Algorithms

Lecture 1

Fibonacci Sequence

1, 1, 2, 3, 5, 8, . . .

$F_0 = 1, \; F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$

Input: A non-negative integer a in binary

Output: the a^{th} number of the Fibonacci sequence, F_a

Method 1: Recursion

Compute F_a by recursively computing F_{a-1} and F_{a-2}

Psuedocode

Base Case:
 if $a = 0$ or $a = 1$
  return 1

Recursive Case: repeated work
 Return $Fib(a-1) + Fib(a-2)$

Method 2 - Table Filling

Build sequence from beginning

Psuedocode
 Let f be an array with indicies $0, \dots a$
 Initialize $f[0] \leftarrow 1$ and $f[1] \leftarrow 1$
 For $i = 2$ to a:
  $f[i] \leftarrow f[i-1] + f[i-2]$

Improve algorithm by only retaining 2 numbers (Improves memory)

Runtimes

Recursive Routine:

$T(a) = T(a-1) + T(a-2) + \text{extra}$

$T(a) \geq T(a-1) + T(a-2)$ w/ $T(1) \geq T(0) \geq 1$

runtime looks like the fibonacci sequence!

Proof by Induction

Base Case: $T(1) \geq F_1$ and $T(0) \geq F_0$

Inductive Step:
 Assume: $T(i) \geq F_i$ for all $i < a$ and $a \geq 2$ ← Strong inductive assumption
 from above, we know that $T(a) \geq T(a-1) + T(a-2)$
 Applying the inductive hypothesis,

$$T(a) \geq T(a-1) + T(a-2) \geq F_{a-1} + F_{a-2} = F_a$$

$\therefore$ Takes at least $O(F_a)$

$F_a = F_{a-1} + F_{a-2}$

$= F_{a-2} + F_{a-3} + F_{a-2}$

$\geq 2 \cdot F_{a-2} \longrightarrow F_a > 2^{a/2}, \ \Omega(2^{a/2})$

Table Fill :

Cost of initializing table + cost of For loop

$O(a) \quad + \quad O(a)$

Total Cost : $O(a)$

Fibonacci Matrix Method

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F_{a-2} \\ F_{a-1} \end{bmatrix} = \begin{bmatrix} F_{a-1} \\ F_{a-1} + F_{a-2} \end{bmatrix}$$

Simply, $$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{a-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} F_{a-1} \\ F_a \end{bmatrix}$$

↖ can be done in $O(\log a)$

Lecture 2

Asymptotic Notation

Characterize long term behavior and hide lower order terms and constant factors

$O, \Omega, \Theta, o, \omega$

$\leq \ \ \geq \ \ = \ \ < \ \ >$

Big O

$f(n) \leq O(g(n))$ if for f is eventually bounded above by $c \cdot g$ for some constant c

$\exists c, n_0 \ \forall n \geq n_0; f(n) \leq c \cdot g(n)$

Alternate notation | $f(n) = O(g(n))$
$f(n) \in O(g(n))$

Allows for easy scaling and discussions of order of magnitude

Different languages have varying constants built into them

Provides upperbound for worst case scenarios

Any constant is $O(1)$

Big Omega

Provides a lower bound

We say $f(n) \geq \Omega(g(n))$ if $f(n)$ is at least $c \cdot g(n)$ for some c and sufficiently large n

$c > 0$

Big Theta

We say $f(n) = \Theta(g(n))$ if $f(n) \leq O(g(n))$ and $f(n) \geq \Omega(g(n))$

Symmetric relation

* Constant factor for change of log base

Asymptotic Arithmetic

If $f(n) \leq O(g(n))$

$f(n) + g(n) \leq O(g(n))$

In general, $f(n) + g(n) = \Theta\left(\max\left(f(n), g(n)\right)\right)$

Lecture 3

Mergesort

Easy to merge two sorted lists into 1 big sorted list

Input: Two sorted lists A and B of n integers each

Output: A sorted list C of all $2n$ elements of A and B

Least element of C must be the least element of A or B

repeated comparisons of first item in A and B

Correctness arguement is an arguement of invariance (minimum value is in first of A or B)

Runtime: $O(n)$

Each round is just a comparison and add to C (constant time)

At most $2n$ rounds

Sorting a list A of n integers

split it in half and recursively sort L and R

Merge L and R into one sorted list

Pseudocode for Mergesort

Input: list A of n integers

Output: sort(A)

Let $L \leftarrow$ Mergesort $(A[1, \ldots n/2])$

$R \leftarrow$ Mergesort $(A[n/2+1, \ldots n])$

Return Merge (L,R)

Correctness: Induction

Base Case: Mergesort is correct on lists of length 1

Inductive Step: Assume Mergesort is correct on inputs of length $1 \ldots n-1$ ← Strong Induction (True on all previous values)

Since it works for lists of length $n/2$ we know that L and R is correctly sorted

We further know that merge is correct so merge (L,R) is correctly sorted

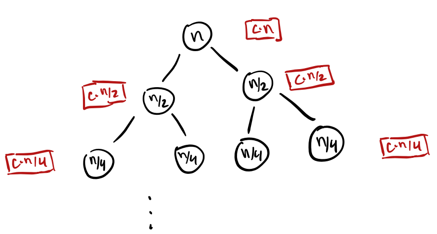
Runtime: $O(n \log n)$

Recurrence Relation

Let $T(n)$ be the worst-case runtime of Mergesort of inputs of length n

$T(n) \leq T(n/2) + T(n/2) + O(n)$ ← Any algorithm of this form is $O(n \log n)$

$\leq 2 \cdot T(n/2) + c \cdot n$



Input size for layer i is $n/2^i$

2^i nodes in layer i

Total Extra work $= 2^i \cdot c \cdot \left(\frac{n}{2^i}\right) = c \cdot n$

nodes ↗ ↖ work per node

$\log_2 n + 1$ layers ← number of times you can half a list

Total Work: $O(\log(n) \cdot c \cdot n) = O(n \cdot \log(n))$

Divide and Conquer

<u>Divide</u> problem into smaller subproblems

splitting array

<u>Conquer</u> the subproblems

sorting half arrays

<u>Post process</u> to form a solution to original problem

Merging sorted lists

Analyzing runtimes will almost alway involve a recurrence expression

Correctness is typically argued via induction

Tends to involve separate analysis of subroutines (correctness and runtimes)

Primitive Operations

Operations that run in constant time

- Addition
- Subtraction
- Arithmetic ← not always constant time
- Ifs
- Boolean operations
- Indexing
- bitwise/string operations

Integer Multiplication Problem

Input: two n-bit integers x and y in binary

Output: x·y written in binary

$$\left.\begin{array}{l} \text{for } i=0 \text{ to } x-1: \ O(2^n) \\ \quad \text{val} += y \leftarrow O(n) \end{array}\right\} O(2^n \cdot n)$$

Elementary Algorithm (Traditional Multiplication)

$$\begin{array}{r} 101 \\ \underline{011} \\ 101 \\ 1010 \\ \underline{00000} \\ 1111 \end{array}$$

$O(n^2)$ look at each bit of x and multiply across n bits in y

Divide and Conquer Algorithm

Think of x as an n-bit string

a represents first $n/2$ bits and b as the second $n/2$ bits

$$x = a \cdot 2^{n/2} + b$$

↖ shifting

$$y = c \cdot 2^{n/2} + d$$

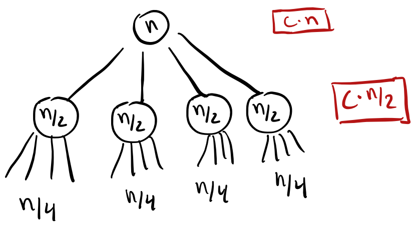
$$x \cdot y = 2^n \cdot a \cdot c + 2^{n/2} ad + 2^{n/2} bc + bd$$

$$= 2^n \, a \cdot c + 2^{n/2} (ad + bc) + bd$$

Recursively comput $a \cdot c$, ad, bc, and bd and plug into expression

$O(n)$ per shift and addition

Recurrence : $T(n) \leq 4 \cdot T(n/2) + O(n)$



Input size at layer i: $n/2^i$

Nodes at layer i: 4^i

Extra work in layer i: $\left(c \cdot \frac{n}{2^i}\right) \cdot 4^i$

Total Work : $\sum_{i=0}^{\log_2(n)} 2^i cn$

work in last layer is $\Theta(n^2)$

Lecture 4

Binary Search

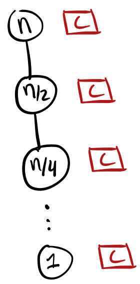
If list of integers A is sorted, can binary search for target value t

Check middle element m of A

- If $m=t$, done!
- If $m>t$, recursively search left half
- If $m<t$, recursively search right half

Recurrence: $T(n) \leq T\left(\frac{n}{2}\right) + O(1)$

$O(\log n)$



log n layers!

In an unsorted list we run in $O(n)$ ← search entire list

For k searches, at what point should we sort?

Sort: $O(n\log(n) + k \cdot \log(n))$ no-sort: $O(k \cdot n)$

$k > \log n$ it is worthwhile to sort

Karatsuba Trick

We only need $ad+bc$ in $x \cdot y = 2^n ac + 2^{n/2}(ad+bc) + bd$

$$\underbrace{(a+b)(c+d)}_{\text{contains all the values we need}} = ac + bd + ad + bc$$

Psuedocode: Karatsuba's Algorithm

Input: two n-bit integers x, y

Output: $x \cdot y$

Define a,b,c,d as before

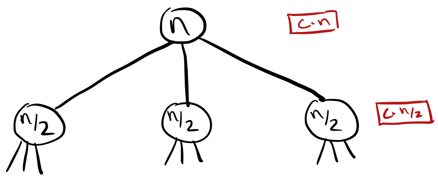
$m_1 \leftarrow \text{mult}(a,c)$
$m_2 \leftarrow \text{mult}(b,d)$
$m_3 \leftarrow \text{mult}(a+b, c+d)$

return $2^n \cdot m_1 + 2^{n/2}(m_3 - m_1 - m_2) + m_2$

shift and addition

Recurrence: $T(n) \le 3T(n/2) + O(n)$

$\le 3T(n/2) + c \cdot n$



Input size: $n/2^i$

Nodes: 3^i

Extra work: $3^i \cdot c \cdot \frac{n}{2^i} = \left(\frac{3}{2}\right)^i \cdot c n$

Total Work: $\sum_{i=0}^{\log_2(n)} \left(\frac{3}{2}\right)^i \cdot c n = c n \sum_{i=0}^{\log_2(n)} \left(\frac{3}{2}\right)^i \approx c n \cdot \Theta\left(\left(\frac{3}{2}\right)^{\log_2(n)}\right)$

geometric series!
Sum is dominated by largest term

If $a > 1$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}$$

$$= \Theta\left(n \cdot \frac{3^{\log_2(n)}}{2^{\log_2(n)}}\right) = \Theta\left(3^{\log_2 n}\right) = \Theta\left(2^{\log_2(n)\log_2(3)}\right) = \Theta\left(n^{\log_2(3)}\right)$$

$3 = 2^{\log_2(3)}$ $\approx n^{1.5}$

Master Theorem

Typical Recurrence Expression

$$T(n) \le a \cdot T(n/b) + O(n^d)$$

a: recursive calls
b: splitting factor
$O(n^d)$: post-processing

* technically only considers polynomial post processing

Three Common Cases

Work per layer can
- stay constant throughout the tree
- Grows
- Shrink

Work in layer i: [# nodes] · [work per node]

nodes in layer i: a^i

Work per node: $c \cdot (n/b^i)^d$

$$a^i \cdot c \cdot (n/b^i)^d = \left(\frac{a}{b^d}\right)^i \cdot c \cdot n^d$$

Runtime Analysis by Work per Layer

1) Stays same when $a = b^d$

Work-per-layer: $c \cdot n^d$

Total Work: $\log_b(n) \cdot c \cdot n^d$

$\log_b(n)$: number of layers

$O(n^d \log(n))$

2) Grows when $a > b^d$

Work-per-layer: $c n^d \left(\frac{a}{b^d}\right)^i$

Total Work: $c n^d \cdot \sum_{i=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^i$

$\approx c n^d \cdot \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right)$

Notice $(b^d)^{\log_b(n)} = \left(b^{\log_b(n)}\right)^d = n^d$

$\cancel{c n^d} \cdot \Theta\left(\frac{a^{\log_b(n)}}{\cancel{n^d}}\right)$

$\Theta\left(a^{\log_b(n)}\right) \xrightarrow[a = b^{\log_b(a)}]{} \Theta\left(b^{\log_b(n)\log_b(a)}\right)$

$O\left(n^{\log_b(a)}\right)$

3) Shrinks when $a < b^d$

Work-per-layer: $c n^d \left(\frac{a}{b^d}\right)^i$

Total Work: $c n^d \cdot \sum_{i=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^i$

$\approx c n^d \cdot \Theta(1)$

$O(n^d)$

Lecture 5

Counting List Inversions

In a list A, A[i] and A[j] is an inversion if $A[i] > A[j]$ when $i < j$

"unsorted pairs"

Naïve Algorithm

Iterate down the list and count ← nested for loops

$\Theta(n^2) \leftarrow \frac{n(n-1)}{2}$ operations

Divide and Conquer Approach

Split the list in half

Recursively count # of inversions

↑ misses cross list inversion

A cross inversion is an inversion where $i \le n/2$ and $j > n/2$

Total Inversions = Left inversions + Right Inversions + Cross Inversions

Counting Cross Inversions

Naïve approach requires $\frac{n}{2} \cdot \frac{n}{2}$ calculations

Suppose we sort LHS and RHS first

Finding one inversion gives us all subsequent inversions (linear time)

You can easily count inversions while merging

Whenever front element of L is larger than R, add # of remaining elements in L to inversion count

Notice that sorting the list after counting inversions does not change the # of inversions

Algorithm:

$\left.\begin{array}{l} L, \ell \leftarrow \text{Sort-and-Inv}(A[1,\dots,n/2]) \\ R, r \leftarrow \text{Sort-and-Inv}(A[n/2+1,\dots,n]) \end{array}\right\}$ Added convenience of sorting while counting

$C, c \leftarrow$ Merge-and-In (L, R)

Return $C, \ell + r + c$

Runtime: $O(n \log n)$ ← mergesort + bookkeeping

$$T(n) \le 2 \cdot T(n/2) + O(n)$$

Closest Pair

Input: A list A points $(x_1, y_1) \dots (x_n, y_n)$ from $\mathbb{R}^2$

Output: Closest pair of distinct points in A

Assumptions

- Can compute the distance between points in $O(1)$ time
- Every point has a distinct x_i, y_i value

Naive Implementation

Try all pairs of points and remember smallest distance

$\Theta(n^2)$ ← Nested for loops

Divide and Conquer

- $O(n \log n)$ preprocessing step
- Divide and conquer approach $O(n \log n)$

Sort coordinates by x- coordinate

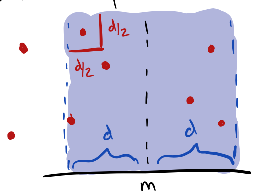
Recursively find closest pair in left half and right half

What if closest pair is split across halves?

Solution: let d be the shortest distance found from recursive calls

let m be an intermediate value between the two lists

Only need to check points within d distance of m



Takes $O(n)$ time to determine the points in the box

check every x-value

Maximum of n points in the box

Only one point can be in $d/2 \times d/2$ box

Only ≤ 4 points per row

Only have to consider 3 rows ← constant number of checks

(from top row)

small integer

Points sorted by y order are sorted by row value

Algorith Idea

1) Sort points by x-coordinate and store
2) Sort points by y-coordinate and store
3) Select m, an x-value between RHS and LHS
4) Find set of B points w x-coor between $[m-d, m+d]$
5) Find set of B sorted by y-coordinate
   ↳ use preprocessing y-sorted list
6) For each coordinate search for pairs within 17 coordinates of it

For $i = 1$ to $n-17$:
  For $j = i+1$ to $i+17$:
    Consider $(B[i], B[j])$

} $O(n)$

Final Runtime

Preprocessing: $O(n \log n)$

Parse P_x, P_y into LHS and RHS: $O(n)$

Recursive calls of size $(n/2)$: $2 \cdot T(n/2)$

Search for candidate crossing points: $O(n)$

Total: $O(n \log n)$

Lecture 6

Fast Matrix Multiplication

- Assume $n \times n$ square matricies
- If X and Y are n by n matrices

$$X \cdot Y = Z \text{ where } Z_{ij} = \sum_{k=1}^{n} X_{i,k} Y_{k,j}$$

Naive Implementation

Each entry takes $O(n)$ ← Assuming entry mult is $O(1)$

n^2 enteries results in $\Theta(n^3)$

Strassen's Algorithm

Attempted to prove matrix multi $\Omega(n^3)$ arithmetic operations

For a 2x2 matrix

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$ ← recursive multiplications + additions

8 recursive multiplications + $O(n^2)$ per addition

Recurrence : $T(n) \leq 8 \cdot T(n/2) + O(n^2)$

Master Theorem: $O(n^3)$

Strassen represented this via 7 multiplications

Runtime: $O(n^{\log_2 7})$

Strassen's Trick

$P_1 = A \cdot (F-H)$

$\vdots$

$P_7 = (A-C) \cdot (E+F)$

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

Matrix Multiplication Context

$O(n^{\omega})$

ω is the matrix multiplication constant

Fast Fourier Transform

Input : Two polynomials

Output: $p(x) \cdot q(x)$

- Assume coefficients can multiply in $O(1)$ time

Naive Algorithm runs in $O(n^2)$

Fast Fourier Transform runs in $O(n \log n)$

Integer multiplication can be done with FFT

Integer c: $c_n c_{n-1} \cdots c_0 = c_n \cdot 2^n + \cdots c_0 \cdot 2^0$

$p(x) = c_n x^n + \cdots c_0 x^0$ $\quad$ $c = p(2)$

$q(x) = d_n x^n + \cdots + d_0 x^0$

$r(x) = p(x) \cdot q(x)$ and evaluate $r(2)$

Most integer multiplication is done by look-up tables

Runtime : If multiplication is $O(1)$, then it runs in $O(n \log n)$

In reality it runs $O(n \log n \cdot \log \log n)$

complex number multiplication

Quicksort and Quickselect Algorithm

Algorithm Idea:

Input: $A[1,\dots,n]$

1) pick a pivot element $A[i]$

2) Divide

$$\left.\begin{array}{l} L: \text{the elements} < A[i] \\ A[i] \text{ itself} \\ R: \text{the elements} > A[i] \end{array}\right\} O(n)$$

Pivot is in correct position!

Runtime :

If we select median of list: ← difficult to do

$O(n \log n)$

If we always select min or max:

$O(n^2)$ ← each partition is 1 vs $n-1$

$T(n) \leq T(n-1) + O(n)$

Solution: Randomly select pivots

Expected runtime : $O(n \log n)$

Quicksort is more space efficient than mergesort

Allows you to run the algorithm in place (doesn't require as much extra memory)

In-place Partition

- Pick pivot and move to front
- Scan from left until you find something larger than pivot
- Scan from right until you find an element smaller than pivot

Swap elements!

- Repeat until scanners cross

Swap first element (Pivot) with last of small things (left most pointer)

Median Finding and Order Statistics

i^{th} order statistic is element at i^{th} position in sorted list

Input: $A[1,\ldots,n]$ and integer $i \in \{1,\ldots n\}$

Output: i^{th} order statistic of A

In quicksort we know the place of the pivot in sorted list

Quickselect Algorithm

Pick a random pivot p and partition A around p

Let j be the position of p

Case 1: $i=j$

Return p

Case 2: $i<j$

Return Quickselect $(A[1,\ldots j-1], i)$

Case 3: $i>j$

Return Quickselect $(A[j+1,\ldots n], i-j)$

Runtime: Expected runtime $O(n)$

Randomized Algorithms tend to simplify complex arguments at the cost of complexity of analysis

Make-Up Lectures

Graphs: Visual represent pairwise relationships

Terminology

verticies/nodes (V)

Edges (E) ← exist between pairs of vertices

Undirected Graphs ← unordered pairs

Directed graphs ← arcs

Cut of a graph is a partition of V into two non-empty sets A and B

some edges are restricted to the set while others can cross the cut

Sparse vs. Dense Graphs

$n=$ # of vertices, $m=$ # of edges

usually m is $\Omega(n)$ and n is $O(n^2)$

Sparse graph has m near $O(n)$

Dense graph has m closer to $O(n^2)$

Adjacency Matrix

Represent G by a nxn matrix A where $A_{ij}=1$ is there is an edge between i,j

Easily modified for parallel, weighted, or directed edges

Adjacency List	Space
- Array of vertices	$O(n)$
- Array of edges	$O(m)$
- Each edge points to endpoints	$O(m)$
- Each vertex points to incident edges	$O(m)$

Generic Graph Search

Goals: ① Find everything findable from a given start vertex

② Efficient search $O(m+n)$

- Note whether each vertex has been explored

begin with s explored and all others unexplored

- While possible:

choose edge (u,v) s.t. u is explored and v is unexplored ← How do we select the next node?

mark v explored

Halt if no such edge exists

Two Major Methods

Breadth-First Search (BFS)
- explore nodes in layers
- Computes shortest paths
- Computes connected components in undirected graphs

vs.

Depth-First Search (DFS)
- explore aggressively and only backtrack when necessary
- compute topological ordering
- compute connected components in directed graphs

Both approximately linear time $O(m+n)$

Breadth-First Search (BFS)

- All nodes are initially unexplored
- Let Q=queue, initialized with s

↖ first in first out

- While $Q \neq \emptyset$:
  - remove the first node of Q, v
  - for each edge (v,w)
    - if w unexplored
      - mark w as explored
      - Add w to Q ← at the end

Runtime

runtime of main while loop is $O(n_s + m_s)$

nodes reached from s ; edges reached from s

Finding Shortest Path

goal: Compute dist(v) the fewest # of edges on a path s to v

Extra code: Initialize $dist(v) = \begin{cases} 0 & \text{if } v = s \\ +\infty & \text{if } v \neq s \end{cases}$

- when considering edge (v,w)
  - if w unexplored, set $dist(w) = dist(v) + 1$

At termination $dist(v) = i \iff v$ is in the i^{th} layer from starting node

Undirected Graph Connectivity

Let $G = (V,E)$ is an undirected graph

Goal: compute all connected components

- All nodes unexplored
- for i=1 to n:
  - if i not yet explored
    - BFS (G,i) ← # calls of BFS = # connected components

Run time: $O(m+n)$

O(1) per edge in BFS

O(1) per node

Depth - First Search (DFS)

- more aggressive cousin of BFS
- Only backtracks when necessary
- Mimics BFS but uses a stack instead of a queue

Recursive Algorithm

DFS (graph G, start vertex s)
  mark s as explored
  for every edge (s,v):
    if v unexplored
      DFS

- Same runtime as BFS $O(n_s + m_s)$

Topological Sort

Topological ordering labels G's nodes such that

- $f(v)$ are the set $\{1, \dots, n\}$
- $(u,v) \in G \Rightarrow f(u) < f(v)$

"only forward arrows in the ordering"

G must be acyclic ← no topological order

No directed cycles → topological order

Every acyclic graph has a sink vertex

Straight forward solution is a simple recursive algorithm that places sink vertices at the end of the list

Modified - DFS for Topological Sort

DFS-Loop (G):
  mark all nodes unexplored
  current_label = n
  for each vertex $v \in G$:
    if v not yet explored
      DFS (G,v)

DFS (G,v)
  mark s explored
  for every edge (s,v):
    if v not yet explored
      DFS(G,v)
  set f(s) = current_label
  current_label --

Essentially, DFS burrows to quickly find sink vertex and then places a label

Runtime : $O(m+n)$

O(1) per node, O(1) per edge

Correctness : If (u,v) is an edge, then $f(u) < f(v)$

Case 1: u visited before v

recursive call to v finishes before u ← DFS structure

Case 2: v visited before u

v recursive call is finished first

Strongly Connected Components

SCCs of a directed graph G are the equivalence relation $u \sim v \iff \exists$ path $u \rightsquigarrow v$ + path $v \rightsquigarrow u$

DFS for SCC computation

envoking DFS from a node produces an SCC but can find extra SCCs depending on where you start

Kosaraju's Two-Pass Algorithm

Compute SCCs in $O(m+n)$ time

Algorithm:

① Reverse all the arcs in G

② Run DFS-Loop on G^{rev} ← computes ordering of nodes

Notes $f(v)$ = finishing time of

③ Run DFS-Loop on G ← finds SCCs

processes nodes in decreasing order of finish time

SCCs are nodes with the same leader

DFS-Loop (G)

global variable $t = 0$ ← finishing time

global variable s = NULL ← Leaders (2^{nd} pass)

Assume nodes labelled 1 to n:

For $i = n$ down to 1

if i not yet explored

$s = i$

DFS (G, i)

DFS (G, i)

mark i as explored

set leader (i) = node s

for each arc $(i,j) \in G$:

if j not yet explored:

DFS (G, j)

t ++

set $f(i) = t$

Runtime: $O(m+n)$

Correctness Arguement

The SCCs of a directed graph induce an acyclic "meta-graph"

meta-nodes are simply SCCs

Connections in meta-nodes are connections between SCCs

SCCs are necessarily acyclic

otherwise collapse into one SCC

Reversing the graph doesn't change SCCs

Lemma: Consider two "adjacent" SCCs in G

$C_1 \to C_2$

Let $f(v)$ = finishing times of DFS-loop in G^{rev}

then $\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v)$

Corollary: maximum f-value of G must lie in SCC sink

First call to DFS discovers C^* and nothing else

← sink SCC

Subsequent DFS calls function analogously to recursing on G w/ C^* deleted

starts in sink SCCs

Proof of Lemma

in G^{rev} $\underbrace{C_1}_{SCC} \leftarrow \underbrace{C_2}_{SCC}$

Case 1: $v \in C_1$ ← encounter C_1 before C_2

Explore all of C_1 before C_2 is ever reached

Every single finishing time in $C_2 > \forall v \in C_1$

Case 2: $v \in C_2$

DFS (G^{rev}, v) won't finish until all of $C_1 \cup C_2$ is completely explored

$f(v) > f(w) \quad \forall w \in C_1$

Lecture 9

Dijkstra's Algorithm

high level idea is to solve a growing cut of the graph

Begin w/ one solved vertex s

Solve for another vertex b

- Consider edges starting at a solved edges and ending in unsolved edges
- Shortest path ending in (u,v) is the shortest path to u with (u,v) added on
- Select edge (u,v) s.t. dist(u) + $w(u,v)$ is smallest

Solve for vertex c

Input : A graph $G = (V,E)$ and edge weights $w(e_1), \dots w(e_m)$ and a source vertex $s \in V$

Output : A table encoding the shortest paths from s to each node in V

Initialize table w/ 3 values

is_solved ← False

Distance ← ∞

Predecessor ← null

Initialize s

is_solved ← True

Distance ← 0

Predecessor ← null

Update Neighbors

update table if $s \to v$ is smaller than v distance and set predecessor to s

Algorithm proceeds in rounds

each round solves a new vertex ← select vertex w/ shortest known distance

updates neighbors

Follow predecessors to retrieve shortest path from table

Essentially creates a shortest path tree

Runtime Analysis

iterate over n verticies ← n times

Find unsolved vertex u $O(n)$

Mark u as solved $O(1)$

update neighbors $O(n)$

$O(n^2)$

Lecture 10

Dijksta's Algorithm

Input: Graph $G = (V,E)$ with weights $w(\cdot)$ and a source node s

Initialize table (is_solved, distance, predecessor)

n times For $i = 1$ to n:

$O(n)$ Find un-solved vertex u with min distance value

$O(1)$ Mark u as solved

Update neighbors of u

$O(n)$ for every vertex v s.t. $(u,v) \in E$

if distance $(u) + w(u,v) <$ distance (v)

set new distance and predecessor for v

Total Runtime : $O(n^2)$

For graphs with $\Omega(n^2)$ edges this is a great runtime, but for more sparse graphs this runtime is less than ideal

Total of $O(m)$ updates across all update neighbors steps

Algorithm Runtime

$n \cdot$ [time to find and remove element w/ min distance] $O(n)$

+

$m \cdot$ [time to update neighbor distances] $O(1)$

If we organize our data we can change these run times

$$n \cdot O(\log n) + m \cdot O(\log n) = O((n+m)\log n)$$

← much better for sparse graphs

Data Structures

Objects that store data and has a designated set of supported operations

Arrays

Gets and sets elements in $O(1)$

Fundamental structures

Lists

Operations	ArrayList	LinkedList
Append	$O(1)$ Amortized	$O(1)$
Deletion	$O(n)$	$O(n)$

Stacks and Queues

Add operation :

Remove operation :

peak operation

Size operation :

$\Big\}$ $O(1)$

Stacks only allow remove/peak of latest element

Queus only allow remove/peak of earliest element

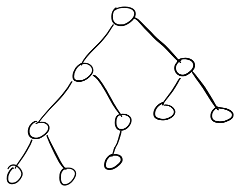
Heaps

Enforce order on retrival of objects

Each object in the heap has an associated key

smallest key object is at the "front" of the heap

Conceptualize heaps as binary trees



Every Node has a smaller key than its children

No nodes in layer i+1 unless i is maximized

Nodes fill in from left to right

Operations

- Find Min: returns top element
  - $O(1)$
- Extract Min: selects and removes top element
  - replace top node with the last element of the bottom layer ← $O(1)$
  - Select one of its new child nodes to maintain heap property
    - move top node down to keep children larger than parental nodes
    - Only need to check size vs children for each step

  } Bubble down recursively

  $O(\log n)$
- Insert
  - Add element to end of heap and fix it
    - maintain heap property: repeatedly compare w/ parent "bubble up"

  $O(\log n)$
- Delete
  - Move final element to target position
  - Fix the heap

  $O(\log n)$
- Update
  - Same as insert + delete

  $O(\log n)$
- Heapify: produces heap from array of n elements

In Dijkstra's Algorithm we keep a heap of all verticies

Each vertex is keyed by its current distance

Initialize an empty heap

- Sourse vertex with key 0
- Remaining verticies with key ∞

Find unsolved vertex u with minimum known distance and mark as solved

- whatever is at top of the heap (Extract min)

Update neighbors

- update key in heap

Runtime w/ Heaps

Initialize heap: n insertions → $O(n \log n)$

Min distance vertex: n Extract-Mins → $O(n \log n)$

Neighbor updates: m updates → $O(m \log n)$

Total: $O((n+m) \log n)$

Implementation Notes

Denote Heaps as arrays

- First index → first layer
- 2, 3 → Second layer
- ⋮

Double indicies to find children

$2i, 2i+1$

Don't need to store pointers or more complex datastructures

Lecture 11

Binary Search Trees

maintains same guarantees as a sorted list

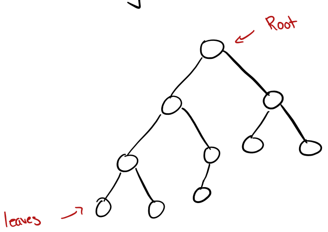
Supported Operations

Sorted Array Runtime*

- Search given a key K
  - $O(\log n)$ with binary search

- Min/Max
  O(1)
- Predecessor/Successor ← closest object with smaller/greater key
  O(1) just look at adjacent indicies
- Select
  O(1) just look for i^{th} index
- Rank ← given a key report its index if in list
  O(log n) with Binary Search
- Insert/Delete
  O(n)

Visualizing BST



BST Property

Every key v's left subtree has a smaller key than v

Every key in v's right subtree has a larger key than v

Height of a BST is the length of the longest path from root to a leaf

BST Search

1) Start at root
   if root's key is k done!
2) Check appropriate subtree
   left for < and right for >

Search Correctness

1) BST Property
   if key is in tree, it is
   - root
   - left subtree
   - right subtree
2) Proof by induction for recursive calls

Runtime

O(h)
↖ height of tree

$T(h) = T(h-1) + c$

worst case

$T(h) \leq c \cdot h = O(h)$

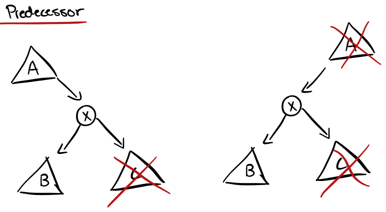
BST Min

Walk left as far as you can ← easy to do recursively

Correctness : BST property and induction

Runtime : O(h)

BST Predecessor



Best element to pick in B is max in B

Only check A when B is empty

B is in right subtree of parent of X

When we search A we just check parents of X

Use lowest common ancestor argument

Runtime
- Check for left subtree O(1)
  ① Find max of left subtree O(h)
  ② Walk-up O(h)

} O(h)

In-Order Traversal

print all elements in order

print left, print node, print right

Nodes predecessor is the one printed immediately before it

Essentially just left-biased DFS so runs in $O(n)$

$m = n-1$ for any tree

$T(n) \leq T(l) + T(r) + O(1)$

$0 \leq l$ and $0 \leq r$

$l + r = n-1$

$T(n) \leq c \cdot n$ by induction

Base Case: $T(0) \leq c$

Inductive Step: Assume $T(i) \leq c \cdot i$ for all $i < k$

prove: $T(k) \leq c \cdot k$

$$T(k) \leq T(l) + T(r) + c$$
$$\leq c \cdot l + c \cdot l + c$$
$$\leq c\,(\underbrace{l+r+1}_{k})$$
$$\leq ck$$

BST Select

Add a size arguement so each node knows how large its subtree is

```
Select (v,i):
    m = v.left_child.size + 1
    if m = i
        return r's key and object
    if m > i
        Return Select (v.left_child, i)
    if m > i
        Return Select (v.right_child, i-m)
```

adjusting order statistic

Runs in $O(h)$

BST Rank

Runs analagously to search

Lecture 12

Balanced Binary Search Trees

Operations	Runtime
Search	$O(h)$
Min/Max	$O(h)$
Predecessor/Successor	$O(h)$
In-order traversal	$O(n)$
Select	$O(h)$
Rank	$O(h)$
Insert	$O(h)$
Delete	$O(h)$

BST Insert

Recursively select correct subtree and place at end point

Runs search and adds when you reach an end point

update sizes accordingly

BST Delete

Deleting leaves and nodes with one child is easy

replace node w/ child

Find a nodes predecessor p, and then delete it and replace the node with p

Since the target node has 2 children, the predecessor is simply max of left subtree

Predecessor is easy to delete since it is a max value (no right child)

Replacing node with predecessor retains BST property since

L subtree < pred(x)

R subtree > pred(x)

Successor also works

Need to carefully update size

O(h) is not necessarily better since the graph can be unbalanced

BST height-Balance

A node is height balanced if the height of the left and right child differs by at most 1

A tree is said to be height balanced if each node is height balanced

A height-balanced binary tree with n nodes will have height $O(\log n)$

$O(h) \to O(\log n)$

Let $S(h)$ be the minimum number of nodes in balanced tree of height h

Easy to see $S(h) > h$

$S(h) \geq S(h-1) + S(h-2)$ ← Fibonacci Sequence!

Balanced Children

$\geq 2 \cdot S(h-2)$

$\geq 2^{h/2}$

For a height-balanced tree with n nodes and a height of h

$n \geq 2^{h/2}$

$\log_2(n) \geq h/2$

$2 \cdot \log_2(n) \geq h$

$h \leq O(\log n)$

Implementing BST height-balance: AVL Trees

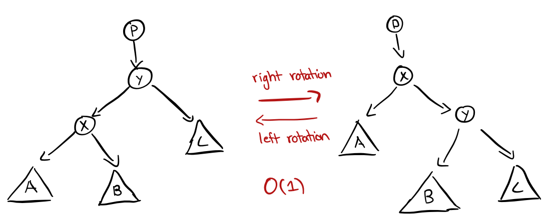
Height-balance value: height of right child − height of left child

null children have a height of −1

Imbalanced nodes have value $> |1|$

AVL trees maintain tree structure through rotations

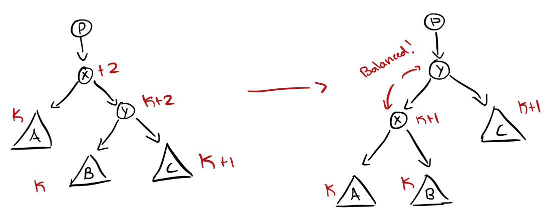
Rotations



Insertions impact the balance value for the sequence of ancestors by −1, 0, or 1

Fix imbalances as we add/delete

WLOG assume the imbalance is +2



more complicated cases exist

Lecture 13

Hash Table

Store keys and associated objects

Operations
- Search/ look-up/get
- Insert
- Delete

} Aim for $O(1)$

Runtime depends on inputs and implementation

Example Use cases

De-duplication

Input: list of n integers $A[1, \dots, n]$

Output: List B with no duplicates

Naive: nested loops and check pairs $O(n^2)$

$O(n\log n)$ solution: Sort array and iterate through

Hash solution: Use hash table to keep track of seen elements

2-sum

Input: A list of n integers $A[1, \dots, n]$ and a target t

Output: A pair of integers A which sum to t or "no pair"

Naive: Nested loops and iteratively check $O(n^2)$

$O(n\log n)$: Sort array + post processing

Hashing: Put every element in hash table

For each $A[i]$ check if $t - A[i]$ is in H

Hash Functions

- Large set of possible keys U
- Small set of keys S

given an array with indices $0, \dots, n-1$

Hash function maps keys in U to $\{0, \dots, n-1\}$

$h: U \to [n]$

Hash table

1. Array with n indices
2. Hash function $h: U \to [n]$

Insertion

given key k and value v

1. Compute $h(k)$
2. Assign $A[h(k)] \leftarrow (k,v)$

- Something might already be there

Deletion

given key k

1. Compute $h(k)$
2. Return $A[h(k)]$

A Collision is a pair of keys with the same hash value

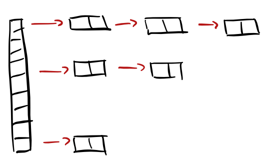
Resolved by holding an array at each hash location (Chaining)

Probing is a complex strategy to select a new position if the hash is taken

- Better theoretical guarantees but hard to study

Chaining

pointer to a linked list at each position



Insert
1) Compute $h(k)$
2) Get list $L \leftarrow A[h(k)]$
3) Append (k,v) to

Search
1) Compute $h(k)$
2) Get list $L \leftarrow A[h(k)]$
3) Search L

Delete
1) Compute $h(k)$
2) Get list $L \leftarrow A[h(k)]$
3) Delete (k,v) from list

Runtime Analysis
Consider a table w/ n entries and s elements
Assume $h(k)$ takes constant time

Insert : $O(1)$
1) Compute $h(k)$ $O(1)$
2) Get list $L \leftarrow A[h(k)]$ $O(1)$
3) Append (k,v) to $O(1)$
$O(s)$ for duplicates

Search: $O(s)$
1) Compute $h(k)$ $O(1)$
2) Get list $L \leftarrow A[h(k)]$ $O(1)$
3) Search L $O(s)$

Delete : $O(s)$
1) Compute $h(k)$ $O(1)$
2) Get list $L \leftarrow A[h(k)]$ $O(1)$
3) Delete (k,v) from list $O(s)$

Good runtimes come from avoiding collisions
1. Resize table as it gets full
2. Choose a good hash function

Re-sizing
guaranteed to have collision when $s > n$
After $n \cdot m$ insertions the average number of objects per bucket is m
When $m = \Theta(\log(m))$ our runtime guarantees are shot
We need $m \leq O(1)$
Afterwards we take a bigger array ($\sim$ double list)
m is the load factor
~ 0.7 is a good rule of thumb
Re-insert everything in new table $O(n)$
n slots with $\sim 0.7n$ insertions

Hash Function
All hash functions are bad due to the pigeon hole principle
call these datasets pathological
Select a random hash function w/ a pre-selected seed

Lecture 14

Greedy Algorithms I: Scheduling Problems
Pick the best option at each step

Scheduling Problems - "given a set of tasks, select best order"

Change-Making Problem
Input: non-negative integer r
Output: a, b, c s.t. $a+b+c$ is minimized and $a \cdot 10 + b \cdot 5 \cdot c \cdot 1 = r$
$5 \rightarrow 6$ the algorithm is wrong!

Generally greedy algorithms are easy to state and conduct runtime-analysis
Easy to come up with
Difficulty lays in selecting a good greedy algorithm ← correct

Interval Scheduling
Input: List of intervals $[s_1, f_1], [s_2, f_2], \ldots [s_n, f_n]$
Output: Longest set of non-overlapping intervals
$s_1 \geq f_2$ or $s_2 \geq f_1$

Greedy Strategy
Add intervals S that do not create conflicts + Strategy to select next interval

1. Largest First

Add largest interval that can be added to S

Counter Example

2. Earliest Start Time

Counter Example

3. Smallest Interval

Counter Example

4. Fewest Conflicts

Counter Example

5. Earliest Finish time

Let S be $\emptyset$

Let F be list of intervals sorted by finish time $O(n \log n)$

For each $x \in F$: n loops

if x is compatible with S, add x to S $O(n)$

↑ efficiently check against last element of S $O(1)$

Correctness Argument

"Greedy Stays Ahead"

1) If our algorithm is wrong, the correct output has at least one more item

2) The i^{th} element in our S will never have later finish time than i^{th} element of any other list

Lemma 2

Let T be any list of non-overlapping intervals sorted by finish time

Let S be the output of our algorithm

$$\forall i : S[i].f \leq T[i].f$$

Proof by Induction

Base Case $(i=1)$:

Trivially true since $S[1]$ is the earliest finish possible

Inductive Assumption: $1 - k^{th}$ finish times obey the property

$\Rightarrow S[k+1].f \leq T[k+1].f$ ← would have considered it earlier

Suppose $T[k+1].f < S[k+1].f$

$$S[k].f \leq T[k].f \leq T[k+1].f$$

would have picked $T[k+1]$ instead

Proving Optimality

Suppose there is a set of non-overlapping intervals T s.t. $|T| > |S| = m$

Treat S and T as sorted by finish time

From Lemma 2

$$S[1].f \leq T[1].f \quad \ldots \quad S[m].f \leq T[m].f$$

We know that $T[m+1].s > T[m].f$

Doesn't overlap with S!

Item should've been added

$S[m].f \leq T[m].f \leq T[m+1].s$

Scheduling to Minimize Lateness

Input: A list of jobs $j_1, \ldots, j_n$ such that each job j_i

time required: t_i

Deadline: d_i

Jobs start immediately after finishing previous one

Output: Order of completing jobs that minimizes lateness

We define lateness as

$$l_i = \max(f_i - d_i, 0)$$

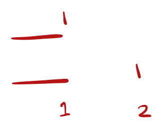
Lateness of a schedule is max lateness

$$L = \max_i l_i$$

Greedy Strategy

1. Latest Deadline First

Counter example



$L = 1$

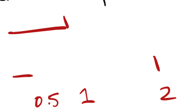
2. Most Slack Time

Counter Example

$L = 1$

3. Shortest Completion time

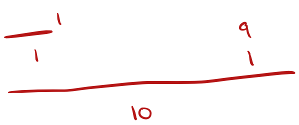
Counter Example



$L = 0.5$

4. Shortest Slack Time

Counter Example



5. Earliest Deadline

Sort and output list $O(n \log n)$

Lecture 15

Greedy Algorithms II: Finishing Scheduling Problems and Minimum Spanning Trees

Earliest Deadline Solution for minimum lateness problem

Correctness

Consider two jobs j_1 and j_2 w/ $d_1 \le d_2$

$j_1 \to j_2$ case:

$\ell_1 = t_1 - d_1 \qquad \ell_2 = t_1 + t_2 - d_2$

$j_2 \to j_1$ case:

$\ell_1 = t_1 + t_2 - d_1 \qquad \ell_2 = t_2 - d_2$

worst possible lateness!

could be 0

Greedy Exchange Arguement

If a schedule has two jobs in reverse-order by deadline

Can use two-job arguement to show that swapping these two jobs can't hurt

Any optimal solution can be sorted into our solution without hurting the solution

Lemma

Let $j_1 \ldots j_n$ be a schedule of jobs

If there is an index i such that $d_i > d_{i+1}$ ← Exists if list isn't sorted

Then jobs j_i and j_{i+1} can be swapped without increasing the lateness of the schedule

Before Swap

$\ell_i = t^* + t_i - d_i$ (start time: t^*)

$\ell_{i+1} = t^* + t_i + t_{i+1} - d_{i+1}$

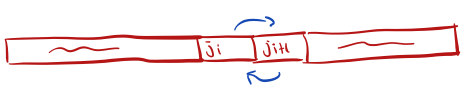
biggest lateness

After Swap

$\ell_i' = t^* + t_{i+1} - d_{i+1}$

$\ell_{i+1}' = t^* + t_{i+1} + t_i - d_i$

Swapping necessarily doesn't increase lateness! All other jobs are untouched!



Proof of Correctness

Suppose T is an optimal solution

Run bubble sort

Each change doesn't increase lateness ← Lemma

We end up with our schedule

↑ Our solution is no worse than the optimal solution

Graph Algorithms

Trees: undirected connected graph w/ no cycles

1) Connected
2) Acyclic
} from definition

3) n-1 edges

Any graph with any 2 of the properties is a tree

Proof: Any tree with n nodes has n-1 edges

$\geq n$ edges implies a cyclic graph

$\leq n-2$ edges is too few

Proof by Induction

Base Case: Graph with 0 edges has n connected components

$n-0 = n$

Inductive Step: Suppose all graphs with n nodes and k edges have at least n-k components

Need to show: All graphs with n nodes and k+1 edges have at least n-k-1 components

Let $G=(V,E)$ be a graph with n nodes with k+1 edges

Let $E' = E \setminus \{e_{k+1}\}$. Suppose $G' = (V, E')$

← has at least n-k components by inductive hypothesis

Inserting edge reduces number of components by at most 1

G has at least $(n-k)-1$ components

generally graph proofs involve selecting a graph, reducing it, and applying known properties

Spanning Trees

Subgraph of G that contains all nodes of G and is a tree

Define spanning tree by edges

$T \subseteq E$ s.t. (V,T) is a tree

Minimum (weight) Spanning Tree

Input: A weighted undirected graph $G=(V,E)$

Output: A spanning tree T of G with minimum total weight

Greedy Solutions

1. Least Weight Edge First

Pick the least weight edges first

Need to make sure we end up w/ a tree

Never add an edge that creates a cycle

Let T be empty

Let E be edges sorted by weight

For each $e \in E$:

If $T \cup \{e\}$ is acyclic, insert e into T

Lecture 16

Minimum Spanning Trees

Subgraph that contains all nodes of G and is a tree

Define tree as a set of edges T

$T \subseteq E$ s.t. (V,T) is a tree

Goal is to produce a minimum total weight tree

General greedy strategy is to select the lightest edge

Kruskal's Algorithm: Choose lightest edge that doesn't create a cycle

Prim's Algorithm: Chooses lightest edge that grows the existing tree

Why do they produce a spanning tree? ← Feasibility

Why is that spanning tree of minimum weight? ← optimality

Naive Implementations

Kruskal's Algorithm

Let T be empty

Sort e by edge weight ← m log m → O(m log n)

For each e in E:

If T ∪ {e} is acyclic, insert e in T ← BFS for path from u to v where e = {u,v}

O(n)

Total Runtime: $O(m \log(n) + m \cdot n) \le O(m \cdot n)$

Prim's Algorithm

Let T be empty

Let C = {s} for some s∈V

Let E be sorted by weight

Until C contains all vertices:

Check edges until one has endpoint in C and an endpoint outside of C ← O(m)

Can check in O(1) time

Total runtime: $O((n-1) \cdot m)$

Efficient implementations reduce runtime to $O(m \log n)$

Kruskal: "Union-find" or "Merge-find"

Prims: Heap to track min-weight edge crossing cut

Correctness

Feasibility

Kruskal's

If we had at least two components we would've selected an edge crossing the components

Prim's

Essentially connecting a leaf to the tree so no cycles

Will always add a crossing edge

Optimality

Kruskal's ← Greedy Stays ahead

Let T be set of edges produced by Kruskal's algorithm

Consider T sorted by weight

For all spanning trees T' sorted by weight

$\forall i : w(e_i) \le w(f_i)$

Proof by Contradiction

Using lemma from HW

If E_1 and E_2 are both acyclic and $|E_1| > |E_2|$ then there is some edge $e \in E_1 \setminus E_2$ s.t. $E_2 \cup \{e\}$ is acyclic

Suppose $\exists k$ s.t. $w(e_k) > w(f_k)$

Consider $\{e_1, \dots, e_{k-1}\}$ and $\{f_1, \dots, f_k\}$

Apply lemma!

Add edge to e set

Exists some $f_i \in \{f_1, \dots, f_k\} \setminus \{e_1, \dots, e_k\}$ s.t. $\{e_1, \dots, e_{k-1}, f_i\}$ is acyclic

Added edge is compatible with e subset

$w(f_i) < w(e_k)$ ← would've been added!

Interesting Corollary

Every MST has this property

If every edge has a different weight then the MST is unique

Cut Property Lemma

Let $G = (V, E)$ be a weighted undirected graph

For any cut $(S, V \setminus S)$ of G

If there is a unique lightest edge crossing the cut, then e is in every MST of G

Proof

Consider an arbitrary cut of graph G

Suppose there is a MST T that doesn't contain e

T must contain at least one edge crossing the cut $c_1, \dots, c_k$

We can swap e w/ this edge and reduce weight of T

We know that $T \cup \{e\}$ has a cycle

Any path from u to v in T where u and v exist in different cuts must use a cut edge c_i

Swap c_i with e

from u to v

Prim's is a repeated application of cut lemma when edge weights are unique

Lecture 17

For non-unique edge weights we can select any tie breaking method

Consider a tie between e_1 and e_2

Select an edge and modify it slightly ← artificial unique weights

Cheat weight C, produced by Prim's on this graph is at least as good as optimal weight O

Original MST is spanning tree of cheat graph

$$O \geq C$$

Actual value of tree is at most

$$C + (n-1)\varepsilon \geq O$$

candidate tree for original graph

$$C + (n-1)\varepsilon \geq O \geq C$$

or

$$W \geq O \geq W - (n-1)\varepsilon$$

As $\varepsilon \to 0$ then we find $W = O$

Select appropriate values of ε based on edge weight precision

Efficient Implementations

Prim's Algorithm

Almost identical to Dijkstra's algorithm

Build a heap which contains all nodes not in tree

Key for each node is weight from lightest edge in CC to node

∞ for edges outside of CC reach

When you add a node update its neighbors in the heap

$O(m \log(n))$

Kruskal's Algorithm

Speed up ways of checking if an edge creates a cycle by keeping track of connected components

Initialize with n connected components

Each time we add an edge we merge two connected components

We want a data structure that

- Fast check that two elements in same CC
- Fast merge of two CC

Union-Find

- Keeps a collection of sets

- Label of set is root of tree
  - vertex with no parent is its own set
- Merge sets by pointing root of one set to the other
  - Minimize height

Operations

Initialize / Insertion : $O(n)$

Find X: $O(\log n) \leftarrow O(h)$

Merge (X,Y) : $O(\log n)$

Runtime w/ Union-Find

- Initialize union-find $O(n)$
- Sorting E $O(m \log n)$
- m cycle checks $\leftarrow$ $O(\log n)$ each
- n merges $\leftarrow$ $O(\log n)$ each

Total: $O(m \log n)$

Union-Find can be amortized to find $O(\log^* (n))$

$\log^*$ is the number of logs required to reach 1

Lecture 18

Dynamic Programming

Memoization : prevent redundant work by saving answers

top down approach

Table-Filling is considered a bottom up approach

no recursive stack to manage

Generally give a recursive expression to solve solution from smaller solutions

Independent Set

Let $G = (V,E)$ be an undirected graph

A set $I \subseteq V$ is an independent set if no pair of vertices in I are adjacent

$$\forall u,v \in I: \{u,v\} \notin E$$

Input : A graph P is a path with n verticies

Each vertex has weight w_i

Output: The independent set in P w/ maximum total weight

greedy solutions: heaviest first

Brute Force Solution

Enumerating every subset $\leftarrow$ 2^n

We can represent our subset as a binary string

presence of node v_i in string is a 1 in the i^{th} position

Easy to enumerate

Backtracking

Let $x_i \in S$

Recursively print all subsets w/ x_i

Recursively print all subsets without x_i

For each S:

Checking if S is independent $\leftarrow O(n)$

Check if there are adjacent 1 in binary string representation

Update max if S is bigger than best known max $\leftarrow O(n)$

Total Run time: $O(2^n \cdot n)$

DP Solution

Let P_k be the initial segment of k verticies in the input graph

Two - Cases

1. Best independent set for P doesn't use v_n

    Solution is same as P_{k-1}

2. Best independent set for P uses v_n

    Solution for P_{k-2}

Pick best solution from two cases

Computing Weight

Let V_i be the weight of max independent set for P_i

From above, V_n is either V_{n-1} or $V_{n-2} + w_n$

$$V_n = \max(V_{n-1}, V_{n-2} + w_n)$$

Require Base Cases

$V_0 = 0$

$V_1 = \max(w_1, 0)$

Table - Filling Algorithm ← $O(n)$

$\left.\begin{array}{l}\text{Initialize table } V[0,\dots,n] \\ V[0]=0,\ V[1] = \max(w_1, 0)\end{array}\right\} O(n)$

$\left.\begin{array}{l}\text{For } i=2 \text{ to } n: \\ \quad V[i] = \max\left(V[i-1], V[i-2] + w_i\right)\end{array}\right\} O(n)$

Return $V[n]$ ← $O(1)$

Finding Set

We can recover the set from the table

If $V[n] = V[n-1]$ then optimal solution w/o v_n

If $V[n] = V[n-2] + w_n$ then optimal solution w/ v_n

Repeat process

DP proof of correctness is given by arguing recurrence and base cases

Weighted Interval Scheduling

Input: list of intervals $[s_1, f_1], \dots [s_n, f_n]$ w/ weights $w_1, \dots, w_n$

Output: Set of non-overlapping intervals that maximize weight

Brute Force Solution

Enumerate all sets of intervals ← 2^n

For each interval check it it is a valid schedule ← $O(n^2)$

If valid schedule and better weight, update max counter

Total Runtime: $O(2^n \cdot n^2)$

DP Solution

For each interval i, either i is in optimal solution or not

Case 1: i isn't in optimal solution

Throw away i and continue

Case 2: i is in optimal solution

Throw away intervals that conflict w/ i

This requires a table of 2^n entries!

Sort intervals by finish time to reduce subproblem size

Solution either
1) Contains $f[n]$
   $f[k]$ is the interval w/ latest finish time that finishes before $f[n]$ starts
   Solution is optimal for $f[1, \dots, k] + f[n]$
2) Doesn't contain $f[n]$
   Optimal for $f[1, \dots, n-1]$

Table-filling Algorithm ← $O(n \log n)$

Initialize $V[0, \dots, n]$, $V[0] = 0$

Let F be list of intervals sorted by finishing time

Let w_i be weight of $F[i]$

For $i = 1$ to n:
    Find latest index k s.t. $F[k]$ ends befor $F[i]$ starts
    $V[i] = \max(V[i-1], V[i-k] + w_i)$

Return $V[n]$

Lecture 19

Dynamic Programming II

Knapsack Problem

Maximize value without exceeding weight capacity

Inputs: $C, w_1, \dots, w_n, v_1, \dots, v_n$

Output: value of set S

$$\max \sum_{i \in S} v_i \quad \text{s.t.} \quad \sum_{i \in S} w_i < C$$

Optimal solution either uses object n or not
- Case 1: n not in S
  Consider $1, \dots, n-1$ with same capacity
- Case 2: n in S
  Consider $1, \dots, n-1$ w/ $C - w_n$
  Add value v_n to solution

Subproblems are parameterized by # of objects and Capacity

Case 1: n not in S

$$V_{n,c} = V_{n-1,c}$$

Case 2: n in S

$$V_{n,c} = V_{n-1,c-w_n} + v_n$$

$$V_{n,c} = \max\left(V_{n-1,c},\ V_{n-1,c-w_n} + v_n\right)$$

Base Case:

if $c = 0$ or $n = 0$, $V_{n,c} = 0$

if $c < 0$, $V_{n,c} = -\infty$

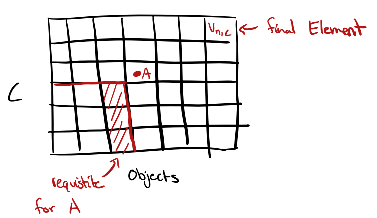


Table-Filling Algorithm

Initialize $V[0,\ldots n][0,\ldots,C]$ with base cases filled

For $i = 1$ to n:

  For $j = 1$ to C:

    $V[i,j] \leftarrow \max(V[i-1,j], V[i-1,j-w_i]+v_i)$ $\leftarrow O(1)$

return $V[n,C]$ $\leftarrow O(1)$

Total Runtime: $O(n \cdot C)$

Psuedopolynomial Runtime

$O(n \cdot C)$

If C is written in binary, the input length for C is $\log C$

↑ think about runtime as length of input

Sequence Alignment

Input: A sequence of n bases where the label is given by b_i

$b_i \in \{A, C, G, U\}$

Output: Maximum number of matched bases

Alignment Rules

1. Matching: Each base i can only match with one other base j
2. Compatibility: Matches must be between $A \leftrightarrow U$ and $C \leftrightarrow G$
3. No sharp turns: if $i < j$ and i matches j then $j \geq i+5$
4. Non-crossing: If (i,j) and (k,l) are pairs where $i<j$ and $k<l$

   can't have $i<k<j<l$

Recurrence

Either base n is matched or not

Case 1: not matched

optimal solution is $1, \ldots, n-1$

Case 2: Matched

If n matches with k

solution is optimum of 1 to $k-1$ and $k+1$ to $n-1$

Let $V_{i,j}$ be the maximal number of matchings for bases $i, i+1, \ldots, j$

If j isn't matched:

$$V_{i,j} = V_{i,j-1}$$

If j is matched w/ k:

$$V_{i,j} = V_{i,k-1} + V_{k+1,j-1} + 1$$

generally, $\leftarrow$ Matched w/ arbitrary base

$$V_{i,j} = \max_{k} \left(V_{i,k-1} + V_{k+1,j-1}\right) + 1$$

$k \in \{i, \ldots, j-5\}$

b_k can match with b_j

$$V_{i,j} = \max\left(V_{i,j-1}, \max_{k \in M_{i,j}} \left(V_{i,k-1} + V_{k+1,j-1}\right) + 1\right)$$

Base Cases

$V_{i,j} = 0$ whenever $j \le i+4$

Table-filling

Initialize $V[1,\dots,n][1,\dots,n]$ with base cases filled $O(n^2)$

For $i = n$ to 1 $n \cdot n$

For $j = i+5$ to n: $O(n)$

best-match $\leftarrow \max\limits_{k \in M_{i,j}} \left(V[i,k-1] + V[k+1,j] \right) + 1$

$V[i,j]$ $\max(V[i,j-1], \text{best-match})$ $O(1)$

Return $V[1,n]$

Total Runtime: $O(n^3)$

Dynamic Programming on Graphs

DAG is a directed graph w/ no cycles

Node w/ no incoming edges is called a source node

Node w/ no outgoing edges is a sink node

Every DAG has at least one source and one sink

Every DAG has a topological ordering that can be found in $O(m+n)$

Shortest Paths in DAGs

Input: A weighted DAG and a source node s

Output: A table of distances from s to each node in V

Recall that we define $\text{pred}(v) = \{u \in V, (u,v) \in E\}$

We can find $d(v)$ by $\min\{d(u) + u\}$

Recurrence

If we know $d(v_1, \dots, v_i)$

Can compute

switch to max for longest path

$$d(v_{i+1}) = \min_{u \in \text{Pred}(v_{i+1})} \left(d(u) + w(u,v) \right)$$

Base Case: $d(s) = 0$

Lecture 20

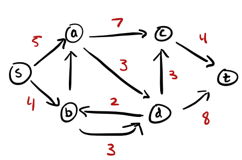
Flow Networks

- Directed graph w/

source node $s \in V$

sink node $t \in V$

Capacity $c(e)$ for each $e \in E$



We define a flow as an assignment of $f(e)$ to each $e \in E$ s.t.

- Non-negative: $\forall e \in E$: $f(e) > 0$
- Capacity: $\forall e \in E$: $f(e) \le c(e)$
- Conservation: $\forall v \in V \setminus s,t$

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out } v} f(e)$$

Max Flow Problem

Input: A flow network: graph G with source s, sink t, and capacities $c(e)$

Output: a flow w/ maximum value

$$|f| = \sum_{e \text{ out } s} f(e) = \sum_{e \text{ in } t} f(e)$$

Conservation enforces

$$\underbrace{\cancel{[\text{out of } t]}}_{0} + [\text{out of } s] + \sum_{u \in V} [\text{flow out of } u] = \sum_{u \in V} [\text{flow into of } u] + \underbrace{\cancel{[\text{flow into } s]}}_{0} + [\text{flow into } t]$$

conservation

$$[\text{out of } s] = [\text{into } t]$$

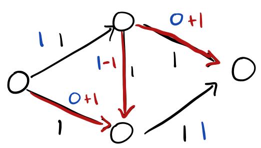
greatest volume to push from s to t

Any cut of the graph that splits s and t provides a flow bottleneck

total flow must cross this cut

Augmenting Paths - Backward Edges

remove flow in backwards edges



A path from s to t which may have backward edges

- room to add flow in forward edges
- flow to remove in backward edges

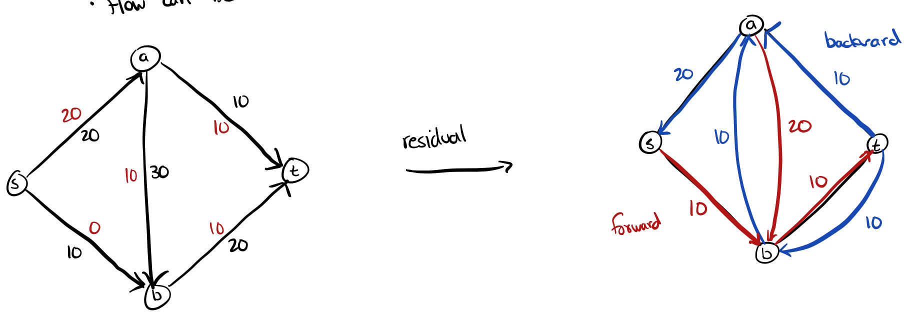
Augmenting path exists iff the flow is not max

Ford-Fulkerson Algorithm

Residual Graph

Given a graph and a flow, the residual graph contains an edge if

- flow can be added from a to b
- flow can be removed from b to a



Capacity of a forward edge is difference between capacity and flow

Capacity of a backward edge is how much flow could be removed just the flow

If both a forward and backward edge exist, capacity is given by the sum of both

Augmenting path is a path from s to t in the residual graph that can push the minimum edgeweight through

Ford-Fulkerson Algorithm

Input: Flow network G with source s, sink t, and edge capacities

Initialize f to trivial flow

Generate residual graph G_f with capacities c

Loop:

- Find path $P = e_1, \dots, e_k$ from s to t in G_f
- If no path exists, end loop
- Let $b \leftarrow \min(c_f(e_1), \dots, c_f(e_k))$
- Update f by pushing b units of flow through P
- Update G_f

Runtime:

If all edge weights are integers flow increases by ≥ 1 per augment

w/ linear time path finding (BFS, DFS)

$$O((n+m) \cdot F)$$

assume $m \geq \Omega(n)$

$$O(m \cdot F)$$ ← pseudopolynomial

Better algorithms can improve to $O(n \cdot m)$

Proof of Correctness

Feasibility: produces a valid flow

Optimality: produces maximum value flow

Arguing feasibility

Augmentation never reduces flow < 0, surpasses capacity, or violate conservation

$$\left.\begin{array}{ll} \text{case 1:} & \to \circ \to \\ \text{case 2:} & \to \circ \leftarrow \\ \text{case 3:} & \leftarrow \circ \to \\ \text{case 4:} & \leftarrow \circ \leftarrow \end{array}\right\} \text{Each case maintains conservation}$$

Optimality

s-t cut is a cut that splits s and t

Capacity of the cut is

$$\sum_{u \in A, v \in B} c(u,v)$$ ← don't consider backward edges

Theorem: given a flow network G

For any flow f and s-t cut (A,B) in G

$$|f| \leq c(A,B)$$

capacity is upperbound of flow

Cut Flow Lemma

For any s-t cut, the flow from s to t is the flow from A that doesn't come back

$|f| = [\text{flow out of } A] - [\text{flow into } A]$

$$|f| = \sum_{\substack{e \text{ from } v \\ \text{in } A}} f(e) - \sum_{\substack{e \text{ into } v \\ \text{in } A}} f(e)$$

$$= \sum_{v \in A} \sum_{e \text{ from } v} f(e) - \sum_{v \in A} \sum_{e \text{ into } v} f(e)$$

$$= \sum_{v \in A} \left(\sum_{e \text{ from } v} f(e) - \sum_{e \text{ into } v} f(e) \right)$$

$$= \sum_{e \text{ from } s} f(e) - 0$$

$$= |f|$$

We now see that

$$|f| \leq [\text{flow out of } A]$$
$$\leq [\text{Capacity of } A]$$
$$\leq c(A,B)$$

Lecture 21

Flow Networks Cont.

Find a cut (A,B) s.t. $|f| = c(A,B)$ ← Saturates upperbound

When FF algo is finished there is a set of vertices reachable from s and a set unreachable from s

note that $t \in$ unreachable set

FF gives us an s-t cut!

Spoiler: $|f| = c(A,B)$

Every edge from A to B is at capacity ← otherwise forward edge would exist in residual

Every edge from B to A is empty ← otherwise backward edge would exist in residual

Recall the lemma

$|f| = [\text{flow out of } A] - [\text{flow into } A]$

$= c(A,B) - 0$

Corollary: Min-Cut

$|f|$ is a lower bound on the capacity of any cut

Theorem

Given G, the minimum capacity over all s-t cuts in G is the value of max flow of G

Min-Cut Problem

Input: A network G

Output: An s-t cut with minimum capacity

Solution:

1. Compute max flow f and residual graph G_f
2. Let A be vertices reachable by s, and B everything else

Same proof of correctness

Runtime: $O(\underbrace{M(n,m)}_{\text{FF runtime}} + n+m) \simeq O(M(n,m))$

Useful to find bottlenecks in supply chains

Maximum Bipartite Matching

Bipartite Graphs

A bipartite graph is a graph $G = (V,E)$ where

- The vertices can be partitioned into two sets s.t.

$X,Y \in L: \{X,Y\} \notin E$
$X,Y \in R: \{X,Y\} \notin E$

Graph Matching

$M \subseteq E$ is a matching if no two edges in M share a vertex

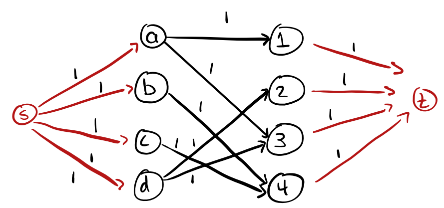
"no vertex is selected twice"

Maximum Bipartite Matching Problem

Input: Bipartite graph

Output: Matching $M \subseteq E$ that maximizes $|M|$

Add a source node connecting to the L set
Add a sink node connecting to the R set



Argue flow is at least as good as matching

Easy to find a flow for a matching ← send across matching edges

Argue matching is at least as good as flow

Easy to find matching by selecting flow edges ← force 0,1 flow and select edges w/ 1 unit flow

If all the capacities in a network are integers, then there is a max-flow on each edge as an integer

FF gives such a flow

Runtime reduces to $O(m \cdot n)$

total flow is # of vertices

Lecture 22

P vs. NP

Efficiently solvable vs. Efficiently Verifiable

Polynomial $\simeq$ Efficient

Every input has some length n

Polynomial run time means polynomial in number of bits
$O(n^c)$

Takes n^2 bits to represent a graph with n nodes and m edges (Adjacency matrix)

$n^c = N^{c/2}$ ← anything polynomial in n is polynomial in N

$m \le N$

$n^c \cdot m^c \le N^{c/2} \cdot N^c \le N^{2c}$

In an adjacency list it takes $O(\log n)$ to write each vertex and each edge

$N = \Theta\left((n+m) \log n\right)$

$n^c \cdot m^c \le N^c \cdot N^c \le N^{2c}$

Search and Decision Problems

A search problem is one where given x you are asked to find y that satisfies some conditions

Decision problems require a binary response

For any search problem there is a corresponding decision problem

The Class P

The class of decision problems that can be solved in polynomial time

Verification can be easy even if finding the solution is difficult

A verifier takes in an additional input c, the certificate, w/ the goal of convincing the verifier that the answer is yes

witness or proof

- Some certificate value should be yes
- Every incorrect certificate is always no

Can't give a false positive

Efficient Verifier

1. Only allowed to recieve certificates of polynomial length
2. Runs in polynomial time

NP is the class of problems that are efficiently verifiable

P vs. NP

If P = NP, every problem with efficient verifier has a poly-time algorithm

P ≠ NP some efficiently verifiable problems do not have poly-time algorithms

$P \subseteq NP$

Lots of practical problems are NP problems

No known proof P ≠ NP but overwhelming belief in CS community

Lecture 23

Polynomial Time Many-One Reduction

A function f which maps yes for A to yes for B
no for A to no for B

No bijections or any of that stuff

Runs in poly-time

Reduce problem A to problem B

Example: Independent set → Clique Problem

(no edges) (maximal edges)

given a graph and integer k, does G have a independent set or clique of size ≥ k

Toggle edges : remove all edges and add missing edges to same vertex set

Reduction Requirements

1. Polynomial time $O(n^2)$
2. Yes → Yes
3. No → No → argue through contrapositive

$\text{INDSET} \leq_p \text{CLIQUE}$

In general if $A \leq_p B$ and B can be solved in poly-time

A can be solved poly-time

NP Hardness

If we can find a problem H so that every problem in NP reduces to H

Then showing $H \in P$ would show that $P = NP$

reduce problem to H + poly-time algorithm

A problem is NP-hard if

$\forall A \in NP : A \leq_p H$

NP Completeness

Is there a problem in NP which is NP-hard?

Hardest problem in NP

If C is NP complete, then resolving whether or not $C \in P$ resolves P vs. NP

We have come across a handful of NP-complete problems

Proving NP-Hard

Reductions are transitive

If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$

We can show a problem is NP complete by showing its NP and then showing a NP complete problem B reduces to it

3-SAT problem

Canonical NP-complete problem

A boolean formula has variables

We make a formula by joining these variables w/ logical operations

Formula is in conjunctive normal form, if it is written as AND of clauses (variables, ors, + negations)

CNF

3-CNF formula has 3 literals in each clause

A formula is satisfiable if you can assign variables s.t. the formula is true

---

Input: A 3-CNF problem

Output: Yes if φ is satisfiable and no otherwise

3-SAT to INDSET

Label the formula as follows

$C_i = (l_i^1 \vee l_i^2 \vee l_i^3)$

A graph G with 3m verticies for each literal

Choosing l_i^j to be in ind set is equivalent to setting it true

Add edges if literals are negations of eachother

Add edges in every clause

Allows us to select m verticies (1 per clause)

Argue

1. Polynomial Time $O(m^2)$
2. φ satisfiable then G has ind set
3. φ is unsatisfiable then G has no ind set

Contrapositive: G has ind set of size m, then φ is satisfiable

2. Select one true literal per clause

3. Oops skipped

Lecture 24

NP-complete Problems

- Set packing problem

  Input: List U of elements, list of sets $S_1, \dots, S_n$ (Subsets of U) and int k

  Decision: Is there a collection of k disjoint subsets from $S_1 \dots S_n$?

  NP-complete: reduction from INDSET

- Set covering problem

  Input: List U of elements, list of sets $S_1, \dots, S_n$ (Subsets of U) and int k

  Decision: Is there a collection of k elements $x_1 \dots x_k \in U$ s.t.

  $$\forall i : S_i \cap \{x_1, \dots x_k\} \neq \emptyset$$

  NP-complete: reduction from Vertex Cover

- Subset sum problem

  Input: List of integers $A[1, \dots, n]$ and target integer t

  Decision: Is there a set of indices $x_1 \dots x_k$ s.t.

  $$\sum_{i=1}^{k} A[x_i] = t$$

  NP-complete: Reduction from VC

- Partition Problem

  Input: List of integers $A[1, \dots, n]$

  Decision: Is there a partition of indices $\{1, \dots, n\}$ into L, R s.t.

  $$\sum_{i \in L} A[i] = \sum_{i \in R} A[i]$$

  NP-complete: Reduction from subset sum

- Graph Coloring

  Input: Graph G and integer k

  Decision: Is there a proper coloring of vertices of G using k colors

  NP-complete: 3-Coloring
  ↓
  3-SAT reduction

Path Problems

Negative edge weights provide issues

Bellman-Ford Algorithm is a DP algo that detects negative weight cycles

Floyd-Warshall Algorithm finds the shortest path between every pair of verticies

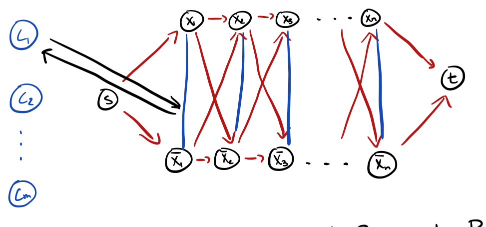
Hamiltonian Path

Reducing 3-SAT to Hamiltonian Path

3-CNF —> G w/ path through every vertex

If we visit x_i before $\bar{x}_i$ we assign x_i to be true

Clause Verticies



If x_i occurs positively we add L-R and R->L if negatively

1. Make graph in poly time ✓
2. If φ is satisfiable then
3. ~~Reverse~~
   Add buffer nodes