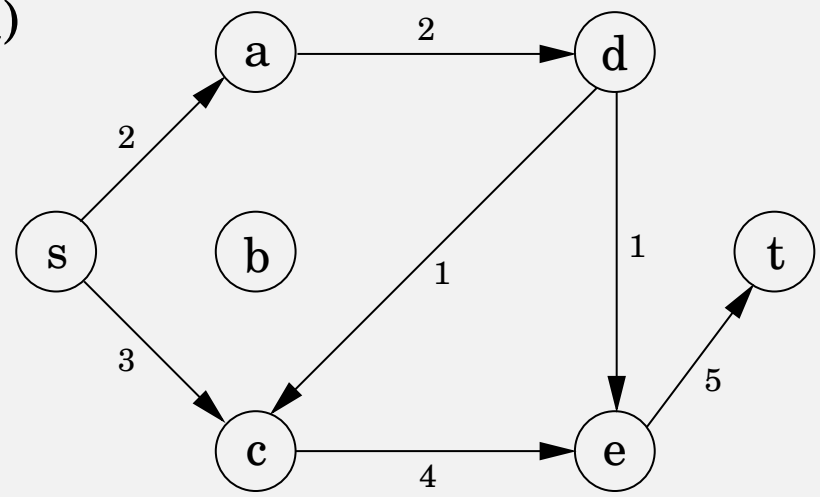
(c)

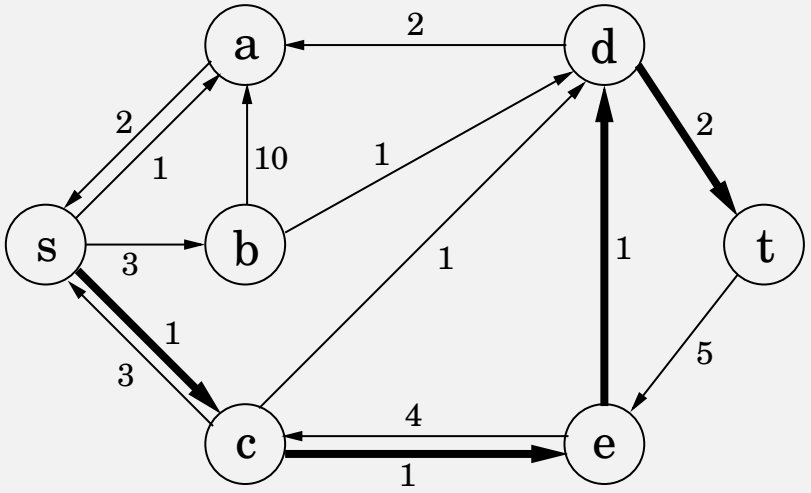




Maxflow Algorithm Illustration (4)

(d)



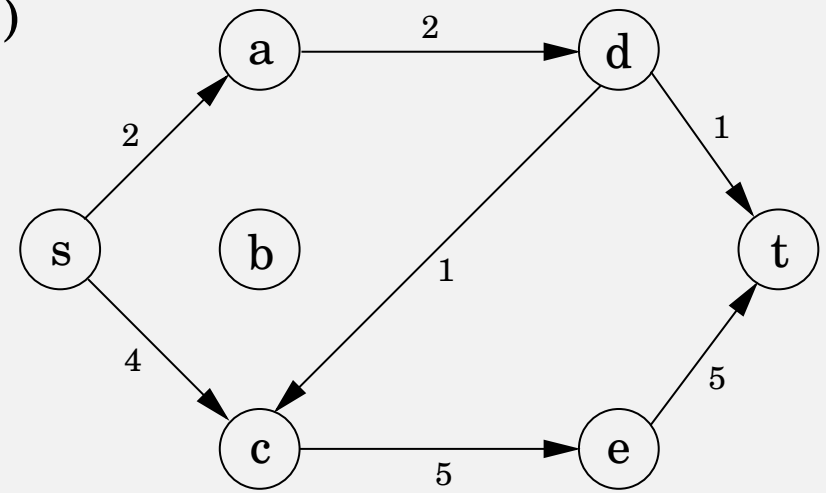


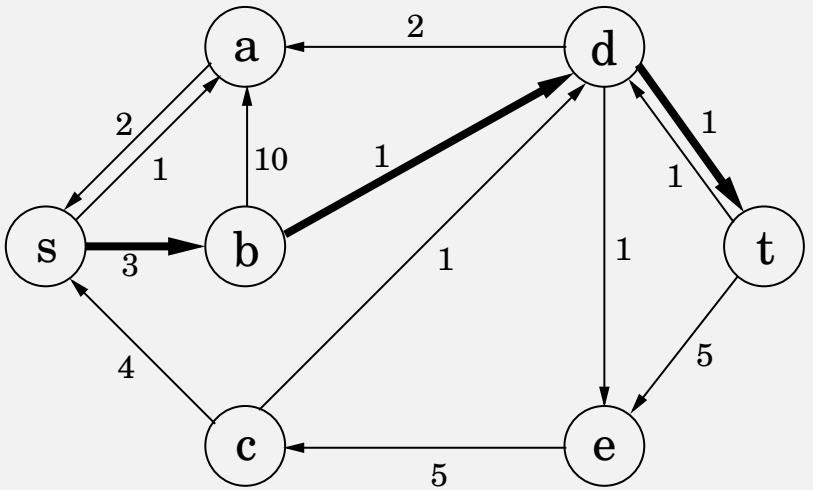
Maxflow Algorithm Illustration (5)

Current Flow

Residual Graph

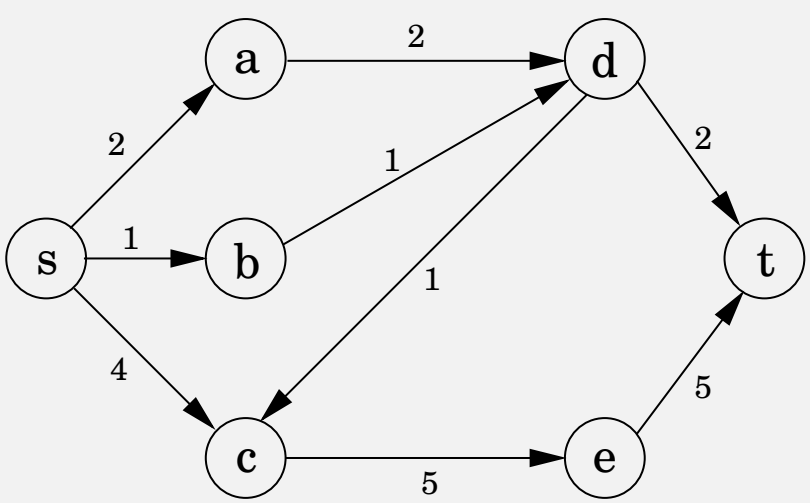
(e)

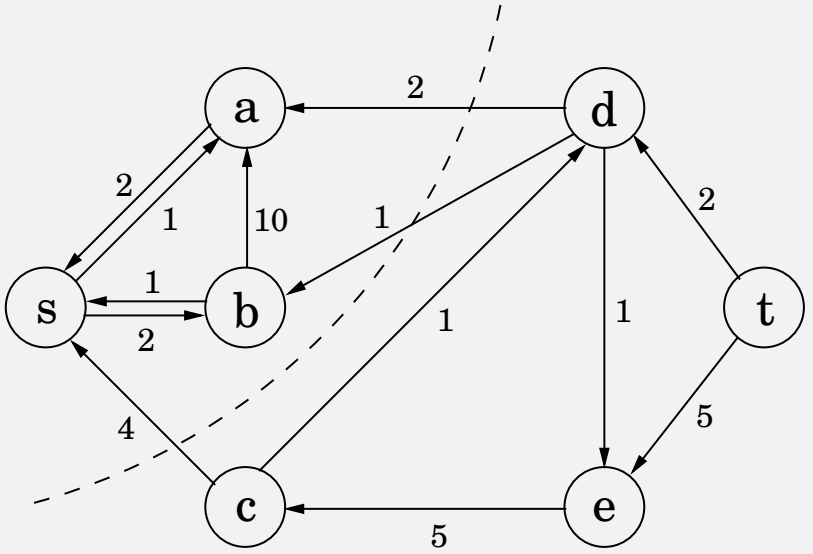




Maxflow Algorithm Illustration (6)

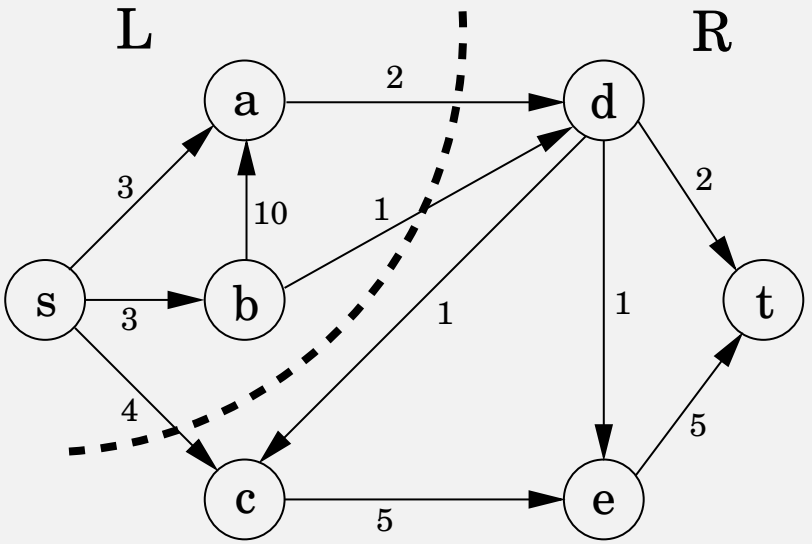
(f)





- No path from s to t in G_f: means we are done.
- Graph highlights a cut-set to show
  - G_f is disconnected, so no more flow can be sent
  - The very same (but inverted) edges in original graph form a *minimal cut-set* that proves we have maximized the flow

Max-flow min-cut theorem



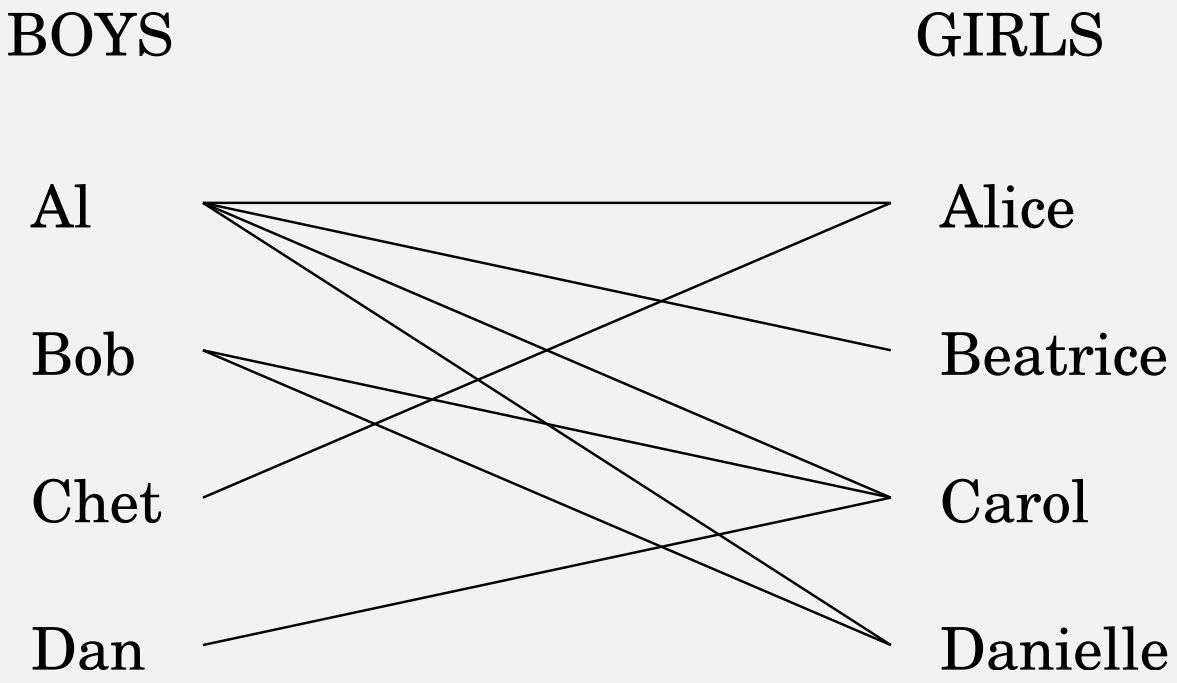
Theorem: The size of maximum flow in a network equals the capacity of the smallest $(s.t)$-cut.

- The dual of maximizing flow: finding a minimum cut-set
- A solution to dual problem is an optimality proof of primal
- Exercise: Find the cutset efficiently in the final G_f.

Runtime of Max-flow Algorithm

- Each path-finding step takes $O(E)$, say, using DFS or BFS
- G_f can be recomputed in the same amount of time
- Each iteration adds at least one unit of flow
- Total runtime: $O(C|E|)$ where C is the maximum flow computed.
  - Note that C can be large.
  - Unfortunately, this worst-case behavior can arise in some graphs if paths are chosen without care
  - If paths are chosen carefully, say, using *BFS*, number of iterations is $O(|V| \cdot |E|)$

Bipartite Matching



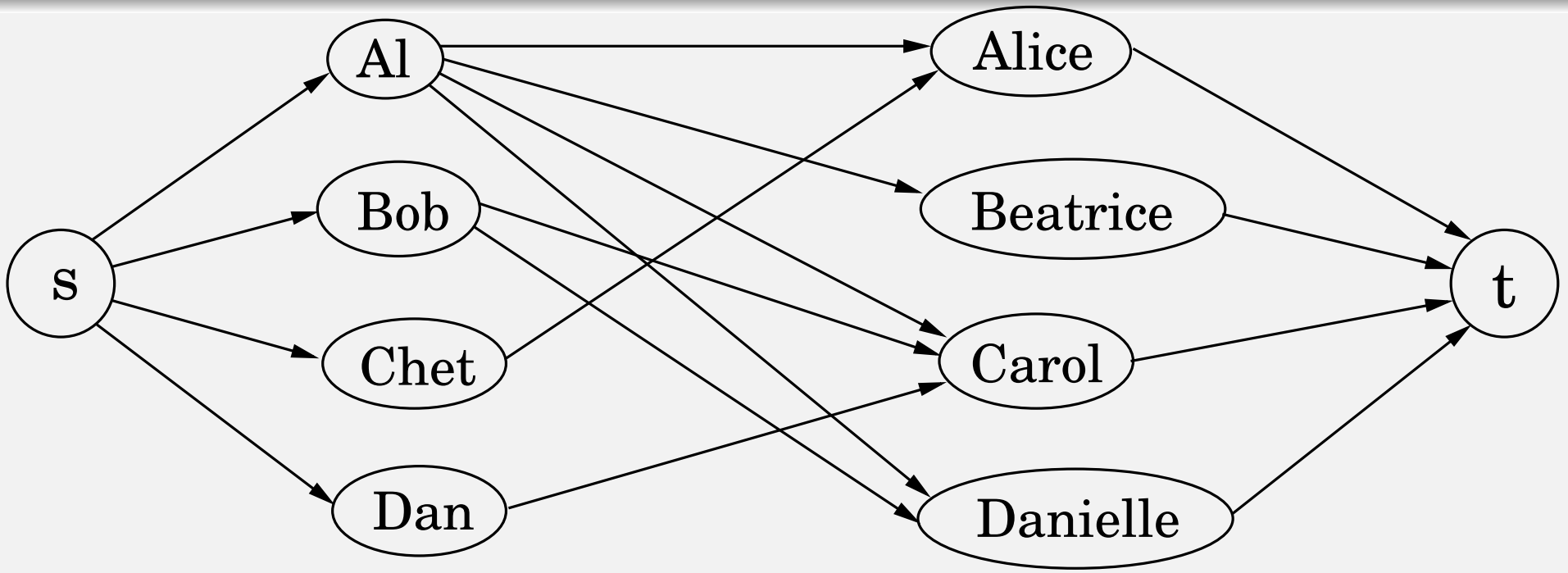
Bipartite: Two disjoint vertex sets, no edges within each set

Matching: Pair each vertex on left with one on right.

Maximal matching: Pairs as many vertices as possible

Exercise: Find an efficient algorithm for this problem

Bipartite Matching and Max-flow



Integral solutions are a must for bipartite matching, but not a real issue for max-flow in general

- As it turns out, Max-flow algorithm does guarantee to produce integral solutions when capacities are integers
- But in general integer optimization problems are much harder then non-integral versions

CSE 548: (*Design and*) Analysis of Algorithms

Randomized Algorithms

R. Sekar

Example 1: Routing

- What is the best way to route a packet from X to Y, esp. in high speed, high volume networks

  A: Pick the shortest path from X to Y

  B: Send the packet to a random node Z, and let Z route it to Y (possibly using a shortest path from Z to Y)

- Valiant showed in 1981 that surprisingly, B works better!
  - Turing award recipient in 2010

Example 2: Transmitting on shared network

- What is the best way for n hosts to share a common a network?

  A: Give each host a turn to transmit

  B: Maintain a queue of hosts that have something to transmit, and use a FIFO algorithm to grant access

  C: Let every one try to transmit. If there is contention, use random choice to resove it.

- Which choice is better?

Topics

1. Intro
2. Decentralize
   - Medium Access
   - Coupon Collection
   - Birthday
   - Balls and Bins
3. Taming distribution
   - Quicksort
   - Caching
   - Closest pair
   - Hashing
   - Universal/Perfect hash
4. Probabilistic Algorithms
   - Bloom filter
   - Rabin-Karp
   - Prime testing
   - Min-cut

Simplify, Decentralize, Ensure Fairness

- Randomization can often:
  - Enable the use of a simpler algorithm
  - Cut down the amount of book-keeping
  - Support decentralized decision-making
  - Ensure fairness
- *Examples:*

  Media access protocol: Avoids need for coordination — important here, because coordination needs connectivity!

  Load balancing: Instead of maintaining centralized information about processor loads, dispatch jobs randomly.

  Congestion avoidance: Similar to load balancing

A Randomized Protocol for Medium Access

- Suppose n hosts want to access a shared medium
  - If mutiple hosts try at the same time, there is contention, and the "slot" is wasted.
  - A slot is wasted if no one tries.
  - How can we maximize the likelihood of every slot being utilized?
- Suppose that a randomized protocol is used.
  - Each host transmits with a probability p
  - What should be the value of p?
- We want the likelihood that one host will attempt access (probability p), while others don't try (probability $(1-p)^{n-1}$)
  - Find p that maximizes $p(1-p)^{n-1}$
  - Using differentiation to find maxima, we get $p = 1/n$

A Randomized Protocol for Medium Access

- Maximum probability (when $p = 1/n$)

$$\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$$

- Note $\left(1-\frac{1}{n}\right)^{n-1}$ converges to $1/e$ for reasonably large n
  - About 5% off e at $n = 10$.
  - So, let us simplify the expression to $1/ne$ for future calculations
- What is the *efficiency* of the protocol?
  - The probability that *some* host gets to transmit is $n \cdot 1/ne = 1/e$
- Is this protocol a reasonable choice?
  - Wasting almost 2/3rd of the slots is rarely acceptable

A Randomized Protocol for Medium Access

- How long before a host i can expect to transmit successfully?
  - The probability it fails the first time is $(1 - 1/ne)$
  - Probability i fails in k attempts: $(1 - 1/ne)^k$
  - This quantity gets to be reasonably small (specifically, $1/e$) when $k = ne$
  - For larger k, say $k = ne \cdot c \ln n$, the expression becomes

$$(\underline{(1 - 1/ne)^{ne}})^{c \ln n} = (1/e)^{c \ln n} = (\underline{e^{\ln n}})^{-c} = n^{-c}$$

- So, a host has a reasonable success chance in $O(n)$ attempts
  - This becomes a virtual certainty in $O(n \ln n)$ attempts

A Randomized Protocol for Medium Access

- What is the expected wait time?
  - "Average" time a host can expect to try before succeeding.

$$E[X] = \sum_{j=0}^{\infty} j \cdot Pr[X = j]$$

- For our protocol, expected wait time is given by

$$1 \cdot p + 2 \cdot (1-p)p + 3 \cdot (1-p)^2 p \cdots = p \sum_{i=1}^{\infty} i \cdot (1-p)^{i-1}$$

- How do we sum the series $\sum ix^{i-1}$?
- Note that $\sum_{i=1}^{\infty} x^i = \frac{1}{(1-x)}$. Now, differentiate both sides:

$$\sum_{i=1}^{\infty} ix^{i-1} = -\frac{1}{(1-x)^2}$$

A Randomized Protocol for Medium Access

- Expected wait time is

$$p \sum_{i=1}^{\infty} i \cdot (1-p)^{i-1} = \frac{p}{p^2} = 1/p$$

- We get an intuitive result — a host will need to wait $1/p = ne$ slots on the average
- *Note:* The derivation is a general one, applies to any event with probability p; it is not particular to this access protocol

A Randomized Protocol for Medium Access

- How long will it be before every host would have a high probability of succeeding?
- We are interested in the probability of

$$S(k) = \bigcup_{i=1}^{n} S(i,k)$$

- Note that failures are not independent, so we cannot say that

$$Pr[S(k)] = \sum_{i=1}^{n} Pr[S(i,k)]$$

but certainly, the rhs is an upper bound on $Pr[F(k)]$.

  - We use this approximate *union bound* for our asymptotic analysis

A Randomized Protocol for Medium Access

- If we use $k = ne$, then
$$\sum_{i=1}^{n} Pr[S(i,k)] = \sum_{i=1}^{n} \frac{1}{e} = n/e$$
which suggests that the likelihood some hosts failed within ne attempts is rather high.
- If we use $k = cn \ln n$ then we get a bound:
$$\sum_{i=1}^{n} Pr[S(i,k)] = \sum_{i=1}^{n} n^{-c/e} = n^{(e-c)/e}$$
which is relatively small — $O(n^{-1})$ for $c = 2e$.
- Thus, it is highly likely that all hosts will have succeeded in $O(n \ln n)$ attempts.

A Randomized Protocol: Conclusions

- High school probability background is sufficient to analyze simple randomized algorithms
- Carefully work out each step
  - Intuition often fails us on probabilities
- If every host wants to transmit in every slot, this randomized protocol is a bad choice.
  - 63% wasted slots is unacceptable in most cases.
  - Better off with a round-robin or queuing based algorithm.
- How about protocols used in Ethernet or WiFi?
  - Optimistic: whoever needs to transmit will try in the next slot
  - Exponential backoff when collisions occur
    - Each collision halves p

Coupon Collector Problem

- Suppose that your favorite cereal has a coupon inside. There are n types of coupons, but only one of them in each box. How many boxes will you have to buy before you can expect to have all of the n types?
- What is your guess?
- Let us work out the expectation. Let us say that you have so far $j-1$ types of coupons, and are now looking to get to the jth type. Let X_j denote the number of boxes you need to purchase before you get the $j+1$th type.

Coupon Collector Problem

- Note $E[X_j] = 1/p_j$, where p_j is the probability of getting the *jth* coupon.
- Note $p_j = (n-j)/n$, so, $E[X_j] = n/(n-j)$
- We have all n types when we finish the X_{n-1} phase:

$$E[X] = \sum_{i=0}^{n-1} E[X_j] = \sum_{i=0}^{n-1} n/(n-j) = nH(n)$$

  - Note $H(n)$ is the harmonic sum, and is bounded by $\ln n$
- Perhaps unintuitively, you need to buy $\ln n$ cereal boxes to obtain one useful coupon.
- *Abstracts the media access protocol just discussed!*

Birthday Paradox

- What is the smallest size group where there are at least two people with the same birthday?
  - 365
  - 183
  - 61
  - 25

Birthday Paradox

- The probability that the $i+1$th person's birthday is distinct from previous i is approx.[1]

$$p_i = \frac{N-i}{N}$$

- Let X_i be the number of *duplicate* birthdays added by i:

$$E[X_i] = 0 \cdot p_i + 1 \cdot (1 - p_i) = 1 - p_i = \frac{i}{N}$$

- Sum up E_i's to find the # of distinct birthdays among n:

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{i}{N} = \frac{n(n-1)}{2N}$$

Thus, when $n \approx 27$, we have one duplicate birthday[2]

[1] We are assuming that $i-1$ birthdays are distinct: reasonable if $n \ll N$

[2] More accurate calculation will yield $n = 24.6$

Birthday Paradox Vs Coupon Collection

- Two sides of the same problem

  Coupon Collection: What is the minumum number of samples needed to cover every one of N values

  Birthday problem: What is the maximum number of samples that can avoid covering any value more than once?

- So, if we want enough people to ensure that every day of the year is covered as a birthday, we will need $365 \ln 365 \approx 2153$ people!
  - Almost 100 times as many as needed for one duplicate birthday!

Balls and Bins

If m balls are thrown at random into n bins:

- What should m be to have more than one ball in some bin?
  - Birthday problem
- What should m be to have at least one ball per bin?
  - Coupon collection, media access protocol example
- What is the maximum number of balls in any bin?
  - Such problems arise in load-balancing, hashing, etc.

Balls and Bins: Max Occupancy

- Probability $p_{1,k}$ that the first bin receives at least k balls:
  - Choose k balls in $\binom{m}{k}$ ways
  - These k balls should fall into the first bin: prob. is $(1/n)^k$
  - Other balls may fall anywhere, i.e., probability 1:[3]

$$\binom{m}{k}\left(\frac{1}{n}\right)^k = \frac{m\cdot(m-1)\cdots(m-k+1)}{k!n^k} \le \frac{m^k}{k!n^k}$$

- Let $m = n$, and use Sterling's approx. $k! \approx \sqrt{2\pi k}(k/e)^k$:

$$P_k = \sum_{i=1}^{n} p_{i,k} \le n\cdot\frac{1}{k!} \le n\cdot\left(\frac{e}{k}\right)^k$$

- Some arithmetic simplification will show that $P_k < 1/n$ when

$$k = \frac{3\ln n}{\ln\ln n}$$

[3] This is actually an upper bound, as there can be some double counting.

Balls and Bins: Summary of Results

m balls are thrown at random into n bins:

- Min. one bin with expectation of 2 balls: $m = \sqrt{2n}$
- No bin expected to be empty: $m = n \ln n$
- Expected number of empty bins: $ne^{-m/n}$
- Max. balls in any bin when $m = n$:

$$\Theta(\ln n / \ln \ln n)$$

  - This is a probabilistic bound: chance of finding any bin with higher occupancy is $1/n$ or less.
  - Note that the absolute maximum is n.

Randomized Quicksort

- Picks a pivot at random. What is its complexity?
- If pivot index is picked uniformly at random over the interval $[l, h]$, then:
  - every array element is equally likely to be selected as the pivot
  - every partition is equally likely
  - thus, *expected* complexity of *randomized* quicksort is given by:

$$T(n) = n + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) + T(n-i))$$

Summary: Input need not be random

- Expected $O(n \log n)$ performance comes from *externally forced* randomness in picking the pivot

Cache or Page Eviction

- Caching algorithms have to evict entries when there is a miss
  - As do virtual memory systems on a page fault
- Optimally, we should evict the "farthest in future" entry
  - But we can't predict the future!
- Result: many candidates for eviction. How can be avoid making bad (worst-case) choices repeatedly, even if input behaves badly?
- Approach: pick one of the candidates at random!

Closest pair

- We studied a deterministic divide-an-conquer algorithm for this problem.
  - Quite complex, required multiple sort operations at each stage.
  - Even then, the number of cross-division pairs to be considered seemed significant
  - Result: deterministic algorithm difficult to implement, and likely slow in practice.
- Can a randomized algorithm be simpler and faster?

Randomized Closest Pair: Key Ideas

- Divide the plane into small squares, hash points into them
  - Pairwise comparisons can be limited to points within the squares very closeby
- Process the points in some random order
  - Maintain min. distance δ among points processed so far.
  - Update δ as more points are processed
- At any point, the "small squares" have a size of $\delta/2$
  - At most one point per square (or else points are closer than δ)
  - Points closer than δ will at most be two squares from each other
    - Only constant number of points to consider
  - Requires rehashing all processed points when δ is updated.

Randomized Closest Pair: Analysis

- Correctness is relatively clear, so we focus on performance
- Two main concerns

  Storage: # of squares is $1/\delta^2$, which can be very large
  - Use a dictionary (hash table) that stores up to n points, and maps $(2x_i/\delta, 2y_i/\delta)$ to $\{1, \ldots, n\}$
  - To process a point (x_j, y_j)
    - look up the dictionary at $(x_j/\delta \pm 2, y_j/\delta \pm 2)$
    - insert if it is not closer than δ

  Rehashing points: If closer than δ — very expensive.
- Total runtime can all be "charged" to insert operations,
  - incl. those performed during rehashing

  so we will focus on estimating inserts.

Randomized Closest Pair: # of Inserts

Theorem

If random variable X_i denotes the likelihood of needing to rehash after processing k points, then

$$X_i \leq \frac{2}{i}$$

- Let $p_1, p_2, \ldots, p_i$ be the points processed so far, and p and q be the closest among these
- Rehashing is needed while processing p_i if $p_i = p$ or $p_i = q$
- Since points are processed in random order, there is a $2/i$ probability that p_i is one of p or q

Randomized Closest Pair: # of Inserts

Theorem

The expected number of inserts is $3n$.

- Processing of p_i involves
  - i inserts if rehashing takes place, and 1 insert otherwise
- So, expected inserts for processing p_i is

$$i \cdot X_i + 1 \cdot (1 - X_i) = 1 + (i-1) \cdot X_i = 1 + \frac{2(i-1)}{i} \leq 3$$

- Upper bound on expected inserts is thus $3n$

Look Ma! I have a linear-time randomized closest pair algorithm—And it is not even probabilistic!

Hash Tables

- A data structure for implementing:

  Dictionaries: Fast look up of a record based on a key.

  Sets: Fast membership check.

- Support expected $O(1)$ time *lookup*, *insert*, and *delete*
- Hash table entries may be:

  fat: store a pair $(key, object)$)

  lean: store pointer to object containing key

- Two main questions:
  - *How to avoid $O(n)$ worst case behavior?*
  - How to ensure *average case performance* can be realized *for arbitrary distribution of keys?*

Hash Table Implementation

Direct access: A fancy name for arrays. Not applicable in most cases where the universe $\mathcal{U}$ of keys is very large.

Index based on hash: Given a hash function h (fixed for the entire table) and a key x, use $h(x)$ to index into an array A.

- Use $A[h(x) \mod s]$, where s is the size of array
  - Sometimes, we fold the mod operation into h.
- Array elements typically called *buckets*
- *Collisions bound to occur* since $s \ll |\mathcal{U}|$
  - Either $h(x) = h(y)$, or
  - $h(x) \neq h(y)$ but $h(x) \equiv h(y)$ (**mod** s)

Collisions in Hash tables

- *Load factor α:* Ratio of number of keys to number of buckets
- *If* keys were random:
  - What is the max α if we want ≤ 1 collisions in the table?
  - If $\alpha = 1$, what is the maximum number of collisions to expect?
- Both questions can be answered from balls-and-bins results: $1/\sqrt{n}$, and $O(\ln n / \ln \ln n)$
- **Real world keys are not random.** Your hash table implementation needs to achieve its performance goals independent of this distribution.

Chained Hash Table

- Each bucket is a linked list.
- Any key that hashes to a bucket is inserted into that bucket.
- What is the *average* search time, as a function of α?
  - It is $1 + \alpha$ if:
    - you assume that the distribution of lookups is independent of the table entries, OR,
    - the chains are not too long (i.e., α is small)

Open addressing

- If there is a collision, probe other empty slots

  Linear probing: If $h(x)$ is occupied, try $h(x) + i$ for $i = 1, 2, \ldots$

  Binary probing: Try $h(x) \oplus i$, where $\oplus$ stands for exor.

  Quadratic probing: For ith probe, use $h(x) + c_1 i + c_2 i^2$

- Criteria for secondary probes

  Completeness: Should cycle through all possible slots in table

  Clustering: Probe sequences shouldn't coalesce to long chains

  Locality: Preserve locality; typically conflicts with clustering.

- Average search time can be $O(1/(1-\alpha)^2)$ for linear probing, and $O(1/(1-\alpha))$ for quadratic probing.

Chaining Vs Open Addressing

- Chaining leads to fewer collisions
  - Clustering causes more collisions w/ open addressing for same α
  - However, for lean tables, open addressing uses half the space of chaining, so you can use a much lower α for same space usage.
- Chaining is more tolerant of "lumpy" hash functions
  - For instance, if $h(x)$ and $h(x+1)$ are often very close, open hashing can experience longer chains when inputs are closely spaced.
  - Hash functions for open-hashing having to be selected very carefully
- Linked lists are not cache-friendly
  - Can be mitigated w/ arrays for buckets instead of linked lists
- Not all quadratic probes cover all slots (but some can)

Resizing

- Hard to predict the right size for hash table in advance
  - Ideally, $0.5 \leq \alpha \leq 1$, so we need an accurate estimate
- *It is stupid to ask programmers to guess the size*
  - Without a good basis, only terrible guesses are possible
- *Right solution:* Resize tables automatically.
  - When α becomes too large (or small), rehash into a bigger (or smaller) table
  - Rehashing is $O(n)$, but if you increase size by a factor, then amortized cost is still $O(1)$
  - Exercise: How to ensure amortized $O(1)$ cost when you resize up as well as down?

Average Vs Worst Case

- Worst case search time is $O(n)$ for a table of size n
- *With hash tables, it is all about avoiding the worst case, and achieving the average case*
- Two main chalenges:
  - *Input is not random,* e.g., names or IP addresses.
  - Even when input is random, h may cause "lumping," or non-uniform dispersal of $\mathcal{U}$ to the set $\{1, \ldots, n\}$
- Two main techniques

  Universal hashing

  Perfect hashing

Universal Hashing

- No single hash function can be good on all inputs
  - Any function $\mathcal{U} \to \{1, \ldots, n\}$ must map $|\mathcal{U}|/n$ inputs to same value!

Note: $|\mathcal{U}|$ can be much, much larger than n.

Definition

A family of hash functions $\mathcal{H}$ is universal if

$$Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{n} \quad \text{for all } x \neq y$$

Meaning: If we pick h at random from the family $\mathcal{H}$, then, probability of collisions is the same for any two elements.

Contrast with non-universal hash functions such as

$$h(x) = ax \bmod n, \qquad (a \text{ is chosen at random})$$

Note y and $y + kn$ collide with a probability of 1 *for every* a.

Universal Hashing Using Multiplication

Observation (Multiplication Modulo Prime)

If p is a prime and $0 < a < p$

- $\{1a, 2a, 3a, \ldots, (p-1)a\} = \{1, 2, \ldots, p-1\}$ *(mod p)*
- $\forall a \; \exists b \; ab \equiv 1 \pmod{p}$

Prime multiplicative hashing

Let the key $x \in \mathcal{U}$, $p > |\mathcal{U}|$ be prime, and $0 < r < p$ be random. Then

$$h(x) = (rx \bmod p) \bmod n$$

is universal.

Prove: $Pr[h(x) = h(y)] = \frac{1}{n}$, for $x \neq y$

Universality of prime multiplicative hashing

- Need to show $Pr[h(x) = h(y)] = \frac{1}{n}$, for $x \neq y$
- $h(x) = h(y)$ means $(rx \textbf{ mod } p) \textbf{ mod } n = (ry \textbf{ mod } p) \textbf{ mod } n$
- Note $a \textbf{ mod } n = b \textbf{ mod } n$ means $a = b + kn$ for some integer k. Using this, we eliminate **mod** n from above equation to get:

$$\begin{aligned} rx \textbf{ mod } p &= kn + ry \textbf{ mod } p, \text{ where } k \leq \lfloor p/n \rfloor \\ rx &\equiv kn + ry \ (\textbf{mod } p) \\ r(x-y) &\equiv kn \ (\textbf{mod } p) \\ r &\equiv kn(x-y)^{-1} \ (\textbf{mod } p) \end{aligned}$$

- So, x, y collide if $r = n(x-y)^{-1}, 2n(x-y)^{-1}, \ldots, \lfloor p/n \rfloor n(x-y)^{-1}$
- In other words, x and y collide for p/n out of p possible values of r, i.e., collision probability is $1/n$

Binary multiplicative hashing

- Faster: avoids need for computing modulo prime
- When $|\mathcal{U}| < 2^w$, $n = 2^l$ and a an odd random number

$$h(x) = \left\lfloor \frac{ax \mod 2^w}{2^{w-l}} \right\rfloor$$

- Can be implemented efficiently if w is the wordsize:

```
(a*x) >> (WORDSIZE-HASHBITS)
```

- Scheme is near-universal: collision probability is $O(1)/2^l$

Prime Multiplicative Hash for Vectors

Let p be a prime number, and the key x be a vector $[x_1, \ldots, x_k]$ where $0 \leq x_i < p$. Let

$$h(x) = \sum_{i=1}^{k} r_i x_i \pmod{p}$$

If $0 < r_i < p$ are chosen at random, then h is universal.

- Strings can also be handled like vectors, or alternatively, as a polynomial evaluated at a random point a, with p a prime:

$$h(x) = \sum_{i=0}^{l} x_i a^i \mod p$$

Universality of multiplicative hashing for vectors

- Since $x \neq y$, there exists an i such that $x_i \neq y_i$
- When collision occurs, $\sum_{j=1}^{k} r_j x_j = \sum_{j=1}^{k} r_j y_j \pmod{p}$
- Rearranging, $\sum_{j \neq i} r_j (x_j - y_j) = r_i (y_i - x_i) \pmod{p}$
- The lhs evaluates to some c, and we need to estimate the probability that rhs evaluates to this c
- Using multiplicative inverse property, we see that $r_i = c(y_i - x_i)^{-1} \pmod{p}$.
- Since $y_i, x_i < p$, it is easy to see from this equation that the collision-causing value of r_i is distinct for distinct y_i.
- Viewed another way, exactly one of p choices of r_i would cause a collision between x_i and y_i, i.e., $Pr_h[h(x) = h(y)] = 1/p$

Perfect hashing

Static: Pick a hash function (or set of functions) that avoids collisions for a given set of keys

Dynamic: Keys need not be static.

Approach 1: Use $O(n^2)$ storage. Expected collision on n items is 0.

But too wasteful of storage.

Don't forget: more memory usually means less performance due to cache effects.

Approach 2: Use a secondary hash table for each bucket of size n_i^2, where n_i is the number of elements in the bucket.

Uses only $O(n)$ storage, *if h is universal*

Hashing Summary

- Excellent average case performance
  - Pointer chasing is expensive on modern hardware, so improvement from $O(\log n)$ of binary trees to expected $O(1)$ for hash tables is significant.
- But all benefits will be reversed if collisions occur too often
  - Universal hashing is a way to ensure expected average case *even when input is not random.*
- Perfect hashing can provide efficient performance even in the worst case, but the benefits are likely small in practice.

Probabilistic Algorithms

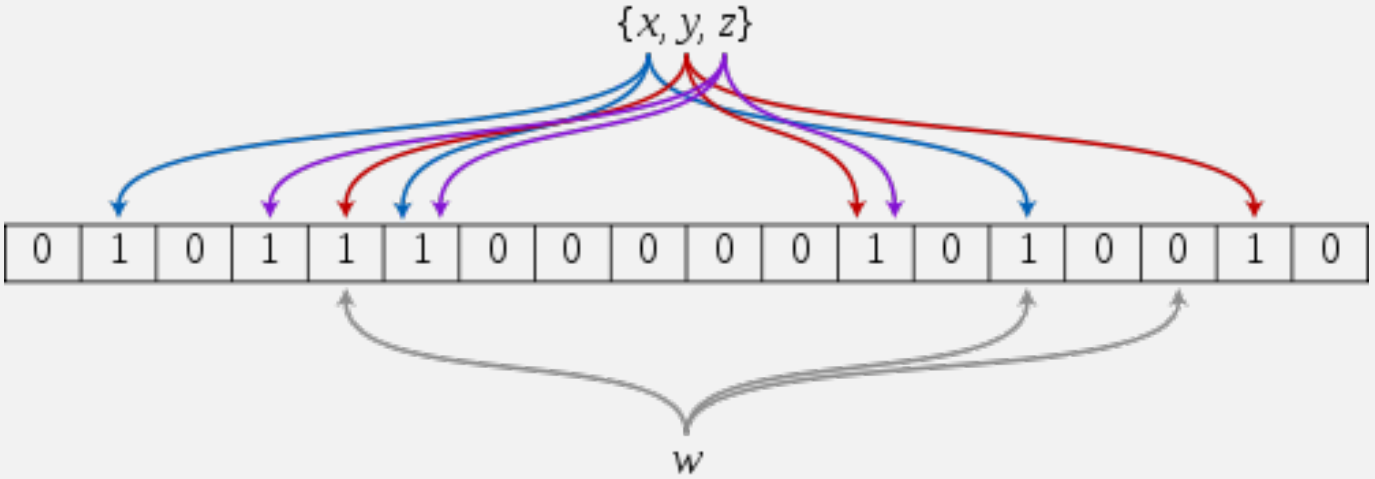
- Algorithms that produce the correct answer with some probability
- By re-running the algorithm many times, we can increase the probability to be arbitrarily close to 1.0.

Bloom Filters

- To resolve collisions, hash tables have to store keys: $O(mw)$ bits, where w is the number of bits in the key
- What if you want to store very large keys?
- *Radical idea:* Don't store the key in the table!
  - Potentially w-fold space reduction

Bloom Filters

- To reduce collisions, use multiple hash functions $h_1, \ldots, h_k$
- Hash table is simply a bitvector $B[1..m]$
- To insert key x, set $B[h_1(x)], B[h_2(x)], \ldots, B[h_k(x)]$



Images from Wikipedia Commons

- Membership check for y: all $B[h_i(y)]$ should be set
  - No false negatives, but false positives possible
- No deletions possible in the current algorithm.

Bloom Filters: False positives

- Prob. that a bit is *not* set by h_1 on inserting a key is $(1 - 1/m)$
  - The probability it is not set by any h_i is $(1 - 1/m)^k$
  - The probability it is not set after r key inserts is $(1 - 1/m)^{kr} \approx e^{-kr/m}$
- Complementing, the prob. p that a certain bit is set is $1 - e^{-kr/m}$
- For a false positive on a key y, all the bits that it hashes to should be a 1. This happens with probability

$$\left(1 - e^{-kr/m}\right)^k = (1 - p)^k$$

Bloom Filters

- Consider
$$\left(1 - e^{-kr/m}\right)^k$$
- Note that the table can potentially store very large number of entries with very low false positives
  - For instance, with $k = 20$, $m = 10^9$ bits (12M bytes), and a false positive rate of $2^{-10} = 10^{-3}$, can store $60M$ keys of arbitrary size!
- *Exercise:* What is the optimal value of k to minimize false positive rate for a given m and r?
  - But large k values introduce high overheads
- *Important:* Bloom filters can be used as a prefilter, e.g., if actual keys are in secondary storage (e.g., files or internet repositories)

Using arithmetic for substring matching

Problem: Given strings $T[1..n]$ and $P[1..m]$, find occurrences of P in T in $O(n+m)$ time.

Idea: To simplify presentation, assume P, T range over `[0-9]`

- Interpret $P[1..m]$ as digits of a number

$$p = 10^{m-1}P[1] + 10^{m-2}P[2] + \cdots 10^{m-m}P[m]$$

- Similarly, interpret $T[i..(i+m-1)]$ as the number t_i
- Note: P is a substring of T at i iff $p = t_i$
- To get t_{i+1}, shift $T[i]$ out of t_i, and shift in $T[i+m]$:

$$t_{i+1} = (t_i - 10^{m-1}T[i]) \cdot 10 + T[i+m]$$

We have an $O(n+m)$ algorithm. Almost: we still need to figure out how to operate on m-digit numbers in constant time!

Rabin-Karp Fingerprinting

Key Idea

- Instead of working with m-digit numbers,
- perform all arithmetic modulo a *random* prime number q,
- where $q > m^2$ fits within wordsize

- All observations made on previous slide still hold
  - Except that $p = t_i$ does not guarantee a match
  - Typically, we expect matches to be infrequent, so we can use $O(m)$ exact-matching algorithm to confirm probable matches.

Carter-Wegman-Rabin-Karp Algorithm

Difficulty with Rabin-Karp: Need to generate random primes, which is not an efficient task.

New Idea: Make the radix random, as opposed to the modulus

- We still compute modulo a prime q, but it is not random.

Alternative interpretation: We treat P as a polynomial

$$p(x) = \sum_{i=1}^{m} P[m-i] \cdot x^i$$

and evaluate this polynomial at a randomly chosen value of x

Like any probabilistic algorithm we can increase correctness probability by repeating the algorithm with different randoms.

- Different prime numbers for Rabin-Karp
- Different values of x for CWRK

Carter-Wegman-Rabin-Karp Algorithm

$$p(x) = \sum_{i=1}^{m} P[m-i] \cdot x^i$$

Random choice does not imply high probability of being right.

- You need to explicitly establish correctness probability.

So, what is the likelihood of false matches?

- A false match occurs if $p_1(x) = p_2(x)$, i.e., $p_1(x) - p_2(x) = p_3(x) = 0$.
- Arithmetic modulo prime defines a *field,* so an mth degree polynomial has $m+1$ roots.
- Thus, $(m+1)/q$ of the q (recall q is the prime number used for performing modulo arithmetic) possible choices of x will result in a false match, i.e., probability of false positive = $(m+1)/q$

Primality Testing

Fermat's Theorem

$a^{p-1} \equiv 1 \pmod{p}$

- Recall $\{1a, 2a, 3a, \ldots, (p-1)a\} \equiv \{1, 2, \ldots, p-1\} \pmod{p}$
- Multiply all elements of both sides:

$$(p-1)!a^{p-1} \equiv (p-1)! \pmod{p}$$

- Canceling out $(p-1)!$ from both sides, we have the theorem!

Primality Testing

- Given a number N, we can use Fermat's theorem as a probabilistic test to see if it is prime:
  - if $a^{N-1} \not\equiv 1 \pmod{N}$ then N is not prime
  - Repeat with different values of a to gain more confidence
- *Question:* If N is *not* prime, what is the probability that the above procedure will fail?
  - For Carmichael's numbers, the probability is 1 — but ignore this for now, since these numbers are very rare.
  - For other numbers, we can show that the above procedure works with probability 0.5

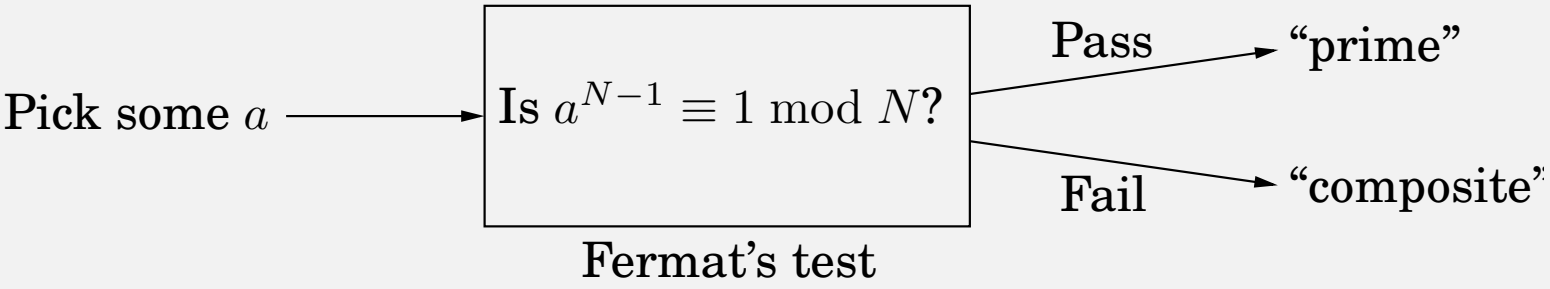
Primality Testing

Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for a relatively prime to N, then it holds for at least half the choices of $a < N$.

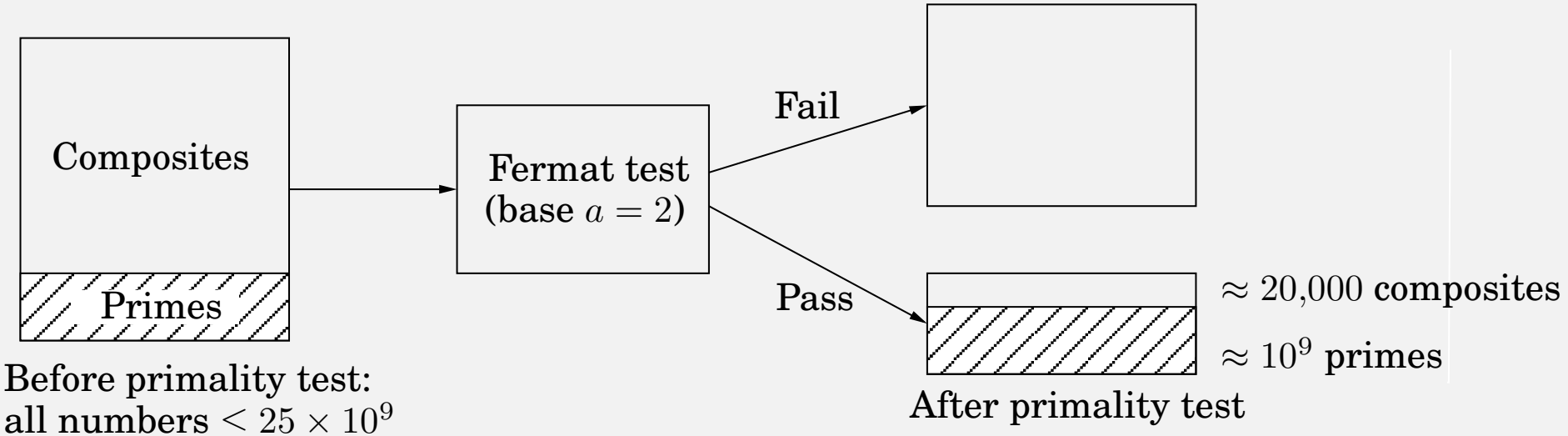
- If there is no b such that $b^{N-1} \equiv 1 \pmod{N}$, then we have nothing to prove.
- Otherwise, pick one such b, and consider $c \equiv ab$.
- Note $c^{N-1} \equiv a^{N-1}b^{N-1} \equiv a^{N-1} \not\equiv 1$
- Thus, for every b for which Fermat's test is satisfied, there exists a c that does not satisfy it.
  - Moreover, since a is relatively prime to N, $ab \not\equiv ab'$ unless $b \equiv b'$.
- Thus, at least half of the numbers $x < N$ that are relatively prime

Primality Testing



- When Fermat's test returns "prime" $Pr[N \text{ is not prime}] < 0.5$
- If Fermat's test is repeated for k choices of a, and returns "prime" in each case, $Pr[N \text{ is not prime}] < 0.5^k$
- In fact, 0.5 is an upper bound. Empirically, the probability has been much smaller.

Primality Testing



- Empirically, on numbers less than 25 billion, the probability of Fermat's test failing to detect non-primes (with $a = 2$) is more like 0.00002
- This probability decreases even more for larger numbers.

Prime number generation

Lagrange's Prime Number Theorem

For large N, primes occur approx. once every $\log N$ numbers.

Generating Primes

- Generate a random number
- Probabilistically test it is prime, and if so output it
- Otherwise, repeat the whole process

- What is the complexity of this procedure?
  - $O(\log^2 N)$ multiplications on $\log N$ bit numbers
- If N is not prime, should we try $N+1, N+2$, etc. instead of generating a new random number?
  - No, it is not easy to decide when to give up.

Rabin-Miller Test

- Works on Carmichael's numbers
- For prime number test, we consider only odd N, so $N-1=2^t u$ for some odd u
- Compute

$$a^u, a^{2u}, a^{4u}, \ldots, a^{2^t u} = a^{N-1}$$

- If a^{N-1} is not 1 then we know N is composite.
- Otherwise, we do a follow-up test on a^u, a^{2u} etc.
  - Let $a^{2^r u}$ be the first term that is equivalent to 1.
  - If $r > 0$ and $a^{2^{r-1}u} \not\equiv -1$ then N is composite
- This combined test detects non-primes with a probability of at least 0.75 for all numbers.

Power of Two Random Choices

If a single random choice yields unsatisfactory results, try making two choices and pick the better of two.

Example applications

Balls and bins: Maximum occupancy comes down from $O(\log n / \log \log n)$ to $O(\log \log n)$

Quicksort: Significantly increase odds of a balanced split if you pick three random elements and use their median as pivot

Load balancing: Random choice does not work well if different tasks take different time. Making two choices and picking the lighter loaded of the two can lead to much better outcomes

CSE 548: (*Design and*) Analysis of Algorithms

Amortized Analysis

R. Sekar

Amortized Analysis

Amortization

The spreading out of capital expenses for intangible assets over a specific period of time (usually over the asset's useful life) for accounting and tax purposes.

- A clever trick used by accountants to average large one-time costs over time.
- In algorithms, we use amortization to spread out the cost of expensive operations.
  - Example: Re-sizing a hash table.

Topics

Summation or Aggregate Method

- Some operations have high worst-case cost, but we can show that the worst case does not occur every time.
- In this case, we can average the costs to obtain a better bound

Summation

Let $T(n)$ be the worst-case running time for executing a sequence of n operations. Then the amortized time for each operation is $T(n)/n$.

Note: We are not making an "average case" argument about inputs.

We are still talking about worst-case performance.

Summation Example: Binary Counter

Incr($B[0..]$)

$i = 0$

while $B[i] = 1$

  $B[i] = 0$

  $i++$

$B[i] = 1$

- What is the worst-case runtime of *incr*?
  - Simple answer: $O(\log n)$, where $n = \#$ of *incr*'s performed
- What is the *amortized* runtime for n *incr*'s?
  - Easy to see that an *incr* will touch $B[i]$ once every 2^i operations.
  - Number of operations is thus

$$n \sum_{i=0}^{\log n} \frac{1}{2^i} = 2n$$

- Thus, amortized cost per *incr* is $O(1)$

Charging Method

Certain operations charge more than their cost so as to pay for other operations. This allows total cost to be calculated while ignoring the second category of operations.

- In the counter example, we charge 2 units for each operation to change a 0-bit to 1-bit.
- Pays for the cost of later flipping the 1-bit to 0-bit.
  - *Important: ensure you have charged enough.*
    - We have satisfied this: a bit can be flipped from 1 to 0 only once after it is flipped from 0 to 1.
- Now we ignore costs of 1 to 0 flips in the algorithm
  - There is only one 0-to-1 bit flipping per call of *incr*!
  - So, *incr* only costs 2 units for each invocation!

Stack Example

- Consider a stack with two operations:

  push(x): Push a value x on the stack

  pop(k): Pop off the top k elements

- What is the cost of a mix of n push and pop operations?
- *Key problem:* Worst-case cost of a *pop* is $O(n)$!
- *Solution:*
  - Charge 2 units for each push: covers the cost of pushing, and also the cost of a subsequent pop
  - A pushed item can be popped only once, so we have charged enough
  - Now, ignore pop's altogther, and trivially arrive at $O(1)$ amortized cost for the sequence of push/pop operations!

Potential Method

Define a potential for a data structure that is initially zero, and is always non-negative. The amortized cost of an operation is the cost of the operation minus the change in potential.

- Analogy with "potential" energy. "Potential" is prepaid cost that can be used subsequently
  - as the data structure changes and "releases" stored energy
- A more sophisticated technique that allows "charges" or "taxes" to be stored within nodes in a data structure and used subsequently at a later time.

Potential Method: Illustration

Stack:

- Each push costs 2 units because a push increases potential energy by 1.
- Pops can use the energy released by reduction in stack size!

Counter:

- Define potential as the number one 1-bits
- Changing a 0 to 1 costs 2 units, one for the operation and one to pay for increase in potential
- Changes of 1 to 0 can now be paid by released potential.

Hash Tables

- To provide expected constant time access, collisions need to be limited
- This requires hash table resizing when they become too full
  - But this requires all entries to be deleted from current table and inserted into a table that is larger — *a very expensive operation.*
- *Options:*
  1. Try to guess the table size right; if you guessed wrong, put up with the pain of low performance.
  2. Quit complaining, bite the bullet, and rehash as needed;
  3. *Amortize*: Rehash as needed, *and* prove that it does not cost much!

Amortized Rehashing

Amortize the cost of rehashing over other hash table operations

Approach 1: Rehash after a large number (say, 1K) operations.

Total cost of 1K ops = 1K for the ops + 1K for rehash = 2K

Note: We may have at most 1K elements in the table after 1K operations, so we may need to rehash at most 1K times.

So, amortized cost is just 2!

Are we done?

Amortized Rehash (2)

Are we done?

Consider total cost after 2K, 3K, and 4K operations:

$$T(2K) = 2K + 1K \text{ (first rehash)} + 2K \text{ (second rehash)} = 5K$$

$$T(3K) = 3K + 1K \text{ (1}^{st}\text{ rehash)} + 2K \text{ (2}^{nd}\text{ rehash)} + 3K \text{ (3}^{rd}\text{...)} = 9K$$

$$T(4K) = 4K + 1K + 2K + 3K + 4K = 14K$$

Hmmm. This is growing like n^2, so amortized cost will be $O(n)$

Need to try a different approach.

Amortized Rehash (3)

Approach 2: Double the hash table whenever it gets full

Say, you start with an empty table of size N. For simplicity, assume only insert operations.

You invoke N insert operations, then rehash to a $2N$ table.

$$T(N) = N + N \text{ (rehashing } N \text{ entries)} = 2N$$

Now, you can insert N more before needing rehash.

$$T(2N) = T(N) + N + 2N \text{ (rehashing } 2N \text{ entries)} = 5N$$

Now, you can insert $2N$ more before needing rehash:

$$T(4N) = T(2N) + 2N + 4N \text{ (rehashing } 4N \text{ entries)} = 11N$$

The general recurrence is $T(n) = T(n/2) + 1.5n$, which is linear.

So, amortized cost is constant!

Amortized Rehash (4)

Alternatively, we can think in terms of *charging*.

Each insert operation can be charged 3 units of cost:

- One for the insert operation
- One for rehashing of this element at the end of this run of inserts
- One for rehashing an element that was already in the hash table when this run began

A run contains as many elements as the hash table at the beginning of run — so we have accounted for all costs.

Thus, rehashing

- increases the costs of insertions by a factor of 3.
- lookup costs are unchanged.

Amortized Rehash (5)

- Alternatively, we can think in terms of *potential*.
- Hash table as a spring: as more elements are inserted, the spring has to be compressed to make room.
- Let $|H|$ denote the capacity and α the occupancy of H
- Define potential as 0 when $\alpha \leq 0.5$ and $2(\alpha - 0.5)|H|$ otherwise.
- Immediately after resize, let the hash table capacity be k. Note $\alpha \leq 0.5$ so potential is 0.
- Each insert (after α reaches 0.5) costs 3 units: one for the operation, and 2 for the increase in potential.
- When α reaches 1, the potential is $2k$. After resizing to $2k$, potential falls to 0, and the released $2k$ cost pays for rehashing $2k$ elements.

Amortized Rehash (6)

- What if we increase the size by a factor less than 2?
  - Is there a threshold $t > 1$ such that expansion by a factor less than t won't yield amortized constant time?
- What happens if we want to support both deletes and inserts, and want to make sure that the table never uses more than k times the actual number of elements?
  - Is there a minimum value of k for which this can be achieved?
  - Do you need a different threshold for expansion and contraction? Are there any constraints on the relationship between these two thresholds to ensure amortized constant time?

Amortized performance of Vectors vs Lists

Linked lists: Data structures of choice if you don't know the total number of elements in advance.

Space inefficient: 2x or more memory for very small objects.

Poor cache performance: Pointer chasing is cache unfriendly.

Sequential access: No fast access to kth element.

Vectors: Dynamically-sized arrays have none of these problems. But resizing is expensive.

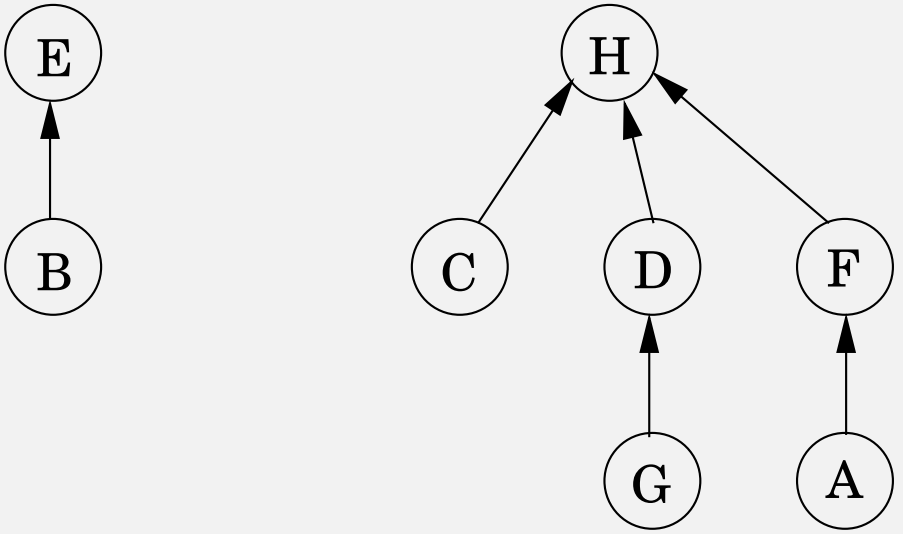
- Is it possible to achieve good amortized performance?
- When should the vector be expanded/contracted?
- What operations can we support in constant amortized time? Inserts? insert at end? concatenation?

Strings: We can raise similar questions as Vectors.

Disjoint Sets

- Represent disjoint sets as "inverted trees"
- Each element has a parent pointer π
- To compute the union of set A with B, simply make B's root the parent of A's root.

A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.



Disjoint Sets (2)

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x) :  x = π(x)
return x

procedure union(x, y)
r_x = find(x)
r_y = find(y)
π(r_y) = r_x
```

Complexity

- `makeset` takes $O(1)$ time
- `find` takes time equal to depth of set: $O(n)$ in the worst case.
- `union` takes $O(1)$ time on a root element; in the worst case, its complexity matches `find`.

Amortized complexity

- Can you construct a worst-case example, where N operations take $O(N^2)$ time?
- Can we improve this?

Disjoint Sets with Union by Depth

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x): x = π(x)
return x
```

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y):
        rank(r_y) = rank(r_y) + 1
```

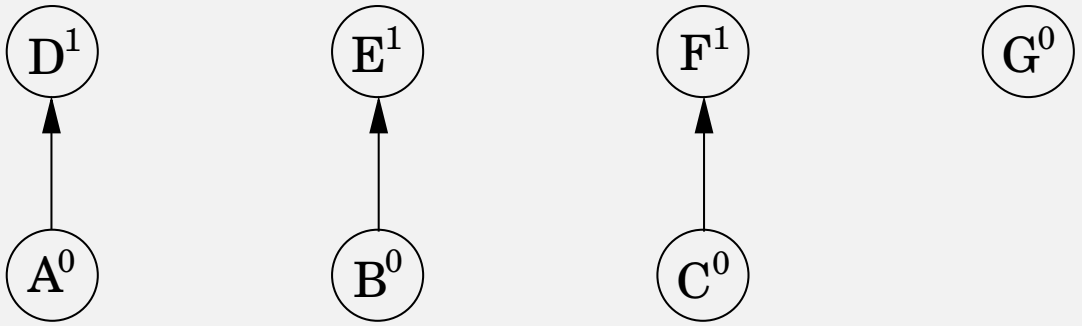
rank of a node is the height of subtree rooted at that node.

Disjoint Sets with Union by Depth (2)

Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.

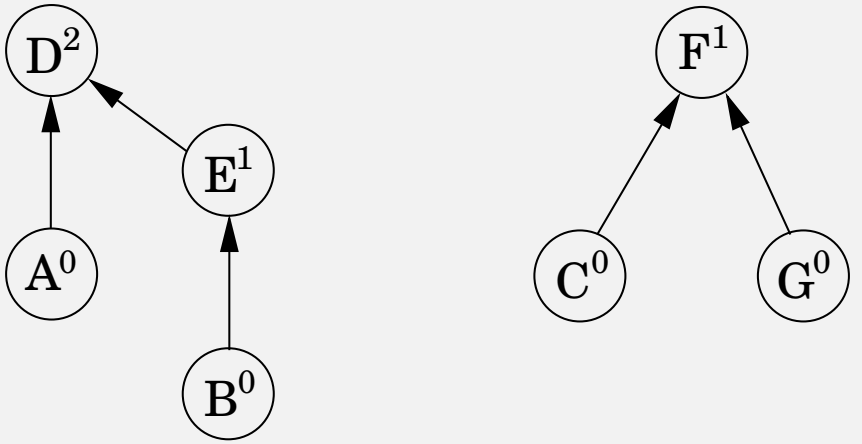
After `makeset`(A), `makeset`(B), ..., `makeset`(G):

After `union`(A, D), `union`(B, E), `union`(C, F):

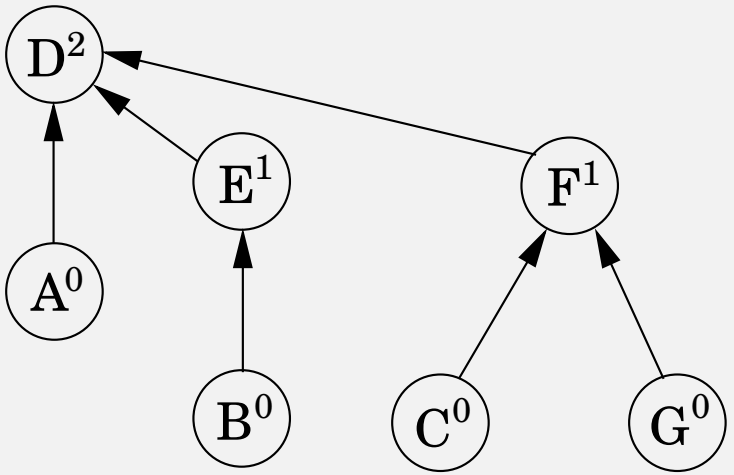


Disjoint Sets with Union by Depth (3)

After `union`(C, G), `union`(E, A):



After `union`(B, G):



Complexity of disjoint sets w/ union by depth

- Asymptotic complexity of `makeset` unchanged.
  - `union` has become a bit more expensive, but only modestly.
- What about `find`?
  - A sequence of N operations can create at most N elements
  - So, maximum set size is $O(N)$
  - With union by rank, each increase in rank can occur only after a doubling of elements in the set

Observation

The number of nodes of rank k never exceeds $N/2^k$

- So, height of trees is bounded by $\log N$

Complexity of disjoint sets w/ union by depth (2)

- Height of trees is bounded by $\log N$
- Thus we have a complexity of $O(\log N)$ for `find`
  - *Question:* Is this bound tight?

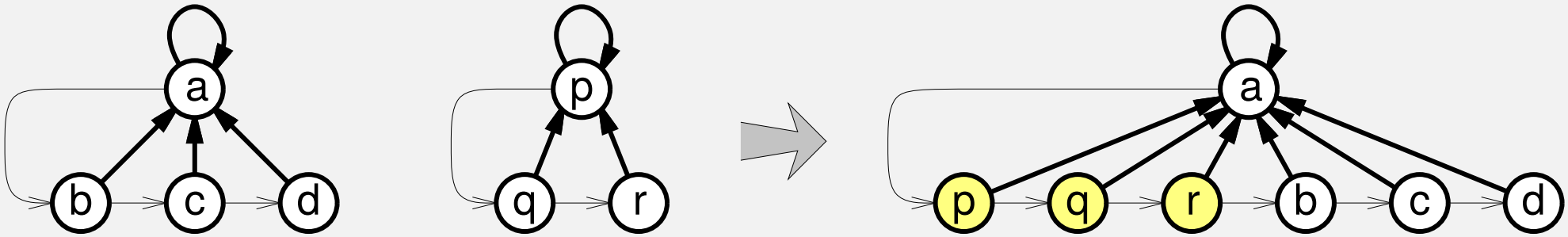
From here on, we limit union operations to only root nodes, so their cost is $O(1)$.

This requires find to be moved out of union into a separate operation, and hence the total number of operations increases, but only by a constant factor.

Improving `find` performance

Idea: Why not force depth to be 1? Then `find` will have $O(1)$ complexity!

Approach: **Threaded Trees**



Problem: Worst-case complexity of `union` becomes $O(n)$

Solution:

- Merge smaller set with larger set
- Amortize cost of `union` over other operations

Sets w/ threaded trees: Amortized analysis

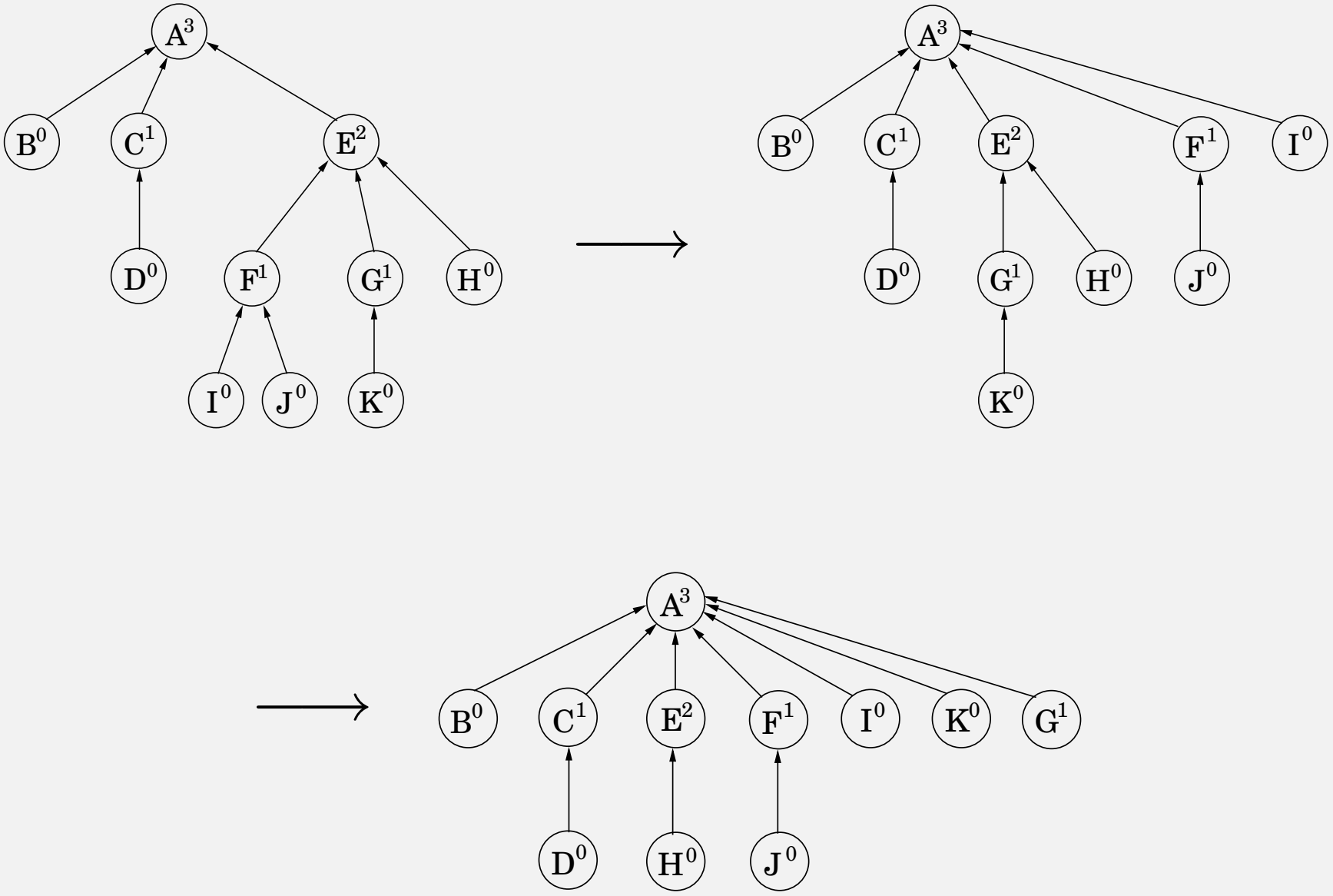
- Other than cost of updating parent pointers, `union` costs $O(1)$
- **Idea:** Charge the cost of updating a parent pointer to an element.
- **Key observation:** Each time an element's parent pointer changes, it is in a set that is twice as large as before
  - So, with n operations, you can at most $O(\log n)$ parent pointer updates per element
- Thus, amortized cost of n operations, consisting of some mix of `makeset`, `find` and `union` is at most $n \log n$

Further improvement

- Can we combine the best elements of the two approaches?
  - Threaded trees employ an *eager* approach to `union` while the original approach used a *lazy* approach
    - Eager approach is better for `find`, while being lazy is better for `union`.
    - So, why not use lazy approach for `union` and eager approach for `find`?
- **Path compression:** Retains *lazy* `union`, but when a `find(x)` is called, *eagerly* promotes x to the level beloe the root
  - Actually, we promote $x, \pi(x), \pi(\pi(x)), \pi(\pi(\pi(x)))$ and so on.
  - As a result, subsequent calls to find x or its parents become cheap.
- From here on, we let *rank* be defined by the *union* algorithm
  - For root node, *rank* is same as depth
  - But once a node becomes a non-root, its *rank* stays fixed,
    - *even when path compression decreases its depth.*

Disjoint sets w/ Path compression: Illustration

`find`(I) followed by `find`(K)



Sets w/ Path compression: Amortized analysis

Amortized cost per operation of n set operations is $O(\log^* n)$ where

$$\log^* x = \text{smallest } k \text{ such that } \underbrace{log(log(\cdots log}_{k \text{ times}}(x)\cdots)) = 1$$

Note: $\log^*(x) \leq 5$ for virtually any n of practical relevance.

Specifically,

$$\log^*(2^{65536}) = \log^*(2^{2^{2^{2}}}) = 5$$

Note that 2^{65536} is approximately a $20,000$ digit decimal number.

We will never be able to store input of that size, at least not in our universe. (Universe contains may be 10^{100} elementary particles.)

So, we might as well treat $\log^*(n)$ as $O(1)$.

CSE 548: (*Design and*) Analysis of Algorithms

String Algorithms

R. Sekar

String Matching

Strings provide the primary means of interfacing to machines.

- programs, documents, ...

Consequently, string matching is central to numerous, widely-used systems and tools

- Compilers and interpreters, command processors (e.g., bash), text-processing tools (sed, awk, ...)
- Document searching and processing, e.g., grep, Google, NLP tools, ...
- Editors and word-processors
- File versioning and compression, e.g., rcs, svn, rsync, ...
- Network and system management, e.g., intrusion detection, performance monitoring, ...
- Computational biology, e.g., DNA alignment, mutations, evolutionary trees, ...

Topics

Terminology

String: List $S[1..i]$ of characters over an alphabet Σ.

Substring: A string $P[1..j]$ such that for $P[1..j] = S[l+1..l+j]$ for some l.

Prefix: A substring P of S occurring at its beginning

Suffix: A substring P of S occurring at its end

Subsequence: Similar to substring, but the the elements of P need not occur contiguously in S.

For instance, *bcd* is a substring of *abcde*, while *de* is a suffix, *abcd* is a prefix, and *acd* is a subsequence. A substring (or prefix/suffix/subsequence) T of S is said to be *proper* if $T \neq S$.

String Matching Problems

Given a "pattern" string p and another string s:

Exact match: Is p a *substring* of s?

Match with wildcards: In this case, the pattern can contain wildcard characters that can match any character in s

Regular expression match: In this case, p is regular expression

Substring/prefix/suffix: Does a (sufficiently long) substring/prefix/suffix of p occur in s?

Approximate match: Is there a substring of s that is within a certain edit distance from p?

Multi-match: Instead of a single pattern, you are given a set $p_1, .., p_n$ of patterns. Applies to all above problems.

String Matching Techniques

Finite-automata and variants: Regexp matching, Knuth-Morris-Pratt, Aho-Corasick

Seminumerical Techniques: Shift-and, Shift-and with errors, Rabin-Karp, Hash-based

Suffix trees and suffix arrays: Techniques for finding substrings, suffixes, etc.

Language of Regular Expressions

Notation to represent (potentially) infinite sets of strings over alphabet Σ.

Let R be the set of all regular expressions over Σ. Then,

Empty String : $\epsilon \in R$

Unit Strings : $\alpha \in \Sigma \Rightarrow \alpha \in R$

Concatenation : $r_1, r_2 \in R \Rightarrow r_1 r_2 \in R$

Alternative : $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$

Kleene Closure : $r \in R \Rightarrow r^* \in R$

Regular Expression

a : stands for the set of strings {a}

$a \mid b$: stands for the set {a, b}

- *Union* of sets corresponding to REs a and b

ab : stands for the set {ab}

- Analogous to set *product* on REs for a and b
  - $(a|b)(a|b)$: stands for the set {aa, ab, ba, bb}.

a^* : stands for the set $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$ that contains all strings of zero or more a's.

- Analogous to *closure* of the product operation.

Regular Expression Examples

$(a|b)^*$: Set of strings with zero or more a's and zero or more b's:

$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

(a^*b^*) : Set of strings with zero or more a's and zero or more b's such that all a's occur before any b:

$\{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, \ldots\}$

$(a^*b^*)^*$: Set of strings with zero or more a's and zero or more b's:

$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

Semantics of Regular Expressions

Semantic Function $\mathcal{L}$: Maps regular expressions to sets of strings.

$$
\begin{aligned}
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(\alpha) &= \{\alpha\} \qquad (\alpha \in \Sigma) \\
\mathcal{L}(r_1 \mid r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1\ r_2) &= \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{aligned}
$$

Finite State Automata

Regular expressions are used for *specification,* while FSA are used for computation.

FSAs are represented by a labeled directed graph.

- A finite set of *states* (vertices).
- *Transitions* between states (edges).
- *Labels* on transitions are drawn from $\Sigma \cup \{\epsilon\}$.
- One distinguished *start* state.
- One or more distinguished *final* states.

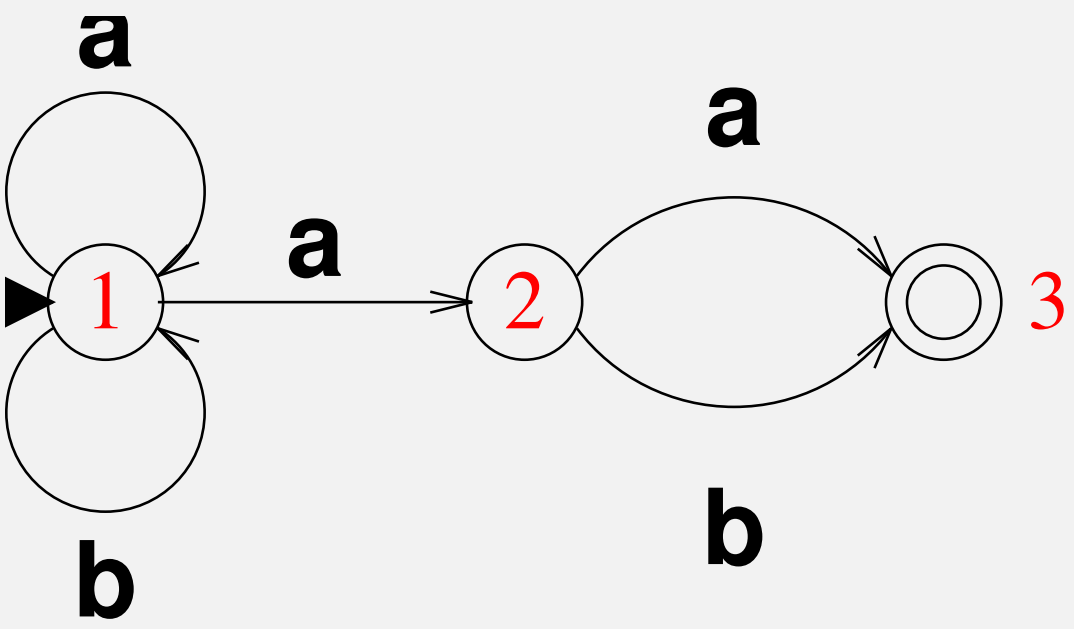
Finite State Automata: An Example

Consider the Regular Expression $(a \mid b)^* a(a \mid b)$.

$\mathcal{L}((a \mid b)^* a(a \mid b)) = \{\texttt{aa}, \texttt{ab}, \texttt{aaa}, \texttt{aab}, \texttt{baa}, \texttt{bab},$

$\texttt{aaaa}, \texttt{aaab}, \texttt{abaa}, \texttt{abab}, \texttt{baaa}, \ldots\}$.

The following (non-deterministic) automaton determines whether an input string belongs to $\mathcal{L}((a \mid b)^* a(a \mid b))$:

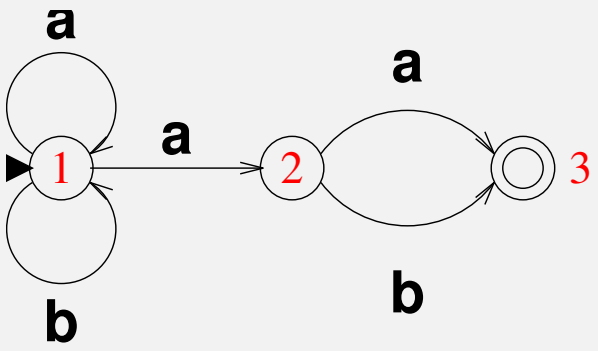


Determinism

$(a \mid b)^* a(a \mid b)$:

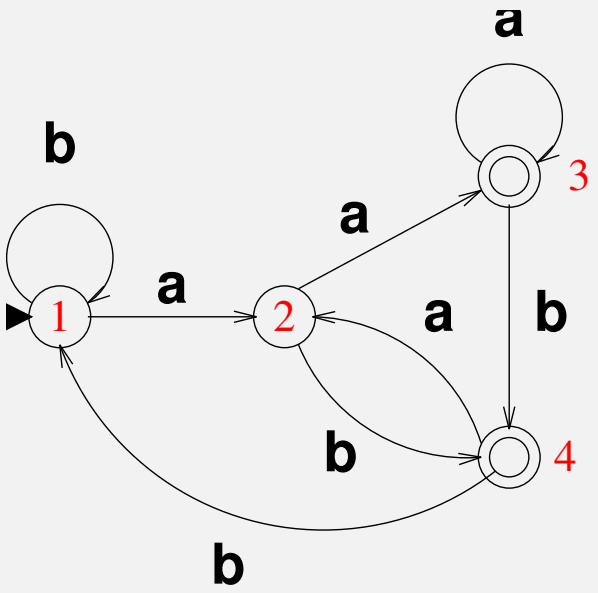
Nondeterministic:

(NFA)



Deterministic:

(DFA)



Acceptance Criterion

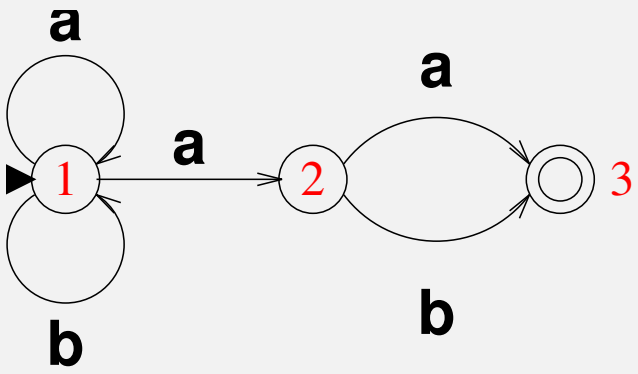
A finite state automaton (NFA or DFA) *accepts* an input string x

- . . . if beginning from the start state
- . . . we can trace some path through the automaton
- . . . such that the sequence of edge labels spells x
- . . . and end in a final state.

Or, there exists a path in the graph from the start state to a final state such that the sequence of labels on the path spells out x

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?

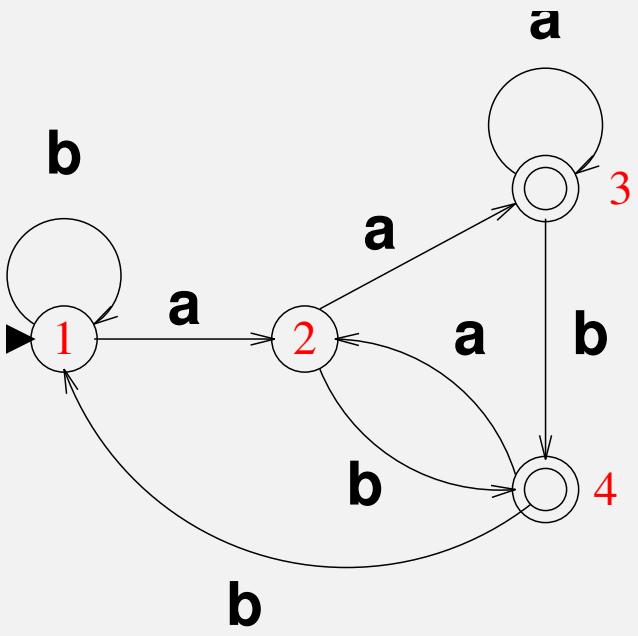


Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	

Accept

Recognition with a DFA

Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	b	a	b	
Path:	1	2	4	2	4	Accept

NFA vs. DFA

For every NFA, there is a DFA that accepts the same set of strings.

- NFA may have transitions labeled by ϵ.

  (Spontaneous transitions)

- All transition labels in a DFA belong to Σ.
- For some string x, there may be *many* accepting paths in an NFA.
- For all strings x, there is *one unique* accepting path in a DFA.
- Usually, an input string can be recognized *faster* with a DFA.
- NFAs are typically *smaller* than the corresponding DFAs.

NFA vs. DFA

n = Size of Regular Expression (pattern)

m = Length of Input String (subject)

	NFA	**DFA**
Size of Automaton	$O(n)$	$O(2^n)$
Recognition time per input string	$O(n \times m)$	$O(m)$

Converting RE to FSA

NFA: Compile RE to NFA (Thompson's construction [1968]), then match.

DFA: Compile to DFA, then match

(A) Convert NFA to DFA (Rabin-Scott construction), minimize

(B) Direct construction: RE derivatives [Brzozowski 1964].

- More convenient and a bit more general than (A).

(C) Direct construction of [McNaughton Yamada 1960]

- Can be seen as a (more easily implemented) specialization of (B).
- Used in Lex and its derivatives, i.e., most compilers use this algorithm.

Converting RE to FSA

- NFA approach takes $O(n)$ NFA construction plus $O(nm)$ matching, so has worst case $O(nm)$ complexity.
- DFA approach takes $O(2^n)$ construction plus $O(m)$match, so has worst case $O(2^n + m)$ complexity.
- So, why bother with DFA?
  - In many practical applications, the pattern is fixed and small, while the subject text is very large. So, the $O(mn)$ term is dominant over $O(2^n)$
  - For many important cases, DFAs are of polynomial size
  - In many applications, exponential blow-ups don't occur, e.g., compilers.

Derivative of Regular Expressions

The derivative of a regular expression R w.r.t. a symbol x, denoted $\partial_x[R]$ is another regular expression R' such that $\mathcal{L}(R) = \mathcal{L}(xR')$

Basically, $\partial_x[R]$ captures the suffixes of those strings that match R and start with x.

Examples

- $\partial_a[a(b|c)] = b|c$
- $\partial_a[(a|b)cd] = cd$
- $\partial_a[(a|b)* cd] = (a|b)* cd$
- $\partial_c[(a|b)* cd] = d$
- $\partial_d[(a|b)* cd] = \emptyset$

Definition of RE Derivative (1)

inclEps(R): A predicate that returns true if $\epsilon \in \mathcal{L}(R)$

$$
\begin{aligned}
\mathit{inclEps}(a) &= \mathit{false}, \quad \forall a \in \Sigma \\
\mathit{inclEps}(R_1|R_2) &= \mathit{inclEps}(R_1) \vee \mathit{inclEps}(R_2) \\
\mathit{inclEps}(R_1R_2) &= \mathit{inclEps}(R_1) \wedge \mathit{inclEps}(R_2) \\
\mathit{inclEps}(R*) &= \mathit{true}
\end{aligned}
$$

Note *inclEps* can be computed in linear-time.

Definition of RE Derivative (2)

$$
\begin{aligned}
\partial_a[a] &= \epsilon \\
\partial_a[b] &= \emptyset \\
\partial_a[R_1|R_2] &= \partial_a[R_1]|\partial_a[R_2] \\
\partial_a[R*] &= \partial_a[R]R* \\
\partial_a[R_1R_2] &= \partial_a[R_1]R_2|\partial_a[R_2] \quad \text{if } \mathit{inclEps}(R_1) \\
&= \partial_a[R_1]R_2 \quad \text{otherwise}
\end{aligned}
$$

Note: $\boxed{\mathcal{L}(\epsilon) = \{\epsilon\}} \neq \boxed{\mathcal{L}(\emptyset) = \{\}}$

DFA Using Derivatives: Illustration

Consider $R_1 = (a|b)* a(a|b)$

$\partial_a[R_1] = R_1|(a|b) = R_2$

$\partial_b[R_1] = R_1$

$\partial_a[R_2] = R_1|(a|b)|\epsilon = R_3$

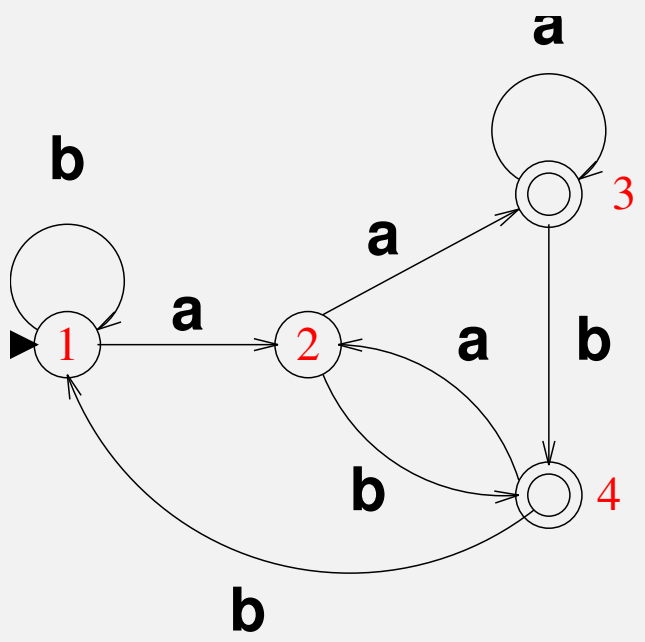
$\partial_b[R_2] = R_1|\epsilon = R_4$

$\partial_a[R_3] = R_1|(a|b)|\epsilon = R_3$

$\partial_b[R_3] = R_1|\epsilon = R_4$

$\partial_a[R_4] = R_1|(a|b) = R_2$

$\partial_b[R_4] = R_1$

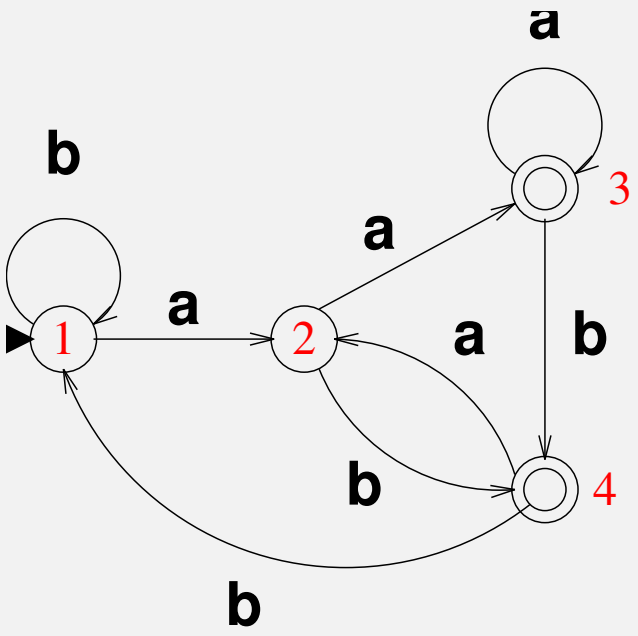


McNaughton-Yamada Construction

Can be viewed as a simpler way to represent derivatives

- Positions in RE are numbered, e.g., ${}^{0}(a^{1}|b^{2})* a^{3}(a^{4}|b^{5})\$^{6}$.
- A derivative is identified by its beginning position in the RE
  - Or more generally, a derivative is identified by a set of positions
- Each DFA state corresponds to a position set (pset)

$$
\begin{aligned}
R_1 &\equiv \{1,2,3\} \\
R_2 &\equiv \{1,2,3,4,5\} \\
R_3 &\equiv \{1,2,3,4,5,6\} \\
R_4 &\equiv \{1,2,3,6\}
\end{aligned}
$$



McNaughton-Yamada: Definitions

first(P): Yields the set of first symbols of RE denoted by pset P

Determines the transitions out of DFA state for P

Example: For the RE $(a^1|b^2)* a^3(a^4|b^5)\$^6$,

$$first(\{1,2,3\}) = \{a,b\}$$

$P|_s$: Subset of P that contain s, i.e., $\{p \in P \mid R \text{ contains } s \text{ at } p\}$

Example: $\{1,2,3\}|_a = \{1,3\}$, $\{1,2,4,5\}|_b = \{2,5\}$

follow(P): Yields the set of positions that immediately follow P.

Note: $follow(P) = \bigcup_{p \in P} follow(\{p\})$

Definition is very similar to derivatives

Example: $follow(\{3,4\}) = \{4,5,6\}$

$follow(\{1\}) = \{1,2,3\}$

McNaughton-Yamada Construction (2)

BuildMY$(R, pset)$

Create an automaton state S labeled *pset*

Mark this state as final if \$ occurs in R at *pset*

foreach symbol $x \in first(pset) - \{\$\}$ **do**

Call *BuildMY*$(R, follow(pset|_x))$ if hasn't previously been called

Create a transition on x from S to

the root of this subautomaton

DFA construction begins with the call *BuildMY*$(R, follow(\{0\}))$. The root of the resulting automaton is marked as a start state.

BuildMY Illustration on $R = {}^{0}(a^{1}|b^{2})* a^{3}(a^{4}|b^{5})\$^{6}$

Computations Needed

$follow(\{0\}) = \{1,2,3\}$

$follow(\{1\}) = follow(\{2\}) = \{1,2,3\}$

$follow(\{3\}) = \{4,5\}$

$follow(\{4\}) = follow(\{5\}) = \{6\}$

$\{1,2,3\}|_a = \{1,3\}, \quad \{1,2,3\}|_b = \{2\}$

$follow(\{1,3\}) = \{1,2,3,4,5\}$

$\{1,2,3,4,5\}|_a = \{1,3,4\}$

$\{1,2,3,4,5\}|_b = \{2,5\}$

$follow(\{1,3,4\}) = \{1,2,3,4,5,6\}$

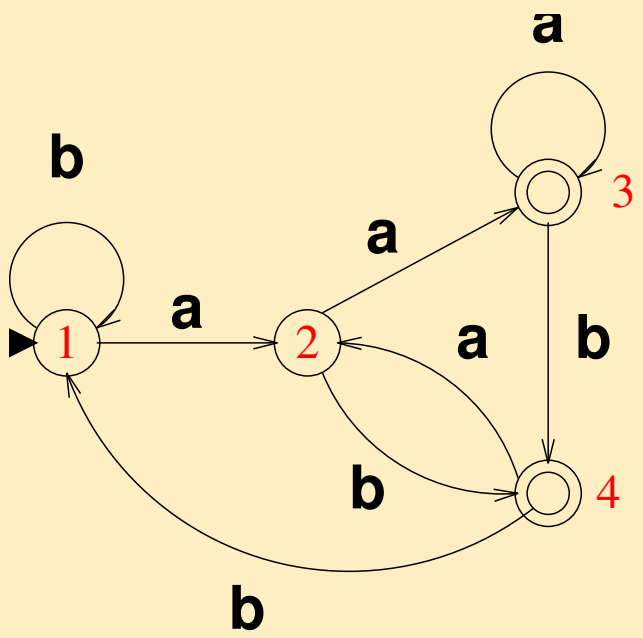
$follow(\{2,5\}) = \{1,2,3,6\}$

$\{1,2,3,4,5,6\}|_a = \{1,3,4\}$

$\{1,2,3,4,5,6\}|_b = \{2,5\}$

$\{1,2,3,6\}|_a = \{1,3\} \quad \{1,2,3,6\}|_b = \{2\}$

Resulting Automaton



State	Pset
1	{1,2,3}
2	{1,2,3,4,5}
3	{1,2,3,4,5,6}
4	{1,2,3,6}

McNaughton-Yamada (MY) Vs Derivatives

- Conceptually very similar
- MY takes a bit longer to describe, and its correctness a bit harder to follow.
- MY is also more mechanical, and hence is found in most implementations
- Derivatives approach is more general
  - Can support some extensions to REs, e.g., complement operator
  - Can avoid some redundant states during construction
    - Example: For $ac|bc$, DFA built by derivative approach has 3 states, but the one built by MY construction has 4 states

      The derivative approach merges the two c's in the RE, but with MY, the two c's have different positions, and hence operations on them are not shared.

Avoiding Redundant States

- Automata built by MY is not optimal
  - Automata minimization algorithms can be used to produce an optimal automaton.
- Derivatives approach associates DFA states with derivatives, but does not say how to determine equality among derivatives.
- There is a spectrum of techniques to determine RE equality
  - MY is the simplest: relies on syntactic identity
  - At the other end of the spectrum, we could use a complete decision procedure for RE equality.
    - In this case, the derivative approach yields the optimal RE!
  - In practice we would tend to use something in the middle
    - Trade off some power for ease/efficiency of implementation

RE to DFA conversion: Complexity

- Given DFA size can be exponential in the worst case, we obviously must accept worst-case exponential complexity.
- For the derivatives approach, it is not immediately obvious that it even terminates!
  - More obvious for McNaughton-Yamada approach, since DFA states correspond to position sets, of which there are only 2^n.
- Derivative computation is linear in RE size in the general case.
- So, overall complexity is $O(n2^n)$
- Complexity can be improved, but the worst-case 2^n takes away some of the rationale for doing so.
  - Instead, we focus on improving performance in many frequently occurring special cases where better complexity is achievable.

RE Matching: Summary

- Regular expression matching is much more powerful than matching on plain strings (e.g., prefix, suffix, substring, etc.)
- Natural that RE matching algorithms can be used to solve plain string matching
  - But usually, you pay for increased power: more complex algorithms, larger runtimes or storage.

We study the RE approach because it seems to not only do RE matching, but yield simpler, more efficient algorithms for matching plain strings.

String Lookup

Problem: Determine if s equals any of the strings $p_1, \ldots, p_k$.

- Equivalent to the question: does the RE $p_1|p_2|\cdots|p_k$ match s?
- We can use the derivative approach, except that derivatives are very easy to compute.
  - Or, we can use *BuildMY* — once again, *follow*() sets are very easy to compute for this class of regular expressions.
- Results in an FSA that is a tree
- More commonly known as a *trie*

Trie Example

$$R_0 = top|tool|tooth|at|sunk|sunny$$

$$R_1 = \partial_t[R_0] = op|ool|ooth$$

$$R_2 = \partial_o[R_1] = p|ol|oth$$

$$R_3 = \partial_p[R_2] = \epsilon$$

$$R_4 = \partial_o[R_2] = l|th$$

$$R_5 = \partial_l[R_4] = \epsilon$$

$$R_6 = \partial_t[R_4] = h, R_7 = \partial_h[R_6] = \epsilon$$

$$R_8 = \partial_a[R_0] = t, R_9 = \partial_t[R_8] = \epsilon$$

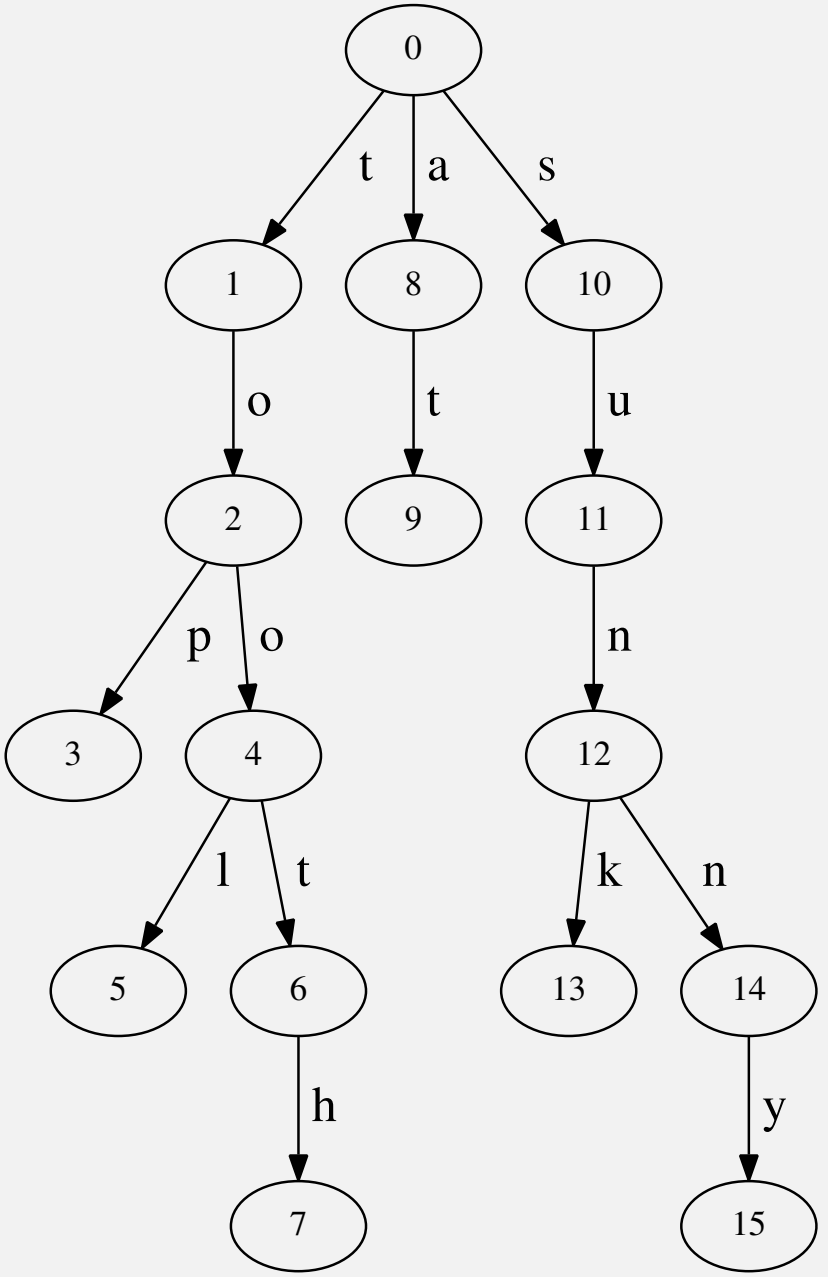
$$R_{10} = \partial_s[R_0] = unk|unny$$

$$R_{11} = \partial_u[R_{10}] = uk|nny$$

$$R_{12} = \partial_n[R_{11}] = k|ny$$

$$R_{13} = \partial_k[R_{12}] = \epsilon$$

$$R_{14} = \partial_n[R_{12}] = y, R_{15} = \partial_y[R_{14}] = \epsilon$$



Trie Summary

- A data structure for efficient lookup
  - Construction time linear in the size of keywords
  - Search time linear in the size of the input string
- Can also support maximal common prefix (MCP) query
- Can also be used for efficient representation of string sets
  - Takes $O(|s|)$ time to check if s belongs to the set
  - Set union/intersection are linear in size of the smaller set
    - Sublinear in input size when one input trie is much larger than the other
  - Can compute set difference as well — with same complexity.

Implementing Transitions

How to implement transitions?

Array: Efficient, but unacceptable space when $|\Sigma|$ is large

Linked list: Space-efficient, but slow

Hash tables: Mid-way between the above two options, but noticeably slower than arrays. Collisions are a concern.

- But customized hash tables for this purpose can be developed.
- Alternatively, since transition tables are static, we can look for perfect hash functions

Specialized representations: For special cases such as exact search, we could develop specialized alternatives that are more efficient than all of the above.

Exact Search

- Determine if a *pattern* $P[1..n]$ occurs within *text* $S[1..m]$
  - Find j such that $P[1..n] = S[j..(j+n-1)]$
- An RE matching problem: Does $\Sigma^* P \Sigma^*$ match S?
  - Note: Σ^* matches any arbitrary string (incl. ϵ)
- We consider $\Sigma^* p$ since it can identify all matches
  - A match can be reported each time a final state is reached.
  - In contrast, an automaton for $\Sigma^* P \Sigma^*$ may not report all matches

Exact Search Example

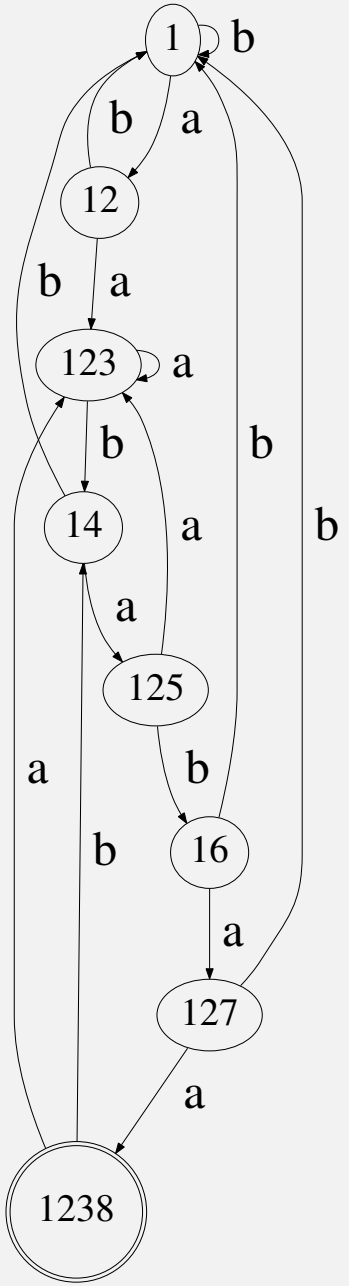
Consider $R_0 = (\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

We use McNaughton-Yamada. Recall that with this technique:

- States are identified by position sets.
- A position denotes a derivative starting at that position
- A position set indicates the union of REs corresponding to each position.

For instance, position set $\{0, 2, 3\}$ represents

$$R_0 | a^2 b^3 a^4 b^5 a^6 a^7 | b^3 a^4 b^5 a^6 a^7$$



Exact Search: Complexity

- **Positives:**
  - Matching is very fast, taking only $O(m)$ time.
  - Only linear (rather than exponential) number of states
- **Downsides:**
  - Construction of psets for each state takes up to $O(n)$ time
  - Thus, overall complexity of automata construction is $O(n^2)$
    - Can be $O(n^2|\Sigma|)$ since each state may have up to $|\Sigma|$ transitions
- **Question:** Can we do better?
  - Faster construction
    - $O(n)$ instead of $O(n^2)$?
  - More efficient representation for transitions.
    - constant number of transitions per state?

Improving Exact Search: Observations

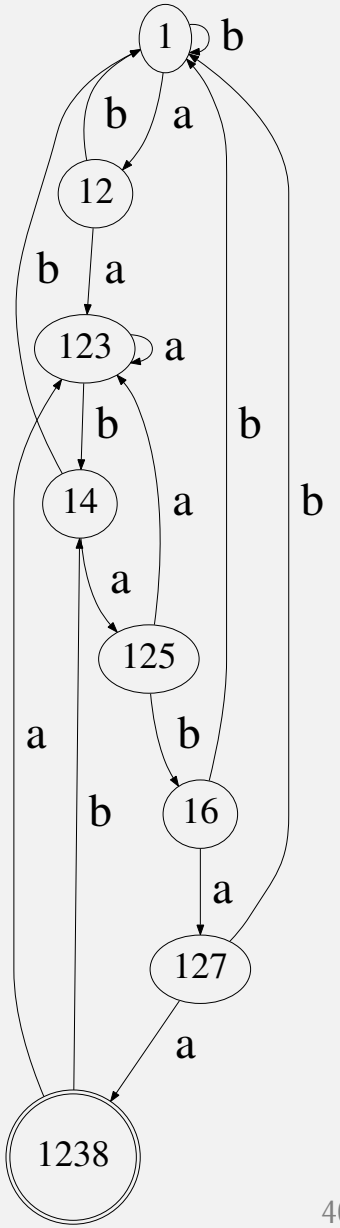
$(\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

The DFA has a linear structure, with states 1 to $n+1$:

- State i is reached on matching the prefix $P[1..i-1]$
- The largest element of $pset(i)$ is i
- If you are in state i after scanning $S[k]$:
  - Let $P' = P[1..i-1] = S[k-i+2..k]$
  - *"Unwinding" of Σ^*:* A prefix of $S[k-i+2..k]$ can be matched with Σ^*, with the rest matching $P[1..j-1]$

  So, $pset(i)$ includes every j such that $S[k-i+2..k] = P[1..j-1] = P[..i-1]$

S	a	a	b	a	b	a	a	
Viable match 1	a^1	a^2	b^3	a^4	b^5	a^6	a^7	$\$^8$
Viable match 2	Σ	Σ	Σ	Σ	Σ	a^1	a^2	b^3
Viable match 3	Σ	Σ	Σ	Σ	Σ	Σ	a^1	a^2
Viable match 4	Σ	Σ	Σ	Σ	Σ	Σ	Σ	a^1

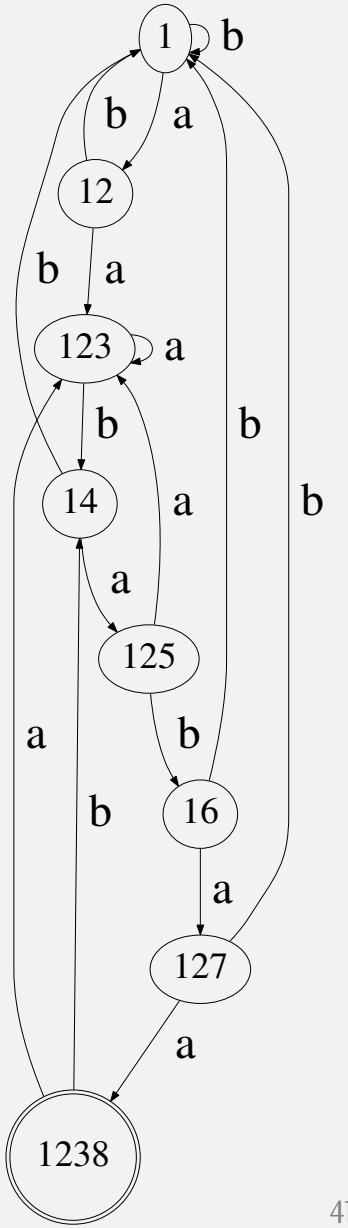


Improving Exact Search: Key Ideas

$(\Sigma^0)^* a^1 a^2 b^3 a^4 b^5 a^6 a^7 \$^8$

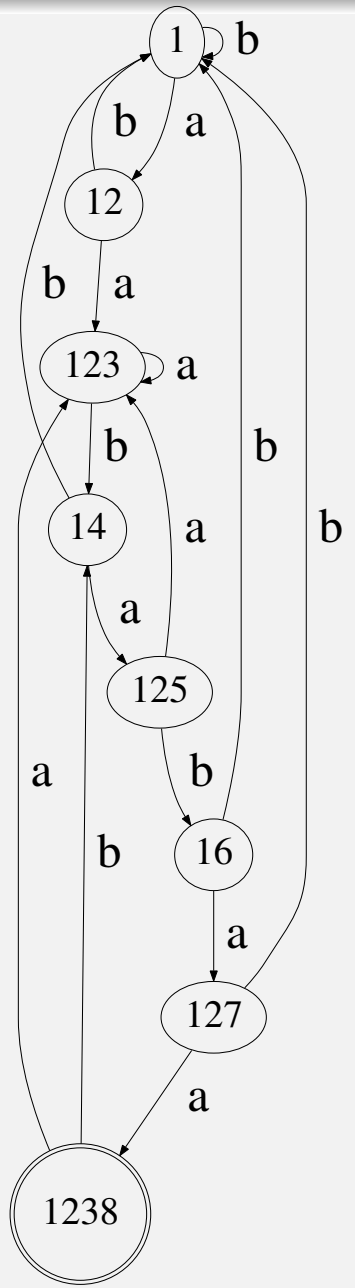
Main Idea

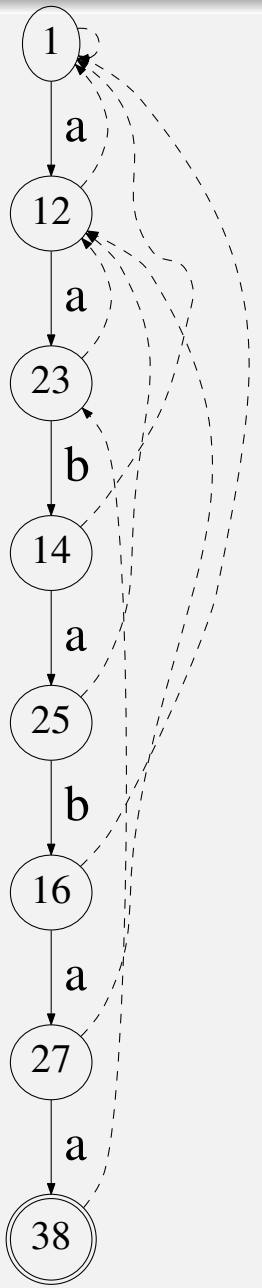
- Remember only the largest $j < i$ in $pset(i)$
  - You can look at $pset(j)$ for the next smaller element
  - Add *failure* links from state i to j for this purpose
- Two positions per $pset \Longrightarrow O(n)$ construction time



Exact Search: KMP Automaton

- Only two positions per state: $\{j, i\}$
- Two trans per state: forward and fail
- If the symbol at both positions is the same, then the next state has the pset $\{j+1, i+1\}$
- Otherwise, the match at j cannot advance on the symbol at i. So, we use the fail link to identify the next shorter prefix that can advance:
  - Follow fail link to state u with pset $\{k, j\}$ and see if that match can advance
  - Otherwise, follow the fail link from u and so on.
- Failure link chase is amortized $O(1)$ time, while other steps are $O(1)$ time.





KMP Algorithm

$BuildAuto(P[1..m])$

$j = 0$
for $i = 1$ **to** m **do**
  $fail[i] = j$
  while $j > 0$ **and** $P[i] \neq P[j]$ **do**
    $j = fail[j]$
  $j++$

$KMP(P[1..m], S[1..n])$

$j = 0$; $\underline{BuildAuto(P)}$
for $i = 1$ **to** $\underline{n}$ **do**
  while $j > 0$ **and** $\underline{T[i]} \neq P[j]$ **do**
    $j = fail[j]$
  $j++$
  if $j > m$ **then return** $i - m + 1$

- Simple, avoids explicit representation of states/transitions.
- Each state has two transitions: normal and failure.
  - Normal transition at state i is on $P[i]$
  - Fail links are stored in an array *fail*
- *BuildAuto* is like matching pattern with itself!
- Algorithm is unbelievably short and simple!

Multi-pattern Exact Search

- Can we extend KMP to support multiple patterns?
- Yes we can! It is called Aho-Corasick (AC) automaton
  - Note that AC algorithm was published before KMP!
  - Today, many systems use AC (e.g., grep, snort), but not KMP.
- KMP looks like a linear automaton plus failure links.
  - Aho-Corasick looks like a trie extended with failure links.
  - Failure links may go to a non-ancestor state
- Failure link computations are similar
- McNaughton-Yamada and the derivatives algorithms build an automaton similar to AC, just as they did for KMP.
  - One can understand Aho-Corasick as a specialization of these algorithms, as we did in the case of KMP,
  - Or, as a generalization of KMP

Aho-Corasick Automaton

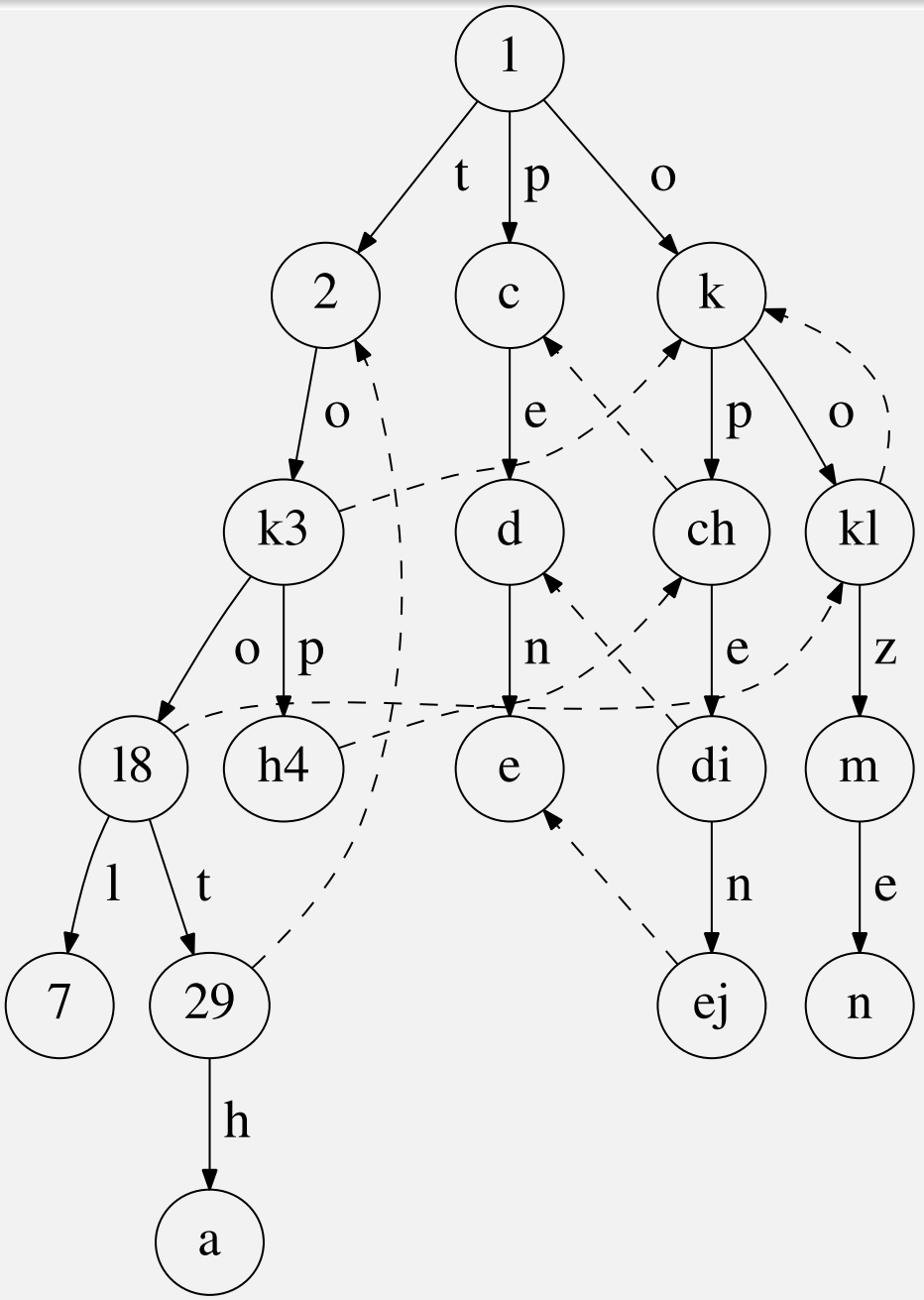
- As with KMP, we can think of AC as a specialization of MY.
  - Retain just the largest two numbers i and j in the pset.
  - Use the value of j as target for failure link, and to find j' in the successor state's pset $\{j', i+1\}$
- But there is an extra wrinkle:
  - With KMP, there is one pattern; we keep two positions from it.
  - With AC, we have multiple patterns, so a state's pset will contain positions from multiple patterns.
    - If two patterns share a prefix, the automaton state reached by this prefix will contain the next positions from both patterns.
  - We will simply retain one one of these positions, say, from the higher numbered pattern.
  - To avoid clutter in our example, we omit numbering of positions that will be dropped this way.

Aho-Corasick Example

Consider RE

$$(\Sigma^0)^*(t^1o^2p^3\$^4|too^5l^6\$^7|toot^8h^9\$^a$$
$$|p^be^cn^d\$^e|o^fp^ge^hn^i\$^j|oo^kz^le^m\$^n)$$

- To reduce clutter, positions that occur with previously numbered positions are *not* explicitly numbered, e.g., *o*'s in *tooth* (occurs with the *o*'s in *tool*)
- Figure omits failure links that go to start state.



Alternative Approaches for Exact Search

- DFA approach had significant preprocessing ("compiling") costs, but optimized runtime — exactly m comparisons.
- KMP reduces compile-time[1] by shifting more work (up to $2m$ comparisons) to runtime.
  - DFA states contain information about all matching prefixes, but KMP states retain just the two longest ones.
  - Other prefixes are essentially being computed at runtime by folowing fail links.
- Can we remember even less in automaton states?
  - Can we leave all matching prefixes to be computed at runtime?

[1] while also simplifying automaton structure

Approximate Search

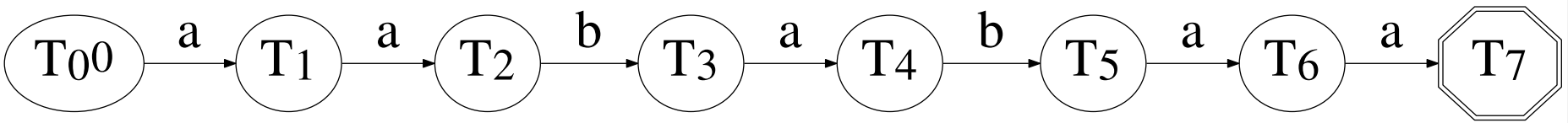
Approach 1: Use edit-distance algorithm

- Expensive
- Does not allow for multiple patterns
  - Unless you try the patterns one-by-one

Approach 2: Levenshtein Automaton

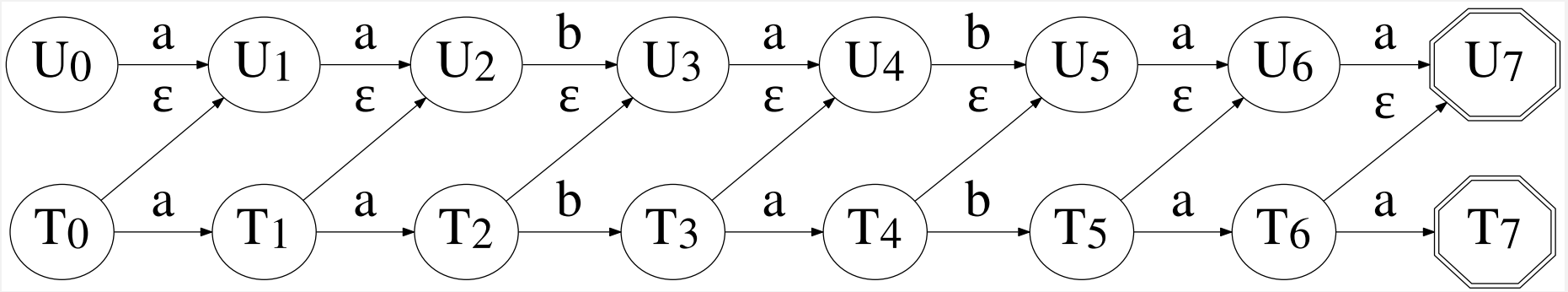
- Can be much faster, especially when p is small.
- Supports multiple patterns
- Enables applications such as spell-correction

Levenshtein Automaton



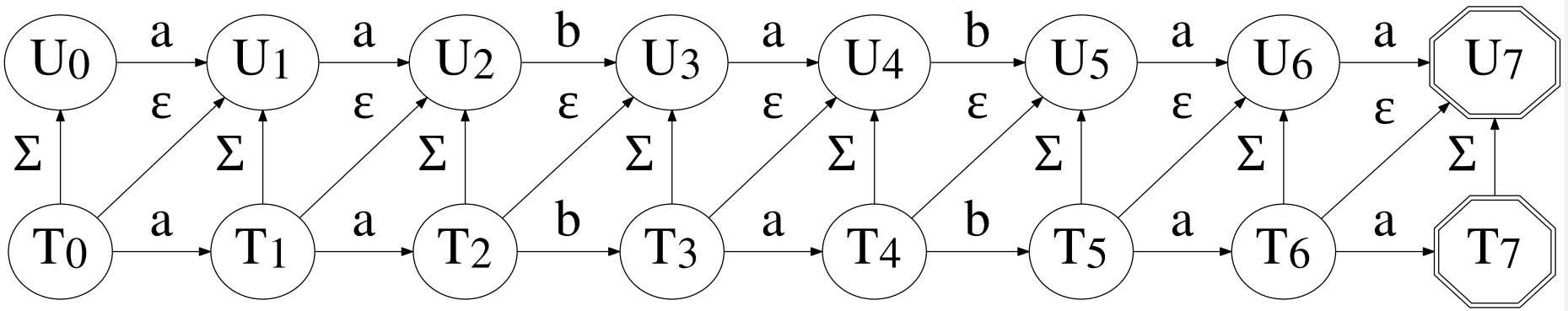
- No errors permitted.

Levenshtein Automaton



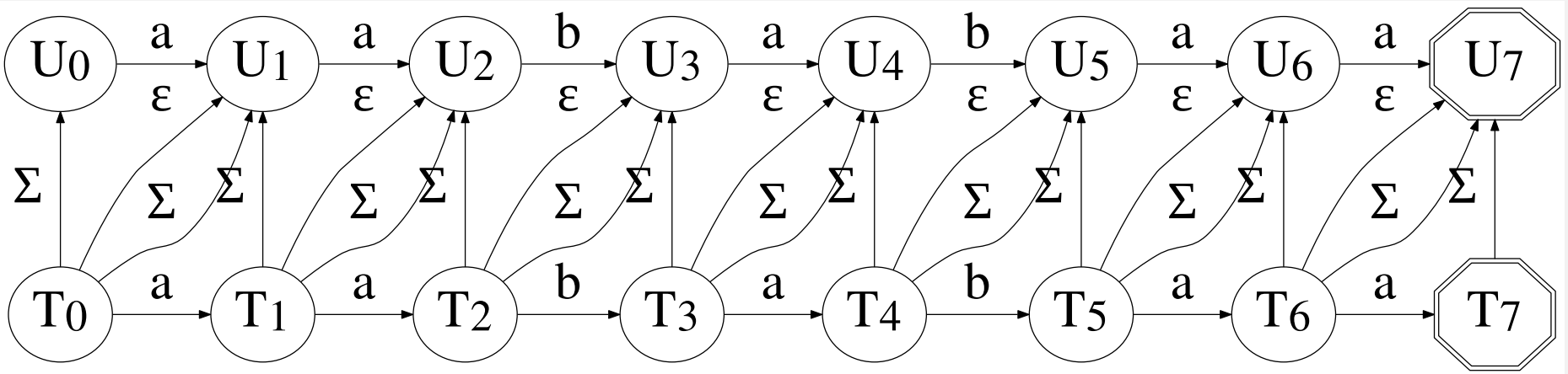
- Up to one missing character (deletion).

Levenshtein Automaton



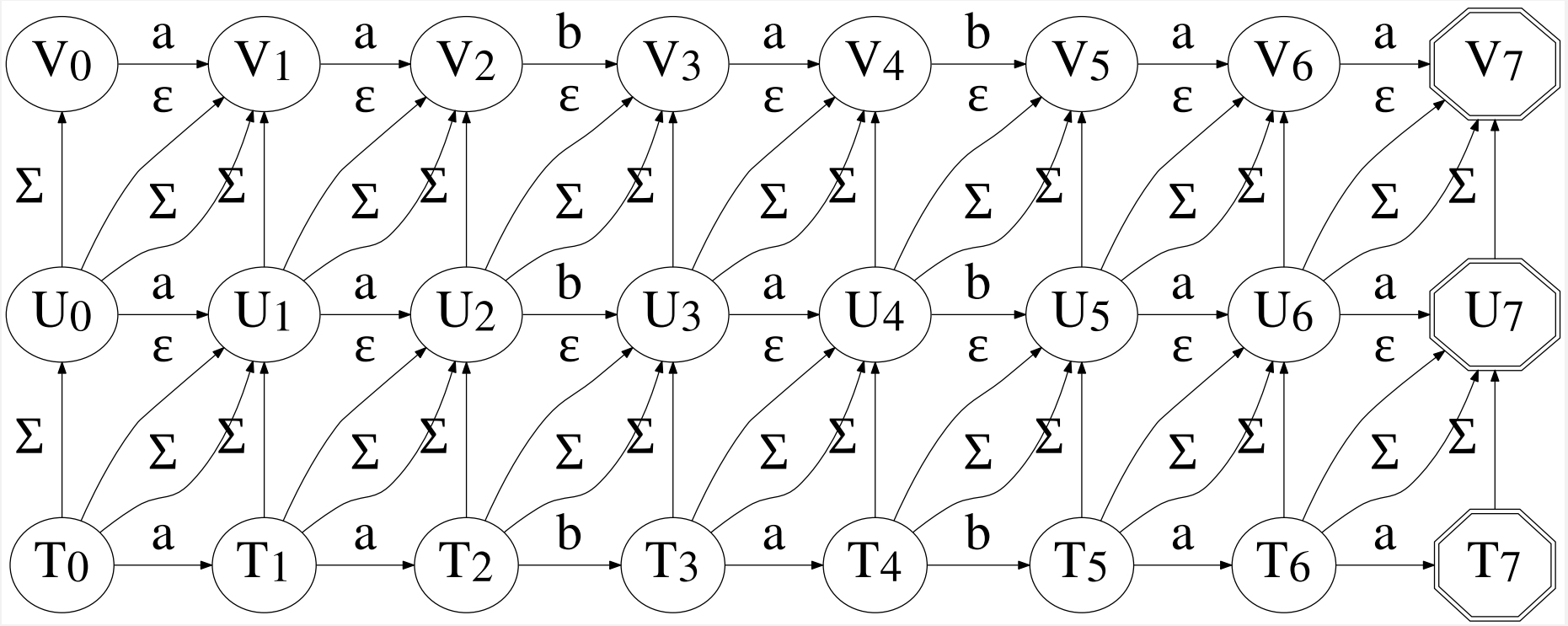
- Up to one deletion and one insertion.

Levenshtein Automaton



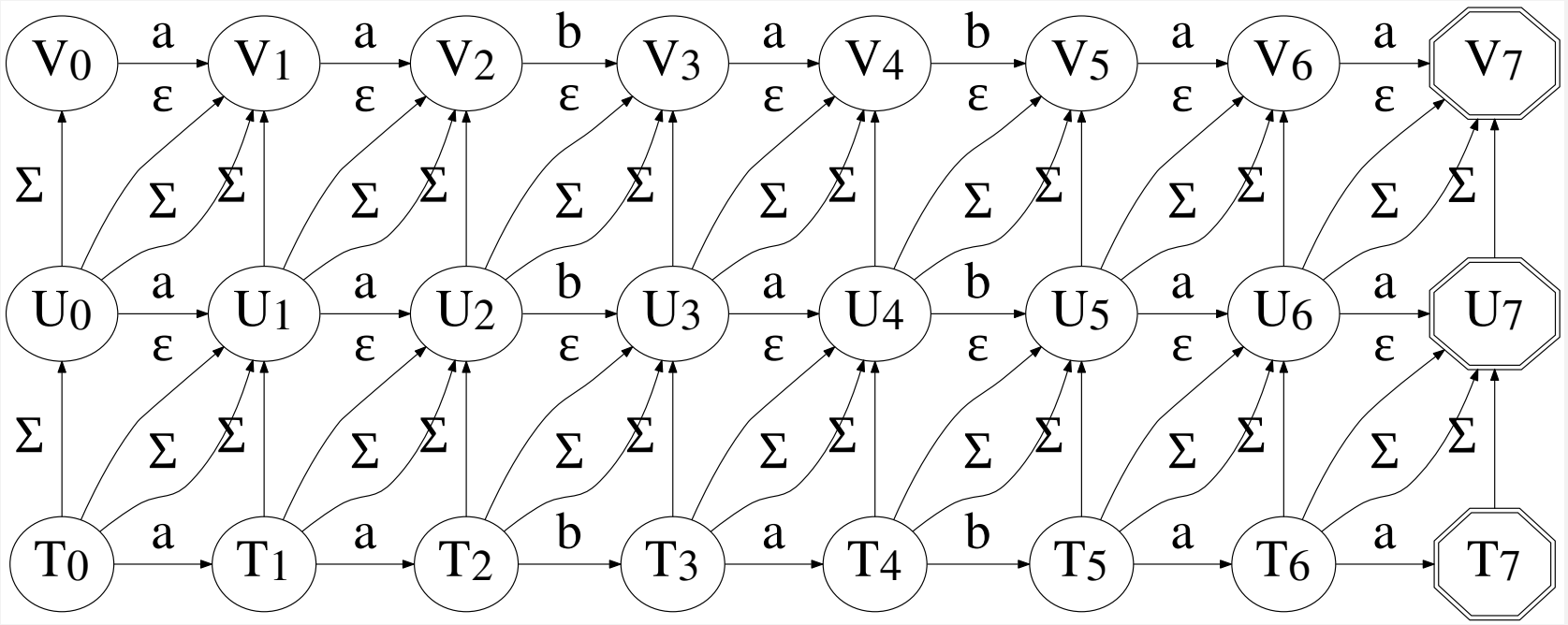
- Up to one deletion, or insertion, or substitution

Levenshtein Automaton



- Up to a total of two deletions, insertions, or substitution

Levenshtein Automaton

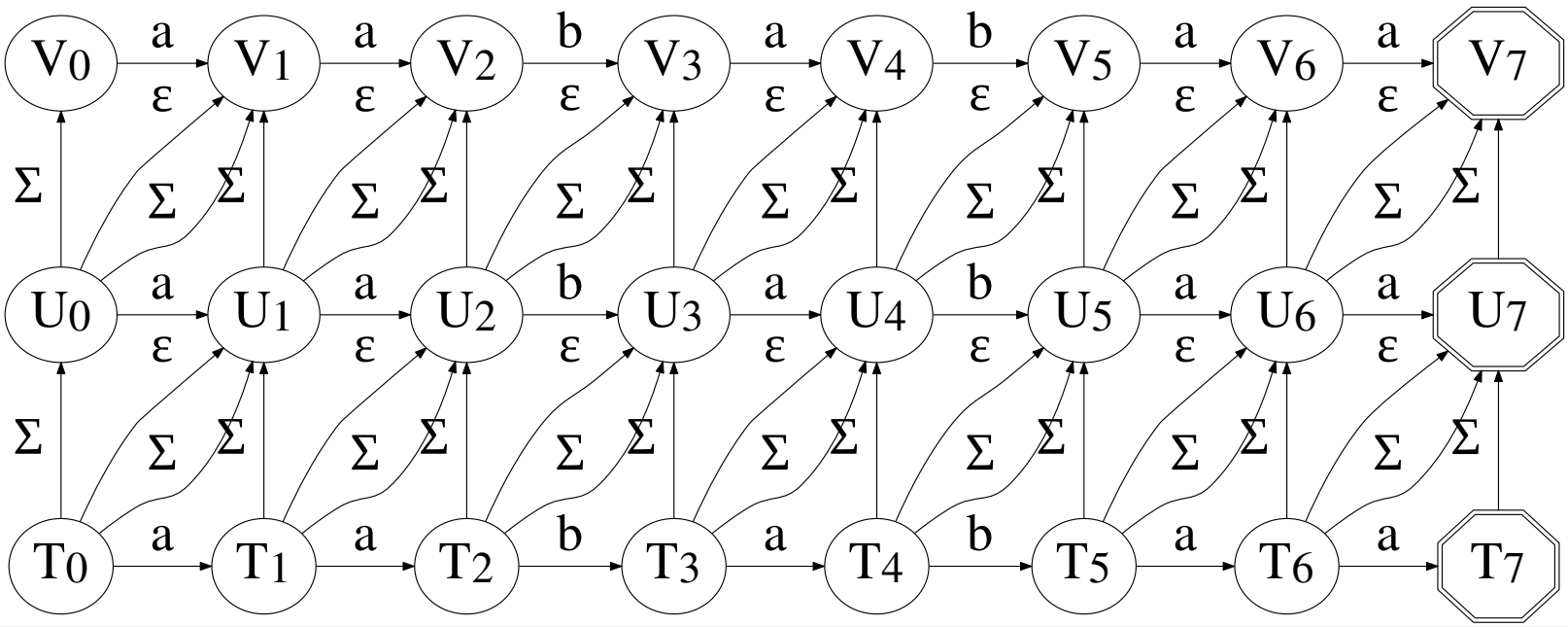


Compare with:

- Structure of cost matrix for edit-distance problem
- Finding least-cost paths from T_0 to T_7, U_7 or V_7

Illustrates the relationship between shortest path and edit-distance problem

Matching Using Levenshtein Automaton



- *Convert to DFA (subset construction)*
  - Potentially $O(n^k)$ states, where k is the max edit distance permitted
- *Adapt Shift-and algorithm*
  - We already know how to maintain $T[0..n]$
  - Need to extend to compute U from T, V from U and so on.

Levenshtein automaton and spell-correction

- When a word w is misspelled, we want to find the closest matching word in the dictionary
  - Or, list all matches within an edit distance of l
- *Approach:*
  - Build Levenshtein automaton for w with $l+1$ "layers"
  - Run the dictionary trie through the automaton
  - List all matches
- Alternatively, a DFA for the Levenshtein automaton could be built, and the trie run through this DFA.
  - The DFA could be directly constructed as well, without going through an NFA and powerset construction.

Rolling Hashes

RK and CWRK are examples of rolling hashes

- Hash computed on text within a sliding window
- *Key point:* Incremental computation of hash as the window slides.

Polynomial-based hashes are easy to compute incrementally:

$$t_{i+1} = (t_i - x^{n-1}T[i]) \cdot x + T[i+n]$$

Complexity:

- x^{n-1} is fixed once the window size is chosen
- Takes just two multiplications, one modulo per symbol
- $O(m+n)$ multiplication/modulo operations in total

Other Rolling Hashes

In some contexts, multiplication/modulo may be too expensive.

Alternatives:

- Use shifts, cyclic shifts, substitution maps and xor operations, avoiding multiplications altogether
  - Need considerable research to find good fingerprinting functions.
- Example: Adler32 — used in zlib (used everywhere) and rsync.

$$A_l = 1 + \sum_{k=0}^{l-1} t_{i+k} \mod 65521$$

$$B = \sum_{k=1}^{n} A_k = n + \sum_{k=0}^{n-1} (n-k) t_{i+k} \mod 65521$$

$$H = (B \ll 16) + A$$

Rolling Hash and Common Substring Problem

- To find a common substring of length l or more
  - Compute rolling hashes of P and T with window size l
    - Takes $O(n+m)$ time.
  - $O(nm)$ comparisons, so expected number of collisions increases.
    - Unless collision probability is $O(1/nm)$, expected runtime can be nonlinear
- Can find longest common substring (LCS) using a binary-search like process, with a total complexity of $O((n+m)\log(n+m))$

zlib/gzip, rsync, binary diff, etc.

rsync: Synchronizes directories across network

- Need to minimize data transferred
  - A diff requires entire files to be copied to client side first!
- Uses timestamps (or whole-file checksums) to detect unchanged files
- For modified files, uses Adler-32 to identify modified regions
  - Find common substrings of certain length, say, 128-bytes
- Relies on stronger MD-5 hash to verify unmodified regions

gzip: Uses rolling hash (Adler-32) to identify text that repeated from previous 32KB window

- Repeating text can be replaced with a "pointer:" (offset, length).

Binary diff: Many programs such as xdelta and svn need to perform diffs on binaries; they too rely on rolling hashes.

- `diff` depends critically on line breaks, so does poorly on binaries

Suffix Trees [Weiner 1973]

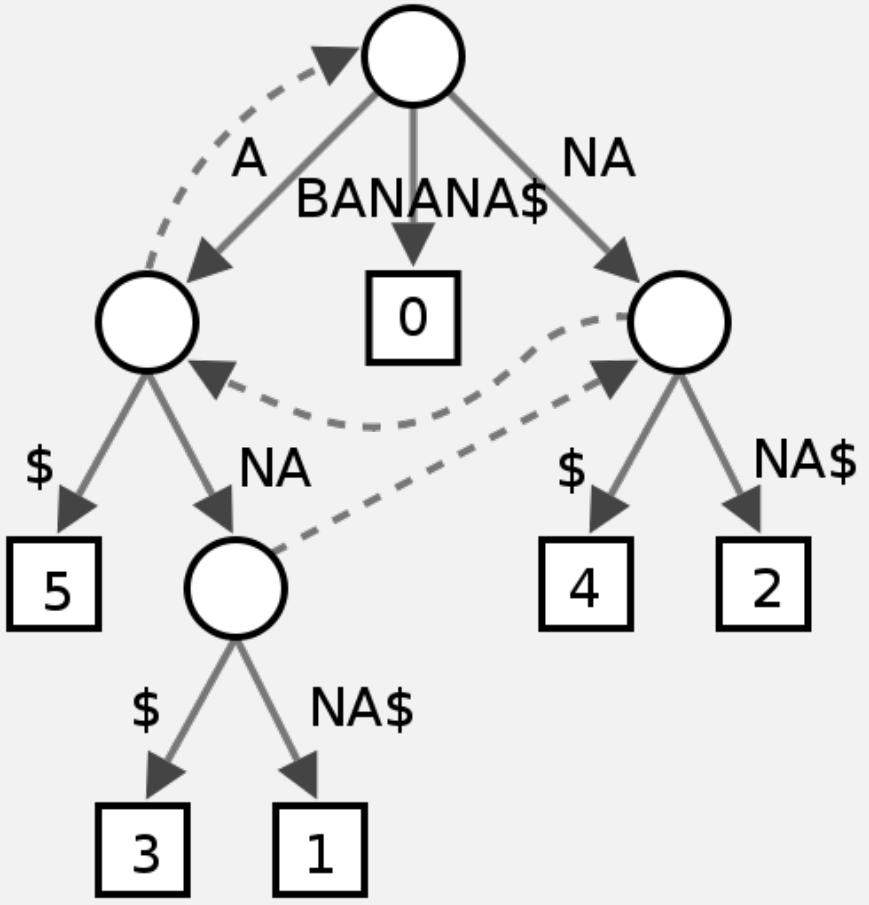
- A versatile data structure with wide applications in string search and computational biology
- *"Compressed" trie of all suffixes of a string appended with "$"*
  - Linear chains in the trie are compressed
    - Edges can now be substrings.
    - Each state has at least two children.
  - Leaves identify starting position of that suffix.
- *Key point: Can be constructed in linear time!*
- *Supports* sublinear *exact match queries, and* linear *LCS queries*
  - With linear-time preprocessing on the text (to build suffix tree),
  - yields better runtime than techniques discussed so far.
- *Applicable to single as well as multiple patterns or texts!*

Suffix Tree Example

Key Property Behind Suffix Trees

Substrings are prefixes of suffixes

- Failure links used only during construction
- Uses end-marker "$"
- Leaves identify starting position of suffix
- Typically, it is the text we preprocess, not the pattern.



Images from Wikipedia commons

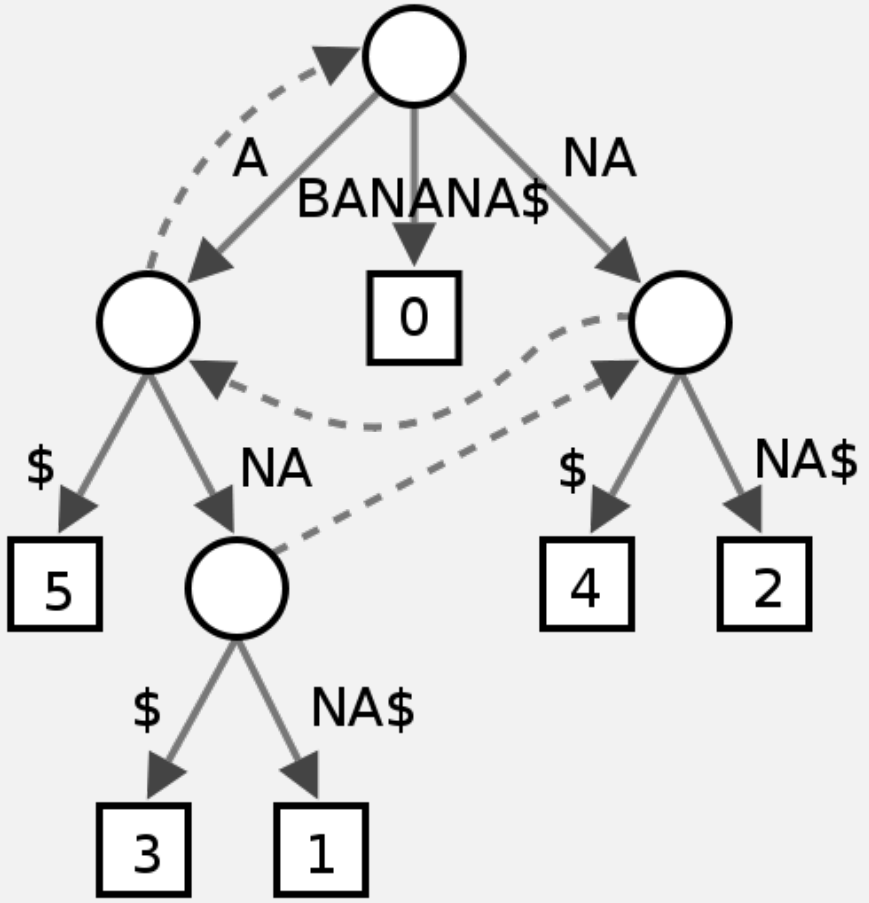
Finding Substrings and Suffixes

Is p a substring of t?

Example: Is *anan* a substring of *banana*?

Solution:

- Follow path labeled p from root of suffix tree for t.
- If you fail along the way, then "no," else "yes"
- p is a *suffix* if you reach a leaf at the end of p
- $O(|p|)$ time, independent of $|t|$ — great for large t

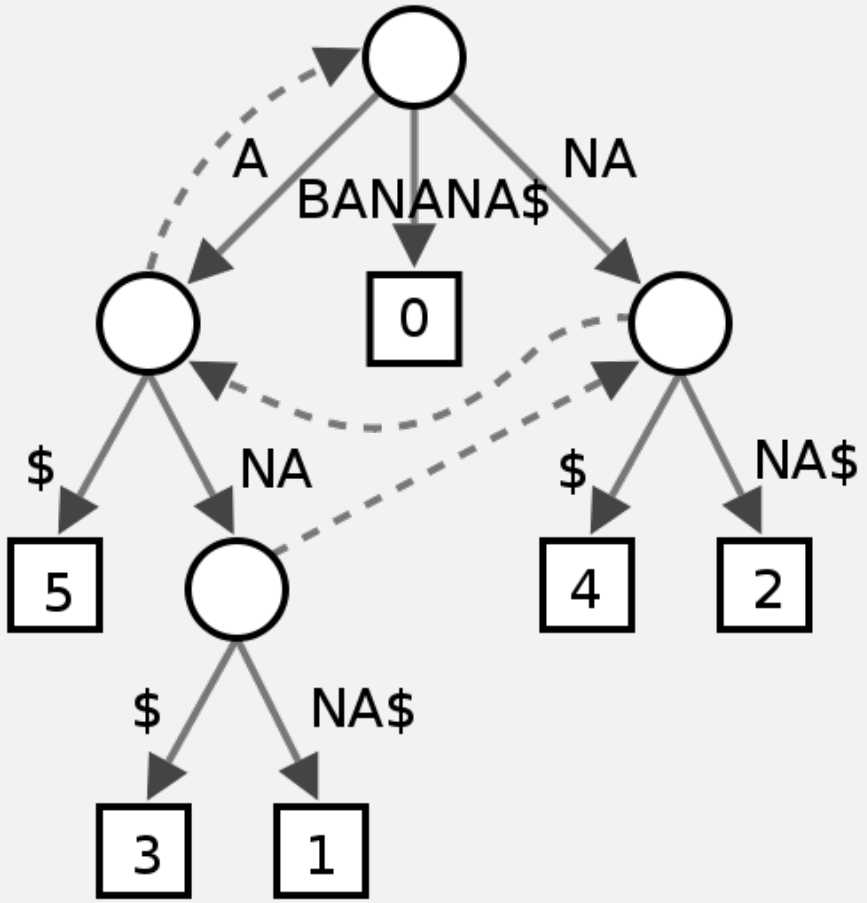


Counting # of Occurrences of p

How many times does "an" occur in t?

Solution:

- Follow path labeled p from root of suffix tree for t.
- Count the number of leaves below.
- $O(|p|)$ time if additional information (# of leaves below) maintained at internal nodes.

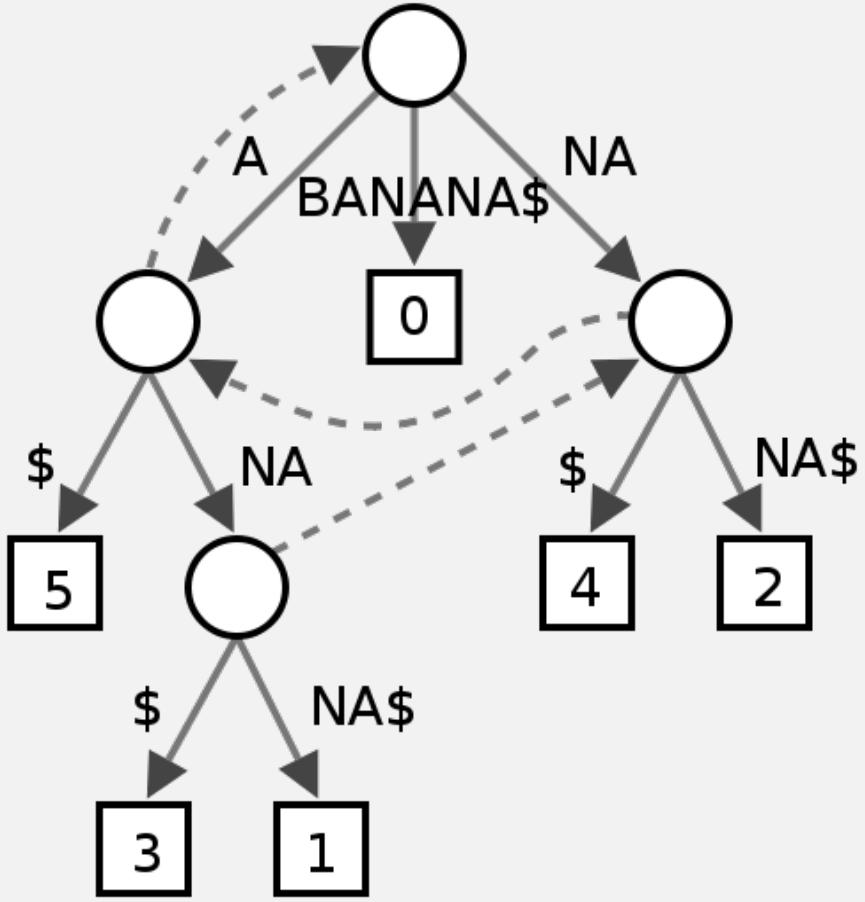


Self-LCS (Or, Longest Common Repeat)

What is the longest substring that repeats in t?

Solution:

- Find the deepest non-leaf node with two or more children!
- In our example, it is *ana*.

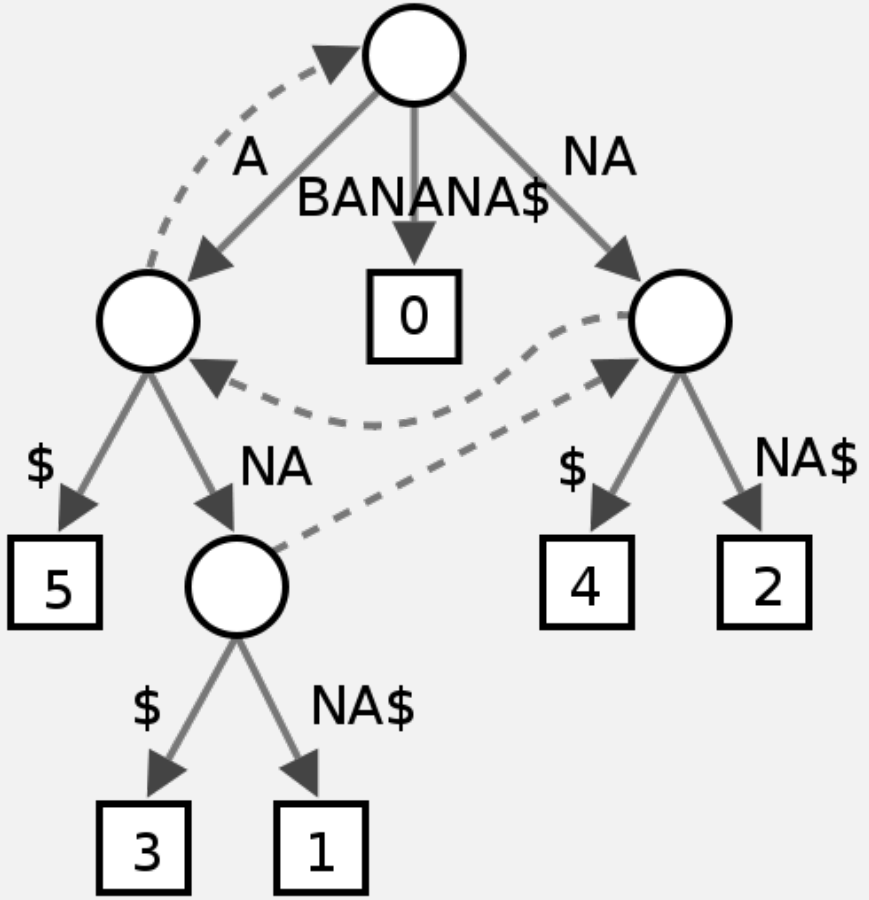


LC extension of i and j

Longest Common Extension

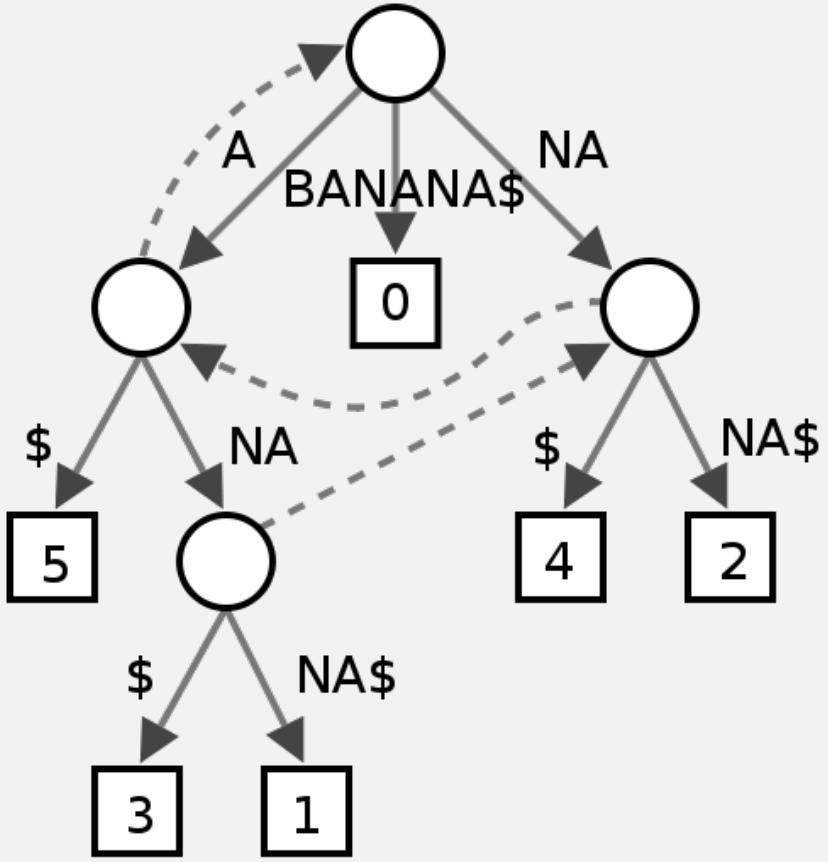
Longest common prefix of suffixes starting at i and j

- Locate leaves labeled i and j.
- Find their least common ancestor (LCA)
- The string spelled out by the path from root to this LCA is what we want.



LCS with another string *p*

- We can use the same procedure as LCR, *if suffixes of p were also included in the suffix tree*
- Leads to the notion of *generalized suffix tree*



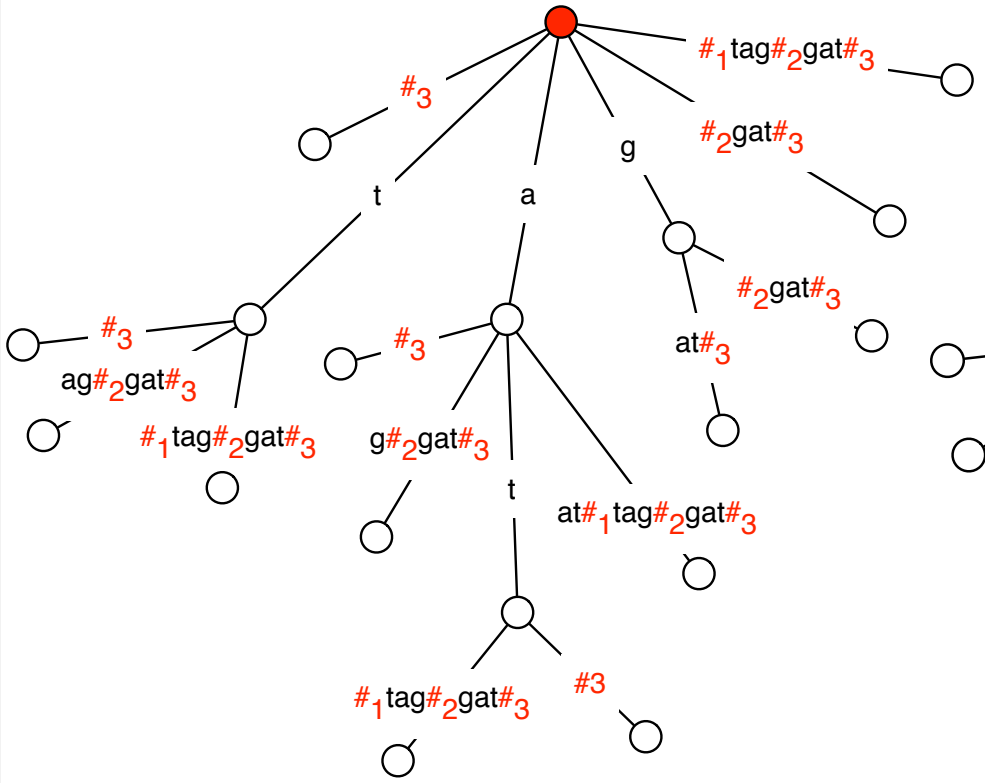
Generalized Suffix Trees

Suffix trees for multiple strings $p_1, \ldots, p_n$

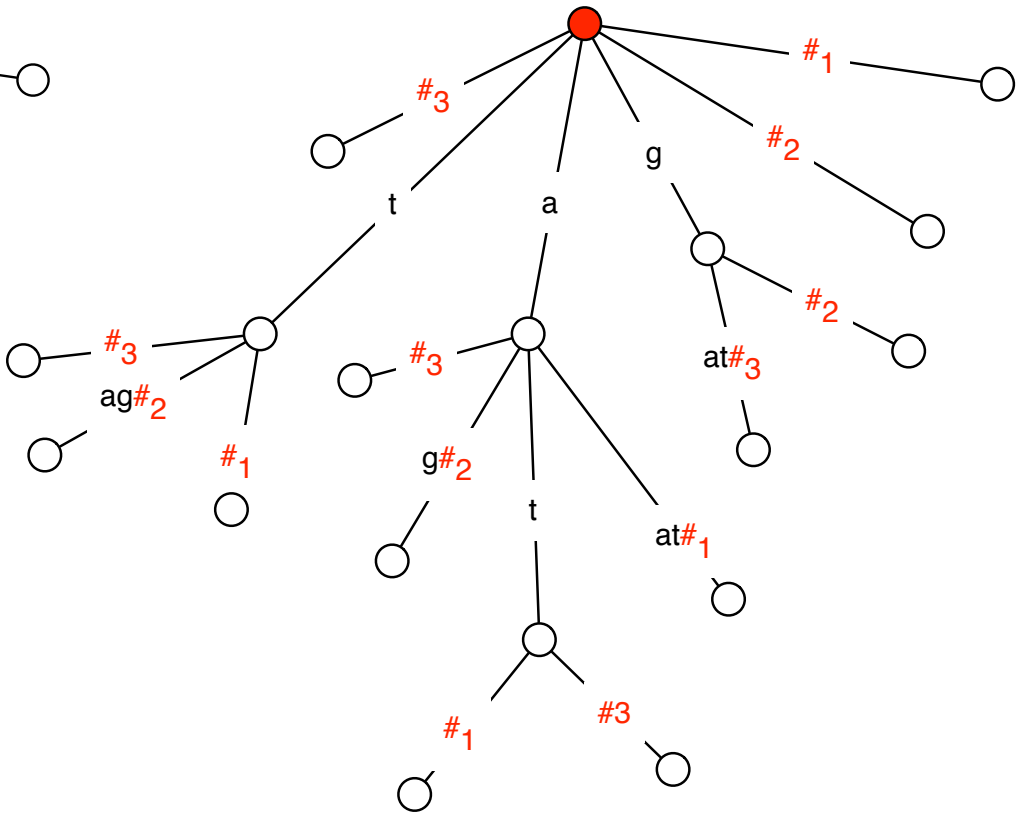
Example. att, tag, gat

Simple solution:

(1) build suffix tree for string aat#$_1$tag#$_2$gat#$_3$



(2) For every leaf node, remove any text after the first # symbol.



Images from UMD CMSC 423 Slides

Generalized Suffix Tree: Applications

LCS of p and t: Build GST for s and t, find deepest node that has descendants corresponding to s and t

LCS of $p_1, \ldots, p_k$: Build GST for p_1 to p_k, find deepest node that has descendants from all of $p_1, \ldots, p_n$

Find strings in database containing q:

- Build a suffix tree of all strings in the database
- follow path that spells q
- q occurs in every p_i that appears below this node.

Suffix Arrays [Manber and Myers 1989]

- *Drawbacks of suffix trees:*
  - Multiple pointers per internal node: significant storage costs
  - Pointer-chasing is not cache-friendly
- Suffix arrays address these drawbacks.
  - Requires same asymptotic storage ($O(n)$) but constant factors a lot smaller — 4x or so.
  - Instead of navigating down a path in the tree, relies on binary search
    - Increases asymptotic cost by $O(\log n)$, but can be faster in practice due to better cache performance etc.

CSE 548: (*Design and*) Analysis of Algorithms

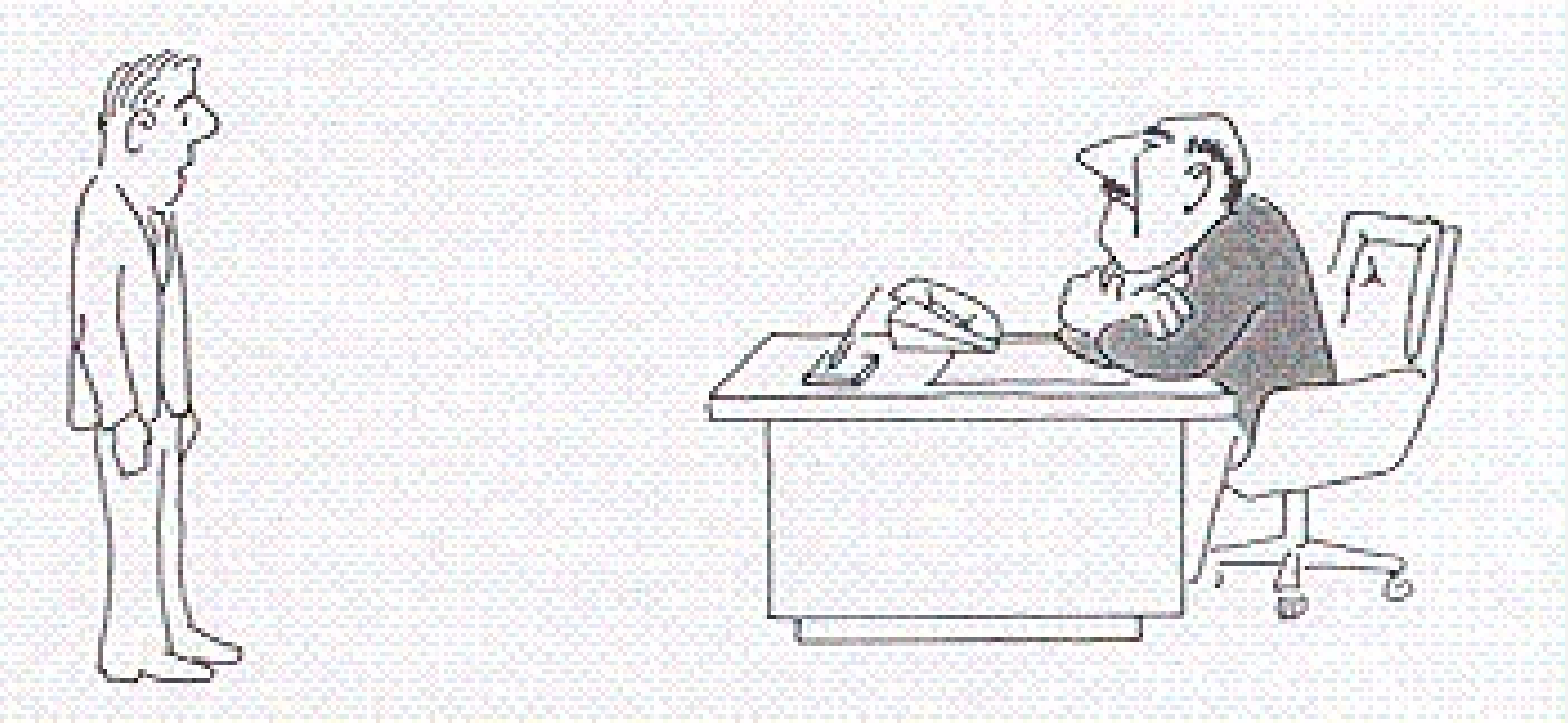
NP and Complexity Classes

R. Sekar

Search and Optimization Problems

- Many problems of our interest are search problems with exponentially (or even infinitely) many solutions
  - Shortest of the paths between two vertices
  - Spanning tree with minimal cost
  - Combination of variable values that minimize an objective
- We should be surprised we find efficient (i.e., polynomial-time) solutions to these problems
  - It seems like these should be the exceptions rather than the norm!
- What do we do when we hit upon other search problems?

Hard Problems: Where you find yourself ...



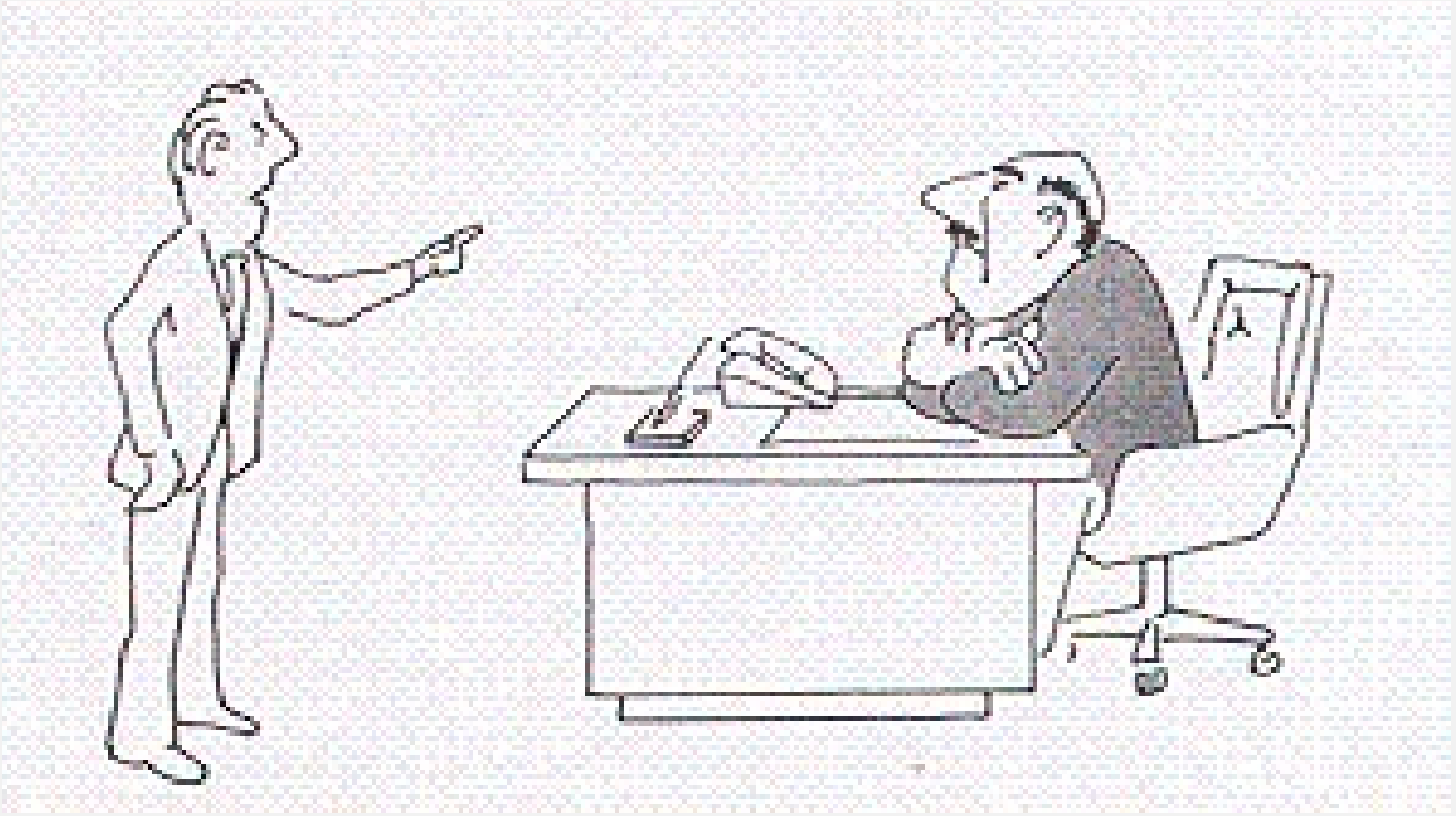
I can't find an efficient algorithm, I guess I'm just too dumb.

Images from "Computers and Intractability" by Garey and Johnson

Search and Optimization Problems

- What do we do when we hit upon hard search problems?
  - Can we prove they can't be solved efficiently?

Hard Problems: Where you would like to be ...



I can't find an efficient algorithm, because no such algorithm is possible.

Images from "Computers and Intractability" by Garey and Johnson

Search and Optimization Problems

- Unfortunately, it is very hard to prove that efficient algorithms are impossible
- Second best alternative:
  - Show that the problem is as hard as many other problems that have been worked on by a host of brilliant scientists over a very long time
- Much of complexity theory is concerned with categorizing hard problems into such *equivalence classes*

P, *NP*, *Co-NP*, *NP*-hard and *NP*-complete

Nondeterminism and Search Problems

- Nondeterminism is an oft-used abstraction in language theory
  - Non-deterministic FSA
  - Non-deterministic PDA
- So, why not non-deterministic Turing machines?
  - Acceptance criteria is analogous to NFA and NPDA
    - if there is a sequence of transitions to an accepting state, an NDTM will take that path.
- What does nondeterminism, a theoretical construct, mean in practice?
  - You can think of it as a boundless potential to search for and identify the correct path that leads to a solution
  - So, it does not change the class of problems that can be solved, just the time/space needed to solve.

Class *NP*: Non-deterministic Polynomial Time

How they operate:

- Guess a solution
- verify correctness in polynomial time

Polynomial time verifiability is the key property of *NP*.

- This is how you build a path from *P* to *NP*.
- Ideal formulation for search problems, where correct solutions are hard to find but easy to recognize.

Example: Boolean formula satisfiability (*SAT*)

- Given a boolean formula in CNF, find an assignment of {`true`, `false`} to variables that makes it true.
  - Why not DNF?

What are the bounds of *NP*?

- *Only* **Decision problems:**
  - Problems with an "yes" or "no" answer
  - Optimization problems are generally not in *NP*
    - But we can often find optimal solutions using "binary search"
- *"No" answers are usually not verifiable in P-time*
  - So, complement of *NP* problems are often not *NP*.
  - *UNSAT* — show that a CNF formula is false for all truth assignments[1]
- **Key point:** You cannot negate nondeterministic automata.
  - So, we are unable to convert an NDTM for *SAT* to solve *UNSAT* in *NP*-time.

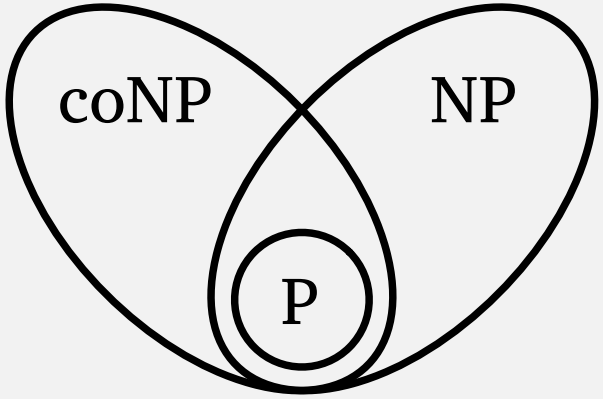
[1] Whether $UNSAT \in NP$ is unknown!

What are the bounds of *NP*?

- *Existentially quantified vs Universally quantified formulas*
  - *NP* is good for $\exists \overline{x}\ P(\overline{x})$: guess a value for $\overline{x}$ and check if $P(\overline{x})$ holds.
  - *NP* is not good for $\forall \overline{x}\ P(\overline{x})$:
    - Guessing does not seem to help if you need to check all values of $\overline{x}$.
- Negation of existential formula yields a universal formula.
  - No surprise that complement of *NP* problems are typically not in *NP*.
  - *UNSAT*: $\forall \overline{x} \neg P(\overline{x})$ where P is in CNF
  - *VALID*: $\forall \overline{x} P(\overline{x})$, where P is in DNF
- *NP seems to be a good way to separate hard problems from even harder ones!*

Co-NP: Problems whose complement is in NP

- Decision problems that have a polynomially checkable proof when the answer is "no"



What we *think* the world looks like.

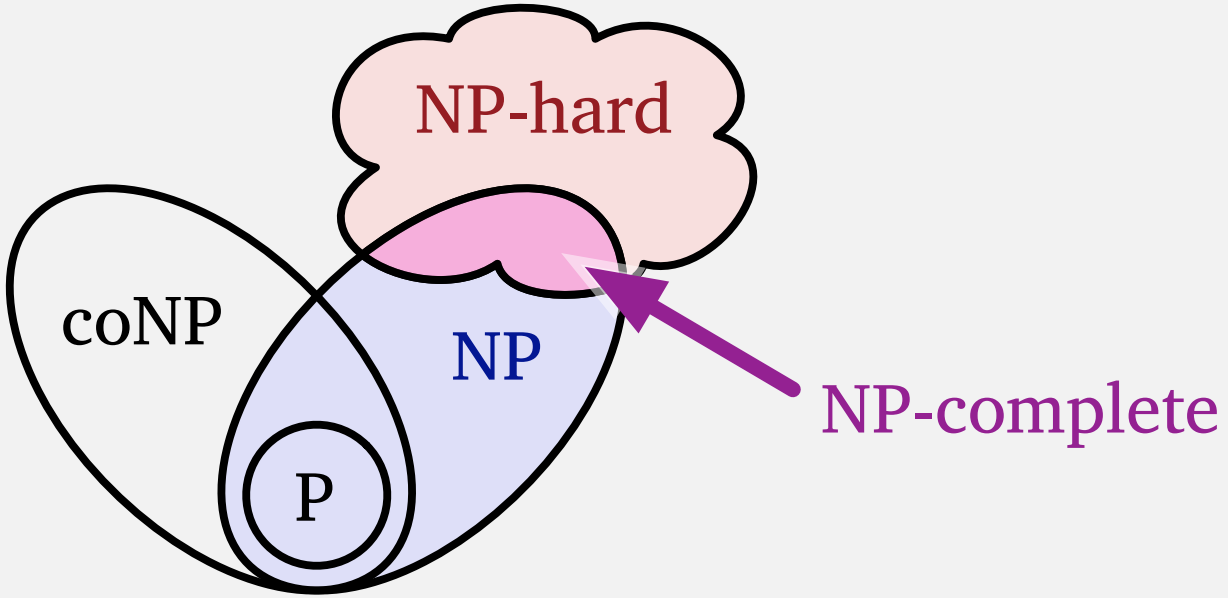
- Biggest open problem: Is $P = NP$?
  - Will also imply $co\text{-}NP = P$

The class *Co-NP* ∩ *NP*

- Often, problems that are in *NP* ∩ *co-NP* are in *P*
- It requires considerable insight and/or structure in the problem to show that something is both *NP* and *co-NP*
  - This can often be turned into a *P*-time algorithm
- Examples
  - Linear programming [1979]
    - Obviously in *NP*. To see why it is in *co-NP*, we can derive a lower bound by multiplying the constraints by a suitable (guessed) number and adding.
  - Primality testing [2002]
    - Obviously in co−NP; See "primality certificate" for proof it is *NP*
  - Integer factorization?

NP-hard and NP-complete

- A problem Π is *NP*-hard if the availability of a polynomial solution to Π will allow *NP*-problems to be solved in polynomial time.
  - Π is *NP*-hard $\Leftrightarrow$ if Π can be solved in *P*-time, $P = NP$
- *NP*-complete = *NP*-hard $\cap$ *NP*



More of what we *think* the world looks like.

Polynomial-time Reducibility

- Show that a problem A could be transformed into problem B in polynomial time
  - Called a polynomial-time reduction of A to B
  - The crux of proofs involving *NP*-completeness
- *Implication:* if B can be solved in P-time, we can solve A in P-time
- *An NP-complete problem* is one to which any problem in *NP* can be reduced to.
- **Never forget the direction:** To prove a problem Π is *NP*-complete, need to show how all other *NP* problems can be solved using Π, not vice-versa!

Wait! How can I reduce *every NP* to my problem?

- If a particular *NP*-problem A is given to you, then you can think of a way to reduce it to your problem B
- But how do you go about proving that *every NP* problem X can be reduced to B
  - You don't even know X — indeed, the class *NP* is infinite!
- *If you already knew an NP-complete problem, your task is easy!*
  - Simply reduce this *NP*-complete problem to B, and by transitivity, you have a reduction of every $X \in NP$ to B
- So, who will bell the cat?
  - Stephen Cook [1970] and Leonid Levin [1973] managed to do this!
  - Cook was denied reappointment/tenure in 1970 at Berkeley, but won

The first *NP*-complete problem: *SAT*

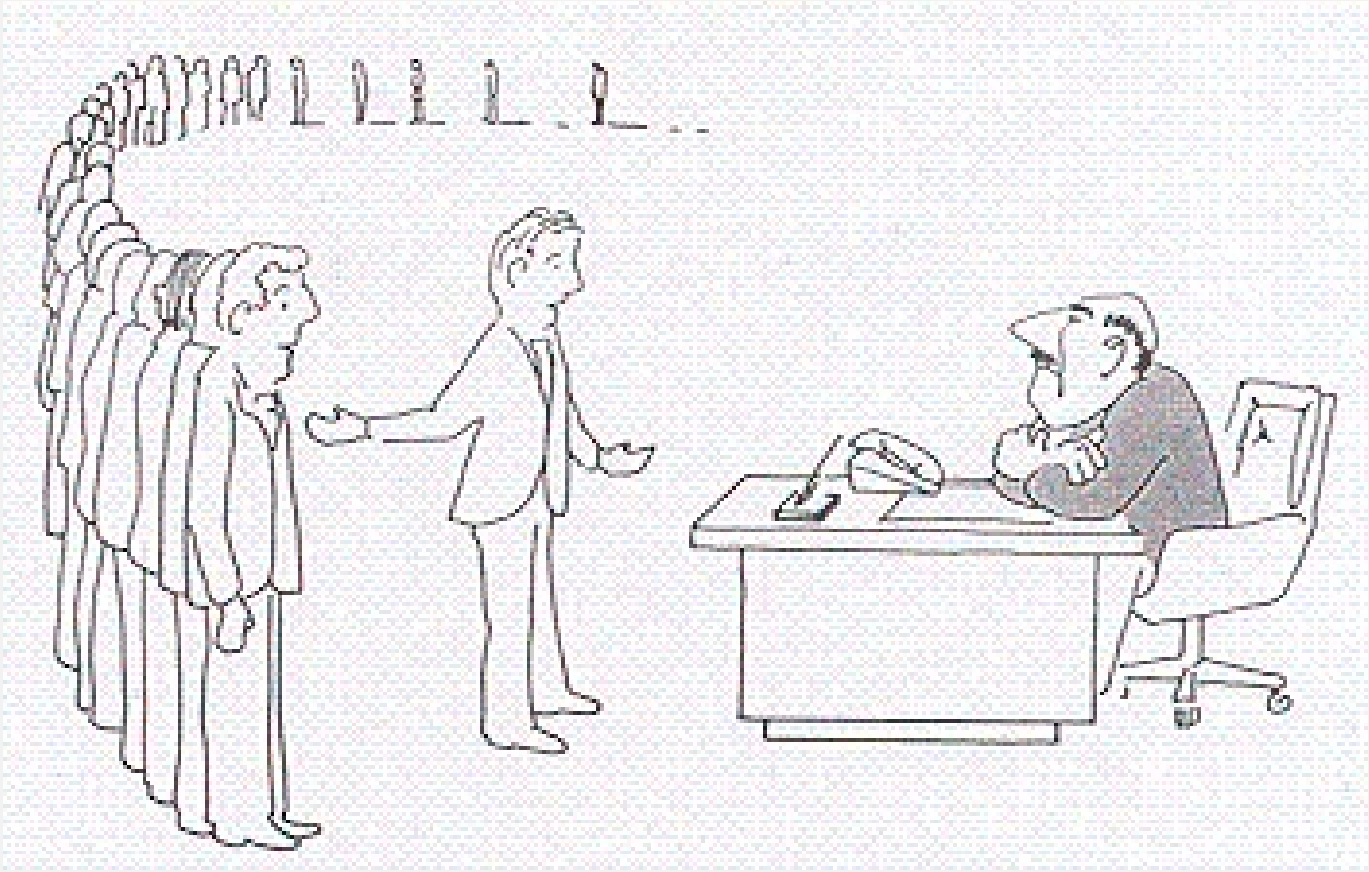
How do you show reducibility of arbitrary *NP*-problems to *SAT*? You start from the definition, of course!

- The class *NP* is defined in terms of an NDTM
  - X is in *NP* if there is an NDTM T_X that solves X in polynomial time
- Use this NDTM as the basis of proof.

Specifically, show that acceptance by an NDTM can be encoded in terms of a boolean formula

- Model T_X tape contents, tape heads, and finite state at each step as a vector of boolean variables
  - Need $(p(n))^2$ variables, where $p(n)$ is the (polynomial) runtime of T_X
- Model each transition as a boolean formula

Thanks to Cook-Levin, you can say ...

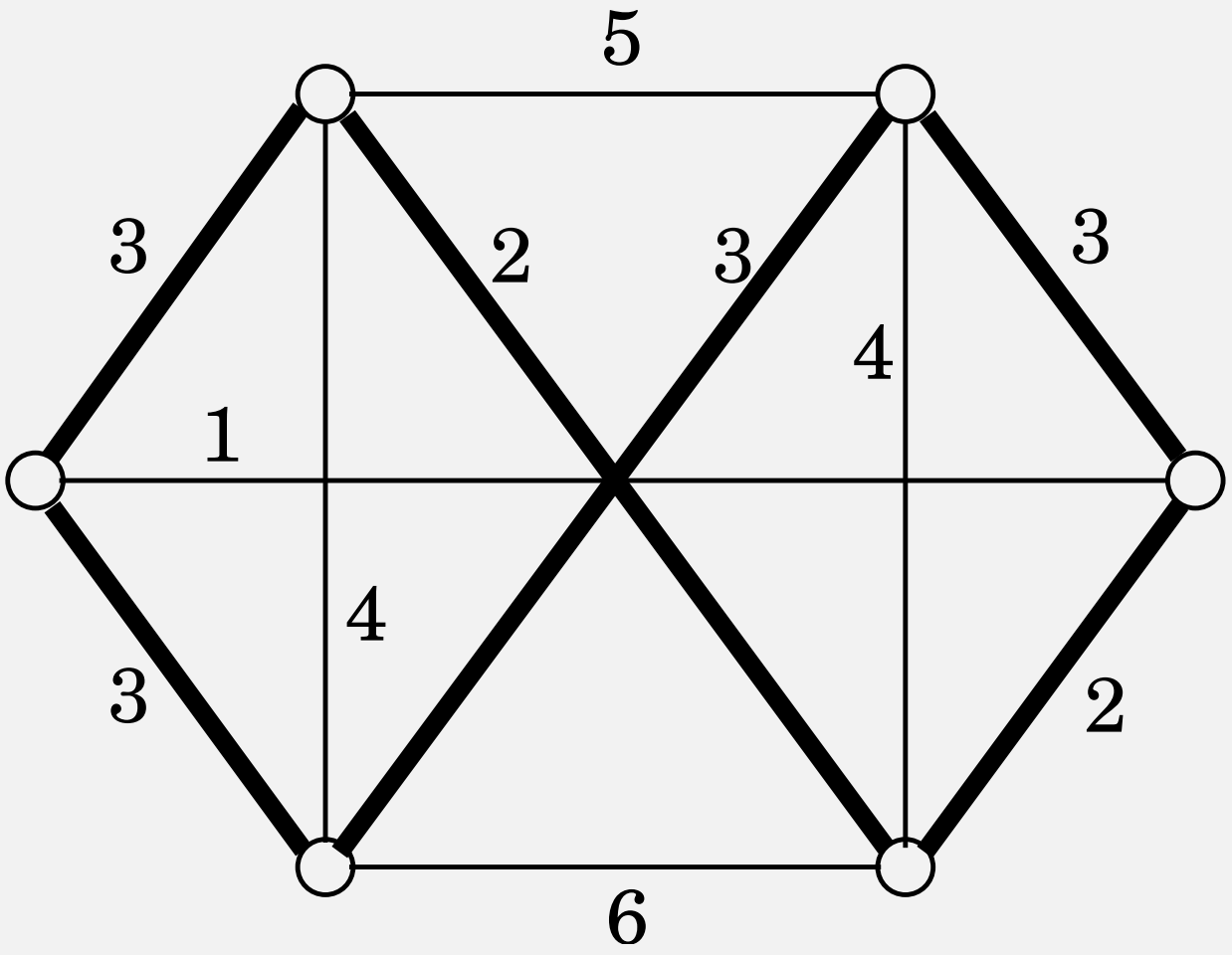


I can't find an efficient algorithm, but neither can all these famous people.

Thanks to *NP*-completeness results, you can say this even if you have been working on an obscure problem that no one ever looked at!

Some Hard Decision Problems

Traveling Salesman Problem



Given n vertices and $n(n-1)/2$ distances between them, is there a *tour* (i.e., cycle) of length b or less that passes through all vertices?

Hamiltonian Cycle

- Simpler than TSP
  - Is there a cycle that passes through every vertex in the graph?
- Earliest reference, posed in the context of chess boards and knights ("Rudrata cycle")
- *Longest path* is another version of the same problem
  - When posed as a decision problem, becomes the same as Hamiltonian path problem

Balanced Cuts

Does there exist a way to partition vertices V in a graph into two sets S and T such that

- there are at most b edges between S and T, and
- $|S| \geq |T| \geq |V|/3$

Integer Linear Programming (ILP) and Zero-One Equations (ZOE)

ILP: Linear programing, but solutions are limited to integers

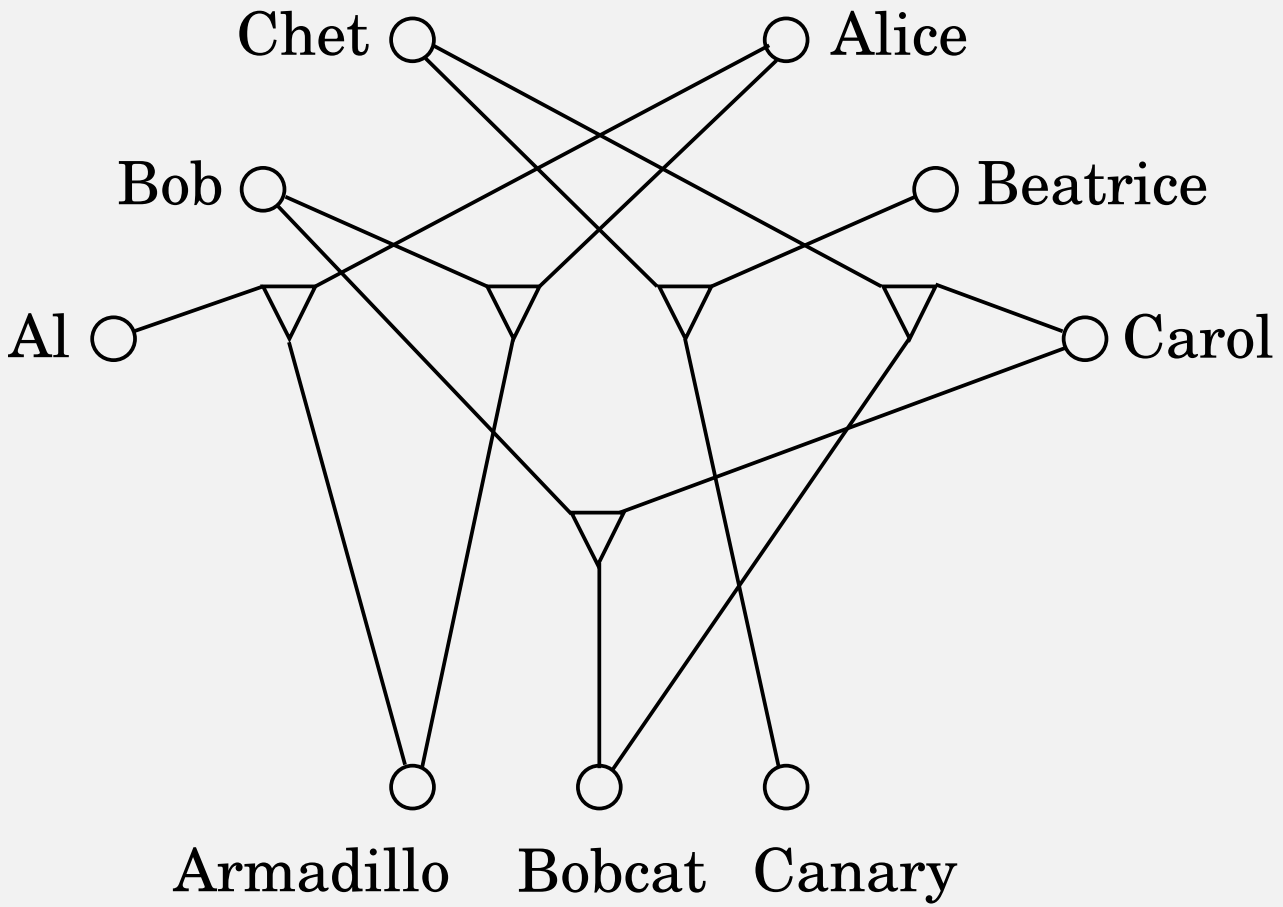
- Many problems are easy to solve over real numbers but much harder for integers.
- Examples:
  - Knapsack
  - solutions to equations such as $x^n + y^n = z^n$

ZOE: A special case of ILP, where the values are just 0 or 1.

- Find $\mathbf{x}$ such that $\mathbf{A}x = \mathbf{1}$ where $\mathbf{1}$ is a column matrix consisting of 1's.

3d-Matching

- Given triples of compatibilities between men, women and pets, find perfect, 3-way matches.

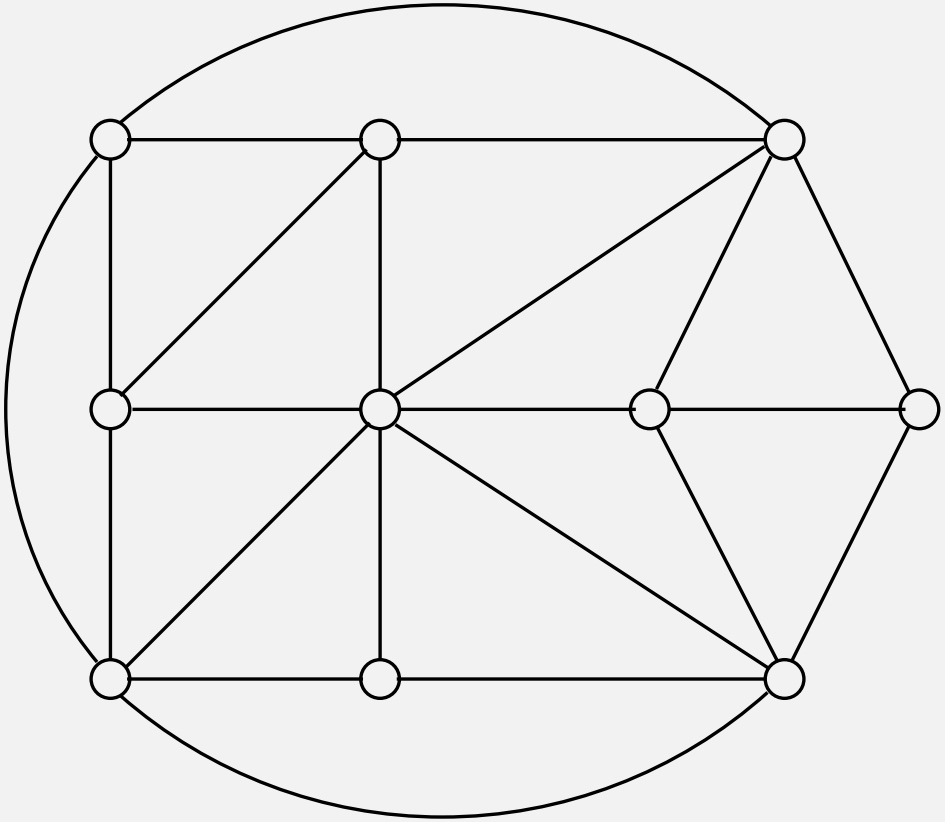


Independent set, vertex cover, and clique

Independent set: Does this graph contain a set of at least k vertices with no edge between them?

Vertex cover: Does this graph contain a set of at least k vertices that cover all edges?

Clique: Does this graph contain at least k vertices that are fully connected among themselves?

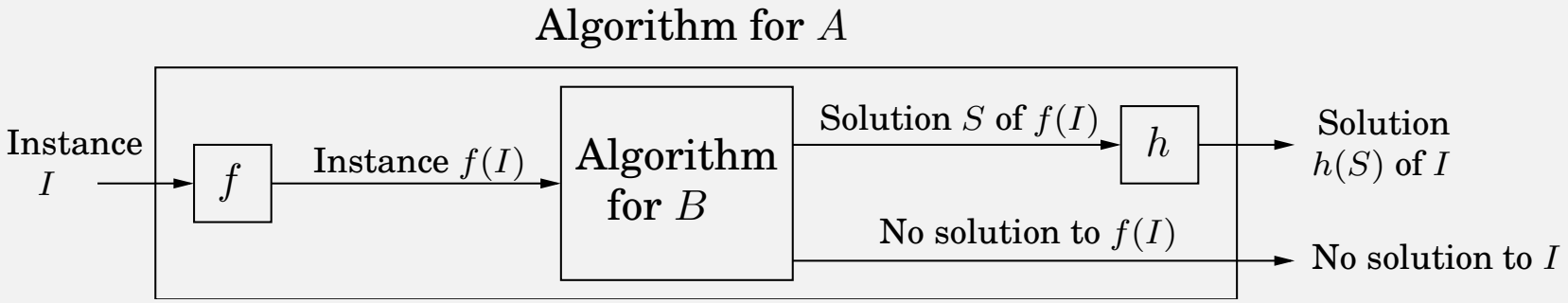


Easy Vs Hard Problems

Hard	Easy
3SAT	2SAT, HORN SAT
TSP	MST
Longest path	Shortest path
3d-matching	bipartite match
Independent set	Indep. set on trees
ILP	Linear programming
Hamiltonian cycle	Euler path,
	Knights tour
Balanced cut	Min-cut

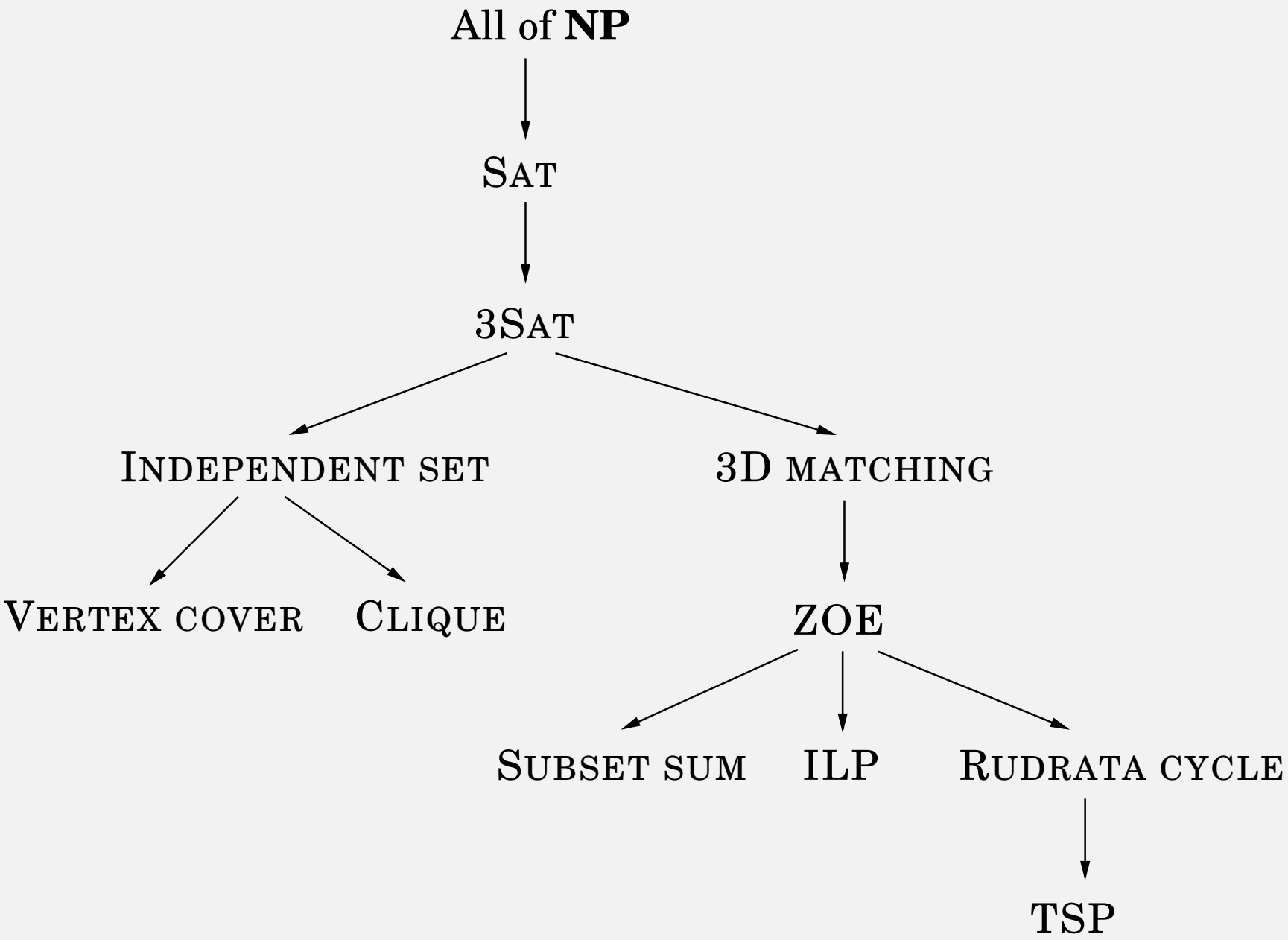
NP-completeness: Polynomial-time Reductions

- Show that a known *NP*-complete problem *A* could be transformed into problem *B* in polynomial time



- *Implication:* if *B* can be solved in *P*-time, we can solve *A* in *P*-time
- **Never forget the direction:**
  - We are proving that *B* is *NP*-complete here.

NP-completeness Reductions



Reducing all of *NP* to *SAT*

- We already discussed this
  - Show how to reduce acceptance by an NDTM to the *SAT* problem.
- *Exercise:* Show how to transform acceptance by an FSA into an instance of *SAT*

Reducing *SAT* to 3*SAT*

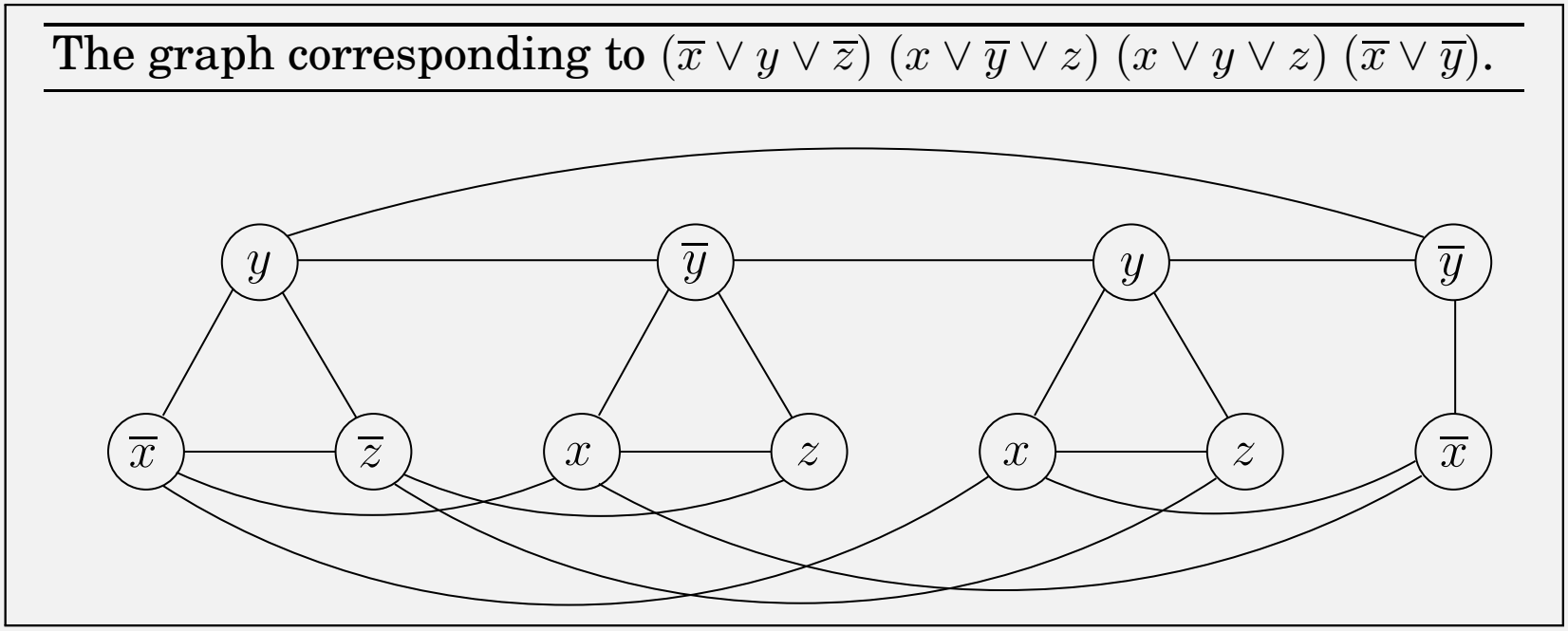
- 3*SAT:* A special case of *SAT* where each clause has ≤ 3 literals
- Reduction involves transforming a disjunction with many literals into a CNF of disjunctions with ≤ 3 literals per term
- The transformation below at most doubles the problem size.
- **Key Idea:** Introduce additional variables:
  - *Example:* $l_1 \vee l_2 \vee l_3 \vee l_4$ can be transformed into:

    $$(l_1 \vee l_2 \vee y_1) \wedge (\overline{y_1} \vee l_3 \vee l_4)$$

    For this conjunction to be true, one of $\{l_1, \ldots, l_4\}$ must be true:
    - So a solution to the transformed problem is a solution to the original — simply discard assignments for the new variables y_i.

Reducing 3*SAT* to Independent set

- Nontrivial reduction, as the problems are quite different in nature
- **Idea:** Model each of k clauses of 3*SAT* by a "triangle" in a graph



  - Independent set of size k must contain one literal from each clause
    - By setting that literal to *true*, we obtain a solution for 3*SAT*
  - *Key point:* Avoid conflicts, e.g., assigning *true* to both x and $\overline{x}$
    - ensure using edges between every variable and its complement

Reducing Independent set to Vertex Cover

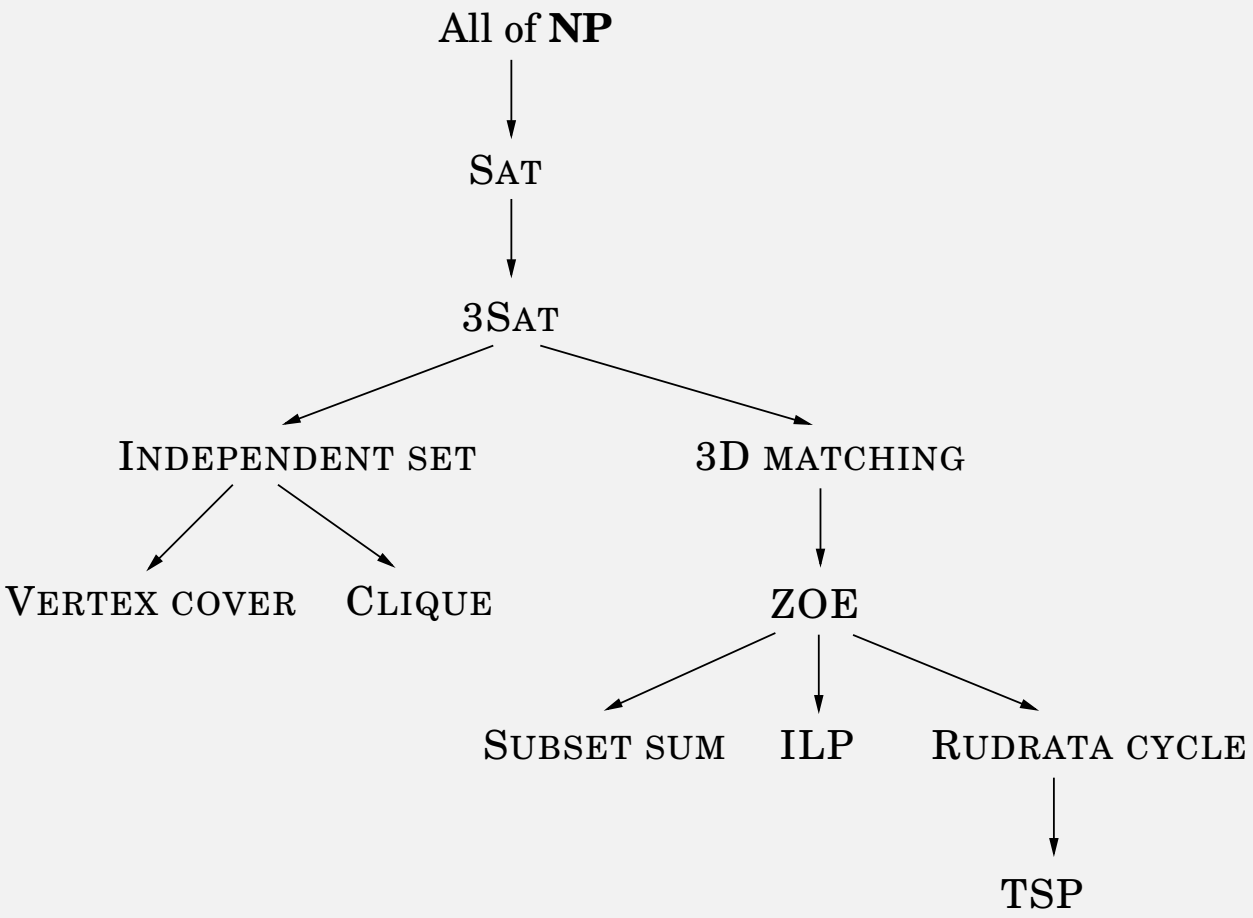
- If S is an independent set then $V - S$ is a vertex cover
  - Consider any edge e in the graph
  - *Case 1:* Both ends of e are in $V - S$
  - *Case 2:* At least one end of e is S. The other end of e cannot be in S or else S won't be independent.
  - Thus, in both cases, at least one side of e must go to $V - S$.
  - In other words $V - S$ is a vertex cover
- Thus, we have reduced independent set to vertex cover problem.

Reducing Independent set to Clique

- If S is an independent set then S is clique in $\overline{G} = (V, \overline{E})$
  - For any pair $v_1, v_2 \in S$ there is no edge in E
    - means that there is an edge between any such pair in G'
    - i.e, S is a clique in $\overline{G}$
- Thus, we have reduced independent set to the clique problem, while only using polynomial time and space.

NP-completeness Reductions

- We have discussed the left half of this picture
- We won't discuss the right half, since the proofs are similar in many ways, but are more involved.
  - You can find those reductions in the text book.



Beyond NP: PSPACE

- **PSPACE:** The class of problems that can be solved using only polynomial amount of space.
  - It is OK to take exponential (or super-exponential) time.
- *Key point:* Unlike time, space is reusable.
  - Result: many exponential algorithms are in PSPACE.
    - Consider universal formulas. We can check them in polynomial space by rerunning the same computation (say, $check(v)$) for each v.
    - The space used for *check* is recycled, but the time adds up for different v's.
- **Note:** *SAT* is in PSPACE
  - Try every possible truthe assignment for variables.
- Thus, all *NP*-complete problems are in PSPACE.

PSPACE-hard and PSPACE-complete

PSPACE-hard: A problem Π is PSPACE-hard if for any problem Π' in PSPACE there is a *P*-time reduction to Π.

PSPACE-complete: PSPACE-hard problems that are in PSPACE.

- *Examples:*

  QBF: Quantified boolean formulae

  NFA totality: Does this NFA accept all strings?

Is $NP \subsetneq$ PSPACE?

- We think so, but we can't even prove $P \subsetneq PSPACE$

CSE 548: (*Design and*) Analysis of Algorithms

Coping with NP-Completeness

R. Sekar

Coping with NP-Completeness

- Sometimes you are faced with hard problems — problems for which no efficient solutions exist.
- **Step 1:** Try to show that the problem is *NP*-complete
  - This way, you can avoid wasting a lot of time on a fruitless search for an efficient algorithm
- **Step 2a:** Sometimes, you may be able to say "let us solve a different problem"
  - you may be able leverage some special structure of your problem domain that enables a more efficient solution
- **Step 2b:** Other times, you are stuck with a difficult problem and you need to make the best of it.
  - We discuss different coping strategies in such cases.

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

  Hamiltonian Tour: Consider an edge e.
  - Either $e = (u, v)$ is part of the tour, in which case you can complete the tour by finding a path from u to v in $G - e$.
  - Or, e is not part of the tour, in which case you can find the tour by searching $G - e$.

  Either case leads to a recurrence $T(m) = 2T(m-1)$, i.e., $T(m) = O(2^m)$. (Here m is the number of edge in G.)

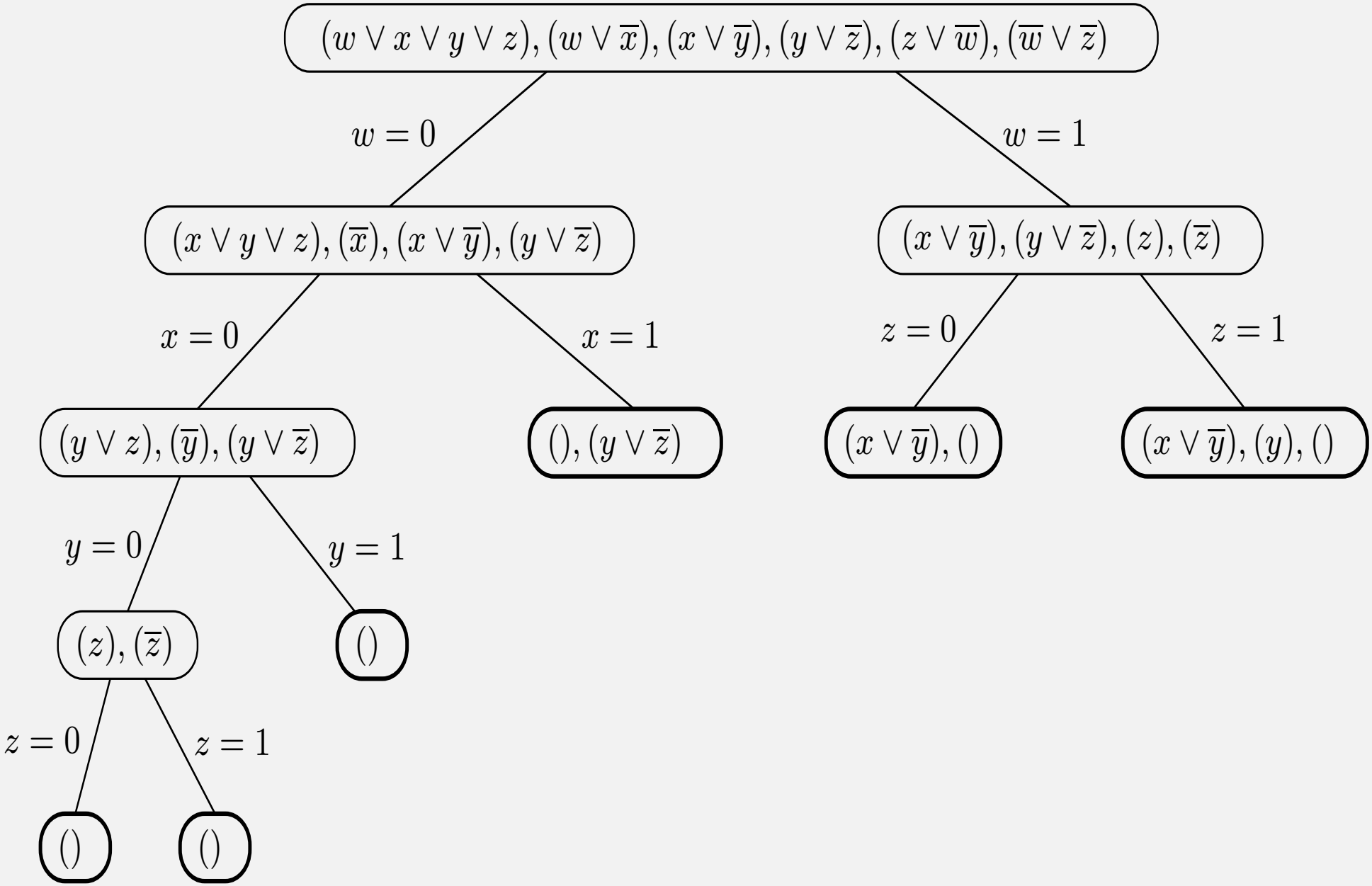
  SAT: Try all 2^n possible truth assignments to the n variables in your formula.

- The key point is to be intelligent in the way this search is conducted, so that the algorithm is faster than 2^n in practice.

Backtracking

- Depth-first approach to perform exhaustive search
  - In the above example, first try to find a solution that includes e
    - Looking down further, the algorithm will make additional choices of edges to include: $e_1, e_2, \ldots, e_k$
  - Only when all paths that include e fail to be Hamiltonian, we consider the alternative (i.e., Hamiltonian path that doesn't include e)
- Key goal is to recognize and prune failing paths as quickly as possible.

Backtracking Approach for *SAT*



Branch and Bound

- Generalization of backtracking to support optimization problems
- Requires a lower bound on the cost of solutions that may result from a partial solution
  - If the cost is higher than that of a previously encountered solution, then this subproblem need not be explored further.
- Sometimes, we may rely on estimates of cost rather than strict lower bounds.

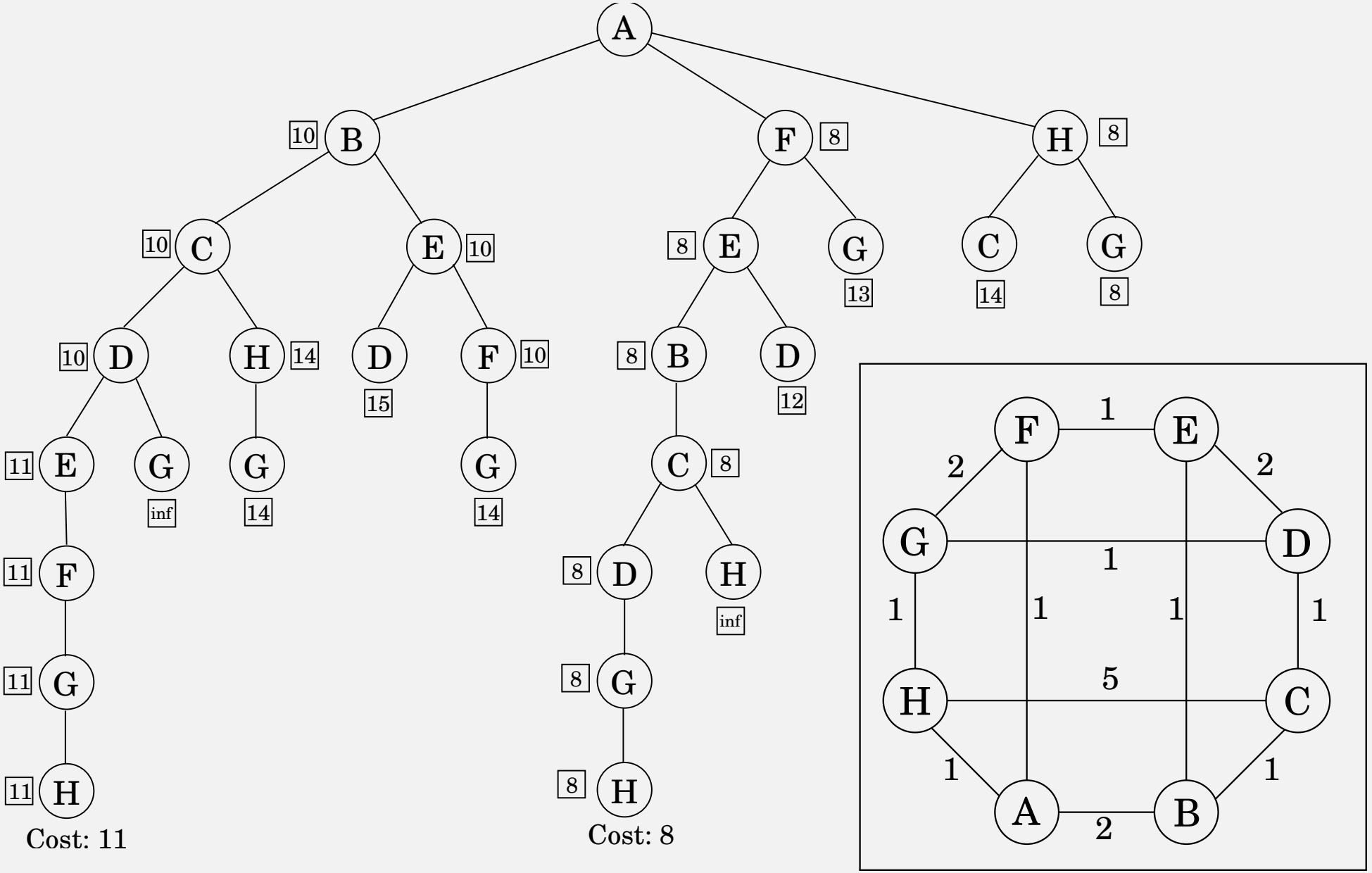
Branch and Bound for TSP

- Begin with a vertex a — the goal is to compute a TSP that begins and ends at a.
- We begin the search by considering an edge from a to its neighbor x, another edge from x to a neighbor of x, and so on.
- *Partial solutions* represent a path from a to some vertex b, passing through a set $S \subset V$ of vertices.
- *Completing a partial solution* requires the computation of a low cost path from b to a using only vertices in $V - S$

Lower bound on costs of partial TSP solutions

- To complete the path from b to a, we must incur at least the following costs
  - Cost of going from b to a vertex in $V - S$, i.e, the minimum weight edge from b to a vertex in $V - S$
  - Cost of going from a $V - S$ vertex to a, i.e, the minimum weight edge from a to a vertex in $V - S$
  - Minimal cost path in $V - S$ that visits all $v \in V - S$
    - *Note:* Lower bound is the cost of MST for $V - S$
- By adding the above three cost components, we arrive at a lower bound on solutions derivable from a partial solution.

Illustration of Branch-and Bound for TSP



Approximation Algorithms

- Relax optimality requirement: permit *approximate* solutions
  - Solutions that are within a certain distance from optimum
- *Not heuristics:* Approximate algorithms *guarantee* that solutions are within a certain distance from optimal
  - Differs from heuristics that can sometimes return very bad solutions.
- How to define "distance from optimal?"

  Additive: Optimal solution S_O and the Solution S_A returned by approximation algorithm differ only by a constant.
  - Quality of approximation is extremely good, but unfortunately, most problems don't admit such approximations

  Factor: S_O and S_A are related by a factor.
  - Most known approximation algorithms fall into this category.

Approximation Factors

Constant: $S_A \leq kS_O$ for some fixed constant k.

- *Examples:* Vertex cover, Facility location, ...

Logarithmic: $S_A \leq O(\log^k n) \cdot S_O$.

- *Examples:* Set cover, dominating set, ...

Polynomial: $S_A \leq O(n^k) \cdot S_O$.

- *Examples:* Max Clique, Independent set, graph coloring, ...

PTAS: $S_A \leq (1+\epsilon) \cdot S_O$ for any $\epsilon > 0$.

("Polynomial-time approximation scheme")

FPTAS: PTAS with runtime $O(\epsilon^{-k})$ for some k. ("Fully PTAS")

- *Examples:* Knapsack, Bin-packing, Euclidean TSP, ...

Bin Packing

Problem

Pack objects of different weight into bins that have a fixed capacity in such a way that minimizes bins used.

- Obvious similarity to Knapsack
- Bin-packing is NP-hard
- Very good (and often very simple) approximation algorithms exist

First-fit Algorithm

A simple, greedy algorithm

FirstFit$(x[1..n])$

for $i = 1$ **to** n **do**

Put $x[i]$ into the first open bin large enough to hold it

Theorem

All open bins, except possibly one, are more than half-full

Proof: Suppose that there are two bins b and b' that are less than half-full. Then, items in b' would have fitted into b, and so the FF algorithm would never have opened the bin b' — a contradiction ■

Theorem

First-fit is optimal within a factor of 2: specifically, $S_A < 2S_O + 1$.

Best-Fit Algorithm

- Another simple, greedy algorithm
- Instead of using the first bin that will can hold $x[i]$, use the open bin whose remaining capacity is closest to $x[i]$
  - Prefers to keep bins close to full.
- Factor-2 optimality can established easily.

Other algorithms for Bin-packing

- *First-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs first-fit.
- *Best-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs best-fit.
- Both FFD and BFD achieve approximation factors of $11/9 S_O + 6/9$.
- Due to the additive term, bin-packing cannot have a PTAS unless $P = NP$.
  - But $S_A = (1+\epsilon)S_O + 1$ is easy to achieve for any $\epsilon > 0$

Set Cover

Problem

Given a collection $S_1, \ldots, S_m$ of subsets of B, find a minimum collection $S_{i_1}, \ldots, S_{i_k}$ such that $\bigcup_{j=1}^{k} S_{i_j} = B$

Greedy Set Cover Algorithm

$GSC(S, B)$

  $cover = \emptyset$; $covered = \emptyset$

  while $covered \neq B$ **do**

    Let *new* be the set in $S - cover$ containing

      the maximum number of elements of $B - covered$

    add *new* to *cover*; $covered = covered \cup new$

return *cover*

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$*, where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of *GSC*
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ each of these k sets must cover at least n_t/k uncovered elements.
- Thus, *GSC* will find at least one set that covers n_t/k elements.
- This yields the recurrence for bounding uncovered elements: $U(t+1) = n_t - n_t/k = n_t(1-1/k) = U(t)(1-1/k)$
- The solution to recurrence is $n(1-1/k)^t < ne^{-t/k}$
- Thus, after $t = k \ln n$ steps, less than 1 (i.e., no) elements uncovered
- Thus, GSC computes a cover at most $\ln n$ times the optimal cover. ■

Vertex Cover

- Note that a vertex cover is a set cover for $(\mathcal{S}, E)$, where $\mathcal{S} = \{\{(v,u)|v \in V \text{ and } (v,u) \in E\}|v \in V\}$,
  - $\mathcal{S}$ contains a set for each vertex; this set lists all edges incident on v
- Thus *GSC* is an approximate algorithm for vertex cover.
- But $\ln n$ is not a factor to be thrilled about — can we do better?
  - Actually, we can do much better! That too with a very simple algorithm.

Vertex Cover

Consider any edge (u, v).

- Either u or v must belong to any vertex cover.
- If we accept $S_A = 2S_O$, then we can avoid the guesswork by simply picking both vertices!

Approximate Vertex Cover Algorithm

$AVC(G = (V, E))$

$C = \emptyset$

while G is not empty

pick any $(u, v) \in E$

$C = C \cup \{u, v\}$

$G = G - \{u, v\}$

return C

k-Cluster

Problem

Given $X = \{x_1, \ldots, x_n\}$ and distances between x_i, partition X into k clusters in a way that minimizes maximum cluster diameter.

Approximate k-Cluster Algorithm (AC)

Pick any point $\mu_1 \in X$ as the first cluster center

for $i = 2$ **to** k **do**

Choose μ_i to be the farthest point from $\mu_1, \ldots, \mu_{i-1}$

Create k clusters $C_i = \{x \in X | \mu_i \text{ is the closest center to } x\}$

Analysis of k-Cluster

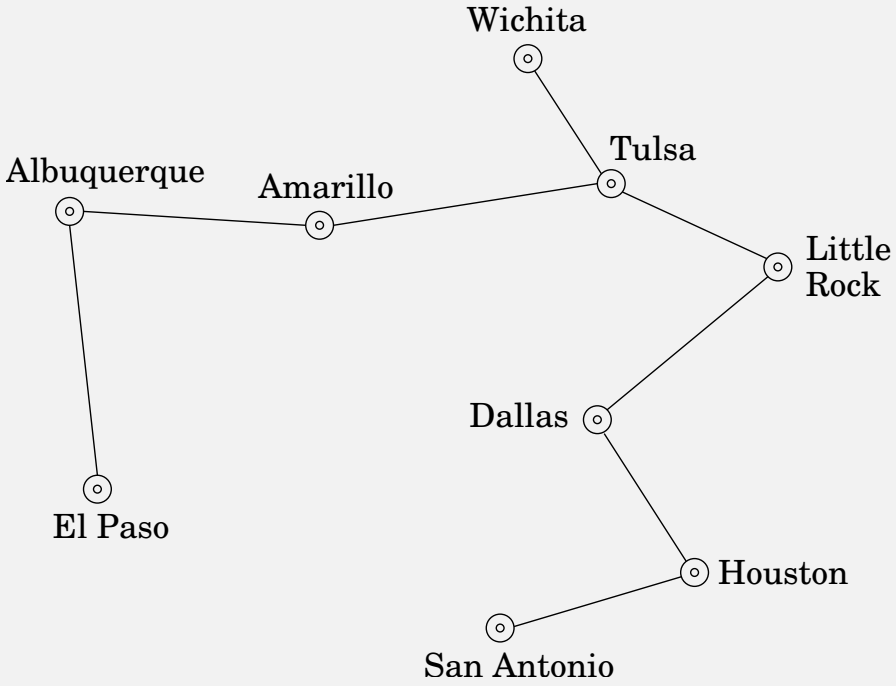
Let x be the farthest point from $\mu_1, \ldots, \mu_k$, and let r the distance to its closest center. Then, we can say:

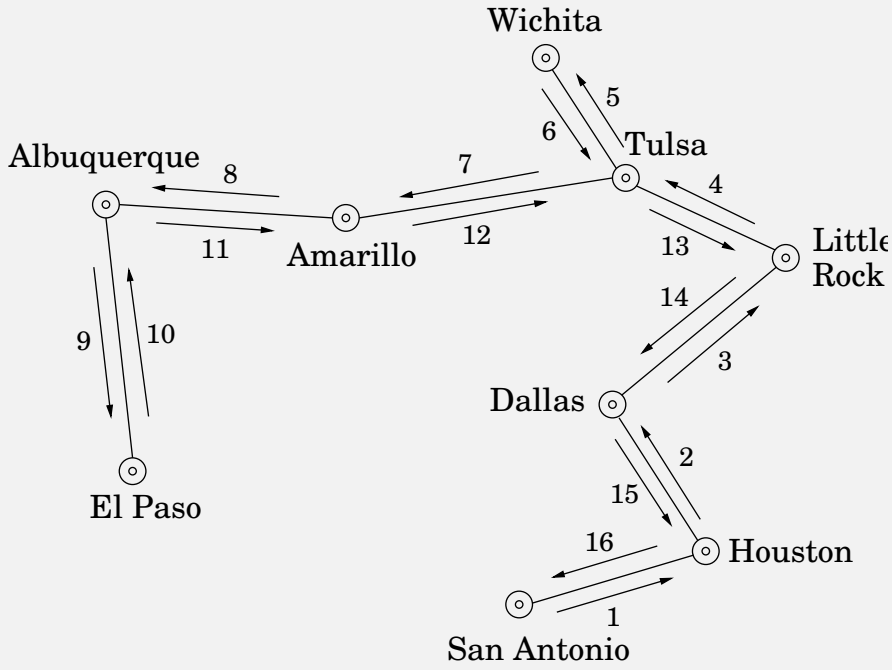
- Cluster diameter of $C_1, \ldots, C_k$ is at most $2r$
- The distance between any 2 points in $\{x, \mu_1, \ldots, \mu_k\}$ is at least r. This follows from:
  - how μ_i's was chosen to be the farthest point from μ_j for $j < i$,
  - this distance to μ_i must decrease with i, and
  - when $i = k + 1$, this distance is r
- Thus, any k-Cluster must have a diameter of at least r
  - With k circles, at least two of $k + 1$ points must be within one of them.
  - This circle's diameter must hence be r or greater
- Thus, AC is approximate within a factor of 2.

Euclidean TSP

- Our starting point is once again the MST
- Note that no TSP solution can be smaller than MST
  - Deleting an edge from TSP solution yields a spanning tree
- **Simple algorithm:**
  - Start with the MST

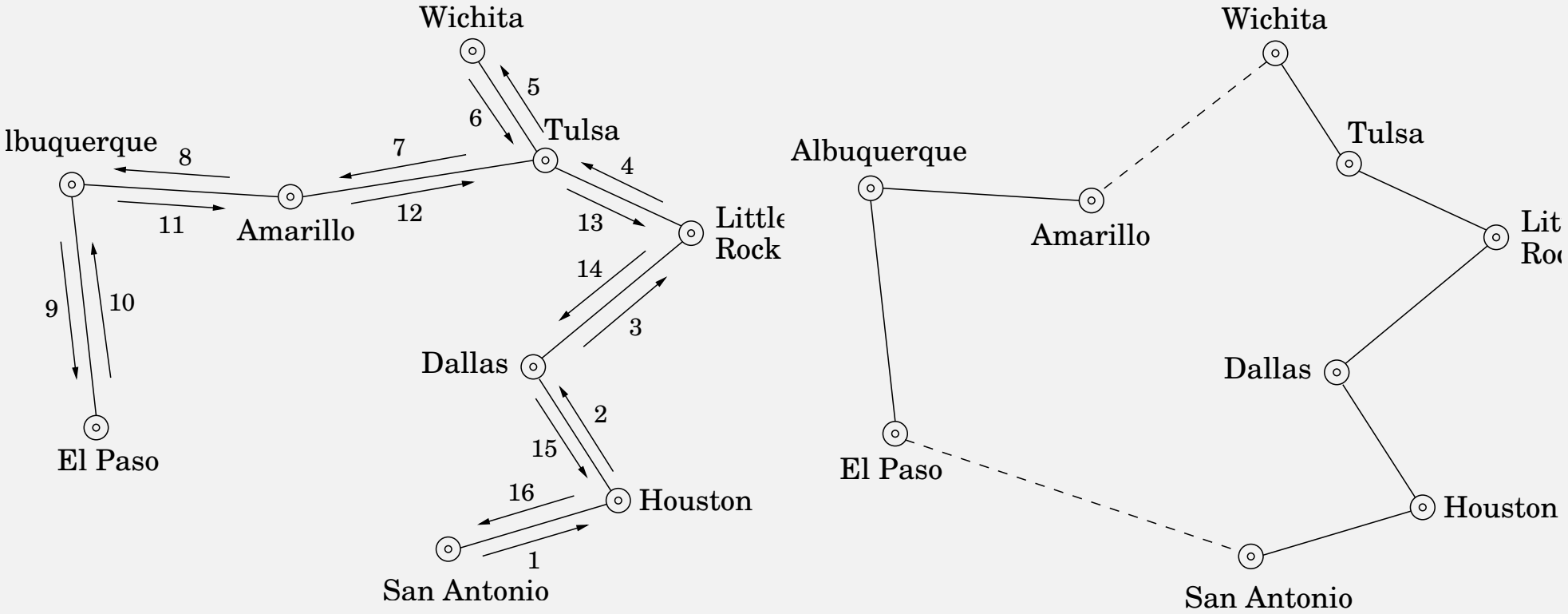
Approximating Euclidean TSP: An Illustration





- Start with the MST
- Make a tour that uses each MST edge twice (forward and backward)
  - This tour is like TSP in ending at the starting node, and differs from TSP by visiting some vertices and edges twice

Approximating Euclidean TSP: An Illustration (2)



- Avoid revisits by short-circuiting to next unvisited vertex
- By triangle inequality, short-circuit distance can only be less than the distance following MST edges.
  - Thus, tour length less than 2xMST, i.e., approximate within a factor 2.

Knapsack

$Knap01(w, v, n, W)$

$V = \sum_{j=0}^{n} v[j]$
$K[0, v] = 0, \forall 0 \leq v \leq V$
for $j = 1$ to n **do**
  for $v = 1$ to V **do**
    if $v[j] > v$ **then** $K[j, v] = K[j-1, v]$
    else $K[j, v] = min(K[j-1, v], K[j-1, v-v[j],] + w[j])$
return maximum v such that $K[n, v] \leq W$

- Computes minimum weight of knapsack for a given value.
- Iterates over all possible items and all possible values: $O(nV)$
  - we derive a polynomial time approximate algorithm from this

FPTAS for 0-1 Knapsack

*Knap*01*FPTAS*(w, v, n, W, ϵ)

$$v'_i = \left\lfloor \frac{v_i}{max_{1 \le j \le n}\, v_j} \cdot \frac{n}{\epsilon} \right\rfloor, \text{ for } 1 \le i \le n$$

*Knap*01(w, v', n, W)

- Rescaling consists of two steps:
  - Express value of each item relative to the most valuable item
    - If we worked with real values, this step won't change the optimal solution
  - Multiply relative values by a factor n/ϵ to get an integer
- Floor operation introduces an error ≤ 1 in v'_i (e.g., $\lfloor 3.99 \rfloor = 3$)
- Error in *Knap*01 output = error in $\sum v'_i$, which is at most $n \cdot 1$
- We scale each v'_i by n/ϵ, so relative error is $n/(n/\epsilon) = \epsilon$
  - Thus we have achieved the desired approximation.

FPTAS for 0-1 Knapsack: Runtime

$Knap01FPTAS(w, v, n, W, \epsilon)$

$$v'_i = \left\lfloor \frac{v_i}{max_{1 \le j \le n}\ v_j} \cdot \frac{n}{\epsilon} \right\rfloor, \text{ for } 1 \le i \le n$$

$Knap01(w, v', n, W)$

- Note that we are using $Knap01$ with rescaled values, so the complexity is $O(nV')$.
- Note: $V' = \sum_1^n v'_i \le n \cdot max_{1 \le j \le n}\ v'_j$
- It is easy to see from definition of v'_i that $max_{1 \le j \le n}\ v'_j = n/\epsilon$. Substituting this into the above equation yields a complexity of:

$$O(nV') \le O(n(n \cdot max_{1 \le i \le n}\ v'_i)) = O(n(n \cdot (n/\epsilon))) = O(n^3/\epsilon)$$

- By varying ϵ, we can trade off accuracy against runtime.