Algorithm Design & Analysis (CSE222)

Lecture-1

Outline

- General info & Policies
- Course Outline
- Introduction
  - What are algorithms
  - Examples

General info & Policies

Lecture Days: Monday and Wednesday @C201

Time: 2pm to 330pm

Tutorials: Tuesdays 1:30pm to 3pm. (Tomorrow no tutorials)

General info & Policies

Instructor: Supratim Shit

Office Hours: By appointment, Tuesday 10-12 at B512, R & D Block

Always send calendar invite (to supratim@...) and book your appointment.

Teaching Fellow (TF): To be announced.

Teaching Assistants (TA): To be announced.
Every students will be assigned to one TA for doubt clarification.

General info & Policies

Addressing queries/doubts:

Your Friends/Peers ⇒ Your TA ⇒ Other TAs ⇒ TF ⇒ me.

General info & Policies

Course Page: bit.ly/ADA-b

Class code: ubidmle

All announcements will be posted in google classroom.

All evaluation components are to be submitted and graded in this google classroom.

Do not post your solutions in public.

General info & Policies

Evaluation Scheme

- Attendance: 5%
- Quiz: 15% (n-1)
- Assignments:
  - Theory: 15% (Only pdf submissions are allowed. Latex template will be shared soon)
  - Programming: 10% (Platform to be announced soon. Programming language should be C++/Java, not Python)
- Mid Sem: 25%
- End Sem: 30%

General info & Policies

Evaluation Type
Quizzes: Short and surprise (n-1)
Mid Sem & End Sem: Proctored offline

No Re-Quiz at any ground.

Make-up exam only for Mid-sem and End-sem with approved medical leave.

Note: Questions need not be same for both sections.

Assignments

- Theory
  - You can submit it in a group of 2 students. Mention your group members and make sure to only submit one assignment file. You may discuss the problem with your friends (other than group member) but write it in your own words and mention your collaborators name. If you have referred to online sources then mention the URL.
- Programming
  - Submit your assignments individually.

Anti-Plagiarism

- Assignment
  - First Instance: 0 in the assignment & one letter grade less.
  - Next Instance: Fail (F-grade)

- Quizzes/Exams
  - First Instance: Fail (F-grade)

Late Submission

- Assignments
  - Theory: 1 day late ⇒ 25% deduction. 2 days late ⇒ 50% deduction. 3 or more days ⇒ 100%.
  - Programming: 1 day late ⇒ 100% deduction.
- General: Always submit your assignments at least an hour before the deadline. Do not submit in the last hour of the deadline. Any GC system related issues due to last hour submission will not be considered.
- Only pdf format either from MS word, G-doc or Latex will be accepted.

General Decoram

Be polite while communicating with others.

Keep checking google classrooms for updates

Do not spam in the google classroom.

In case of discrepancy, be discreet with your arguments. Vague or irresponsible arguments will not be entertained and it can lead to reevaluation and subsequently reduction in marks.

If attendance falls below 40% then faculty may opt, not to put up the scribble in the google classroom.

Outline

- General info & Policies
- Course Outline
- Introduction
  - What are algorithms
  - Examples

Course Outline

Introduction, Recurrence relations

Divide & Conquer

Dynamic Programming

Greedy Methods

Graph Algorithms: DFS and its applications, shortest path, spanning tree

Network Flow

NP-Completeness

Outline

- General info & Policies
- Course Outline
- Introduction
  - What are algorithms
  - Examples

Algorithms

Step of process needs to carried out to solve a problem.

Sequence of instructions that acts on an input (or values) to produce an output in finite number of steps.

Input: May or may not take input. If it takes input then it must be well defined.
Output: Must produce at least one well defined output.
Finiteness: An algorithm must terminate after finite number of steps.
Definiteness: The instructions must be unambiguous. At every step there has to be one and only possible outcome.
Effectiveness: One should be able to follow each and every instructions to solve the problem.

Algorithms

Mathematics: Solving a system of linear equations

Artificial Intelligence: Used in developing intelligent systems image recognition, voice recognition etc.

Data Science: Analyze and extract insights from large datasets.

Examples

Where have you seen algorithms?

Coming to class

Finding a book in the library

Following recipes

First Algorithm

Finding largest and the smallest element in the following array A.

A =

3	8	2	6	9	4	5
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Let **Largest** = **Smallest** = **A[1]**

Go over every element in A and compare each with **Largest** and **Smallest**.
Update **Largest** if an item is bigger than **Largest**.
Update **Smallest** if an item is smaller than **Smallest**.

Is this correct? Why?

Pseudo Code - Find Max and Min

Input: An array **A**, Size(**A**)

Output: **Largest**, **Smallest**

```
FIND-MINMAX(A, n)
i = 1
Largest = Smallest = A[i]
i = i + 1
while i < n
    if A[i] > Largest
        Largest = A[i]
    if A[i] < Smallest
        Smallest = A[i]
    i = i + 1
```

Find Max and Min

Input: An array **A**, Size(**A**)

Output: **Largest**, **Smallest**

$\textsc{Find-MinMax}(\mathbf{A}, n)$

```
i = 1
Largest = Smallest = A[i]
i = i + 1
while i < n
    if A[i] > Largest
        Largest = A[i]
    if A[i] < Smallest
        Smallest = A[i]
    i = i + 1
```

Effectiveness

3	8	2	6	9	4	5
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Swap Numbers

Let **a** and **b** be two numbers. Swap the values of **a** and **b**.

Input: **a**, **b**

Output: **a**, **b** (values swapped)

$\textsc{Swap}(\mathbf{a}, \mathbf{b})$

$$
\begin{aligned}
&1 \quad temp = \mathbf{a} \\
&2 \quad \mathbf{a} = \mathbf{b} \\
&3 \quad \mathbf{b} = temp
\end{aligned}
$$

Effectiveness

5	8

Swap Numbers

Let **a** and **b** be two numbers. Swap the values of **a** and **b**.

Input: **a**, **b**

Output: **a**, **b** (values swapped)

$\textsc{Swap}(\mathbf{a}, \mathbf{b})$

$temp = \mathbf{a}$

$\mathbf{a} = \mathbf{b}$

$\mathbf{b} = temp$

Can you do better?

Effectiveness

5	8

Swap Numbers

Let **a** and **b** be two numbers. Swap the values of **a** and **b**.

Input: **a**, **b**
Output: **a**, **b** (values swapped)

$\textsc{Swap-Efficient}(\mathbf{a}, \mathbf{b})$

$$\begin{aligned} &\mathbf{a} = \mathbf{a} + \mathbf{b} \\ &\mathbf{b} = \mathbf{a} - \mathbf{b} \\ &\mathbf{a} = \mathbf{a} - \mathbf{b} \end{aligned}$$

Effectiveness

5	8

Analysis

Analysis of an ‘algorithm’ is done with respect to

Space: The total memory required to run the algorithm.

Time: The total time taken by the algorithm to return the output.

Usually, how these two changes with change in input size.

Running time depends on CPU!!

Analyze the asymptotic running time and working space.
Growth of function with respect to input size.
(To what values the running time and the working space will tend to for large input size, while disregarding the system’s configuration.)

Growth of Function

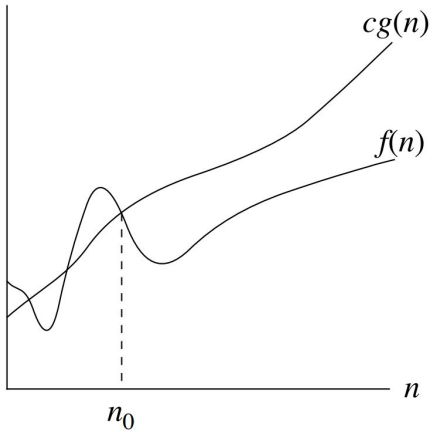
Notations

O	$\approx$	$\leq$
Ω	$\approx$	$\geq$
Θ	$\approx$	$=$
o	$\approx$	$<$
ω	$\approx$	$>$

Growth of Function

O-notation

O(g(n)) = {f(n): there exists positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$}.



g(n) is an asymptotic upper bound for f(n).
So, f(n) = *O*(g(n)).

Growth of Function

O-notation

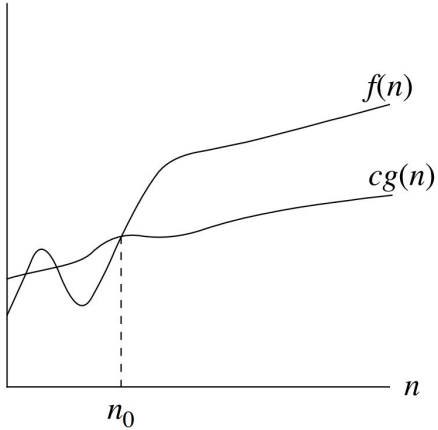
What is the relationship between f(n) and g(n) for the following functions.

- $f(n) = n^2$ and $g(n) = n$
- $f(n) = n^{1/2}$ and $g(n) = n$
- $f(n) = n^{1/5}$ and $g(n) = \log(n)$
- $f(n) = n^{10}$ and $g(n) = 2^n$

Growth of Function

Ω-notation

$\Omega(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0\}$.



g(n) is an asymptotic lower bound for f(n).
So, $f(n) = \Omega(g(n))$.

Growth of Function

Ω-notation

What is the relationship between f(n) and g(n) for the following functions.

- $f(n) = n^{1/2}$ and $g(n) = n$
- $f(n) = n^{-1}$ and $g(n) = \log(n)$
- $f(n) = n^3$ and $g(n) = 10^n$

Reference

Slides

Introduction to Algorithms by CLRS - Chp-3

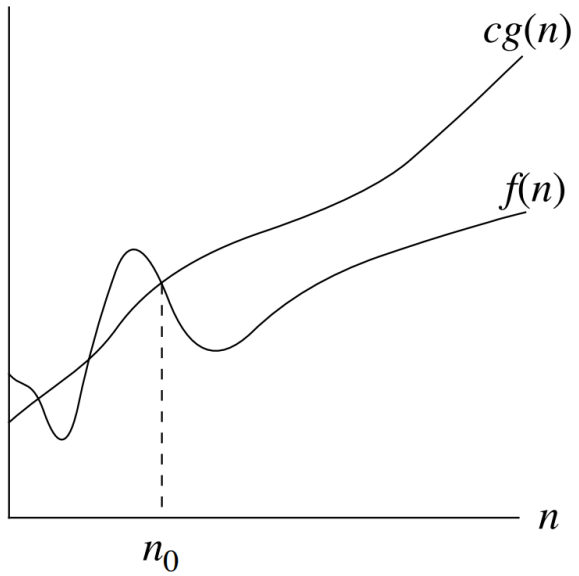
Algorithm Design & Analysis (CSE222)

Lecture-2

Recap

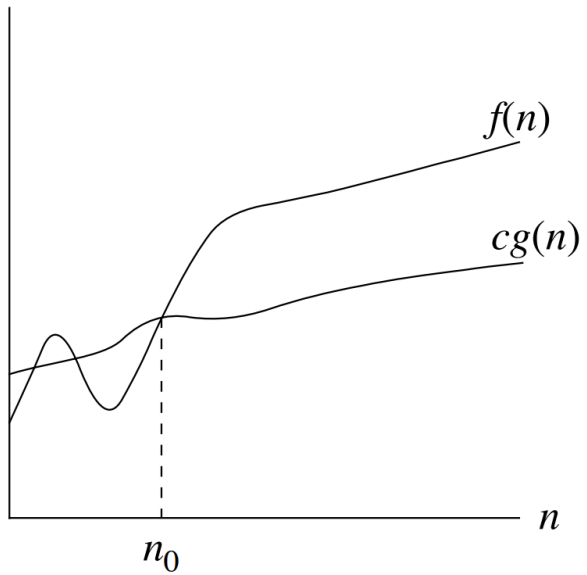
Characteristics of Algorithm: Well defined Input, Output, Finiteness, Definiteness, Effectiveness.

Growth Functions: *O*-notation



$O(g(n))$ = {f(n): there exists positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$}.

Ω-notation



$\Omega(g(n))$ = {f(n): there exists positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0$}.

Outline

- Growth Functions (cont.)
- Graph Notations
- Revisit BFS

Growth of Function

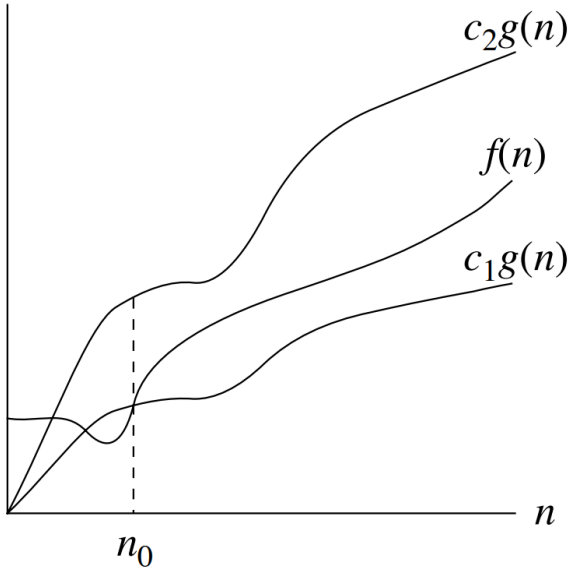
For an input of size n, let the running time of your algorithm is f(n). Then which of the following are true?

- $f(n) = n^2 + 10^{10} \cdot n$
  - $f(n) = O(n)$?
  - $f(n) = O(n^2)$?
- $f(n) = 10^{-10} \cdot n^2 + n$
  - $f(n) = \Omega(n)$?
  - $f(n) = \Omega(n^2)$?

Growth of Function

Θ-notation

$\Theta(g(n)) = \{f(n)$: there exists positive constants c_1, c_2 and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$.



g(n) is an asymptotic tight bound for f(n).
So, $f(n) = \Theta(g(n))$.

Growth of Function

Θ-notation

Can find a relation between the following functions using Θ-notation.

- $f(n) = n^{1/2}$ and $g(n) = n$
- $f(n) = n^{-1}$ and $g(n) = \log(n)$
- $f(n) = n^{1.000000001}$ and $g(n) = n$
- $f(n) = 3n + 4$ and $g(n) = 2n - 1$
- $f(n) = 10^{-10} \cdot n^2 + n$ and $g(n) = n^2 + 10^{10} \cdot n$

Growth of Function

o-notation

$o(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0\}$.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

What is the relationship between f(n) and g(n) for the following functions.

- $f(n) = n$ and $g(n) = n^{1/2}$
- $f(n) = n \cdot \log(n)$ and $g(n) = n^{3/2}$
- $f(n) = n$ and $g(n) = n^2 + n$
- $f(n) = n^2$ and $g(n) = n^2 + n$

Growth of Function

ω-notation

$\omega(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0\}$.

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

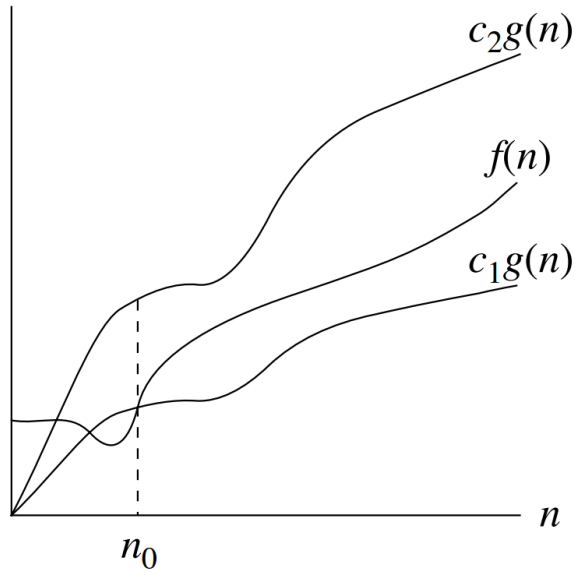
What is the relationship between f(n) and g(n) for the following functions.

- $f(n) = n$ and $g(n) = n^{1/2}$
- $f(n) = n$ and $g(n) = n^2 - n$
- $f(n) = n^2$ and $g(n) = n^2 + n^{-1}$

Growth of Function

Θ-notation

$\Theta(g(n)) = \{f(n)$: there exists positive constants c_1, c_2 and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$.



$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

Relational Properties

<u>Transitivity</u>
If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
This also holds for O, Ω, o and ω

<u>Reflexivity</u>
$f(n) = \Theta(f(n))$
Also holds for O and Ω

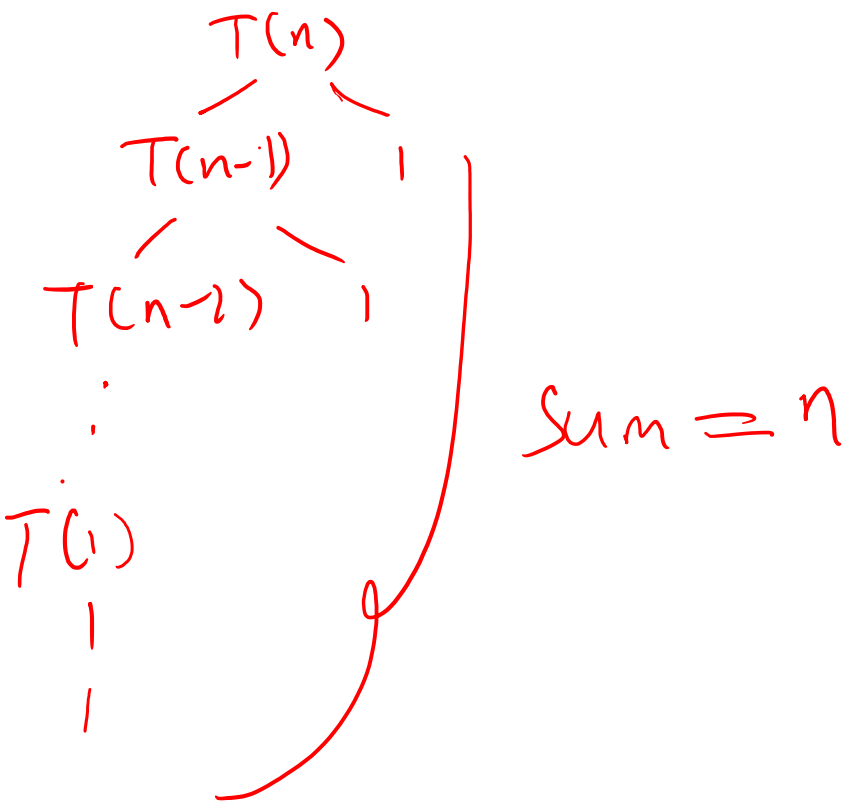
<u>Symmetry</u>
$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

<u>Transpose</u>
$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Recurrences

Recurrences are functions with one or more base cases and itself with smaller arguments.

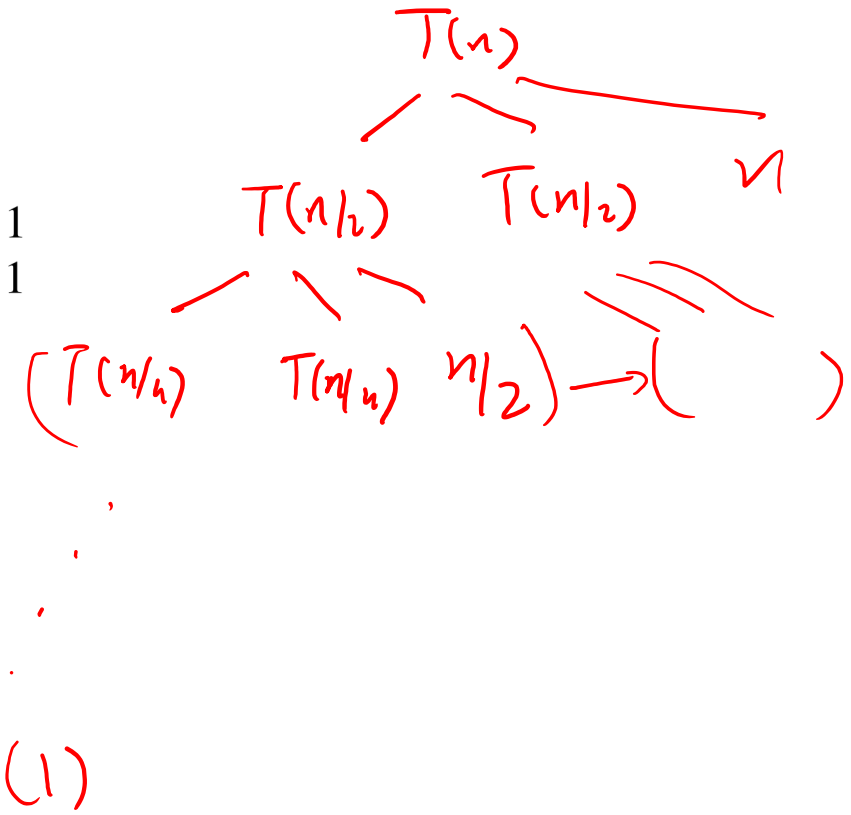
$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1 \end{cases}$$



sum = n

Recurrences

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n \geq 1 \end{cases}$$



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

Master's Theorem

$T(n) = 9T(n/3) + n$, $T(n) = \theta(n^2)$

$T(n) = T(3n/5) + 1$

$a = 1$, $b = \frac{5}{3}$, $f(n) = 1$

$T(n) = aT(n/b) + f(n)$ be defined on a non-negative integer, where,
a ≥ 1 is a constant (# subproblems),
b > 1 is a constant (such that n/b is input size to each subproblem) and
f(n) is a polynomial function (cost on each recursive call).

T(n) has following asymptotic bound for a constant ε > 0.

- If $f(n) = O(n^{\log_b a - \varepsilon})$, then $T(n) = \theta(n^{\log_b a})$.
- If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \log_e n)$.
- If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $af(n/b) < cf(n)$ for some constant $c < 1$ and for all sufficiently large n then $T(n) = \theta(f(n))$.

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1, b \geq 2$, and c and k are positive constants, is

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

Example

$T(n) = 9T(n/3) + n$

$T(n) = T(3n/5) + 1$

$T(n) = 2T(n/4) + n$

Recursive equation should be in form below

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

Where $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real no.

1) If $a > b^k$, then $T(n) = \Theta(n\log_b a)$
2) If $a = b^k$ then
   a. If $p > -1$, then $T(n) = \Theta(n\log_b a \; log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n\log_b a \; loglogn)$
   c. If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$
3) If $a < b^k$ then
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

Outline

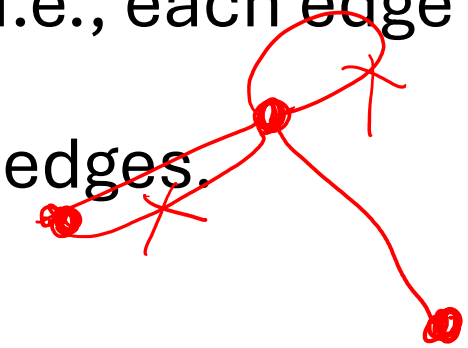
- Growth Functions (cont.)
- Graph Notations
- Revisit BFS

Graph Notations

Graphs: Captures relationship (a.k.a edges) between objects (a.k.a nodes).
Consists of collection of nodes (a.k.a vertices) V and a collection of edges E (sets of nodes).
$G = (V, E)$

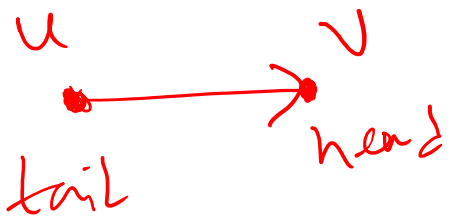
<u>Simple Graphs</u>

- At most one edge between pair of vertices (i.e., each edge is a pair of nodes).
- Undirected (or symmetric) and unweighted edges.
- No self loops.



Graph Notations

Directed graph for asymmetric relationship, $G = (V, E)$. Every $e \in E$ is an ordered pair (u, v).



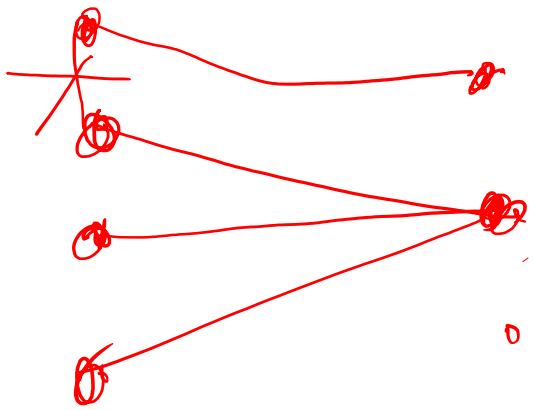
(u,v) ✓

(v,u) ✗

(u,v) or (v,u)

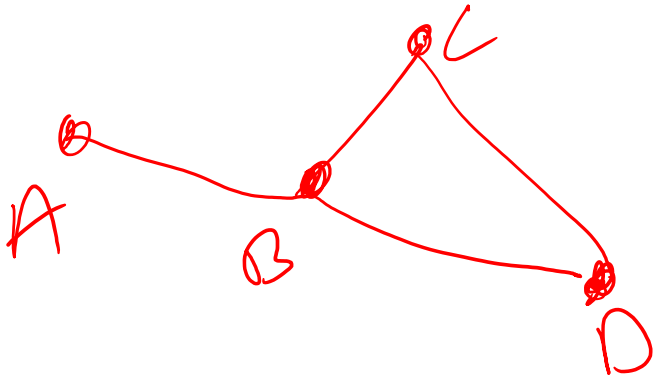
Graph Notations

Bipartite Graph: Vertex set consists of two disjoint sets of vertices, such that every edge is a pair of one vertex from each disjoint sets.



Graph Notations

In an undirected graph $G = (V, E)$, a **<u>path</u>** p is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that each pair of consecutive vertices (v_i, v_{i+1}) is an element in E. That is they are connected by an edge.



A, B, D ✓

(A, B), (B, D)

A, B, C ✓

ACB X

Graph Notations

In an undirected graph $G = (V, E)$, a **<u>path</u>** p is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that each pair of consecutive vertices (v_i, v_{i+1}) is an element in E. That is they are connected by an edge.

A **<u>simple path</u>** is a sequence of vertices, where none of them are repeated.

A **<u>cycle</u>** is a path if all the vertices are unique except for the first and the last.

A graph is called **<u>connected</u>** if there is a path between every pair of vertices.

Graph Notations

In an undirected graph $G = (V, E)$, a **path** p is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that each pair of consecutive vertices (v_i, v_{i+1}) is an element in E. That is they are connected by an edge.

A **simple path** is a sequence of vertices, where none of them are repeated.

A **cycle** is a path if all the vertices are unique except for the first and the last.

A graph is called **connected** if there is a path between every pair of vertices.

A **tree** is a connected graph without any cycle.

Outline

- Growth Functions (cont.)
- Graph Notations
- Revisit BFS

Revisit BFS

Breadth First Search (BFS): Starting from a vertex it traverses the graph layer by layer.

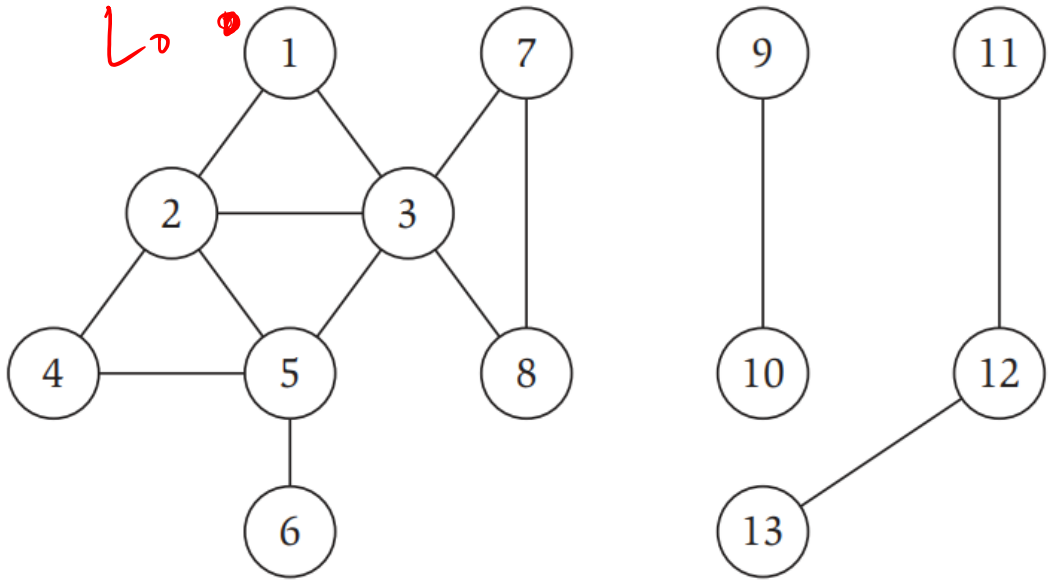
$L_0 = 1,$

$L_1 = 2, 3$

$L_2 = 4, 5, 8, 7$

$L_3 = 6$

L_4



Revisit BFS

<u>Applications</u>

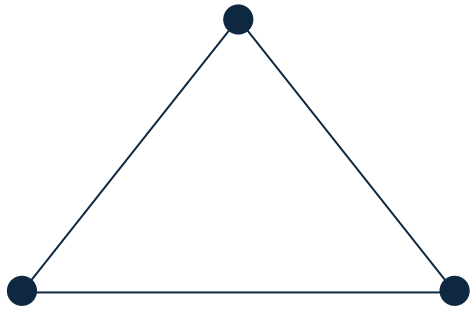
- Finds if two vertices are connected.
- Finds the shortest path between two connected vertices.

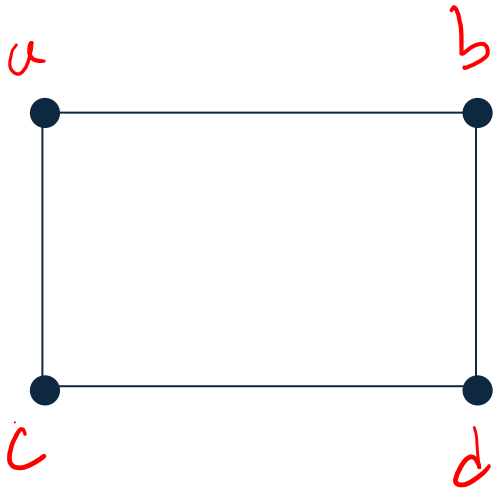
For $j \geq 1$, the layer L_j produced by BFS consists of all the nodes that are exactly at distance j from s (start node).

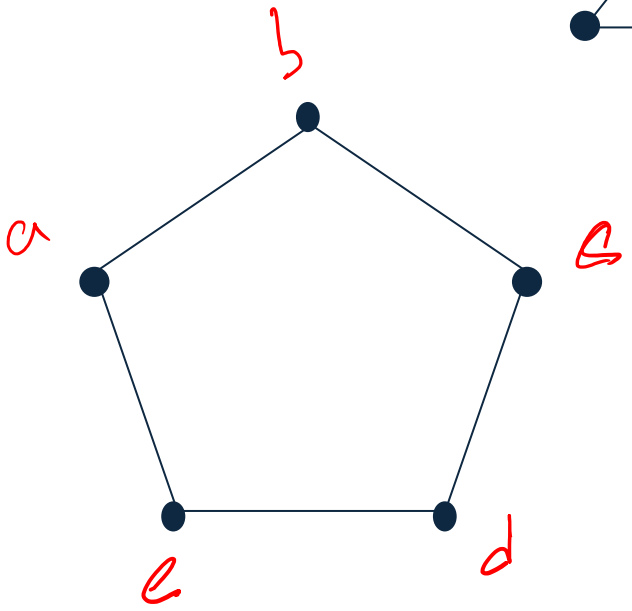
Running time of BFS?

Testing Bipartiteness using BFS

Is it a bipartite graph?



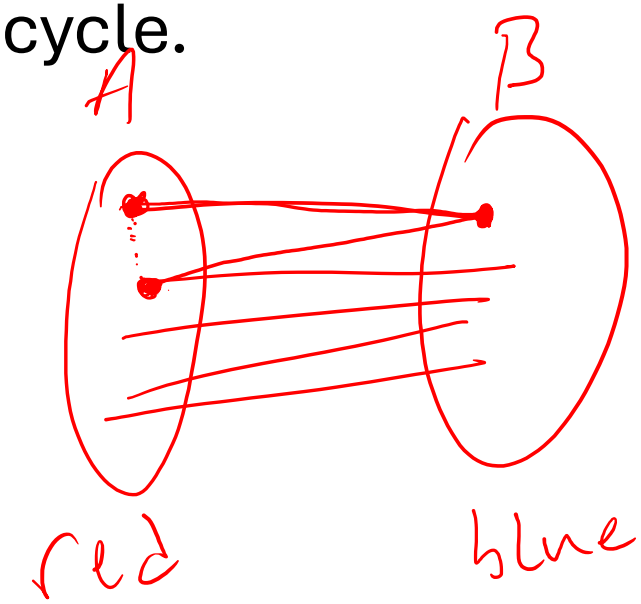




Testing Bipartiteness using BFS

If a graph G having odd length cycle cannot be a bipartite graph.

A bipartite graph cannot contain an odd length cycle.



Testing Bipartiteness using BFS

Start from any vertex and colour it red.

Go to its neighbours and colour them blue.

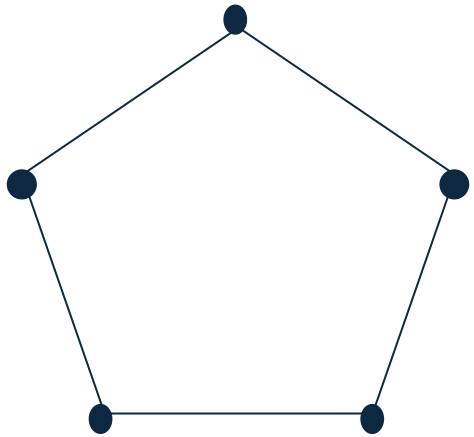
Go to their neighbours and colour them red.

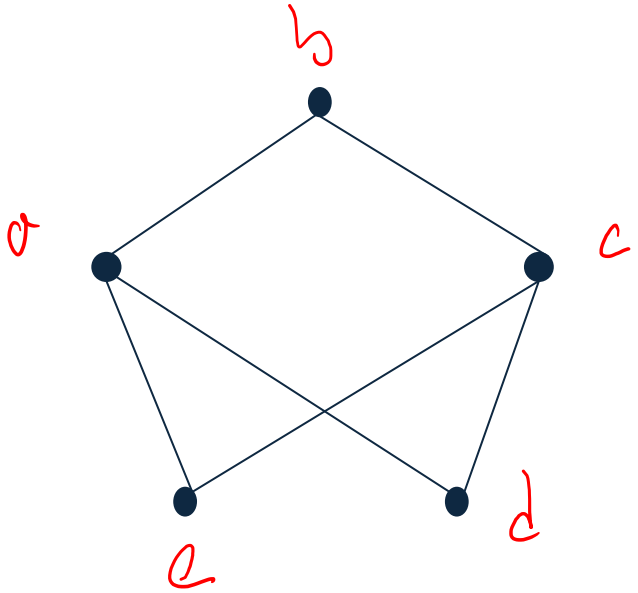
Continue until all the vertices are coloured.

If there is an edge whose vertex pairs are of same colour then the graph is not bipartite. Why?

<u>Think</u>: Let T be a BFS tree, let u and v be nodes in T belonging to layers L_i and L_j respectively, and let (u, v) be an edge of G. Then i and j differ by at most 1.

Testing Bipartiteness using BFS





Testing Bipartiteness using BFS

The bipartite testing algorithm is identical to BFS.

Colour all the vertices in the even numbered layers by red.

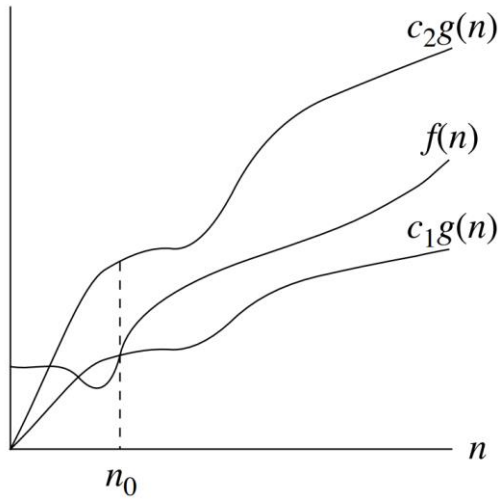
Colour all the vertices in the odd numbered layers by blue.

Algorithm Design & Analysis (CSE222)

Lecture-3

Recap

Growth Functions:



$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

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

f(n) = Θ(g(n)) f(n) = o(g(n)) f(n) = ω(g(n))

- Recurrences & Masters' Theorem
- Graph Notations
- BFS

Outline

- Revisit BFS: Bipartite Testing
- Divide & Conquer
- Counting Inversions

Revisit BFS

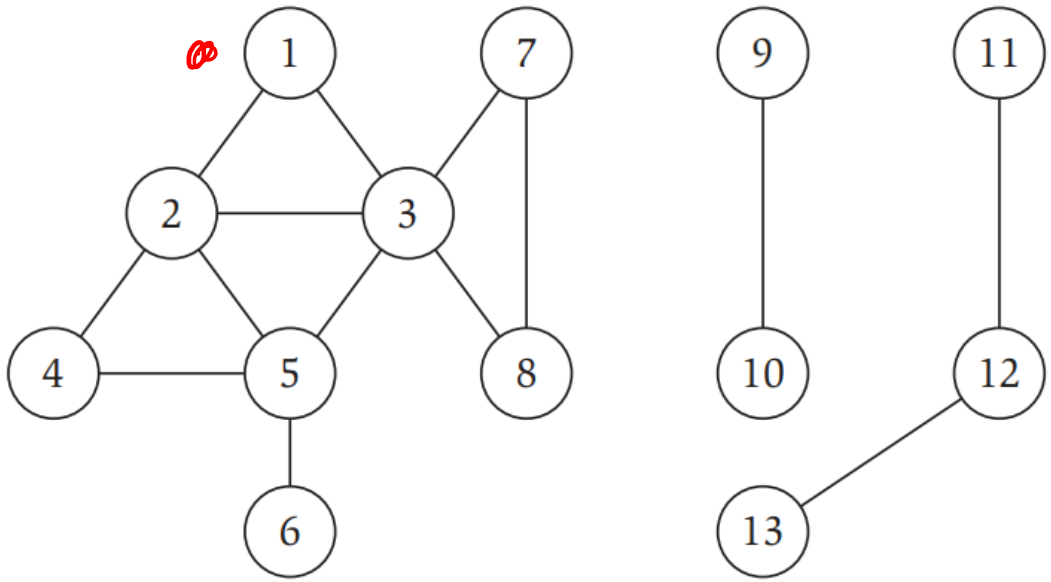
Breadth First Search (BFS): Starting from a vertex it traverses the graph layer by layer.

$L_0 = 1$

$L_1 = 2, 3.$

$L_2 = 4, 5, 8, 7$

$L_3 = 6$



Revisit BFS

<u>Applications</u>

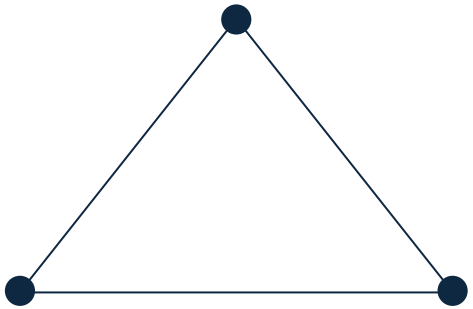
- Finds if two vertices are connected.
- Finds the shortest path between two connected vertices.

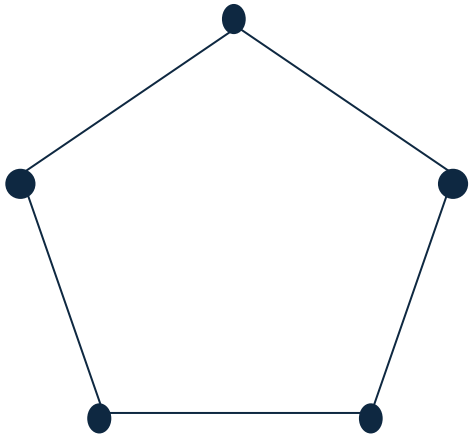
For $j \geq 1$, the layer L_j produced by BFS consists of all the nodes that are exactly at distance j from s (start node).

Running time of BFS? $O(|E| + |V|)$

Testing Bipartiteness using BFS

Is it a bipartite graph?





Testing Bipartiteness using BFS

A graph G having odd length cycle cannot be a bipartite graph.

A bipartite graph cannot contain an odd length cycle.

Testing Bipartiteness using BFS

Start from any vertex and colour it red.

Go to its neighbours and colour them blue.

Go to their neighbours and colour them red.

Continue until all the vertices are coloured.

If there is an edge whose vertex pairs are of same colour then the graph is not bipartite.

<u>Think</u>: Let T be a BFS tree, let u and v be nodes in T belonging to layers L_i and L_j respectively, and let (u, v) be an edge of G. Then i and j differ by at most 1.

Testing Bipartiteness using BFS

The bipartite testing algorithm is identical to BFS.

Colour all the vertices in the even numbered layers by red.

Colour all the vertices in the odd numbered layers by blue.

Claim of the Algorithm

Let G be a connected graph and let layers L_1, L_2, ... be produced by the BFS starting at vertex s.

1. If there is no edge joining two vertices of same layer then every edge has a vertex coloured red and another vertex coloured blue. Hence, G is a bipartite graph.
2. If there is an edge joining two vertices of the same layer then G contains odd length cycle. Hence, G is not a bipartite graph.

Analysis

1. If there is no edge joining two vertices of same layer then every edge has a vertex coloured red and another vertex coloured blue. Hence, G is a bipartite graph.

Recall: Let T be a BFS tree, let u and v be nodes in T belonging to layers L_i and L_j respectively, and let (u, v) be an edge of G. Then i and j differ by at most 1.

- This implies edges in G are either in same layer or adjacent layers.
- No edges in same layer (by assumption)
- So every edge joins two vertices in adjacent layers.
- So, in our colouring every edge has two vertices from different colours.

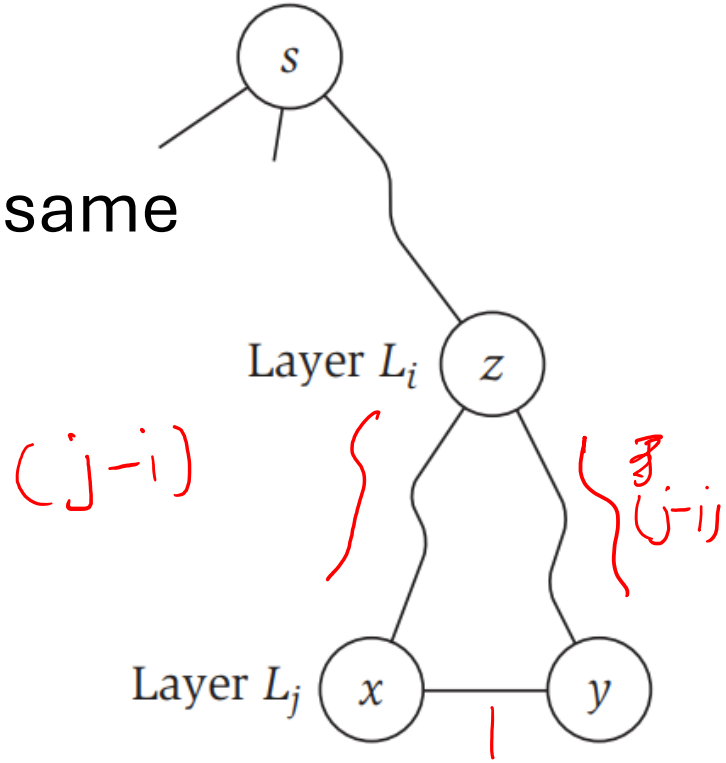
Analysis

2. If there is an edge joining two vertices of the same layer then *G* contains odd length cycle. Hence, G is not a bipartite graph.

As there is an edge between two vertices in the same layer. So consider this situation.

Now compute the length of the cycle (z,x,y,z).

$2(j-i)+1$



Outline

- Revisit BFS: Bipartite Testing
- Divide & Conquer
- Counting Inversions

Divide & Conquer

$T(n) = 3T(n/3) + n$

$T(1) = O(1)$

Divide & Conquer

$T(n) = 3T(n/3) + n$

$T(1) = O(1)$

- Divide the input into small pieces.
- Solve subproblems on these pieces separately by recursions.
  - Subproblems are the same problem solved on a smaller input size
- Combine results into overall solution.

Merge Sort

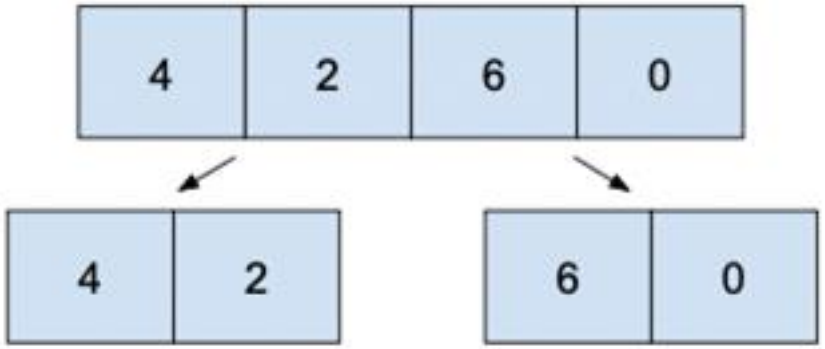
Input: Array ***A*** of n numbers.
Output: Sorted array of n numbers.

- <u>Divide</u> array into $\boldsymbol{A}_{\mathbf{Left}} = \boldsymbol{A}[\mathbf{1,2, \ldots, n/2}]$ and $\boldsymbol{A}_{\mathbf{Right}}. = \boldsymbol{A}[\mathbf{n/2+1, \ldots, n}]$ (Divide).
- <u>Recursively</u> sort $\boldsymbol{A}_{\mathbf{Left}}$ and $\boldsymbol{A}_{\mathbf{Right}}$.(Conquer).
- <u>Merge</u> two sorted arrays from $\boldsymbol{A}_{\mathbf{Left}}$ and $\boldsymbol{A}_{\mathbf{Right}}$ into one sorted array (Combine).

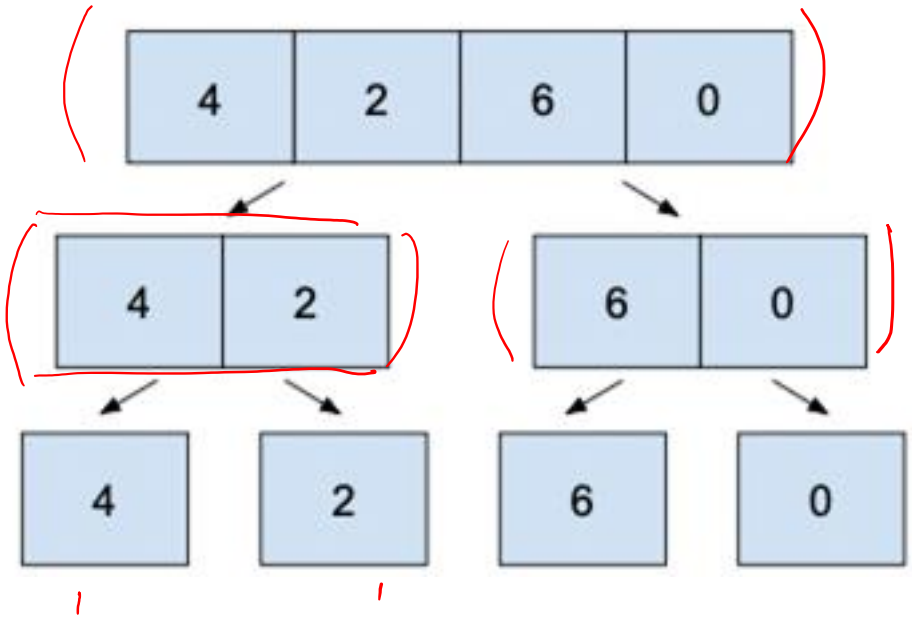
Merge Sort

4	2	6	0

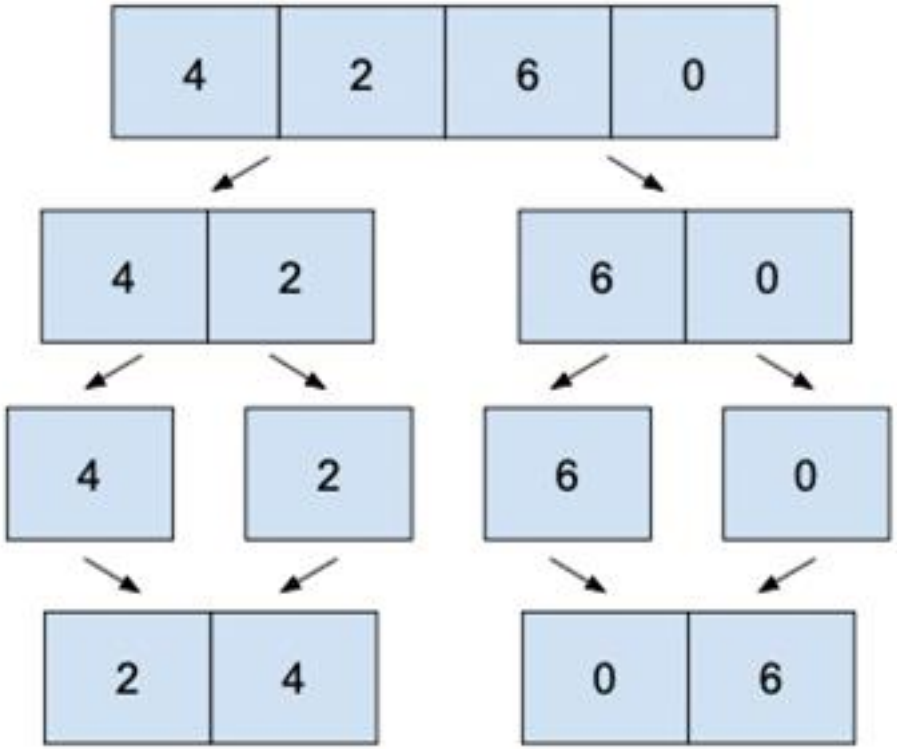
Merge Sort



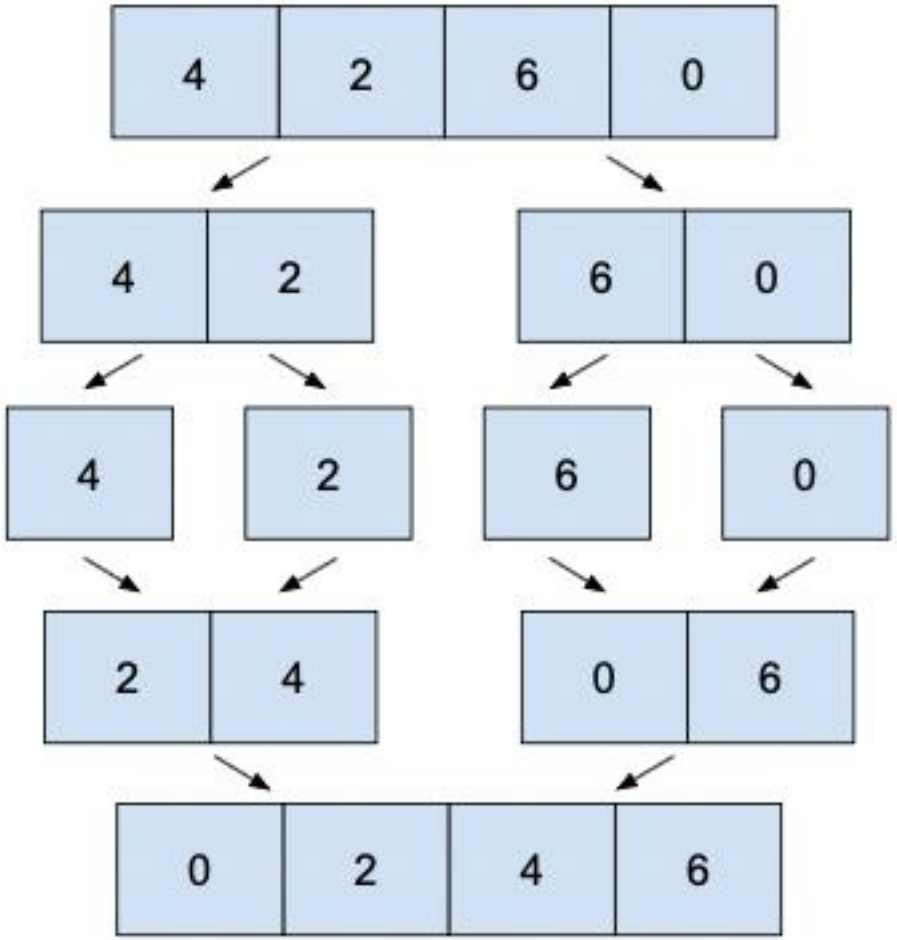
Merge Sort



Merge Sort



Merge Sort



Merge Sort

```
MERGE-SORT(A, ℓ, r)
if ℓ < r
    m = ⌈(ℓ + r)/2⌉
    MERGE-SORT(A, ℓ, m)
    MERGE-SORT(A, m + 1, r)
    MERGE(A, ℓ, m, r)
```

Outline

- Revisit BFS: Bipartite Testing
- Divide & Conquer
- Counting Inversions

Measure Similarity

$A = (1 \quad 3 \quad 2 \quad 6)$

$A(1) \quad A(2) \quad A(3) \quad A(4)$

- Let there are ***n*** songs.
- You rank them as ***1***, ***2***, ..., ***n***
- Your friend ranks them as $\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_n}$
- How to measure similarity?
- Compute the number of <u>*out of order*</u> pairs.
- A pair is out of order if $\boldsymbol{i} < \boldsymbol{j}$ but $\boldsymbol{a_i} > \boldsymbol{a_j}$ (a.k.a. <u>*Inversion*</u>).

$1 \quad 3 \quad 2 \quad 6$

$1 \quad 1$

$1 \quad 2 \quad 3 \quad 6$

Questions

- What is the number of inversions in a sorted (ascending order) array? 0
- What is the maximum possible inversions in a array of length n? Example? n(n-1)/2

Counting Inversions

Problem: In an array of ***n*** numbers count the number of inversions.

A Naive algorithm.

Input: Array ***A*** of ***n*** numbers.
Output: Number of inversion in ***A***.

$$\textsc{Simple-CountIversions}(A, n)$$

$$\begin{array}{ll}
1 & Count = 0 \\
2 & \textbf{for} \text{ every } (i, j) \in n^2 - n \\
3 & \qquad \textbf{if } \big(i > j \text{ and } A[i] < A[j]\big) \\
4 & \qquad\qquad Count = Count + 1
\end{array}$$

Running time?

Count Inversions in Sorted Arrays

$C = \cancel{0}\ 3 - 2 + 1 = \cancel{2}\ \cancel{4}\ 5$

Left

1	5	9

Right

2	4	7

1	2	4	5	7	9

Count Inversions in Sorted Arrays

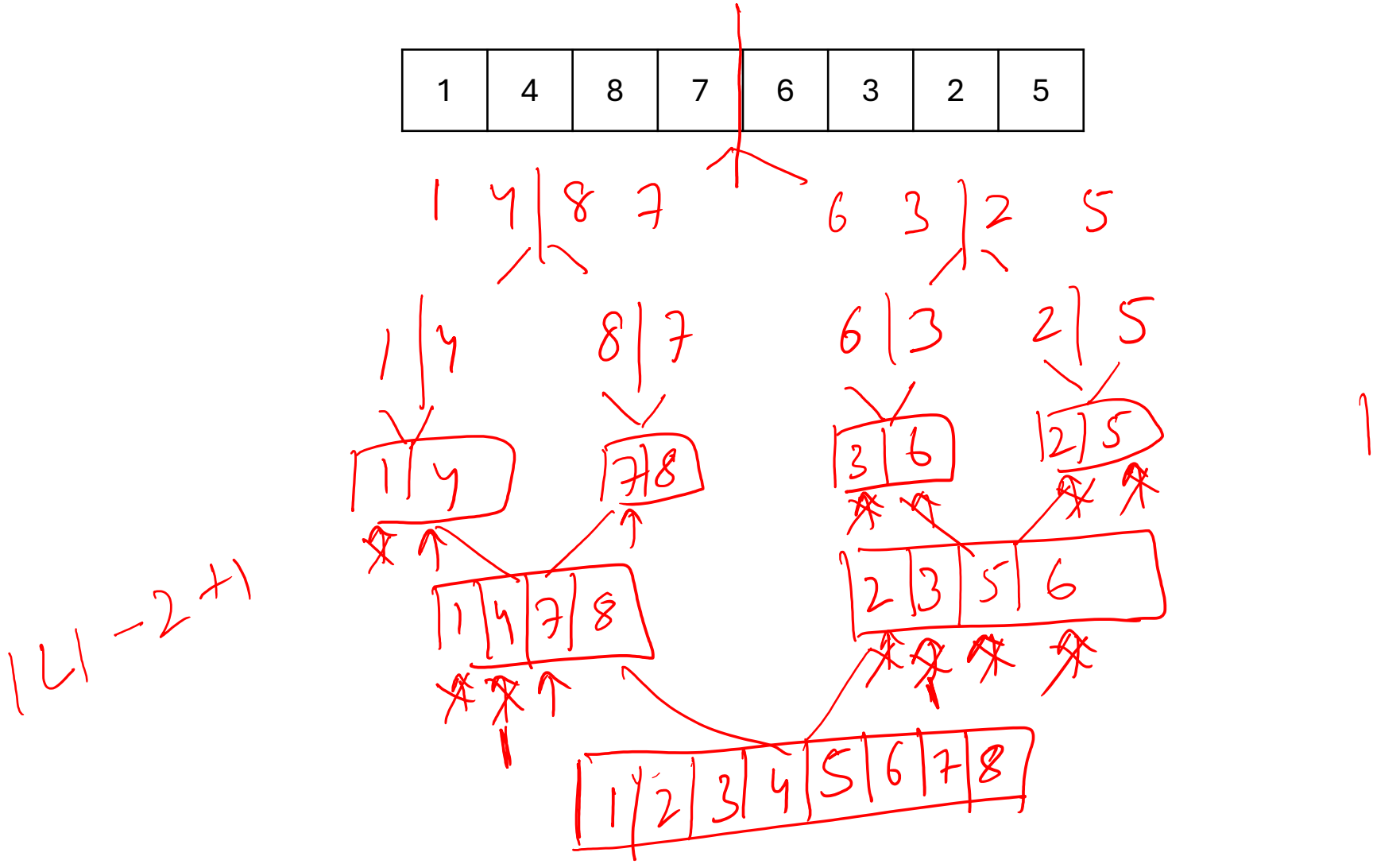
Input: Two Sorted arrays
Output: Count Inversions

```
MERGE-COUNT(L, R)
Initialize pt_ℓ = 1; pt_r = 1; i = 1 and Count = 0
Initialize an empty array A of size |L| + |R|.
while (pt_ℓ ≤ |L| and pt_r ≤ |R|)
    if (L[pt_ℓ] ≤ R[pt_r]
        A[i] = L[pt_ℓ] and pt_ℓ = pt_ℓ + 1
    else
        A[i] = R[pt_r]; pt_r = pt_r + 1 and Count = Count + |L| − pt_ℓ + 1
    i = i + 1
if (pt_ℓ > |L|)
    Append remaining elements of R into A
if (pt_r > |R|)
    Append remaining elements of L into A
Return (Count, A)
```

Running Time? $O(n)$

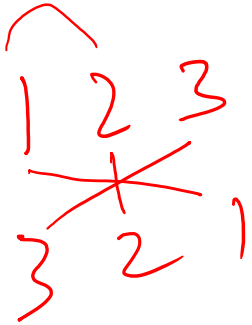
Counting Inversion Using DC

1	4	8	7	6	3	2	5



Counting Inversion Using DC

Input: Unsorted array
Output: Count Inversions



Counting Inversion Using DC

Input: Unsorted array
Output: Count Inversio

```
COUNT-INVERSION(A)
if |A| = 1
else
    m = ⌈|A|/2⌉
    A_Left = A[1, ..., m] and A_Right = A[m + 1, ..., |A|]
    (C_Left, A_Left) = COUNT-INVERSION(A_Left)
    (C_Right, A_Right) = COUNT-INVERSION(A_Right)
    (C_Split, A) = MERGE-COUNT(A_Left, A_Right)
    Count = C_Left + C_Right + C_Split
Return (Count, A)
```

Is it correct? Lets analyze!

Reference

Slides

Algorithms Design by Kleinberg & Tardos - Chp 3.4 & 5.3

Algorithm Design & Analysis (CSE222)

Lecture-4

Recap

- Testing Bipartiteness using BFS

- Counting Inversions

Outline

- Counting Inversions

- Select k Smallest Elements

Count Inversions in Sorted Arrays

Input: Two Sorted arrays
Output: Count Inversions

```
COUNT-INVERSION(A)
if |A| = 1
else
    m = ⌈|A|/2⌉
    A_Left = A[1, ..., m] and A_Right = A[m+1, ..., |A|]
    (C_Left, A_Left) = COUNT-INVERSION(A_Left)
    (C_Right, A_Right) = COUNT-INVERSION(A_Right)
    (C_Split, A) = MERGE-COUNT(A_Left, A_Right)
    Count = C_Left + C_Right + C_Split
Return (Count, A)
```

```
MERGE-COUNT(L, R)
Initialize pt_ℓ = 1; pt_r = 1; i = 1 and Count = 0
Initialize an empty array A of size |L| + |R|.
while (pt_ℓ ≤ |L| and pt_r ≤ |R|)
    if (L[pt_ℓ] ≤ R[pt_r]
        A[i] = L[pt_ℓ] and pt_ℓ = pt_ℓ + 1
    else
        A[i] = R[pt_r]; pt_r = pt_r + 1 and Count = Count + |L| − pt_ℓ + 1
    i = i + 1
if (pt_ℓ > |L|)
    Append remaining elements of R into A
if (pt_r > |R|)
    Append remaining elements of L into A
Return (Count, A)
```

Running Time?

Counting Inversion Using DC (Divide and Conquer)

Input: Unsorted array
Output: Count Inversions

```
COUNT-INVERSION(A)
if |A| = 1
else
    m = ⌈|A|/2⌉
    A_Left = A[1, ..., m] and A_Right = A[m + 1, ..., |A|]
    (C_Left, A_Left) = COUNT-INVERSION(A_Left)
    (C_Right, A_Right) = COUNT-INVERSION(A_Right)
    (C_Split, A) = MERGE-COUNT(A_Left, A_Right)
    Count = C_Left + C_Right + C_Split
Return (Count, A)
```

Is it correct? Lets analyze!

Analysis of Count-Inversion(A)

1 3 4 | 2 5 7

Proof by induction

Base: k = 0. So, n = 2^k = 1. No inversion.

Induction Hypothesis: The algorithm will return correct count for all k = 1, 2, ..., i, i.e., until n = 2^i.

Let n = 2^{i+1}. It has two sorted arrays and calls Merge-Count($\boldsymbol{A_{Left}}$, $\boldsymbol{A_{Right}}$)

Why count is right?

(3,2), (4,2)

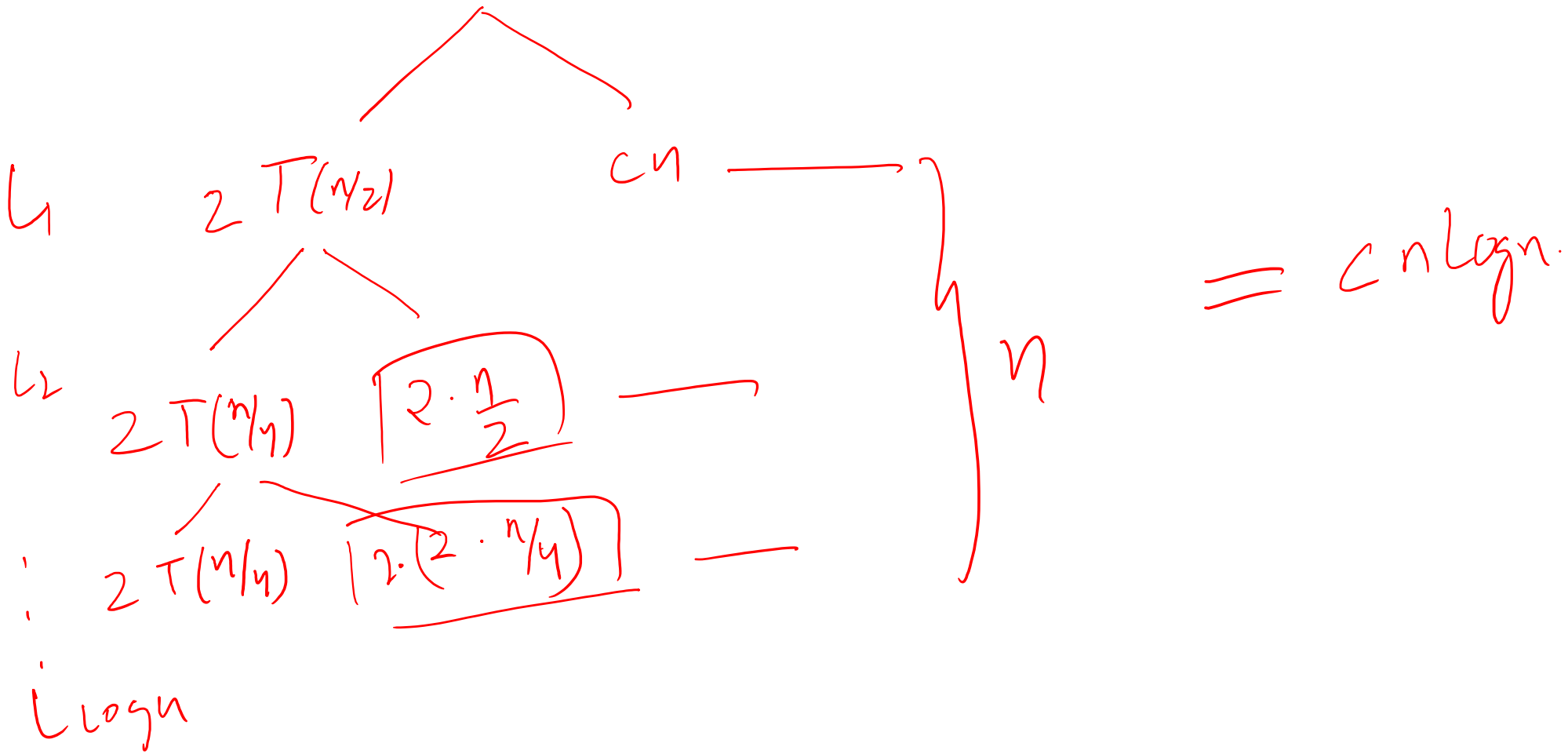
$|A_{left}| - Pt_L + 1$

1

Count = $C_{left} + C_{Right} + C_{split}$

Running Time

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



Outline

- Counting Inversions

- Select k Smallest Elements

k^{th} Smallest Element

Running time for the following problems

- Select smallest element in array of size n. (Can we get o(n) or $O(n^{1-c})$?)
- Select $n/2^{th}$ smallest element in an array of size n.
- Select k^{th} smallest element in an array of size n.

Input: Array of size n
Output: k^{th} smallest element.

1. Sort the input array in ascending order.
2. Return the value at k^{th} index.

Can we do better?

Not interested in the ranking of all the elements.

Only k^{th} smallest element!

QuickSort

Input: Array of size n
Output: Sorted Array

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Partition Function

2	8	7	1	3	5	6	4

2	1	3	4	7	5	6	8

Running time?

k^{th} Smallest Element

Based on the *<u>pivot</u>* element, partition the array into two subarrays
Lesser (All elements smaller than the pivot).
Greater (All elements bigger than the pivot).

Let the *<u>pivot</u>* position returned by Partition() is i.

- If $i = k$ then return the pivot element.
- If $i > k$ then go to *Lesser* and look for k^{th} smallest element.
- Else, go to *Greater* and look for $(k - |Lesser| - 1)^{th}$ smallest element.

Analyze

Divide & Conquer algorithm:
Every Partition() takes O(n).

Worst case could be $O(n^2)$.

Best case could be O(n). When?

1 2 3 4 5 6 7 8

$T(n) = T(n-1) + cn$

$T(n) = T(n/2) + cn$

Randomized Algorithm

Input: Array *A* of size n, integer *k*.
Output: k^{th} smallest element in *A*.

RandQuickSelect(*A*,*k*)

1. Pick a random pivot element *p* from *A*.
2. Split A into subarrays *Lesser* & *Greater*. Set $L = |Lesser|$
3. If $L = k - 1$ then return *p*.
4. If $L > k - 1$ then return RandQuickSelect(*Lesser*, *k*).
5. If $L < k - 1$ then return RandQuickSelect(*Greater*, $k - L - 1$)

Claim: The expected running time of RandQuickSelect() is O(n).

Randomized Algorithm

Input: Array *A* of size n, integer *k*.
Output: k^{th} smallest element in *A*.

```
RandQuickSelect(A, k)
if (|A| == 1)
    Return A
p = ChoosePivot(A)
Lesser = {A[i] : A[i] < p}; Greater = {A[i] : A[i] > p}
if (|Lesser| == k − 1)
    Return p
if (|Lesser| > k − 1)
    Return RandQuickSelect(Lesser, k)
else
    Return RandQuickSelect(Greater, k − |Lesser| − 1)
```

Claim: The expected running time of RandQuickSelect() is O(n).

$$T(n) = cn + \frac{1}{n}\sum_{i=1}^{n} T(i) \le cn + \sum_{i=n/2}^{n} T(i) \cdot \frac{2}{n}$$

$$cn + \frac{2}{n} \cdot c\left(\frac{n}{2}\left(\frac{n}{2}+1\right)/2 + \frac{n}{2}\frac{n}{2}\right) = cn + c\frac{n}{4} + \frac{cn}{2}$$

$$= O(n)$$

Analyze

The expected running time of RandQuickSelect() is O(n).

The exact running time depends on partition sizes due to the random pivot.

Expected running time is the average of all possible partition sizes. (0 & n-1) or (1 & n-2) or (2 & n-3) or ... or (n-1 & 0).

The Partition() takes c·n time. So recurrence of expected running time is,

(0 & n-1) or (n-1 & 0)

$$T(n) \le cn + \frac{2}{n}\sum_{i=n/2}^{n-1} T(i)$$

$$cn + \frac{2}{n}\left(c\frac{n}{2} + c\left(\frac{n}{2}+1\right) + \cdots + c(n-1)\right)$$

$$\le cn + \frac{2}{n}c\left(\frac{n}{2} + \frac{n}{2}+1 + \cdots + n-1 + \frac{n}{2} + \frac{n}{2}\right)$$

Analyze

$$
\begin{aligned}
T(n) &\leq c \cdot n + \frac{2}{n} \sum_{i=\lceil n/2 \rceil}^{n-1} T(i) \\
&\leq c \cdot n + \frac{2}{n} \left(T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil + 1) + \ldots + T(n) \right) \\
&= c \cdot n + \frac{2c}{n} \left(\lceil n/2 \rceil + \lceil n/2 \rceil + 1 + \ldots + n \right) \\
&= c \cdot n + \frac{2c}{n} \left(n/2 \cdot n/2 + 1 + 2 + \ldots + n/2 \right) \\
&= c \cdot n + \frac{2c}{n} \left(\frac{n^2}{4} + \frac{n/2(n/2+1)}{2} \right) \leq 3c \cdot n = O(n)
\end{aligned}
$$

Analyze

The expected running time of RandQuickSelect() is O(n).

The exact running time depends on partition sizes due to the random pivot.

Expected running time is the average of all possible partition sizes. (0 & n-1) or (1 & n-2) or (2 & n-3) or … or (n-1 & 0).

The Partition() takes c·n time. So recurrence of expected running

$$T(n) \leq cn + \frac{2}{n} \sum_{i=n/2}^{n-1} T(i)$$

Can we improve the worst case running time?

Reference

Slides

Algorithms Design by Kleinberg & Tardos - Chp 5.3

Algorithm Design & Analysis (CSE222)

Lecture-5

Recap

- Counting Inversions
  - Merge-Count()

```
MERGE-COUNT(L, R)
Initialize pt_ℓ = 1; pt_r = 1; i = 1 and Count = 0
Initialize an empty array A of size |L| + |R|.
while (pt_ℓ ≤ |L| and pt_r ≤ |R|)
    if (L[pt_ℓ] ≤ R[pt_r]
        A[i] = L[pt_ℓ] and pt_ℓ = pt_ℓ + 1
    else
        A[i] = R[pt_r]; pt_r = pt_r + 1 and Count = Count + |L| − pt_ℓ + 1
    i = i + 1
if (pt_ℓ > |L|)
    Append remaining elements of R into A
if (pt_r > |R|)
    Append remaining elements of L into A
Return (Count, A)
```

```
COUNT-INVERSION(A)
if |A| = 1
else
    m = ⌈|A|/2⌉
    A_Left = A[1, ..., m] and A_Right = A[m + 1, ..., |A|]
    (C_Left, A_Left) = COUNT-INVERSION(A_Left)
    (C_Right, A_Right) = COUNT-INVERSION(A_Right)
    (C_Split, A) = MERGE-COUNT(A_Left, A_Right)
    Count = C_Left + C_Right + C_Split
Return (Count, A)
```

Recap

- Counting Inversions
  - Merge-Count()
- Select k Smallest Elements
  - Randomized Algorithm: Partition()

```
RandQuickSelect(A, k)
if (|A| == 1)
    Return A
p = ChoosePivot(A)
Lesser = {A[i] : A[i] < p}; Greater = {A[i] : A[i] > p}
if (|Lesser| == k − 1)
    Return p
if (|Lesser| > k − 1)
    Return RandQuickSelect(Lesser, k)
else
    Return RandQuickSelect(Greater, k − |Lesser| − 1)
```

Outline

- Select k-Smallest Element

- Fast Fourier transform

Deterministic Algorithm (B. F. P. R. & T. 1973)

Input: Array *A* of size n, integer *k*.
Output: k^{th} smallest element in *A*.

DetQuickSelect(*A*,*k*)

1. Group *A* into n/5 groups, each of size 5. Find median of each group.
2. Recursively find median of the medians. Lets call it p.
3. Split A into subarrays *Lesser* & *Greater*. Set $L = |Lesser|$
4. If $L = k - 1$ then return *p*.
5. If $L > k - 1$ then return DetQuickSelect(*Lesser*, *k*).
6. If $L < k - 1$ then return DetQuickSelect(*Greater*, $k - L - 1$)

Claim: The worst case running time of DetQuickSelect() is O(n).

Deterministic Algorithm (B. F. P. R. & T. 1973)

Input: Array *A* of size n, integer *k*.
Output: k^{th} smallest element in *A*.

$\textsc{ChoosePivot}(A)$

Initialize an empty array C of size $|A|/5$; $i = 1$
Split A into $|A|/5$ groups as $B_1, \ldots B_g$ where $g = |A|/5$.
for $j = 1 \cdots g$
    $q = Median(B_j)$
    $C[i] = q$; $i = i + 1$
$p = \textsc{DetQuickSelect}(C, |C|/2)$
Return p

$\textsc{DetQuickSelect}(A, k)$

if $(|A| == 1)$
    Return A
$p = \textsc{ChoosePivot}(A)$
$Lesser = \{A[i] : A[i] < p\}$; $Greater = \{A[i] : A[i] > p\}$
if $(|Lesser| == k - 1)$
    Return p
if $(|Lesser| > k - 1)$
    Return $\textsc{DetQuickSelect}(Lesser, k)$
else
    Return $\textsc{DetQuickSelect}(Greater, k - |Lesser| - 1)$

Claim: The worst case running time of DetQuickSelect() is O(n).

Analyze

Let T(n) be the running time of DetQuickSelect()

Step1: Takes O(n)
Step2: Takes T(n/5)
Step3: Takes O(n)
Step4: Takes O(1)
Max of Step5 and Step6 is T(7n/10).

Recurrence: $T(n) \quad \leq \quad cn + T(n/5) + T(7n/10)$

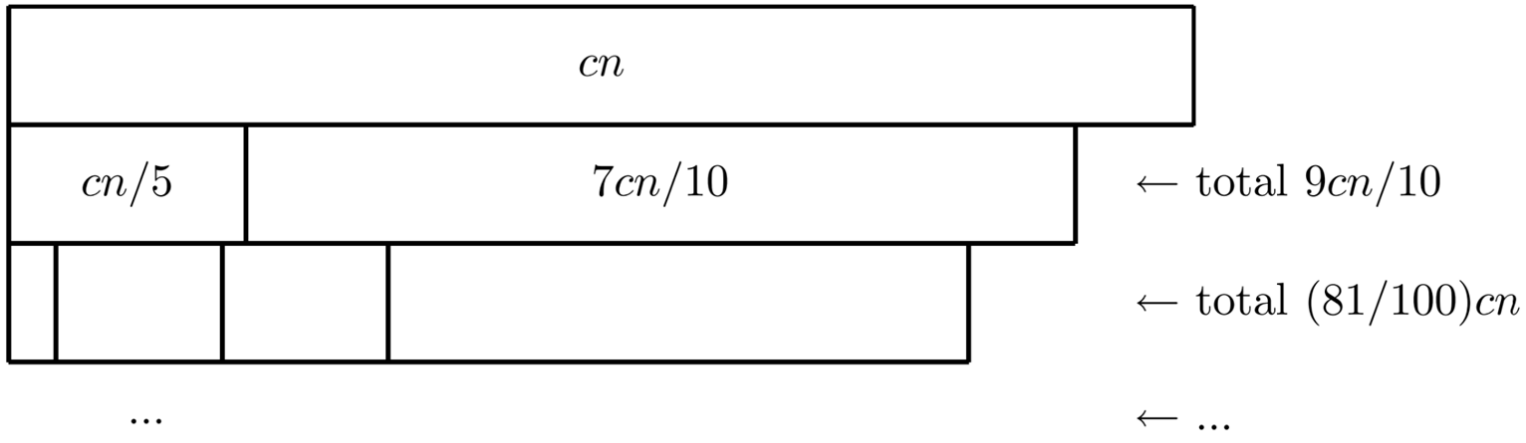
DetQuickSelect(*A*,*k*)

1. Group *A* into n/5 groups, each of size 5. Find median of each group.
2. Recursively find median of the medians. Lets call it p.
3. Split A into subarrays *Lesser* & *Greater*. Set *L* = |*Lesser*|
4. If *L* = *k* - 1 then return *p*.
5. If *L* > *k* - 1 then return DetQuickSelect(*Lesser*, *k*).
6. If *L* < *k* - 1 then return DetQuickSelect(*Greater*, *k* - *L* - 1)

Analyze

$$T(n) \quad \leq \quad cn + T(n/5) + T(7n/10)$$

Recurrence Tree



$$cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \ldots)$$

Sum of gp when r < 1: a/(1-r).

Outline

- Select k-Smallest Element

- Fast Fourier transform

Fast Fourier Transform

Global Positioning System (GPS)

Television network

Wireless Communications

<u>Signal Processing</u>

- Spectrum analysis: Determine frequency content of a signal.
- Filtering: Remove unwanted frequency component.
- Compression: Lossless data compression.
- Convolution: Mathematical operation used to combine two signals.

Fast Fourier Transform

Discrete Fourier Transform

Time domain ⇔ Frequency domain

and more...

Polynomial Multiplication (Convolution)

Given, two polynomials $P(x)$ and $Q(x)$ compute $R(x) = P(x) \cdot Q(x)$.

Let, $P(x) = 1 + 2x^2$ and $Q(x) = 2x + x^2$ then

$R(x) = 2x + x^2 + 4x^3 + 2x^4$.

```
PolyMult(P, Q):
## Compute R(x) = P(x)·Q(x).

R = [0, 2, 1, 4, 2]
```

Coefficient Value Representation.

Polynomial Multiplication (Convolution)

PolyMult(P, Q):
Compute $R(x) = P(x) \cdot Q(x)$.

If $P(x)$ and $Q(x)$ are degree d polynomials then PolyMult(P, Q) will take $O(d^2)$ time.

Polynomial Multiplication (Convolution)

How many points do we need to represent a polynomial of degree 1?

Degree 1 polynomial is a line, so two points enough.

$P(x) = p_0 + p_1 \cdot x$.

If (3, 0) & (0, 3) are two points then $P(x) = 3 - x$.

Point Value Representation

Claim: Any d degree polynomial can be uniquely represented by d + 1 points.

Proof

Claim: Any d degree polynomial can be uniquely represented by d + 1 points.

$$\{(x_0, P(x_0)), (x_1, P(x_1)), \ldots, (x_d, P(x_d))\}$$

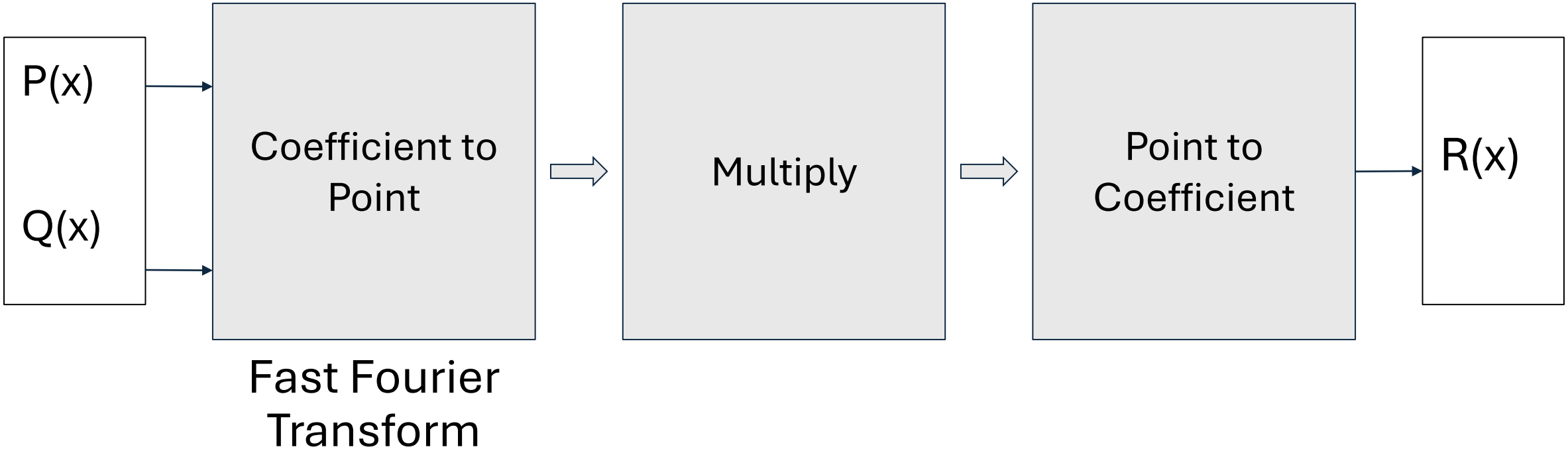
$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d$$

$$\begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_d) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^d \\ 1 & x_1 & x_1^2 & \cdots & x_1^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \cdots & x_d^d \end{bmatrix}}_{M} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_d \end{bmatrix}$$

- M is invertible for unique $x_0, \ldots, x_d$
- So, unique $p_0, \ldots, p_d$, i.e., $p = M^{-1} \cdot y$
- So, unique polynomial.

Polynomial Multiplication (Convolution)

Given point value representation of P(x) and Q(x) of degree d. Propose an efficient algorithm (assume $M^{-1} \cdot y$ takes (O(1) time).



Fast Fourier Transform

Consider the polynomi $P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d$

Compute n ≥ d+1 pairs of (point, value) for P(x).

Coefficient to Point

$$\begin{array}{c} (1, P(1)) \\ (2, P(2)) \\ \vdots \\ (n, P(n)) \end{array} \longrightarrow \begin{array}{c} (1, p_0 + p_1 \cdot 1 + p_2 \cdot 1^2 + \cdots + p_d \cdot 1^d) \\ (2, p_0 + p_1 \cdot 2 + p_2 \cdot 2^2 + \cdots + p_d \cdot 2^d)) \\ \vdots \\ (n, p_0 + p_1 \cdot n + p_2 \cdot n^2 + \cdots + p_d \cdot n^d)) \end{array}$$

P(x) at unique n points.

Running time: O(nd) ~ O(d^2) !!!

Lets improvise.

Fast Fourier Transform

Let $P(x) = x^2$ and $n = 4$.

(1,1) & (-1,1)

(2,4) & (-2,4)

$P(-x) = P(x)$

Let $P(x) = x^3$ and $n = 4$.

(1,1) & (-1,-1)

(2,8) & (-2,-8)

$P(-x) = -P(x)$

Fast Fourier Transform

General function $P(x) = 3x^5 + 2x^4 + x^3 + 7x^2 + 5x + 1$

Calculate P(x) at $\pm x_1, \pm x_2, \ldots, \pm x_{n/2}$

$$P(x) = (2x^4 + 7x^2 + 1) + (3x^5 + x^3 + 5x)$$

$$P(x) = \underbrace{(2x^4 + 7x^2 + 1)}_{P_e(x^2)} + x\underbrace{(3x^4 + x^2 + 5)}_{P_o(x^2)}$$

$$P(x) = P_e(x^2) + xP_o(x^2)$$

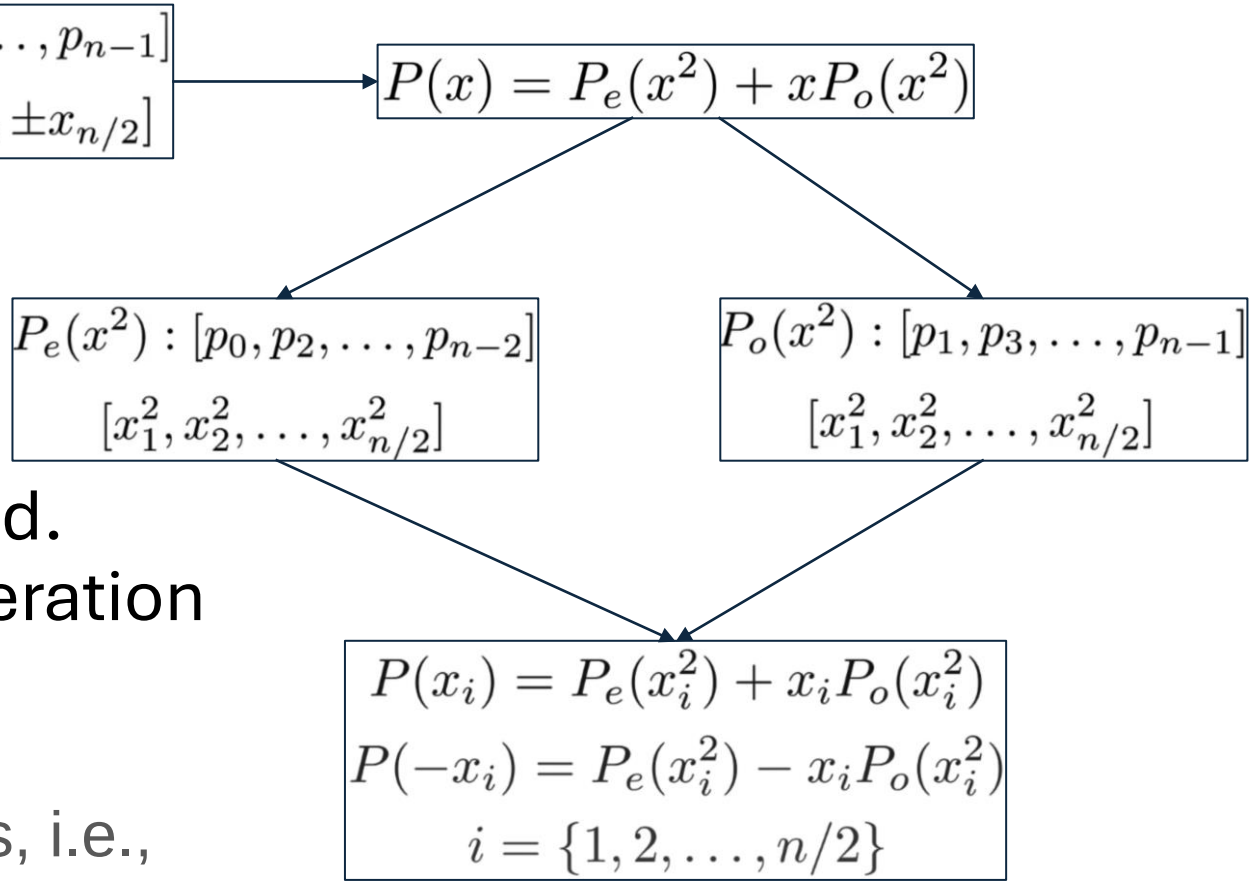
$$P(x_i) = P_e(x_i^2) + x_i P_o(x_i^2)$$

$$P(-x_i) = P_e(x_i^2) - x_i P_o(x_i^2)$$

<u>Observations</u>

- $P_e(x^2)$ and $P_o(x^2)$ are of degree 2, even though P(x) was a 5 degree polynomial.
- Calculate $P_e(x^2)$ and $P_o(x^2)$ $x_1^2, x_2^2, \ldots, x_{n/2}^2$ points.

Fast Fourier Transform

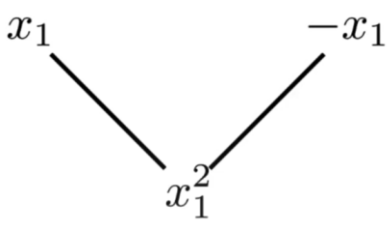


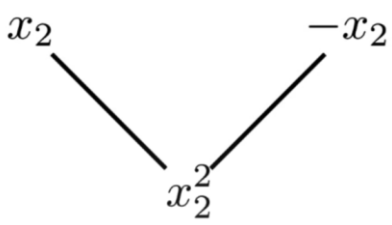
<u>Observations</u>

- Running time O(n·log(n))
- $[\pm x_1, \pm x_2, \ldots, \pm x_{n/2}]$ ls to be paired.
- So, $[x_1^2, x_2^2, \ldots, x_{n/2}^2]$. For next iteration these needs to paired.
- But these are not +ve/-ve paired.
- Expand the domain of initial points, i.e., $[\pm x_1, \pm x_2, \ldots, \pm x_{n/2}]$ complex numbers.

Fast Fourier Transform

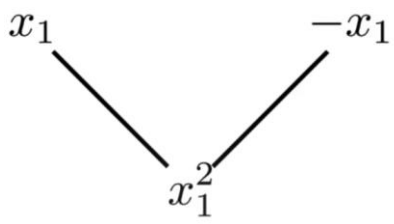
Let $P(x) = x^3 - x^2 + x - 1$ & $n = 4$.

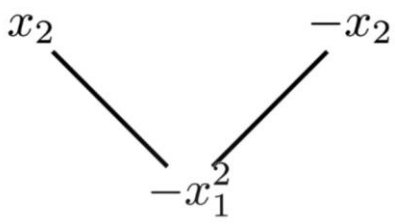




Fast Fourier Transform

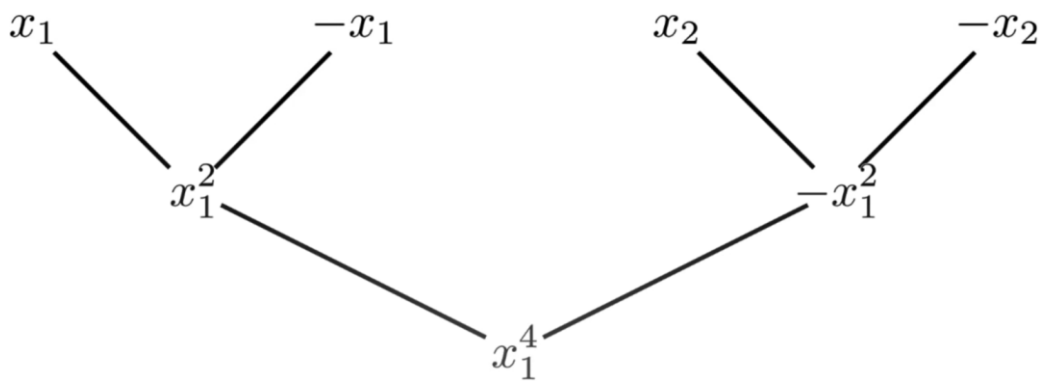
Let $P(x) = x^3 - x^2 + x - 1$ & $n = 4$.





Fast Fourier Transform

Let $P(x) = x^3 - x^2 + x - 1$ & $n = 4$.

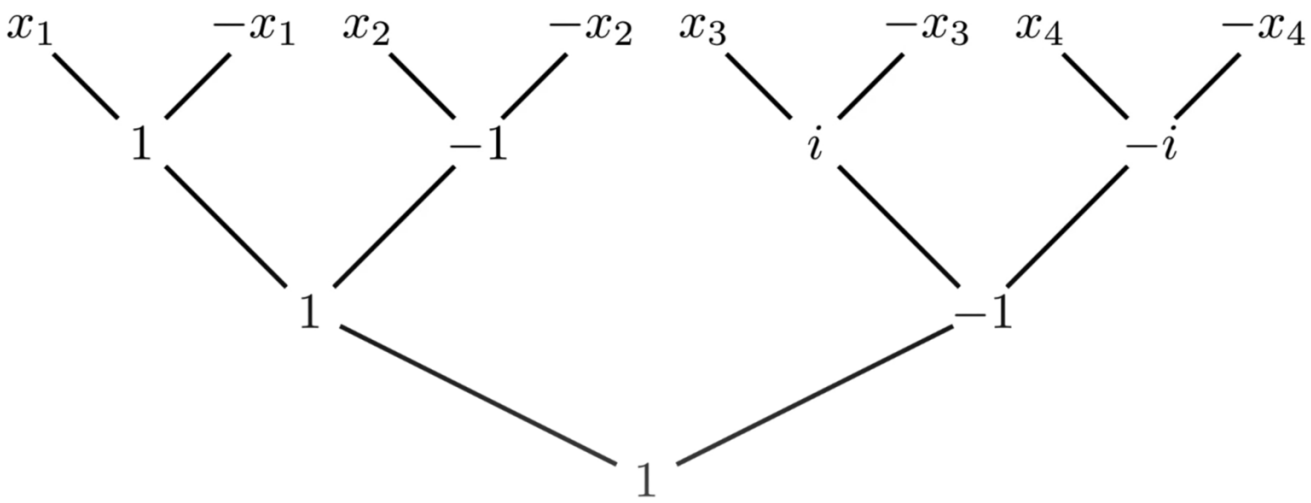


Let $x_1 = 1$.

So, our initial points are solution to $x^4 = 1$.

Fast Fourier Transform

Let $P(x) = x^6 - x^5 + x^4 - x^3 + x^2 - x + 1$ & $n = 8$ (>7).



So, our initial points are solution to $x^8 = 1$.

Reference

Slides

Algorithms Design by Kleinberg & Tardos - Chp 5.3 & 5.6

Jeff Erickson's Lecture notes - 1.8

Algorithm Design & Analysis (CSE222)

Lecture-6

Recap

- Select k Smallest Elements
  - Deterministic Algorithm: Partition()

$\textsc{DetQuickSelect}(A, k)$

```
if (|A| == 1)
    Return A
p = ChoosePivot(A)
Lesser = {A[i] : A[i] < p}; Greater = {A[i] : A[i] > p}
if (|Lesser| == k − 1)
    Return p
if (|Lesser| > k − 1)
    Return DetQuickSelect(Lesser, k)
else
    Return DetQuickSelect(Greater, k − |Lesser| − 1)
```

$\textsc{ChoosePivot}(A)$

```
Initialize an empty array C of size |A|/5; i = 1
Split A into |A|/5 groups as B_1, ... B_g where g = |A|/5.
for j = 1 ··· g
    q = Median(B_j)
    C[i] = q; i = i + 1
p = DetQuickSelect(C, |C|/2)
Return p
```

Recap

- Select k Smallest Elements
  - Deterministic Algorithm: Partition()
- Fast Fourier Transform
  - Convolution

Outline

- Fast Fourier Transform (Contd.)

- Dynamic Programming

Polynomial Multiplication (Convolution)

Given, two polynomials $P(x)$ and $Q(x)$ compute $R(x) = P(x) \cdot Q(x)$.

Let, $P(x) = 1 + 2x^2$ and $Q(x) = 2x + x^2$ then

$R(x) = 2x + x^2 + 4x^3 + 2x^4$.

```
PolyMult(P, Q):
## Compute R(x) = P(x)·Q(x).

R = [0, 2, 1, 4, 2]
```

Coefficient Value Representation.

Polynomial Multiplication (Convolution)

PolyMult(P, Q):
Compute R(x) = P(x)·Q(x).

If P(x) and Q(x) are degree d polynomials then PolyMult(P, Q) will take $O(d^2)$ time.

```
for i = 1 ... d
    for j = 1 ... d
        R[i+j] = R[i+j] + P[i] · Q[j]
```

Polynomial Multiplication (Convolution)

How many points do we need to represent a polynomial of degree 1?

Degree 1 polynomial is a line, so two points enough.

$P(x) = p_0 + p_1 \cdot x$.

If (3, 0) & (0, 3) are two points then $P(x) = 3 - x$. (**Point Value Representation**)

Claim: Any d degree polynomial can be uniquely represented by d + 1 points.

Proof

Claim: Any d degree polynomial can be uniquely represented by d + 1 points.

$$\{(x_0, P(x_0)), (x_1, P(x_1)), \ldots, (x_d, P(x_d))\}$$

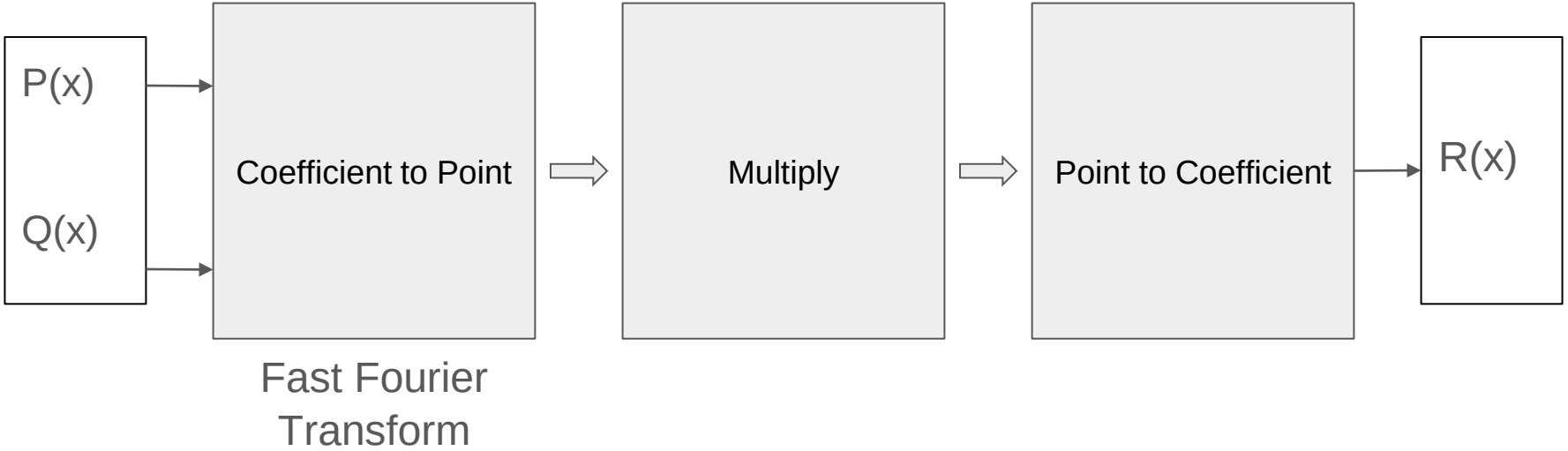
$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d$$

$$\begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_d) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^d \\ 1 & x_1 & x_1^2 & \cdots & x_1^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \cdots & x_d^d \end{bmatrix}}_{M} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_d \end{bmatrix}$$

- M is invertible for unique $x_0, \ldots, x_d$
- So, unique $p_0, \ldots, p_d$, i.e., $p = M^{-1} \cdot y$
- So, unique polynomial.

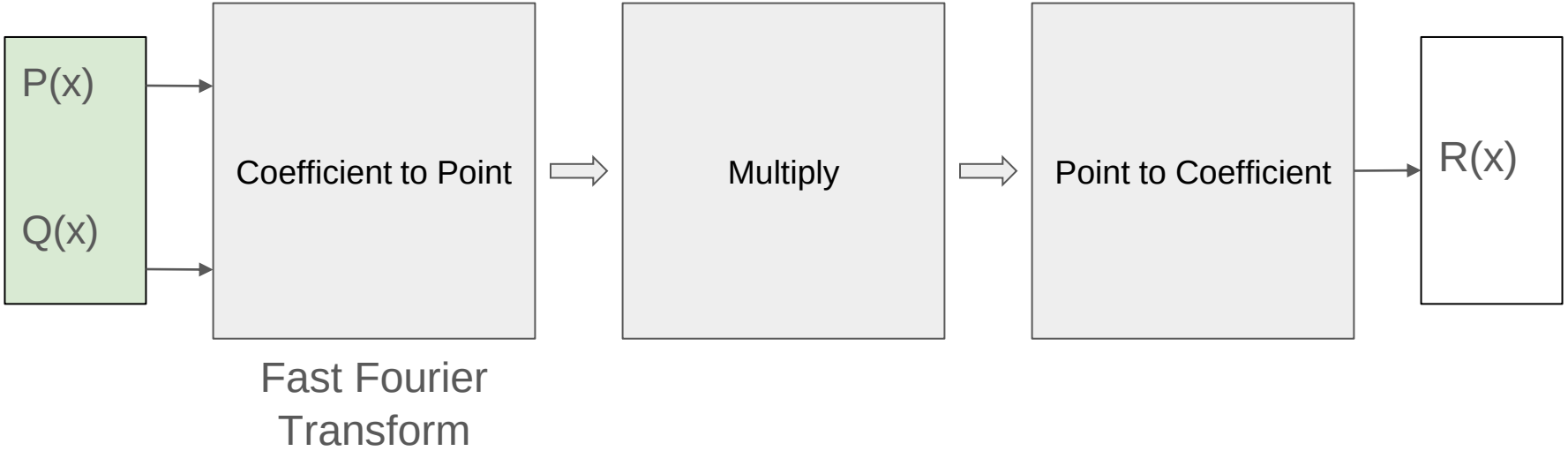
Polynomial Multiplication (Convolution)

Given point value representation of P(x) and Q(x) of degree d. Propose an efficient algorithm to multiply P(x) and Q(x).



Polynomial Multiplication (Convolution)

Given point value representation of P(x) and Q(x) of degree d. Propose an efficient algorithm to multiply P(x) and Q(x).



Fast Fourier Transform

Consider the polynomial $P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d$

Compute n ≥ d+1 pairs of (point, value) for P(x).

Coefficient to Point

$$\begin{array}{c} (1, P(1)) \\ (2, P(2)) \\ \vdots \\ (n, P(n)) \end{array} \longrightarrow \begin{array}{c} (1, p_0 + p_1 \cdot 1 + p_2 \cdot 1^2 + \cdots + p_d \cdot 1^d) \\ (2, p_0 + p_1 \cdot 2 + p_2 \cdot 2^2 + \cdots + p_d \cdot 2^d)) \\ \vdots \\ (n, p_0 + p_1 \cdot n + p_2 \cdot n^2 + \cdots + p_d \cdot n^d)) \end{array}$$

P(x) at unique n points.

Running time: O(nd) ~ O(d^2) !!!

What if, we only need to compute n/2 (point, value) pairs and use it to get n pairs.

Coefficient to Point

Let $P(x) = x^2$ and $n = 4$.

(1,1) & (-1,1) $P(-x) = P(x)$

(2,4) & (-2,4)

Let $P(x) = x^3$ and $n = 4$.

(1,1) & (-1,-1) $P(-x) = -P(x)$

(2,8) & (-2,-8)

Coefficient to Point

General function $P(x) = 3x^5 + 2x^4 + x^3 + 7x^2 + 5x + 1$

Calculate P(x) at $\pm x_1, \pm x_2, \ldots, \pm x_{n/2}$

$$P(x) = (2x^4 + 7x^2 + 1) + (3x^5 + x^3 + 5x)$$

$$P(x) = \underbrace{(2x^4 + 7x^2 + 1)}_{P_e(x^2)} + x\underbrace{(3x^4 + x^2 + 5)}_{P_o(x^2)}$$

$$P(x) = P_e(x^2) + xP_o(x^2)$$

$$P(x_i) = P_e(x_i^2) + x_i P_o(x_i^2)$$

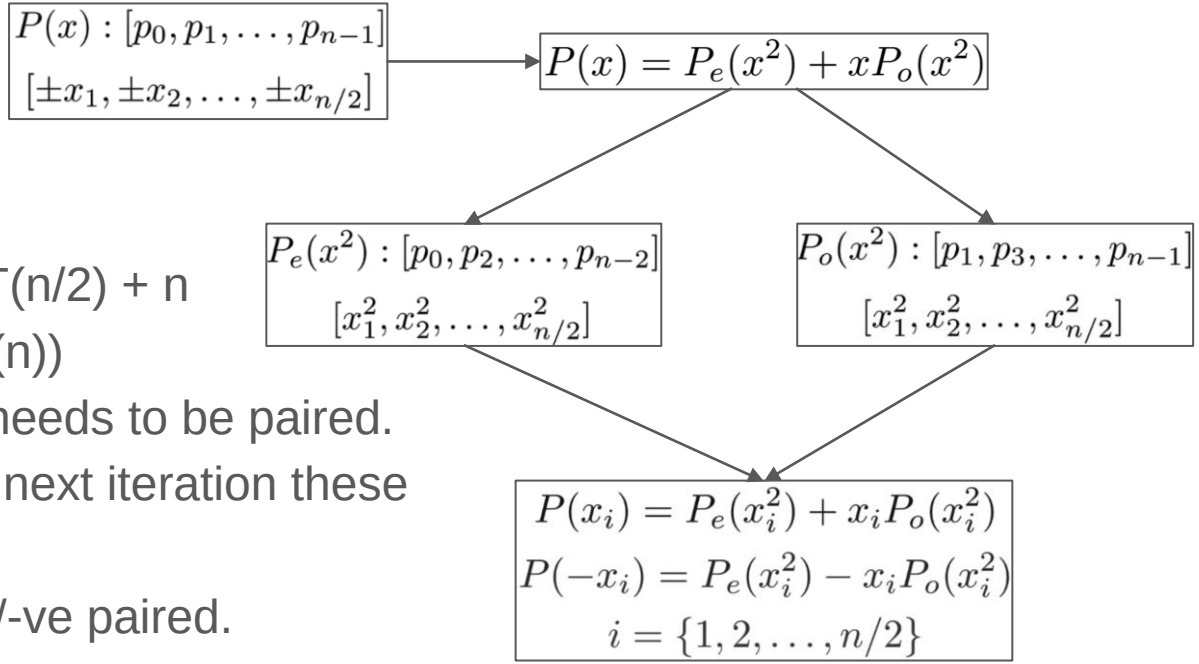
$$P(-x_i) = P_e(x_i^2) - x_i P_o(x_i^2)$$

<u>Observations</u>

- $P_e(x^2)$ and $P_o(x^2)$ are of degree 2, even though P(x) was a 5 degree polynomial.
- Calculate $P_e(x^2)$ and $P_o(x^2)$ at $x_1^2, x_2^2, \ldots, x_{n/2}^2$ points.

Coefficient to Point

I/P: P = [p_0, p_1, …, p_{n-1}]
O/P: [P(x_0), …, P(X_{n-1})]

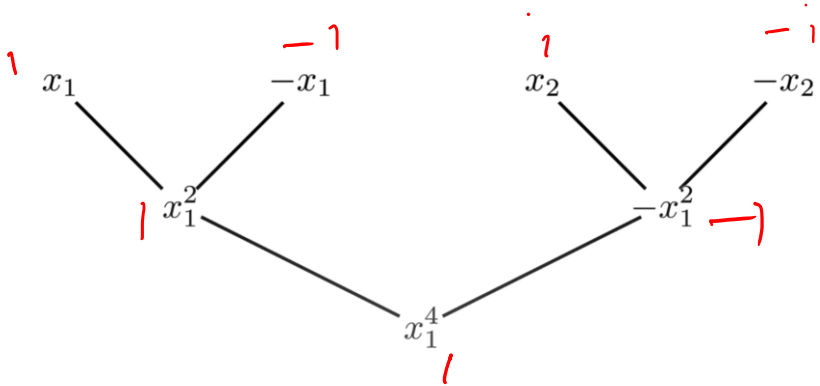


<u>Observations</u>

- Recurrence: T(n) = 2T(n/2) + n
- Running time O(n·log(n))
- $[\pm x_1, \pm x_2, \ldots, \pm x_{n/2}]$ needs to be paired.
- So, $[x_1^2, x_2^2, \ldots, x_{n/2}^2]$. For next iteration these needs to paired.
- But these are not +ve/-ve paired.
- Expand the domain of initial points, i.e., $[\pm x_1, \pm x_2, \ldots, \pm x_{n/2}]$ complex numbers.

Coefficient to Point

Let $P(x) = x^3 - x^2 + x - 1$ & $n = 4$.

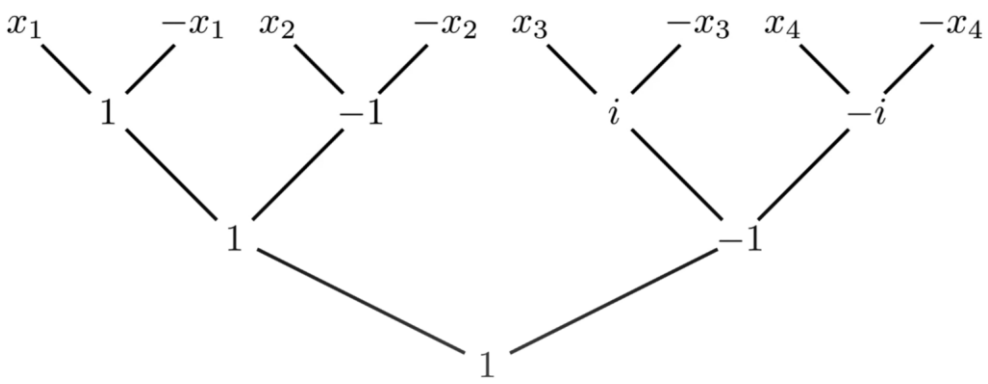


Let $x_1 = 1$.

So, our initial points are solution to $x^4 = 1$.

Coefficient to Point

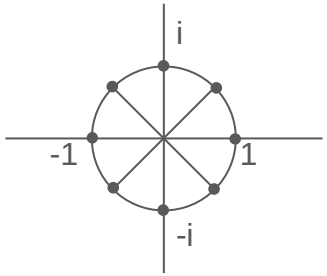
Let $P(x) = x^6 - x^5 + x^4 - x^3 + x^2 - x + 1$ & $n = 8$ (>7).



So, our initial points are solution to $x^8 = 1$.

Coefficient to Point

Solving $x^n = 1$ gives,



$e^{i\theta} = \cos(\theta) + i \cdot \sin(\theta)$

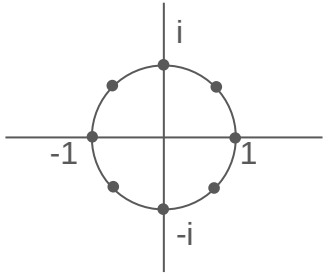
$\omega = e^{\frac{2\pi i}{n}}$

Roots are $[\omega^0, \omega^1, \ldots, \omega^{n-1}]$.

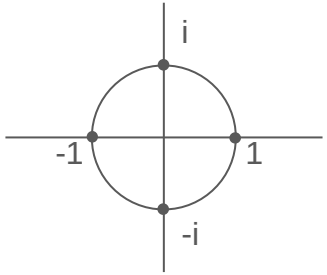
Notice, $\omega^{j + n/2} = -\omega^j$ in pairs!

Coefficient to Point

Squaring n roots of unity results.



Recursion



Evaluate p(x) at [$1, \omega^1, \ldots, \omega^{n-1}$]

Evaluate $P_e(x^2)$ and $P_e(x^2)$ at [$1, \omega^2, \ldots, \omega^{2(n/2-1)}$]

These are also paired.

Coefficient to Point

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$
Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

$$P(x) : [p_0, p_1, \ldots, p_{n-1}]$$

$$\omega = e^{\frac{2\pi i}{n}} : [\omega^0, \omega^1, \ldots, \omega^{n-1}]$$

Base: n = 1, P(1).

$$P_e(x^2) : [p_0, p_2, \ldots, p_{n-2}] \qquad P_o(x^2) : [p_1, p_3, \ldots, p_{n-1}]$$

$$[\omega^0, \omega^2, \ldots, \omega^{n-2}] \qquad [\omega^0, \omega^2, \ldots, \omega^{n-2}]$$

$$P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j})$$

$$P(-\omega^j) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j})$$

$$j \in \{0, 1, \ldots (n/2 - 1)\}$$

Coefficient to Point

Input: P = [p_0, p_1, …, p_{n-1}]
Output y = [P(ω^0), P(ω^1), …, P(ω^{n-1})]

$$P(x) : [p_0, p_1, \ldots, p_{n-1}]$$

$$\omega = e^{\frac{2\pi i}{n}} : [\omega^0, \omega^1, \ldots, \omega^{n-1}]$$

Base: n = 1, P(1).

$$P_e(x^2) : [p_0, p_2, \ldots, p_{n-2}] \qquad P_o(x^2) : [p_1, p_3, \ldots, p_{n-1}]$$

$$[\omega^0, \omega^2, \ldots, \omega^{n-2}] \qquad [\omega^0, \omega^2, \ldots, \omega^{n-2}]$$

$$P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j})$$

$$P(\omega^{j+n/2}) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j})$$

$$j \in \{0, 1, \ldots (n/2 - 1)\}$$

$\mathrm{FFT}(P)$

```
Initialize n = |P|; ω = e^(2πi/n) and y = [0]^n
if n == 1
    Return P
P_e = [p_0, p_2, ..., p_{n-2}] and P_o = [p_1, p_3, ..., p_{n-1}]
y_e = FFT(P_e) and y_o = FFT(P_o)
for j = 0 ··· n/2 − 1
    y[j] = y_e[j] + ω^j y_o[j]
    y[j + n/2] = y_e[j] − ω^j y_o[j]
Return y
```

Coefficient to Point

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$

Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

$\mathrm{FFT}(P)$

1 Initialize $n = |P|; \omega = e^{\frac{2\pi i}{n}}$ and $y = [0]^n$

2 **if** $n == 1$

3 Return P

4 $P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$

5 $y_e = \mathrm{FFT}(P_e)$ and $y_o = \mathrm{FFT}(P_o)$

6 **for** $j = 0 \cdots n/2 - 1$

7 $y[j] = y_e[j] + \omega^j y_o[j]$

8 $y[j + n/2] = y_e[j] - \omega^j y_o[j]$

9 Return y

$P = [4,3,2,1]$

FFT(P):

1: $n = 4$; $\omega = i$; $y = [0,0,0,0]$.

2-3. 4: $P_e = [4, 2]$ and $P_o = [3, 1]$.

5: $\mathrm{FFT}(P_e)$

Coefficient to Point

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$
Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

$\mathrm{FFT}(P)$

1 Initialize $n = |P|; \omega = e^{\frac{2\pi i}{n}}$ and $y = [0]^n$
2 **if** $n == 1$
3 Return P
4 $P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$
5 $y_e = \mathrm{FFT}(P_e)$ and $y_o = \mathrm{FFT}(P_o)$
6 **for** $j = 0 \cdots n/2 - 1$
7 $y[j] = y_e[j] + \omega^j y_o[j]$
8 $y[j + n/2] = y_e[j] - \omega^j y_o[j]$
9 Return y

$P_e = [4,2]$
$\mathrm{FFT}(P_e)$:
1: $n = 2$; $\omega = -1$; $y = [0,0]$.
2-3. 4: $P_e = [4]$ and $P_o = [2]$.
5: $\mathrm{FFT}(P_e) \rightarrow 4$
6: $j = 0$.
7-8: $y[0] = 4 + 2$, $y[1] = 4 - 2$.
9: Return [6, 2].

Coefficient to Point

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$

Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

$\mathrm{FFT}(P)$

1 Initialize $n = |P|; \omega = e^{\frac{2\pi i}{n}}$ and $y = [0]^n$

2 **if** $n == 1$

3 Return P

4 $P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$

5 $y_e = \mathrm{FFT}(P_e)$ and $y_o = \mathrm{FFT}(P_o)$

6 **for** $j = 0 \cdots n/2 - 1$

7 $y[j] = y_e[j] + \omega^j y_o[j]$

8 $y[j + n/2] = y_e[j] - \omega^j y_o[j]$

9 Return y

$P = [4,3,2,1]$

FFT(P):

1: $n = 4$; $\omega = i$; $y = [0,0,0,0]$.

2-3. 4: $P_e = [4, 2]$ and $P_o = [3, 1]$.

5: $[6, 2] = \mathrm{FFT}(P_e)$ & $\mathrm{FFT}(P_0)$

Coefficient to Point

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$
Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

$\mathrm{FFT}(P)$

Initialize $n = |P|; \omega = e^{\frac{2\pi i}{n}}$ and $y = [0]^n$
if $n == 1$
    Return P
$P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$
$y_e = \mathrm{FFT}(P_e)$ and $y_o = \mathrm{FFT}(P_o)$
for $j = 0 \cdots n/2 - 1$
    $y[j] = y_e[j] + \omega^j y_o[j]$
    $y[j + n/2] = y_e[j] - \omega^j y_o[j]$
Return y

$P_0 = [3,1]$
$\mathrm{FFT}(P_o)$:
1: $n = 2$; $\omega = -1$; $y = [0,0]$.
2-3. 4: $P_e = [3]$ and $P_o = [1]$.
5: $\mathrm{FFT}(P_e) \rightarrow 3$ & $\mathrm{FFT}(P_o) = 1$.
6: $j = 0$
7: $y[0] = 3 + 1$, 8: $y[1] = 3 - 1$.
9: Return [4, 2].

Coefficient to Point

$4 + 3x + 2x^2 + x^3$

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$

Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

$\mathrm{FFT}(P)$

1 Initialize $n = |P|; \omega = e^{\frac{2\pi i}{n}}$ and $y = [0]^n$
2 **if** $n == 1$
3 Return P
4 $P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$
5 $y_e = \mathrm{FFT}(P_e)$ and $y_o = \mathrm{FFT}(P_o)$
6 **for** $j = 0 \cdots n/2 - 1$
7 $y[j] = y_e[j] + \omega^j y_o[j]$
8 $y[j + n/2] = y_e[j] - \omega^j y_o[j]$
9 Return y

P = [4,3,2,1]

FFT(P):

1: n = 4; ω = i; y = [0,0,0,0].

2-3. 4: P_e = [4, 2] and P_o = [3, 1].

5: [6, 2] = FFT(P_e) & [4, 2] = FFT(P_0)

6: j = 0

7-8: y[0] = 6 + 4, y[2] = 6 - 4.

6: j = 1

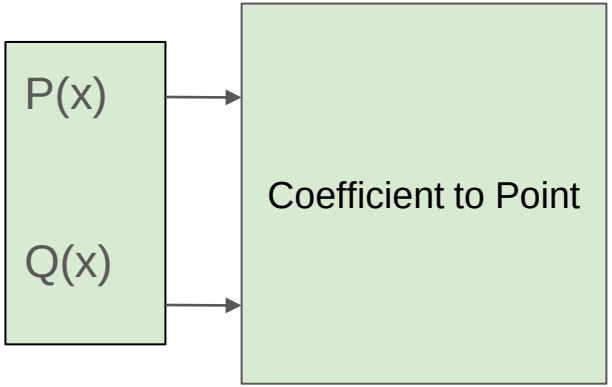
7-8: y[1] = 2 + 2i, y[3] = 2 - 2i

9: [10, 2+2i, 2-2i, 2]

$\omega^0, \omega^1, \omega^2, \omega^3$

Coefficient to Point

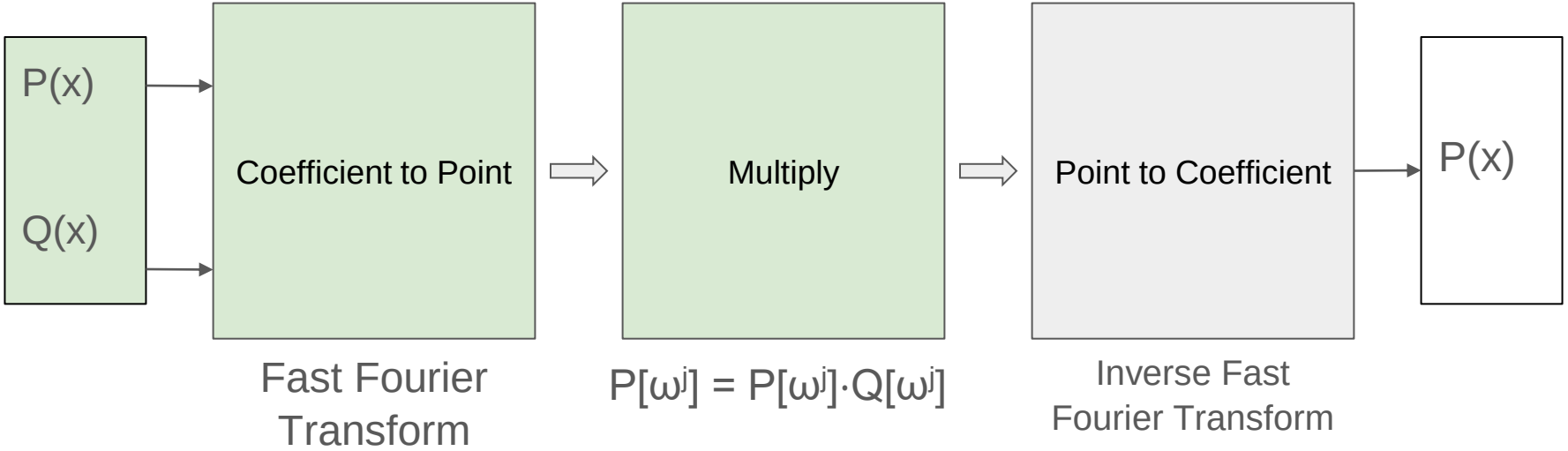
$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$



Inverse Fast Fourier Transform

Given point value representation of P(x) and Q(x) of degree d. Propose an efficient algorithm to multiply P(x) and Q(x).

$P(x) = P(x) \cdot Q(x)$



Point to Coefficient

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}^{-1}}_{\Downarrow} \begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

$$\frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Point to Coefficient

Replace every ω with 1/(n·ω)

Point to Coefficient

Input: $P = [p_0, p_1, \ldots, p_{n-1}]$

Output $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$

IFFT(P)

Initialize $n = |P|; \omega = e^{\frac{-2\pi i}{n}}$ and $y = [0]^n$
if $n == 1$
    Return P
$P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$
$y_e = \text{IFFT}(P_e)$ and $y_o = \text{IFFT}(P_o)$
for $j = 0 \cdots n/2 - 1$
    $y[j] = y_e[j] + \omega^j y_o[j]$
    $y[j + n/2] = y_e[j] - \omega^j y_o[j]$
Return y

$y = y/n.$

FFT(P)

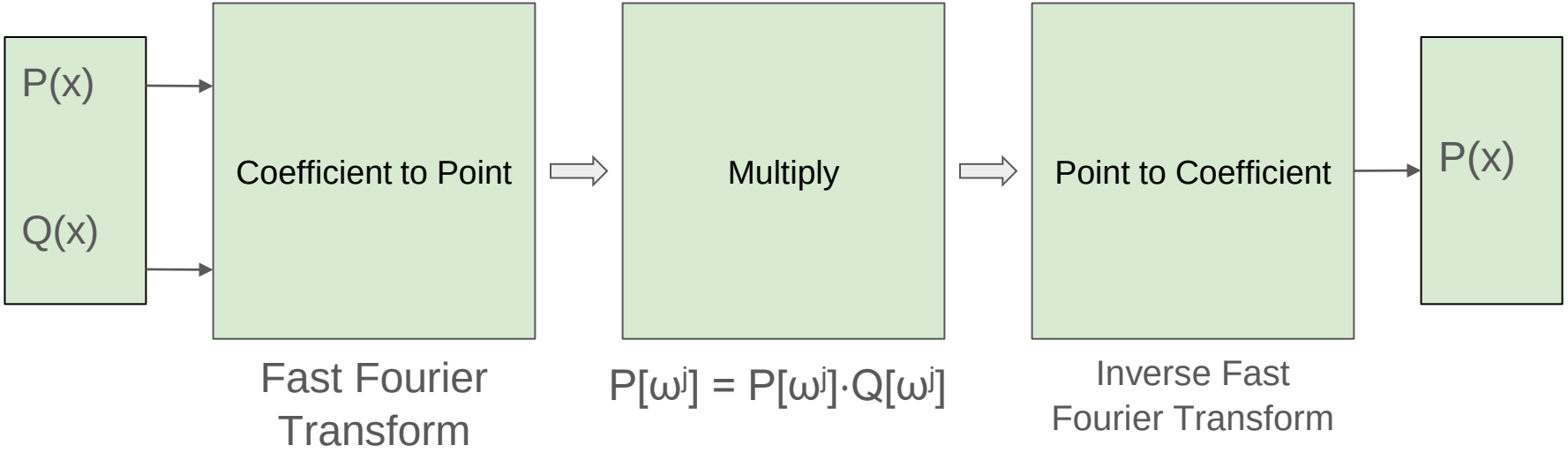
Initialize $n = |P|; \omega = e^{\frac{2\pi i}{n}}$ and $y = [0]^n$
if $n == 1$
    Return P
$P_e = [p_0, p_2, \ldots, p_{n-2}]$ and $P_o = [p_1, p_3, \ldots, p_{n-1}]$
$y_e = \text{FFT}(P_e)$ and $y_o = \text{FFT}(P_o)$
for $j = 0 \cdots n/2 - 1$
    $y[j] = y_e[j] + \omega^j y_o[j]$
    $y[j + n/2] = y_e[j] - \omega^j y_o[j]$
Return y

Observations

- Recurrence: $T(n) = 2T(n/2) + n$
- Running time $O(n \cdot \log(n))$

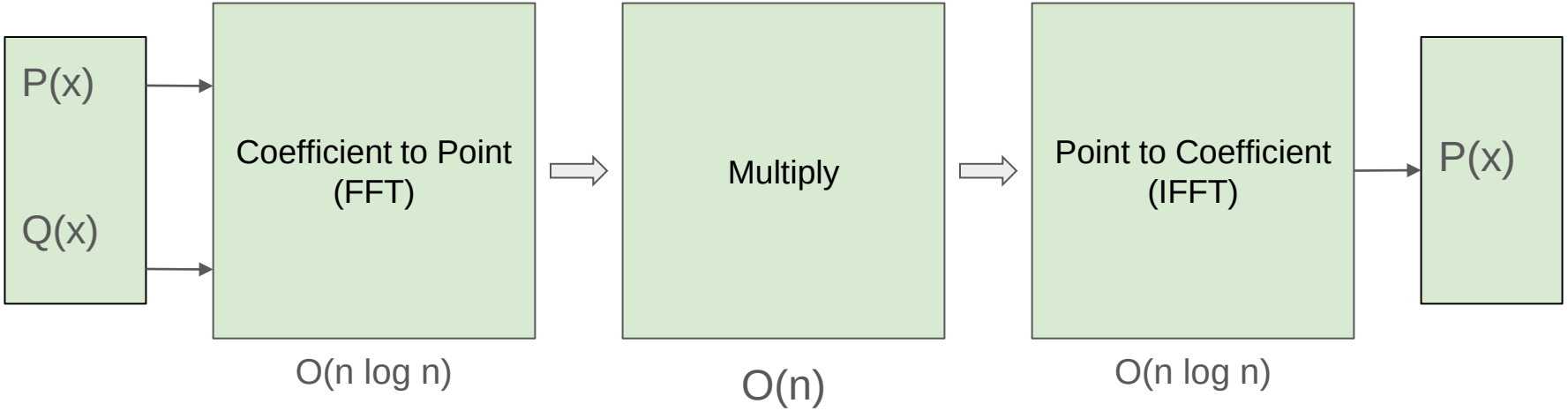
Inverse Fast Fourier Transform

Given point value representation of P(x) and Q(x) of degree d. Propose an efficient algorithm to multiply P(x) and Q(x).



Inverse Fast Fourier Transform

Given point value representation of P(x) and Q(x) of degree d. Propose an efficient algorithm to multiply P(x) and Q(x).



Outline

- Fast Fourier Transform (Contd.)

- Dynamic Programming

Dynamic Programming

Divide & Conquer: Breaks up problem into <u>independent</u> subproblem; solve each subproblem; combine them to get a solution for the original problem.

Dynamic Programming: Breaks up problem into a series of <u>overlapping</u> subproblem; combine solutions of smaller subproblem to to get a solution for the original problem.

Dynamic Programming

The term was coined by *Richard E. Bellman* in 1950s.

Fibonacci Series

Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Compute n^{th} fibonacci number recursively.

$F(n) = F(n-1) + F(n-2)$

$F(2) = 1$

$F(1) = 0$

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

Recurrence: $T(n) = T(n-1) + T(n-2) + 1$

Running time?

Fibonacci Series

Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Compute n^{th} fibonacci number recursively.

```
FIB(n)
if M[n] is not empty
        Return M[n]
if n == 0
else
        f = FIB(n − 1) + FIB(n − 2)
M[n] = f
Return f
```

Bottom up!

Reference

Slides

Algorithms Design by Kleinberg & Tardos - 5.6

Algorithm Design & Analysis (CSE222)

Lecture-8

Recap

- Fibonacci numbers

Recap

- Fibonacci numbers

M = {}

```
FIB(n)
if n == 1
    return 0
if n == 2
    return 1
if M[n] is not empty
    return M[n]
else
    M[n] = FIB(n − 1) + FIB(n − 2)
return M[n]
```

Recap

- Fibonacci numbers

- Collection of non-adjacent heavy set
  - Greedy approach need not be optimal

1, 5, 8, 5, 4, 7

Outline

- Collection of non-adjacent heavy set
  - Dynamic Programming

- Vertex Cover

Collection of non-adjacent heavy set

<u>Subproblem:</u>

MaxWt[k] = The weight of an optimal set $B^* \subseteq \{b_1, b_2, \ldots, b_k\}$ such that no two balls are adjacent to each other and their sum of weights is maximum.

$B^* = \{b_{i_1}, b_{i_2}, \ldots\} \cancel{b_k}\}$

OPT for $\{b_1, b_2 \ldots b_{k-2}\}$

If k = 0 then MaxWt[0] = 0.
If k = 1 then MaxWt[0] = b_1.

Claim: Let B* be a collection of non-adjacent balls and their sum is maximum. We call B* be the optimal solution for the set $\{b_1, b_2, \ldots, b_k\}$.

1. If $b_k \in B^*$, then $B^* \setminus \{b_k\}$ is an optimal solution for $\{b_1, b_2, \ldots, b_{k-2}\}$.
2. If $b_k \notin B^*$, then B* is an optimal solution for $\{b_1, b_2, \ldots, b_{k-1}\}$.

Proof by contradiction

- Identify the proposition (P) that has to be proved.
- Assume P is false, i.e., ~P is true.
- Check if, ~P being true implies falsehood, i.e., two mutually contradictory assertions (e.g., Q and ~Q both are true).
- Since, P being false leads to contradiction, hence P has to be true.

Proof:

$p \rightarrow q$

$(\sim p \cup q)$ $\quad$ $p \cap \sim q$

Claim: Let B* be a collection of non-adjacent balls and their sum is maximum. We call B* be the optimal solution for the set $\{b_1, b_2, \ldots, b_k\}$.

1. If $b_k \in B^*$, then $B^* \setminus \{b_k\}$ is an optimal solution for $\{b_1, b_2, \ldots, b_{k-2}\}$.

p $\quad$ q

<u>Proof:</u> If $b_k \in B^*$ then $b_{k-1} \notin B^*$.

, $B^* \setminus \{b_k\}$ is not opt. of $\{b_1, b_2, \ldots b_{k-2}\}$

C is opt of $(b_1, b_2, \ldots, b_{k-2})$

Proof:

Claim: Let B^* be a collection of non-adjacent balls and their sum is maximum. We call B^* be the optimal solution for the set $\{b_1, b_2, \ldots, b_k\}$.

1. If $b_k \in B^*$, then $B^* \setminus \{b_k\}$ is an optimal solution for $\{b_1, b_2, \ldots, b_{k-2}\}$.

<u>Proof:</u> If $b_k \in B^*$ then $b_{k-1} \notin B^*$. Suppose, $B^* \setminus \{b_k\}$ is not an optimal solution for the set $\{b_1, b_2, \ldots, b_{k-2}\}$. Let, $C \subseteq \{b_1, b_2, \ldots, b_{k-2}\}$ be optimal solution for the set.

$$W(C) = \sum_{b \in C} w(b) > \sum_{b \in B^* \setminus \{b_k\}} w(b)$$

$C \cup \{b_k\}$ vs B^*

$$W(C \cup \{b_k\}) = \sum_{b \in C} w(b) + w(b_k) > \sum_{b \in B^* \setminus \{b_k\}} w(b) + w(b_k)$$

Proof:

Claim: Let B^* be a collection of non-adjacent balls and their sum is maximum. We call B^* be the optimal solution for the set $\{b_1, b_2, \ldots, b_k\}$.

1. If $b_k \in B^*$, then $B^* \setminus \{b_k\}$ is an optimal solution for $\{b_1, b_2, \ldots, b_{k-2}\}$.

<u>Proof:</u> If $b_k \in B^*$ then $b_{k-1} \notin B^*$. Suppose, $B^* \setminus \{b_k\}$ is not an optimal solution for the set $\{b_1, b_2, \ldots, b_{k-2}\}$. Let, $C \subseteq \{b_1, b_2, \ldots, b_{k-2}\}$ be optimal solution for the set.

Compare $W(C)$ with $W(B^* \setminus \{b_k\})$

Proof:

Claim: Let B* be a collection of non-adjacent balls and their sum is maximum. We call B* be the optimal solution for the set $\{b_1, b_2, \ldots, b_k\}$.

1. If $b_k \in B^*$, then $B^* \setminus \{b_k\}$ is an optimal solution for $\{b_1, b_2, \ldots, b_{k-2}\}$.
2. If $b_k \notin B^*$, then B* is an optimal solution for $\{b_1, b_2, \ldots, b_{k-1}\}$.

$\{b_1, b_2 \ldots b_k\}$

(B^*)

$B^* \subseteq B_k$

$B^* \subseteq B_{k-1}$

Proof:

Claim: Let B* be a collection of non-adjacent balls and their sum is maximum. We call B* be the optimal solution for the set $\{b_1, b_2, \ldots, b_k\}$.

1. If $b_k \in B^*$, then $B^* \setminus \{b_k\}$ is an optimal solution for $\{b_1, b_2, \ldots, b_{k-2}\}$.
2. If $b_k \notin B^*$, then B* is an optimal solution for $\{b_1, b_2, \ldots, b_{k-1}\}$.

<u>Proof:</u> If $b_k \notin B^*$ then $B^* \subseteq \{b_1, b_2, \ldots, b_{k-1}\}$. Suppose, $C \subseteq \{b_1, b_2, \ldots, b_{k-1}\}$ be an optimal solution set.

Compare W(C) with W(B*)

$w(k) =$

Recurrence

For k > 1, $\mathrm{MaxWt}[k] = \max\{\mathrm{MaxWt}[k-1], w(k) + \mathrm{MaxWt}[k-2]\}$

s_1 s_2 s_3 s_4 s_5 - s_6

Example, 1, 5, 8, 5, 4, 7

$M(0) = 0$, $M(1) = 1$

$M(2) = \max(1, 5+0) = 5$

$M(3) = \max(5, 8+1) = 9$

$M(4) = \max(9, 5+5) = 10$

Algorithm

Input: Weights of Balls
Output: Maximum Weight of non-adjacent set.

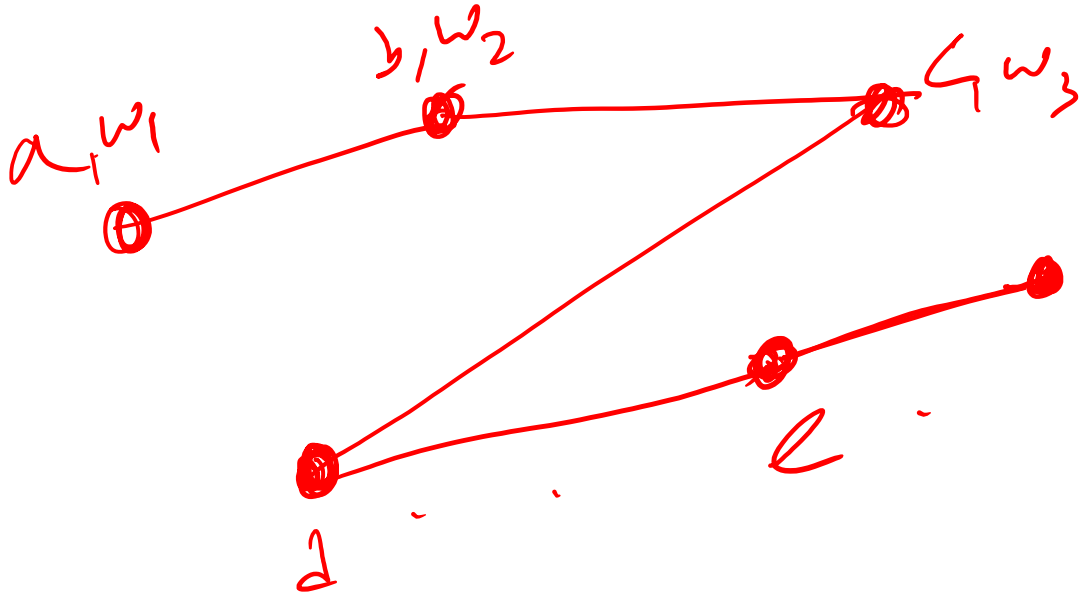
Running time: O(|W|)

```
MaximumWeight(W)
MaxWt[0] = 0; MaxWt[1] = W[1];
for i = 2 ··· |W|
        t1 = W[i] + MaxWt[i − 2];
        t2 = MaxWt[i − 1];
        if (t1 > t2)
                MaxWt[i] = t1;
        else
                MaxWt[i] = t2;
return MaxWt[|W|]
```

Independent Set

For a given graph G = (V,E), a set of vertices S ⊆ V(G) is independent if for every pair u, v ∈ S, (u, v) ∉ E(G).

Question: How can we efficiently compute the maximum weighted independent set in a path graph?



Outline

- Collection of non-adjacent heavy set
  - Dynamic Programming

- Vertex Cover

Vertex Cover

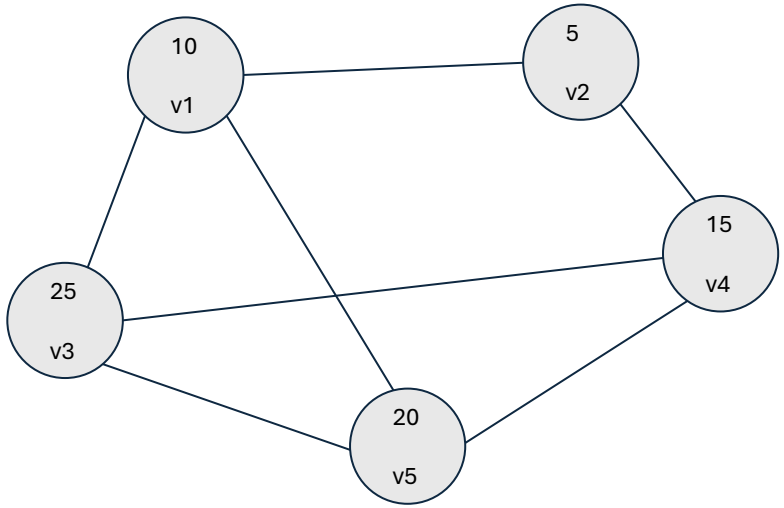
Given an undirected graph G = (V, E), a set of vertices S ⊆ V(G) is a vertex cover if for (u, v) ∈ E(G), u ∈ S or v ∈ S (or both).

In a weighted graph (weights on vertices) G = (V, E, Wt), a vertex cover S is called the minimum weighted vertex cover of G if Wt(S) is minimum among all possible vertex covers.

{V1, V2, V4}?

{V2, V3, V5}? = 50

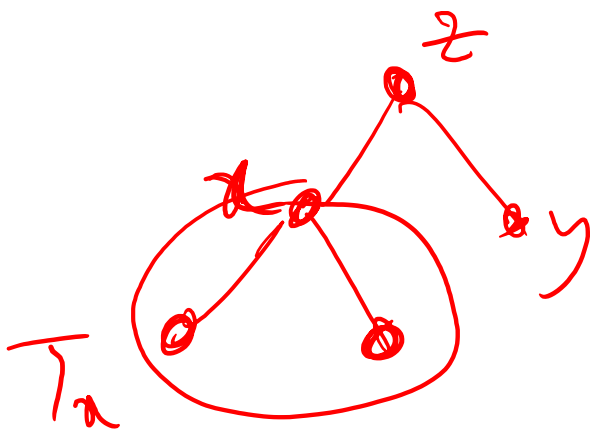
{V1, V4, V5}? = 45



Vertex Cover

Input: Weighted rooted tree. T = (V, E, Wt)
Output: Find vertex cover with minimum total weight.

Assume its a binary tree. For every vertex $x \in V(T)$, T_x be a subtree rooted at x.



Vertex Cover

Input: Weighted rooted tree. T = (V, E, Wt)
Output: Find vertex cover with minimum total weight.

Assume its a binary tree. For every vertex $x \in V(T)$, T_x be a subtree rooted at x.
For every $x \in V(T)$ define following subproblems

- MVC(x,1): Weight of a minimum weight vertex cover of T_x that contains x.
- MVC(x,0): Weight of a minimum weight vertex cover of T_x that does not contains x.
- VC(x): Minimum weight vertex cover of T_x.

Define VC(x): min (MVC(x,1), MVC(x,0))

Subproblem of VC

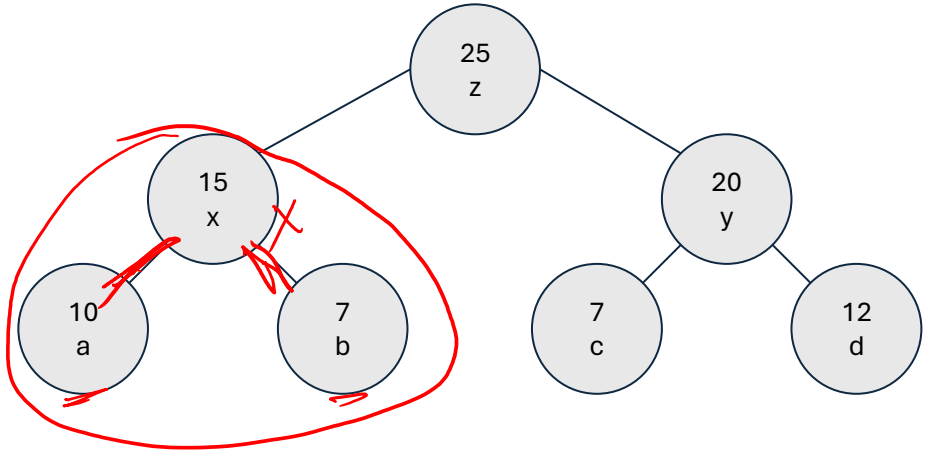
$MVC(x, 0) = 17$

$MVC(x, 1) = 15$, $VC(x) = 15$

$MVC(y, 1) = 20$

$VC(y) = 19$

$MVC(y, 0) = 19$

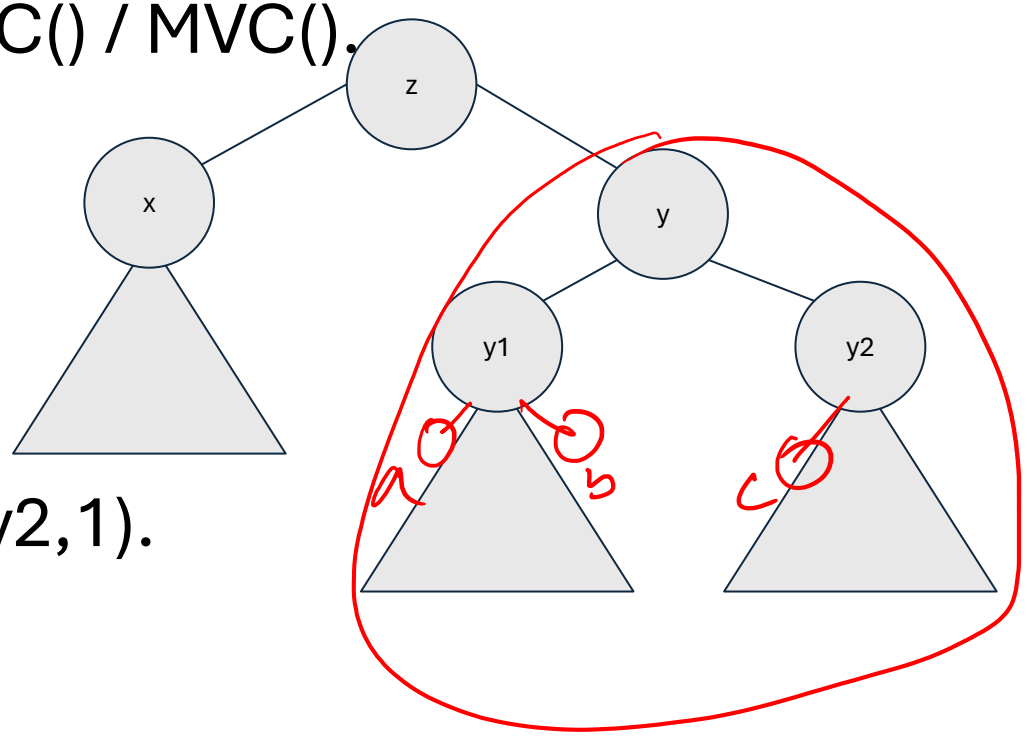


VC(x)?

Recurrence of Subproblem

How to represent MVC(y) using its child's VC() / MVC().

MVC(y,0) =



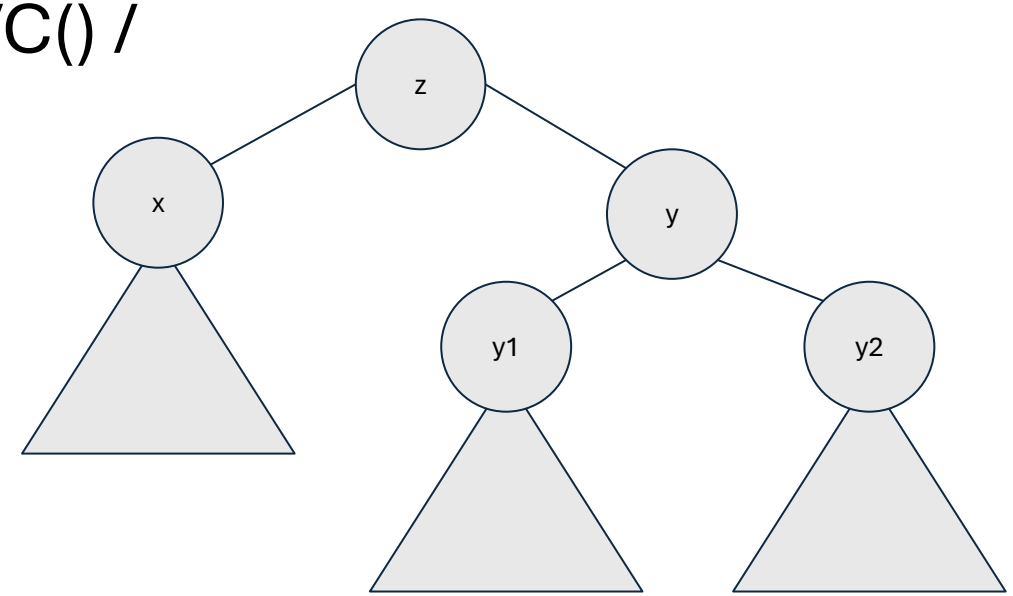
MVC(y,0) must account MVC(y1,1) & MVC(y2,1).

Recurrence of Subproblem

How to represent MVC(y) using its child's VC() / MVC().

$$\mathrm{MVC}(y,0) = \mathrm{MVC}(y1,1) + \mathrm{MVC}(y2,1)$$

$$\mathrm{MVC}(y,1) = Wt(y) + \min \begin{cases} \mathrm{MVC}(y1,0) + \mathrm{MVC}(y2,0), \\ \mathrm{MVC}(y1,0) + \mathrm{MVC}(y2,1), \\ \mathrm{MVC}(y1,1) + \mathrm{MVC}(y2,0) \end{cases}$$



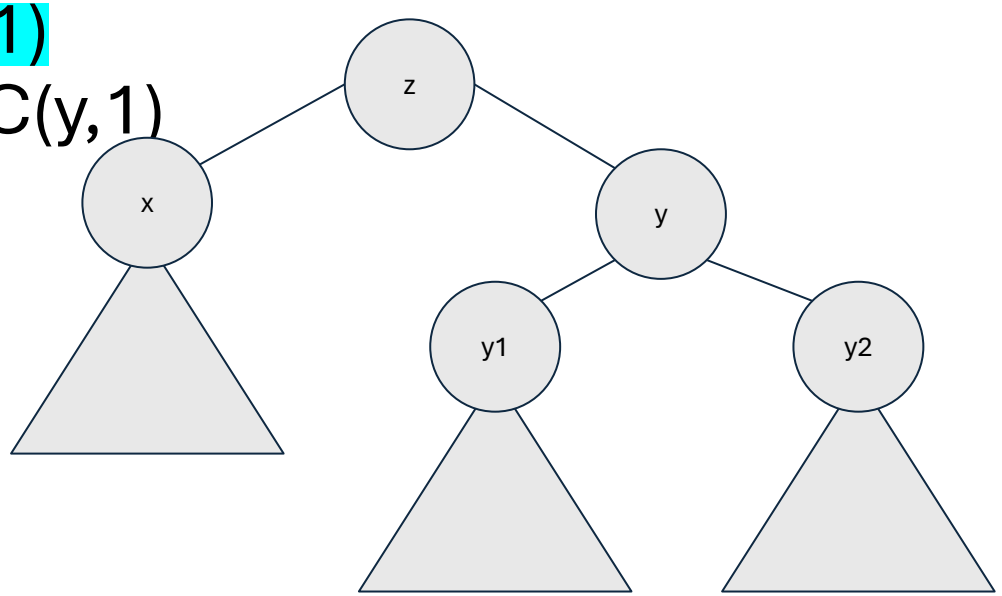
If y is a leaf then, MVC(y,0) = 0 & MVC(y,1) = Wt(y).

MVC(y , 0) means minimum vertex cover not including y
MVC(y , 1) means minimum vertex cover including y

For VC(z), compute MVC(z,0) & MVC(z,1)
For MVC(z,0), compute MVC(x,1) & MVC(y,1)

Algorithm

- For VC(z), compute MVC(z,0) & MVC(z,1)
- For MVC(z,0), compute MVC(x,1) & MVC(y,1)
- So on…

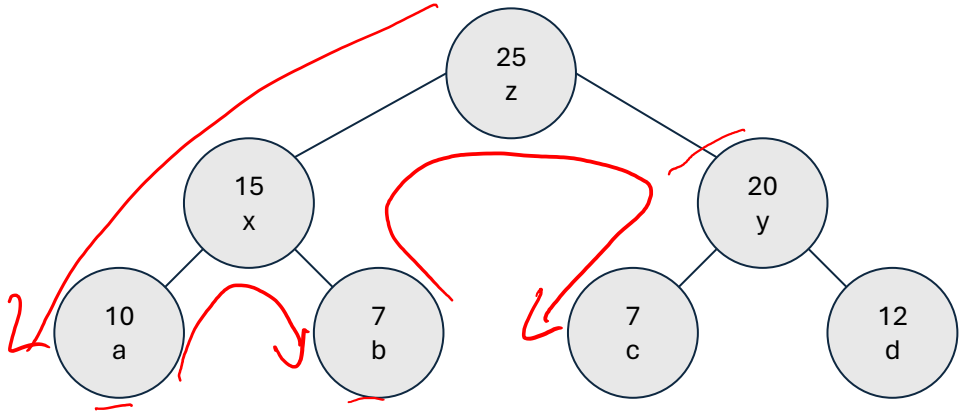


Use postorder traversal.

Vertex Cover

$$\min \left\{ \begin{array}{l} MVC(x,0) + MVC(y,0) \\ MVC(x,1) + MVC(y,0) \\ MVC(x,0) + MVC(y,1) \end{array} \right\} + w(z)$$

	a	b	x	c	d	y	z
0	0	0	17	0	0	19	35
1	10	7	15	2	7	20	59



Reference

Slides

Algorithm Design & Analysis (CSE222)

Lecture-9

Recap

- Collection of non-adjacent heavy set.

- Vertex cover for Tree

Outline

- Matrix Chain multiplication

- Longest common subsequence

Matrix multiplication

$$\begin{bmatrix} 1 & 1 & 1 \\ 3 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} -1 & 2 \\ 2 & 2 \\ 1 & -3 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 2 & 7 \end{bmatrix}$$

$\textsc{MatrixProduct}(A, B)$

```
Initialize p ← # rows in A
Initialize q ← # columns in A
Initialize r ← # columns in B
Initialize C ← {0}^{p×r}
for i = 1 ··· p
    for j = 1 ··· r
        for k = 1 ··· q
            C[i, j] = C[i, j] + A[i, k] · B[k, j]
return C
```

#rows in the first matrix must be equal to the #columns in the second matrix.

Running time: O(pqr)

Matrix multiplication

Let A, B and C be three matrices of dimensions 5 x 3, 3 x 8 and 8 x 2 respectively.

Compute $M = A \cdot B \cdot C$.

Let $T = A \cdot B$, then $M = T \cdot C = (A \cdot B) \cdot C$
Running time:

Let $T = B \cdot C$, then $M = A \cdot T = A \cdot (B \cdot C)$
Running time:

Matrix Chain Multiplication

Input: Sequence of matrices $A_1, A_2, \ldots, A_n$
Problem: Running time of fastest matrix chain multiplication.

Trial-1: Check all possible parenthesization of input.
Let P(k) be the all possible parenthesization of an input of k matrices.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

Check which is the most efficient.

$$|P(n)| = \frac{\binom{2n}{n}}{n+1} = \Omega(2^n)$$

Matrix Chain Multiplication

Input: Sequence of matrices $A_1 \in R^{p0xp1}$, $A_2 \in R^{p1xp2}$, $A_3 \in R^{p2xp3}$, ..., $A_n \in R^{pn-1xpn}$.

Problem: Running time of fastest matrix chain multiplication.

To optimally multiply $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

- Parenthesize $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$, let the split is between A_k and A_{k+1}.
- Compute optimal parenthesization of $(A_i \cdot A_{i+1} \cdot \ldots \cdot A_k)$ & $(A_{k+1} \cdot A_{k+2} \cdot \ldots \cdot A_j)$.
- Subproblem: M[i, j] is the minimum number of scalar multiplication needed to compute $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$. For $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ we look at m[1, n].
  - When i == j then it is trivial!
  - If optimal split between A_k and A_{k+1} then $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \ . \end{cases}$$

Pseudocode

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Running time?

$O(n^3)$

Example

Input: Sequence of matrices $A_1 \in R^{2x3}$, $A_2 \in R^{3x6}$, $A_3 \in R^{6x2}$, $A_4 \in R^{2x9}$, $A_5 \in R^{9x3}$.
Problem: Running time of fastest matrix chain multiplication.

Maintain following tables to solve the problem (bottom up).

	1	2	3	4	5
1					
2					
3					
4					
5					

Maintain multiplication cost for a subset of matrices

	1	2	3	4	5
1					
2					
3					
4					
5					

Maintain multiplication order

Table Filling (or Bottom Up)

Start with simplest subproblem.

Gradually reach to the solution of original problem.

Does not require recursive function call.

Solves subproblems which might not even necessary for solving the original problem.

Recursion

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Recurrence:

$$
\begin{aligned}
T(1) &\geq 1 , \\
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1
\end{aligned}
$$

Recursive running time

$$
\begin{aligned}
T(1) &\geq 1 , \\
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1
\end{aligned}
$$

Memoization (or Top Down)

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length − 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Reference

Slides

Introduction to Algorithms by CLRS - Chp-15.2 & 15.3

Algorithm Design & Analysis (CSE222)

Lecture-10

Recap

- Matrix Chain multiplication
  - $A \in R^{5x3}$, $B \in R^{3x8}$ and $C \in R^{8x2}$.
  - $A \cdot (B \cdot C)$ is more efficient than $(A \cdot B) \cdot C$
- Table Filling (Bottom Up)

- Memoization (Top Down)

Outline

- Longest Common Subsequence
- Edit Distance
- Maximum Job

Computational Biology

DNA (consists of base) sequencing has played crucial role in computational biology.

- Gene identification, prediction
- Gene variations, uncovers evolution
- Personalized medications

Problem

Given two DNA sequences, compare the similarity between them.

- Check if two sequences are equal.
- Check if one sequence is a substring of another sequence.
- Find the longest sequence such that the bases in it are also present in the original sequence in the same order.

Example S_1 = {a**a**g**ct**c**g**} and S_2 = {g**act**a**g**} S_3 = {actg}

Note that S_3 is not a continuous sequence in either S_1 or S_2.

Longest Common Subsequence

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a *subsequence* of X if there is an increasing sequence <i1, i2, ..., ik> of X such that, $x_{ij} = z_j$ for every $j = \{1, 2, \ldots, k\}$.

$X = \langle a\ b\ a\ c\ d \rangle$

$Z = \langle b\ c\ d \rangle$

$Z = \langle a\ a\ c \rangle$

$Z = \langle d\ c \rangle$

Longest Common Subsequence

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, a sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a <u>*subsequence*</u> of X if there is an increasing sequence <i1, i2, ..., ik> of X such that, $x_{ij} = z_j$ for every $j = \{1, 2, \ldots, k\}$.

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, i^{th} prefix of X is $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ for $i = \{0, 1, 2, \ldots, m\}$.

Given two sequences X and Y, Z is a <u>*common subsequence*</u> if it is a subsequence of both X and Y.

Longest such common subsequence is called the <u>*longest common subsequence*</u>.

Given two sequences X and Y, find its longest common subsequence.

Problem

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find its longest common subsequence.

<u>Brute force</u>

1. If $m < n$, then go over every possible subset of X and return the longest subset that is a subsequence of Y.
2. Else, then go over every possible subset of Y and return the longest subset that is a subsequence of X.

Running time: $\min(2^m, 2^n)$

Claim

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be its longest common subsequence.

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is the LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is the LCS of X_{m-1} and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is the LCS of X and Y_{n-1}.

Proof

Given two sequences $X = <x_1, x_2, ..., x_m>$ and $Y = <y_1, y_2, ..., y_n>$, let $Z = <z_1, z_2, ..., z_k>$ be its longest common subsequence.

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is the LCS of X_{m-1} and Y_{n-1}.

Proof by contradiction

- Let z_k of Z is not equal to x_m.
- Since, x_m and y_n are the equal (by assumption), so we can append x_m with Z and get LCS of X and Y of length k+1.
- This contradicts that $Z = <z_1, z_2, ..., z_k>$ is the LCS.
- Let W be LCS of X_{m-1} and Y_{n-1} such that it is greater than k-1, then we can append x_m with W and get LCS of X and Y of length k (Contradiction).
- Hence, Z_{k-1} is LCS of X_{m-1} and Y_{n-1}.

Proof

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be its longest common subsequence.

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is the LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is the LCS of X_{m-1} and Y.

Proof by contradiction

- Since, $z_k \neq x_m$ so at most z_k is equal to x_{m-1} and y_n.
- Let W be LCS of X_{m-1} and Y such that it is greater than k, W is also LCS of X and Y of length k (Contradiction).
- Hence, Z is LCS of X_{m-1} and Y.

Proof

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be its longest common subsequence.

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is the LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is the LCS of X_{m-1} and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is the LCS of X and Y_{n-1}.

Proof by contradiction – same as previous case.

Recursive Solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Running time:

Table filling or Memoization?

Example

X = <a, b, c, a, c, d, b, c> and Y = <b, a, c, a, d, b, c, d, c>

c) b

		b	a	c	a	d	b	c	d	c
	0	0	0	0	0	0	0	0	0	0
a	0	←0	↖1	←1	↖1	←1	←1	←1	←1	←1
b	0	1	←1	←1	←1	←1	2	←2	←2	←2
c	0	↑1	←1	2	←2	←2	←2	3	←3	3
a	0	↑1	2	←2	3	←3	←3	←3	←3	←3
c	0	↑1	↑2	3	←3	←3	←3	4	←4	4
d	0									
b	0									
c	0									

X

j ↑

bca

Z = bcac

Outline

- Longest Common Subsequence
- Edit Distance
- Maximum Job

Levenshtein (Edit) Distance

Given two strings, <u>edit distance</u> measures the number of operations needed to transform one string into another.

<u>Operations</u>

- Insert
- Replace
- Delete

Input: ‘horse’ and ‘rose’
Replacing ‘h’ with ‘r’ results to ‘rorse’
Deleting ‘r’ results to ‘rose’.

Edit Distance

Input: A = horse and B = rose

<u>Case-1:</u> Do nothing
If A[5] equal to B[4]
then transform A[0-4] → B[0-3].

<u>Case-2:</u> Insert
If A[3] not equal to B[3]
then transform A[0-3] → B[0-2].

<u>Case-3:</u> Replace
If A[3] not equal to B[3]
then transform A[0-2] → B[0-2].

<u>Case-4:</u> Delete
If A[3] not equal to B[3]
then transform A[0-2] → B[0-3].

Example

Input: A = horse and B = rose

Maintain the following table

Case 1

case 4: delete 'r'

Case 3: replace 'h' with r

edit distance

	‘ ’	h	o	r	s	e
‘ ’	0	1	2	3	4	5
r	1	1	2	2	3	4
o	2	2	1	2	3	4
s	3	3	2	2	2	3
e	4	4	3	3	3	2

Outline

- Longest Common Subsequence
- Edit Distance
- Maximum Job

Maximum Job

There are two machines A and B. At any given minute we can run job in either machine A or machine B. At ith minute we can run ai steps in machine A or bi steps in machine B. We can move jobs from one machine to another but it will cost us one minute, i.e., no processing for one minute.

Goal: Design an algorithm that determines the maximum number of steps that can be executed in ‘n’ minutes.

Subproblem

Cost(k, A) = The maximum number of steps that can be executed in minutes {1, 2, ..., k} such that machine A executes instruction at minute k.

Cost(k, B) = The maximum number of steps that can be executed in minutes {1, 2, ..., k} such that machine B executes instruction at minute k.

$$\text{cost}(k, A) = \max \begin{cases} a_k + \text{cost}(k-1, A) \\ a_k + \text{cost}(k-2, B) \end{cases}$$

$$\text{cost}(k, B) = \max \begin{cases} b_k + \text{cost}(k-1, B) \\ b_k + \text{cost}(k-2, A) \end{cases}$$

Base Cases

Cost(1, A) = a_1

Cost(1, B) = b_1.

Cost(2, A) = $a_1 + a_2$

Cost(2, B) = $b_1 + b_2$.

Solving Final Problem

$$\max \begin{cases} \text{cost}(n, A), \\ \text{cost}(n, B) \end{cases}$$

Algorithm

$\textsc{Maximum Job}(A, B, n)$

$\text{cost}[1,1] = a_1;\ \text{cost}[1,2] = b_1$

$\text{cost}[2,1] = a_1 + a_2;\ \text{cost}[2,2] = b_1 + b_2$

for $i = 3 \cdots n$

$$\text{cost}(i, A) = \max \begin{cases} a_i + \text{cost}(i-1, A), \\ a_i + \text{cost}(i-2, B) \end{cases}$$

$$\text{cost}(i, B) = \max \begin{cases} b_i + \text{cost}(i-1, B), \\ b_i + \text{cost}(i-2, A) \end{cases}$$

return $\max \begin{cases} \text{cost}(n, A), \\ \text{cost}(n, B) \end{cases}$

Reference

Slides

Introduction to Algorithms by CLRS - Chp-15.4

Algorithm Design & Analysis (CSE222)

Lecture-11

Announcement

Raise assignment-1 concerns via google form in gc by today midnight.

Recap

- Longest Common Subsequence
- Maximum Job
- Edit Distance

Outline

- Edit Distance

- Weighted Subset

Levenshtein (Edit) Distance

Given two strings, <u>edit distance</u> measures the number of operations needed to transform one string into another.

<u>Operations</u>

- Insert
- Replace
- Delete

Input: ‘horse’ and ‘rose’
Replacing ‘h’ with ‘r’ results to ‘rorse’
Deleting ‘r’ results to ‘rose’.

Edit Distance

Input: A = horse and B = rose

<u>Case-1:</u> Do nothing
If A[5] equal to B[4]
then transform A[0-4] → B[0-3].

hors ros

<u>Case-2:</u> Insert
If A[3] not equal to B[3]
then transform A[0-3] → B[0-2].

A[0-3] → B[0-3]

hor → ro r

<u>Case-3:</u> Replace
If A[3] not equal to B[3]
then transform A[0-2] → B[0-2].

hor → ros

ho → ro

<u>Case-4:</u> Delete
If A[3] not equal to B[3]
then transform A[0-2] → B[0-3].

hor → ros

ho → ros

Edit Distance

<u>Subproblem:</u> EDist(A, B, i, j): Compute the edit distance to transform A[0-i]→B[0-j].

<u>Recurrence:</u> $$\mathrm{EDist}(A, B, i, j) = \begin{cases} \mathrm{EDist}(A, B, i-1, j-1) & \text{if } A[i] == B[j] \\ 1 + \min \begin{cases} \mathrm{EDist}(A, B, i-1, j-1), \\ \mathrm{EDist}(A, B, i-1, j), \\ \mathrm{EDist}(A, B, i, j-1) \end{cases} & \text{else} \end{cases}$$

<u>Final Solution:</u> EDist(A, B, |A|, |B|)

Edit Distance

$= ED(A, B, |A|, |B|)$

Initialize global matrix $ed = \{-1\}^{|A| \times |B|}$

```
EDIT-DISTANCE(A, B, i, j)
if (A == " ")
    return j;
if (B == " ")
    return i;
if (ed(i, j) =! -1)
    return ed(i, j);
else
    if (A[i] == B[j])
        return ed(i, j) = EDIT-DISTANCE(A, B, i-1, j-1);
    else
        insert = EDIT-DISTANCE(A, B, i, j-1)
        replace = EDIT-DISTANCE(A, B, i-1, j-1)
        delete = EDIT-DISTANCE(A, B, i-1, j)
        ed[i, j] = 1 + min{insert, replace, delete}
        return ed[i, j]
```

Table filling or Memoization?

<u>Subproblem:</u> EDist(A, B, i, j): Compute the edit distance to transform A[0-i]→B[0-j].

<u>Recurrence:</u>
$$\mathrm{EDist}(A, B, i, j) = \begin{cases} \mathrm{EDist}(A, B, i-1, j-1) & \text{if } A[i] == B[j] \\ 1 + \min \begin{cases} \mathrm{EDist}(A, B, i-1, j-1), \\ \mathrm{EDist}(A, B, i-1, j), \\ \mathrm{EDist}(A, B, i, j-1) \end{cases} & \text{else} \end{cases}$$

<u>Final Solution:</u> EDist(A, B, |A|, |B|)

Outline

- Edit Distance

- Weighted Subset

Weighted Subset

Let there are n items {1, 2, ..., n} and a weight function w: [n] → $R_{>0}$. Let W > 0.

Goal: Find subset S ⊆ {1, 2, ..., n} such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximum.

Trial-1:
Let n = 3 with weights W/2, W/2+1, W/2.
Ensure the maximum possible weighted item in the subset.
Sort the item by weights (say W/2+1, W/2, W/2).

Trial-2:
Sort the item by weights (say W/2, W/2, W/2+1) (It worked!).

Let n = 3 with weights 1, W/2, W/2?

Weighted Subset

$\{ w_{/2+1}, w_{/2}, w_{/2} \}$

Subproblem: Wt(i): Optimum weight from first i elements.

Recurrence (handle cases): At iteration n

Wt(n-1)

If n not in optimum then Wt(n) = Wt(n-1)

Else

Wt(n) = w_n + Wt(n-1) ?

Weighted Subset

Subproblem: Wt(i,T): Optimum weight from first i elements with weight limited to T.

Recurrence (handle cases): At iteration n
If n not in optimum then Wt(n, W) = Wt(n-1, W)
Else
Wt(n,T) = w_n + Wt(n-1, W - w_n).

$$Wt(i, w) = \max\{Wt(i-1, w), Wt(i-1, w - w_i) + w_i\}$$

Final Solution: Wt(n, W)

Algorithm

```
SUBSET-SUM(W, {w_1, w_2, ..., w_n})
for i = 0 ··· n
        Wt[i, 0] = 0
for i = 0 ··· W
        Wt[0, i] = 0
for i = 1 ··· n
        for j = 1 ··· W
                if w_i < j
                        Wt[i, j] = max{Wt(n − 1, w), Wt(n − 1, w − w_n) + w_n}
                else
                        Wt[i, j] = Wt[i − 1, j]
return Wt[n, W]
```

Running time: O(nW)

Memoization?

Problem

Let there are n items {1, 2, ..., n}, a weight function w: [n] → $R_{>0}$ and a value function v: [n] → $R_{>0}$. Let W > 0.

Goal: Find subset S ⊆ {1, 2, ..., n} such that $\sum_{i \in S} w_i \leq W$ and the value of the subset if maximum.

Solution

Define the following

- Subproblem
- Recurrence
- Final Solution
- Algorithm and its running time.

Reference

Slides

Algorithm design by Kleinberg & Tardos - Chp-6.4

Algorithm Design & Analysis (CSE222)

Lecture-12

Recap

- Edit Distance

- Weighted Subset
  - 0/1 Knapsack

0/1 Knapsack

Let there are n items {1, 2, ..., n}, a weight function w: [n] → $R_{>0}$ and a value function v: [n] → $R_{>0}$. Let W > 0.

Goal: Find subset S ⊆ {1, 2, ..., n} such that $\sum_{i \in S} w_i \leq W$ and the value of the subset if maximum.

0/1 Knapsack

Subproblem: Val(i,T): Maximum value from first i elements with weight limited to T.

Recurrence: $Val(i,T) = \max\{Val(i-1,T), Val(i-1,T-w_i) + v_i\}$

Final Solution: Val(n, W)

Running time?

Recurrence Problem

Is the following relation is true?

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Prove without showing LHS = RHS.

Outline

- Graphs

- Depth First Search

Graph Notations

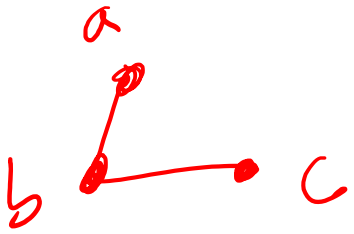
Graphs: Captures relationship (a.k.a edges) between objects (a.k.a nodes).
Consists of collection of nodes (a.k.a vertices) V and a collection of edges E (sets of nodes).

$G = (V, E)$

<u>Simple Graphs</u>

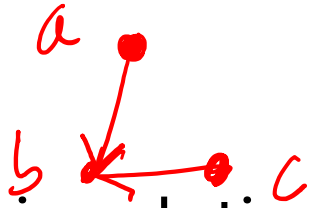
- At most one edge between pair of vertices (i.e., each edge is a pair of nodes).
- Undirected (or symmetric) and unweighted edges.
- No self loops.

Graph Notations



$(a,b) / (b,a) \in E$

Undirected graph for symmetric relationship, $G = (V, E)$. Every $e \in E$ is not an ordered pair (u, v).



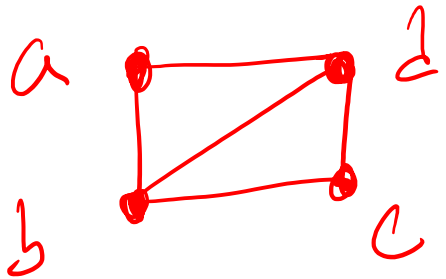
$(a,b) \in E$

$(b,a) \notin E$

Directed graph for asymmetric relationship, $G = (V, E)$. Every $e \in E$ is an ordered pair (u, v).

Graph Notations

Given a graph $G = (V, E)$, a **<u>path</u>** p is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that each pair of consecutive vertices (v_i, v_{i+1}) is an element in E.
A **<u>simple path</u>** is a sequence of vertices, where none of them are repeated.



paths

a, b, c ✓

b c d ✓

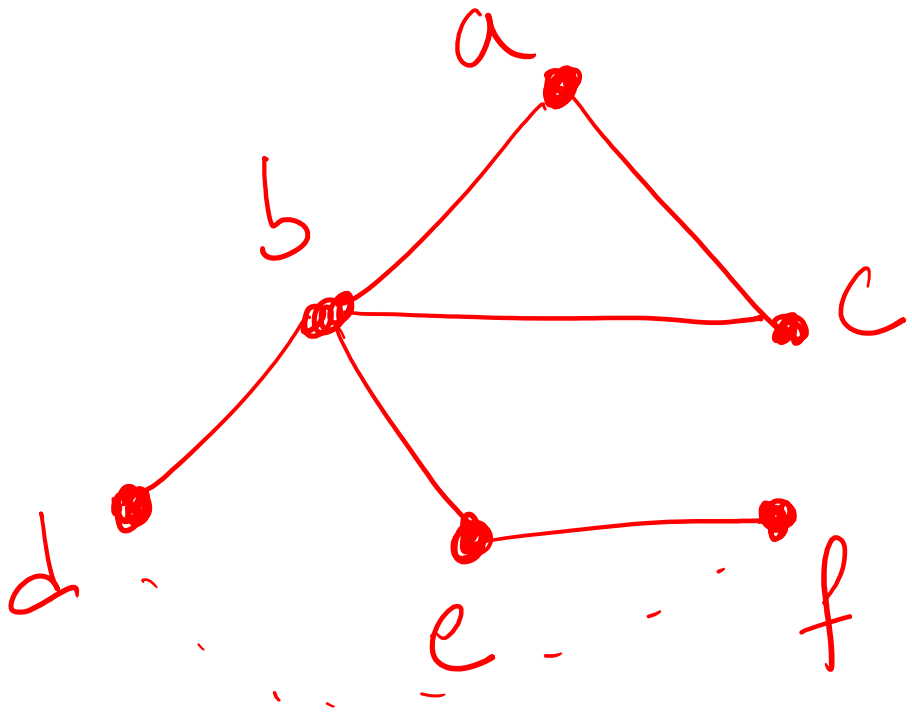
a c b ✗

b c d b

Graph Notations

Given a graph $G = (V, E)$, a **path** p is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that each pair of consecutive vertices (v_i, v_{i+1}) is an element in E.
A **simple path** is a sequence of vertices, where none of them are repeated.
A **cycle** is a path if all the vertices are unique except for the first and the last.
A graph is called **connected** if there is a path between every pair of vertices.
A **tree** is a connected graph without any cycle.
A directed graph with no cycle is called **directed acyclic graph** (DAG).
The shortest path length from u to v is called the **distance** from u to v, i.e., $\delta_G(u, v)$.
The maximum distance between any pair of vertices is the **diameter** of a graph.
Eccentricity of a vertex u $\{e(u)\}$ is maximum distance of any other vertex v from u.
The node with the smallest eccentricity is the **center** and its value is the **radius** of the graph.

Examples



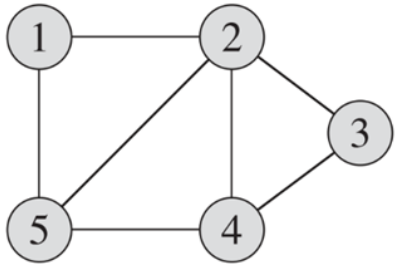
dia = 3

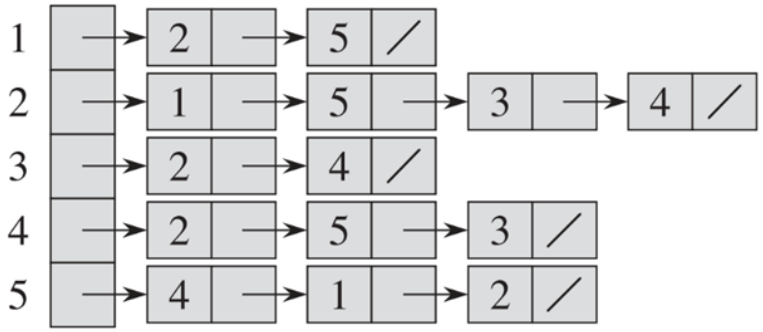
ecc(d) = 3

center = b

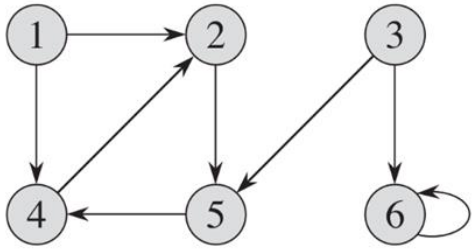
radius = 2

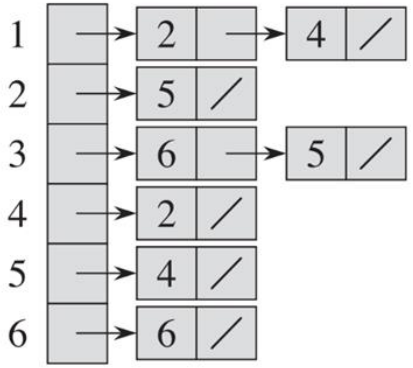
Graph Representation





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





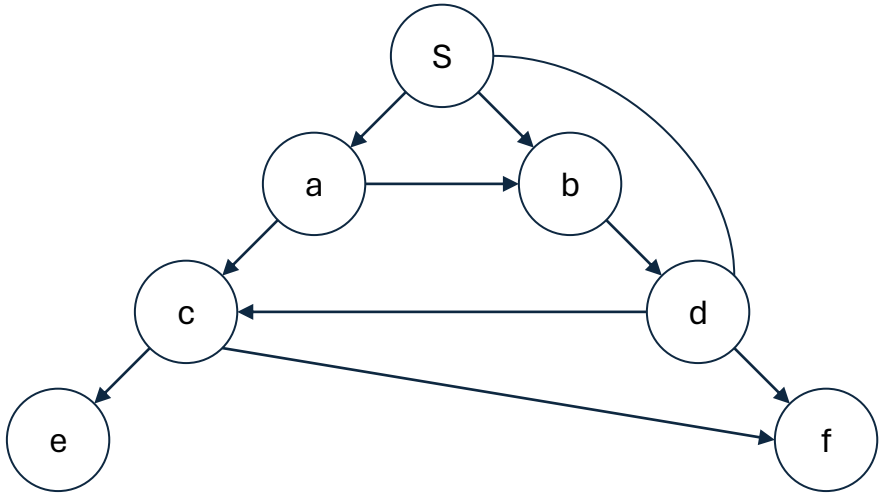
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Graph Properties

A graph G = (V, E) is sparse if $|E| \ll |V|^2$.

Topological Sort of a DAG is a total ordering $v_1 < v_2 < \ldots < v_n$ of vertices in V such that for any edge $(v_i, v_j) \in E$, $j > i$.

If edges of DAG represents dependencies, then topological sort follows all dependencies.



Depth First Search

```
ITERATIVE-DFS(s)
Push(s)
while Stack is not empty
    u ← Pop()
    if u is unmarked
        mark u
        for every edge (u, v)
            Push(v)
```

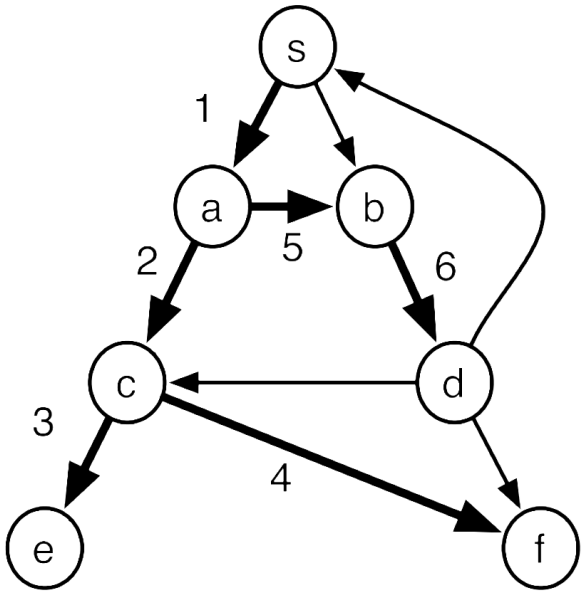
Graph Properties

A graph G = (V, E) is sparse if $|E| \ll |V|^2$.

Topological Sort of a DAG is a total ordering $v_1 < v_2 < \ldots < v_n$ of vertices in V such that for any edge $(v_i, v_j) \in E$, $j > i$.

If edges of DAG represents dependencies, then topological sort follows all dependencies.

Running time?



Outline

- Graphs

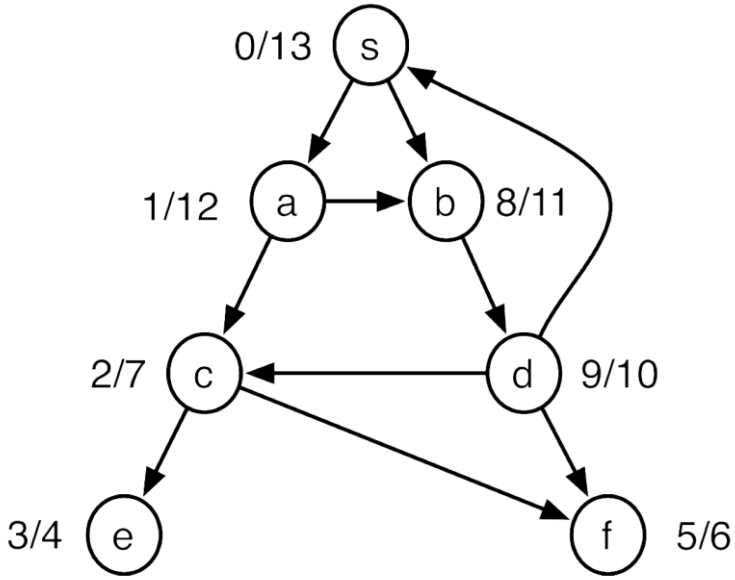
- Depth First Search

Depth First Search

```
DFS(G)
for each u ∈ G.V
    u.color = WHITE
time = 0
for each u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY                    // discover u
for each v ∈ G.Adj[u]             // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                        // finish u
```

Entry and Exit time



DFS Tree Edge Types

Let DFS starts at node s and returns a tree.

We call an edge (u, v) a **<u>tree edge</u>** if it is present it the returned tree.

The rest of the edges in the graph, which are non-tree edges, can further be classified as back edges, forward edges, and cross edges.

An edge $(u, v) \in E$ is called **<u>back edge</u>**, if v is an ancestor of u in DFS tree.
An edge $(u, v) \in E$ is called **<u>forward edge</u>**, if v is decentend of u in DFS tree.
An edge $(u, v) \in E$ is called **<u>cross edge</u>**, if v is neither ancestor nor a decentend of u in DFS tree.

Reference

Slides

Introduction to Algorithms by CLRS - Chp-22.3

Algorithm Design & Analysis (CSE222)

Lecture-13

Recap

- Graph Notations

- DFS Revisit

Outline

- DFS Revisit

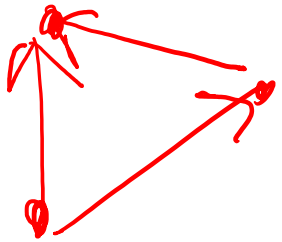
- Applications of DFS

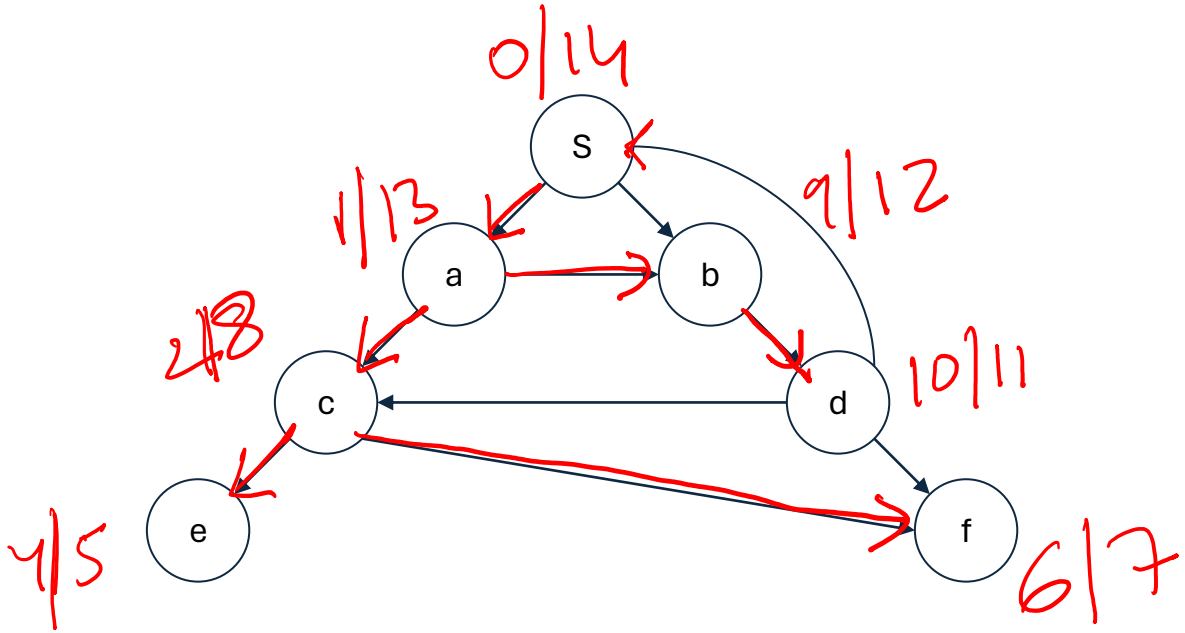
Depth First Search

```
DFS(G)
for each u ∈ G.V
    u.color = WHITE
time = 0
for each u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY                    // discover u
for each v ∈ G.Adj[u]             // explore (u, v)
    if v.color == WHITE
        DFS-VISIT(v)
u.color = BLACK
time = time + 1
u.f = time                        // finish u
```

Entry and Exit time





DFS Tree Edge Types

Let DFS starts at node s and returns a tree.

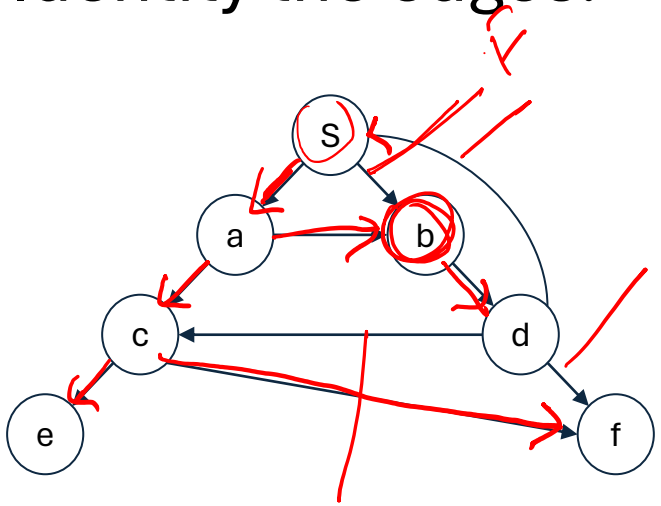
We call an edge (u, v) a **<u>tree edge</u>** if it is present it the returned tree.

The rest of the edges in the graph, which are non-tree edges, can further be classified as back edges, forward edges, and cross edges.

An edge $(u, v) \in E$ is called **<u>back edge</u>**, if v is an ancestor of u in DFS tree.
An edge $(u, v) \in E$ is called **<u>forward edge</u>**, if v is decentend of u in DFS tree.
An edge $(u, v) \in E$ is called **<u>cross edge</u>**, if v is neither ancestor nor a decentend of u in DFS tree.

Example

Identity the edges.



u, v

(S, B) – F

(d, c) – C

(d, f) – C

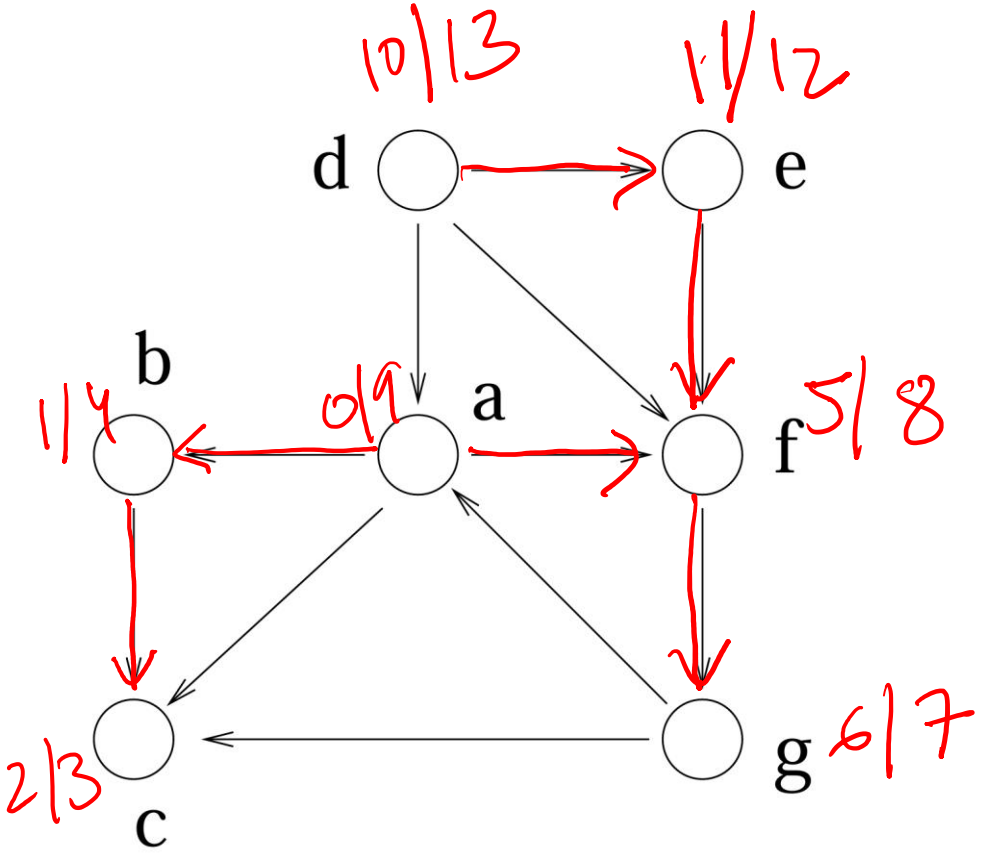
(d, S) – B

An edge $(u, v) \in E$ is called **back edge**, if v is an ancestor of u in DFS tree.

An edge $(u, v) \in E$ is called **forward edge**, if v is decentend of u in DFS tree.

An edge $(u, v) \in E$ is called **cross edge**, if v is neither ancestor nor a decentend of u in DFS tree.

Example



(a, c) — F

(g, a) — B

(g, c) — C

(d, f) — F

(d, a) — C

Properties

For a given directed graph G = (V, E), let DFS(G) returns a tree. For any two nodes u, v ∈ V.

- v is descented of u if and only if start(u) < start(v) < finish(v) < finish(u).
- u and v are not related to each other if finish(u) < start(v) or finish(v) < start(u)
- Forward edge is from a node of higher finish time to node of lower finish time.
- Back edge is from a node of lower finish time to node of higher finish time.

Questions

- What is the finish time relationship between cross edge?
- What is the relationship between u and v if start(u) < start(v) < finish(u) < finish(v)?

Outline

- DFS Revisit

- Applications of DFS

Questions

- How to compute if node v is reachable from node u?
- How can we identify the cycle in a digraph using these edges?

Claim

A digraph G = (V, E) is acyclic if and only if DFS(G) does not yield any back edge.

If there is a back edge in a depth-first search (DFS) traversal of a graph, it indicates the presence of a cycle in the graph.

Proof:
(⇐) Suppose there is no back edge from DFS(G). It implies that all edges go from higher finish time node to node with a lower finish time. So there cannot be any cycle.
(⇒) Proof by contradiction – Let there is no cycle in G.
There is a back edge (u, v) ∈ E.

- So, v is an ancestor of u in the DFS tree.
- So, there is path from v to u in the DFS tree.

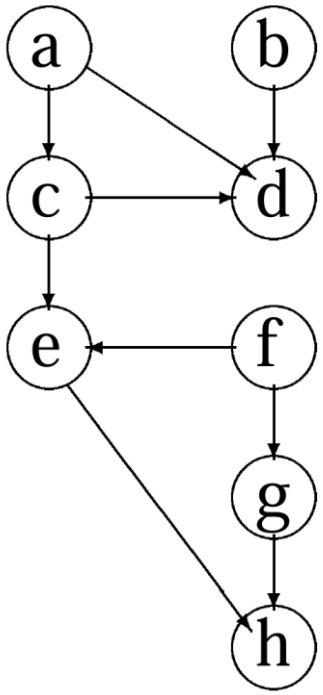
Path and back edge = cycle (Contradiction!)

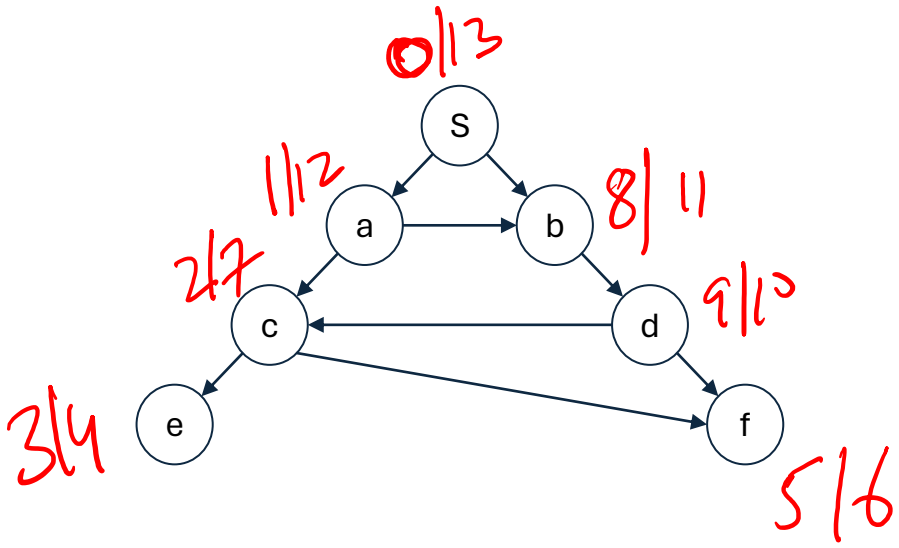
Topological Sorting

If G is DAG then there is an ordering of nodes V such that for each $(u, v) \in E$, u appears before v.

Topological Sort of a DAG is a total ordering $v_1 < v_2 < \ldots < v_n$ of vertices in V such that for any edge $(v_i, v_j) \in E$, $j > i$.

Example





efcdbas

<f, g, b, a, c, e, h, d>

<s, a, b, d, c, f, e>

Algorithm

Input: $G = (V, E)$
Output: Topological sort of V

TopSort(G):

1. Run DFS(G)
2. When computing the finish time for every vertex v, insert it at the front of a list
3. Return the list.

Running time? $O(|V| + |E|)$

Longest Path Weight

Input: Edge weighted directed graph G = (V, E, l). $l : E \to R^+$
Goal: Total weight of a longest path from v to t.

Let, LLP(v) computes the longest path in G from node v to node t.

If v = t then,
LLP(v) = 0
Else
max{l(v,w) + LLP(w) | (v, w) ∈ E}

Longest Path Weight

Input: Edge weighted directed graph G = (V, E, l).
Goal: Total weight of a longest path from v to t.

$$LLP(v) = \begin{cases} 0 & \text{if } v = t, \\ \max\left\{\ell(v \to w) + LLP(w) \;\middle|\; v \to w \in E\right\} & \text{otherwise,} \end{cases}$$

```
LongestPath(v, t):
    if v = t
        return 0
    if v.LLP is undefined
        v.LLP ← −∞
        for each edge v→w
            v.LLP ← max{v.LLP, ℓ(v→w) + LongestPath(w, t)}
        return v.LLP
```

Is there a possible DP?

Dynamic Programming

Compute postorder of G.
<u>Subproblem</u>: w.LLP: Computes the longest path in G from node w (appears after v) to node t.
<u>Recurrence</u>: If v = t then, v.LLP = 0
Else max{l(v,w) + w.LLP | w appears after v in the post order}

```
LongestPath(s, t):
    for each node v in postorder
        if v = t
            v.LLP ← 0
        else
            v.LLP ← −∞
            for each edge v→w
                v.LLP ← max {v.LLP, ℓ(v→w) + w.LLP}
    return s.LLP
```

Reference

Slides

Jeff Erickson Chp-6.1 & 6.3

Algorithm Design & Analysis (CSE222)

Lecture-14

Recap

- DFS Revisit
  - Back Edge, Forward Edge and Cross Edge
  - Start / Finish Time
  - v is descented of u if start(u) < start(v) < finish(v) < finish(u)
  - u and v are not related if finish(u) < start(v) or finish(v) < start(u)

- DFS Application
  - Cycle identification
  - Longest heavy path using topological sorting.

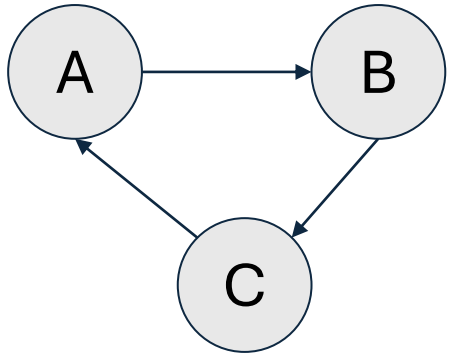
Outline

- Strongly Connected Components

Strongly Connected

Two vertices u and v are **strongly connected** if u can reach v as well as v can reach u.

A directed graph is **strongly connected** if and only if every pair of vertices is strongly connected.



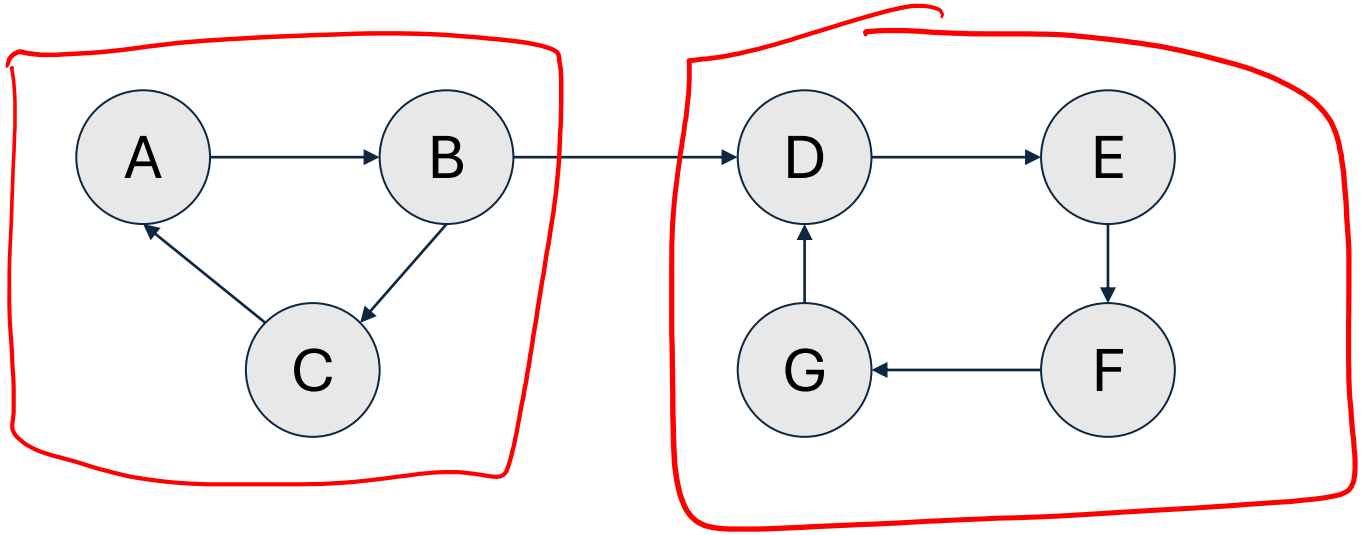
Strongly Connected

Two vertices u and v are **strongly connected** if u can reach v as well as v can reach u.

A directed graph is **strongly connected** if and only if every pair of vertices is strongly connected.

A **strong connected component (SSC or SC)** of a graph G is the maximal strongly connected subgraph of G.

Strong Component



$S_1 = \{A, B, C\} \subseteq V$

$A \to B$, $B \to A$; $B \to C$, $C \to B$; $A \to C$, $C \to A$

$S_2 = \{A, B, C, D\}$

$A \to D$, $D \not\to A$

$B \to D$ $D \not\to B$

Strongly Connected

Two vertices u and v are **strongly connected** if u can reach v as well as v can reach u.

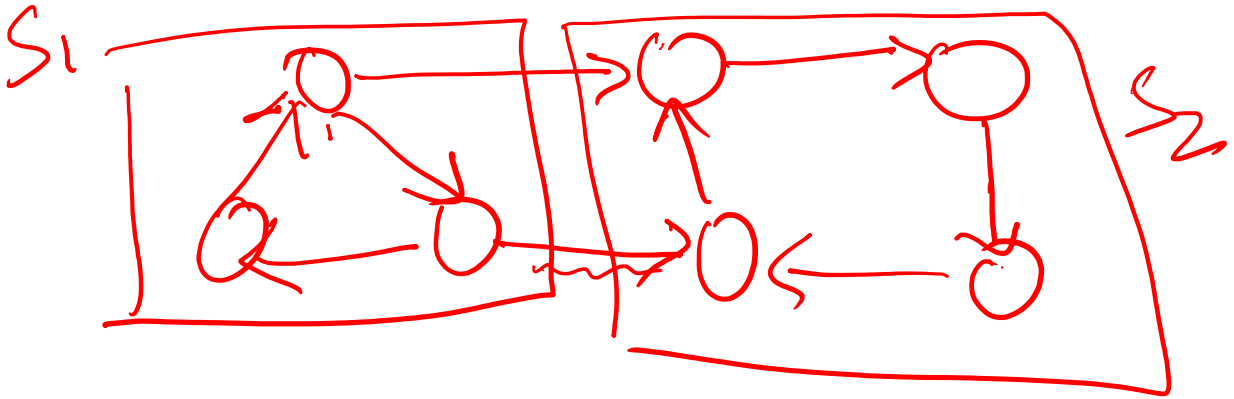
A directed graph is **strongly connected** if and only if every pair of vertices is strongly connected.

A **strong connected component (SSC or SC)** of a graph G is the maximal strongly connected subgraph of G.

A directed graph is strongly connected if and only if G has exactly one strong component.

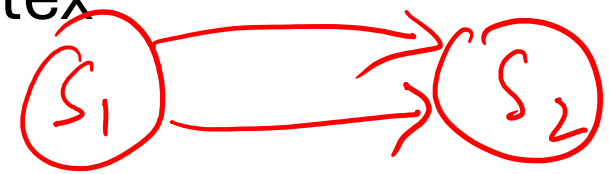
What is the strong component in a DAG?

Strong Component

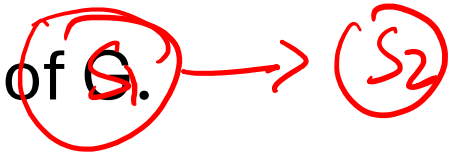


Let G be a directed graph.

- Collapse every strong component to a single vertex
- Collapse parallel edges between these vertex.



The resultant graph is called **meta-graph** or **condensation** of G.



Prove that condensation of G is a DAG.

Strong Component

Design an algorithm to compute all the strong components from a vertex v.

- Run DFS on G and compute reach(v)
- Run DFS on G and compute $\text{reach}^{-1}(v)$.
- Compute $\text{reach}(v) \cap \text{reach}^{-1}(v)$.

What is the running time to check if the complete graph G is strongly connected?

Design an algorithm for computing all strong component in G.

Strong Component

<u>Claim:</u> Fixing a DFS traversal of any directed graph G. Each strong component C of G has exactly one node that does not have a parent in C.

<u>Proof:</u>

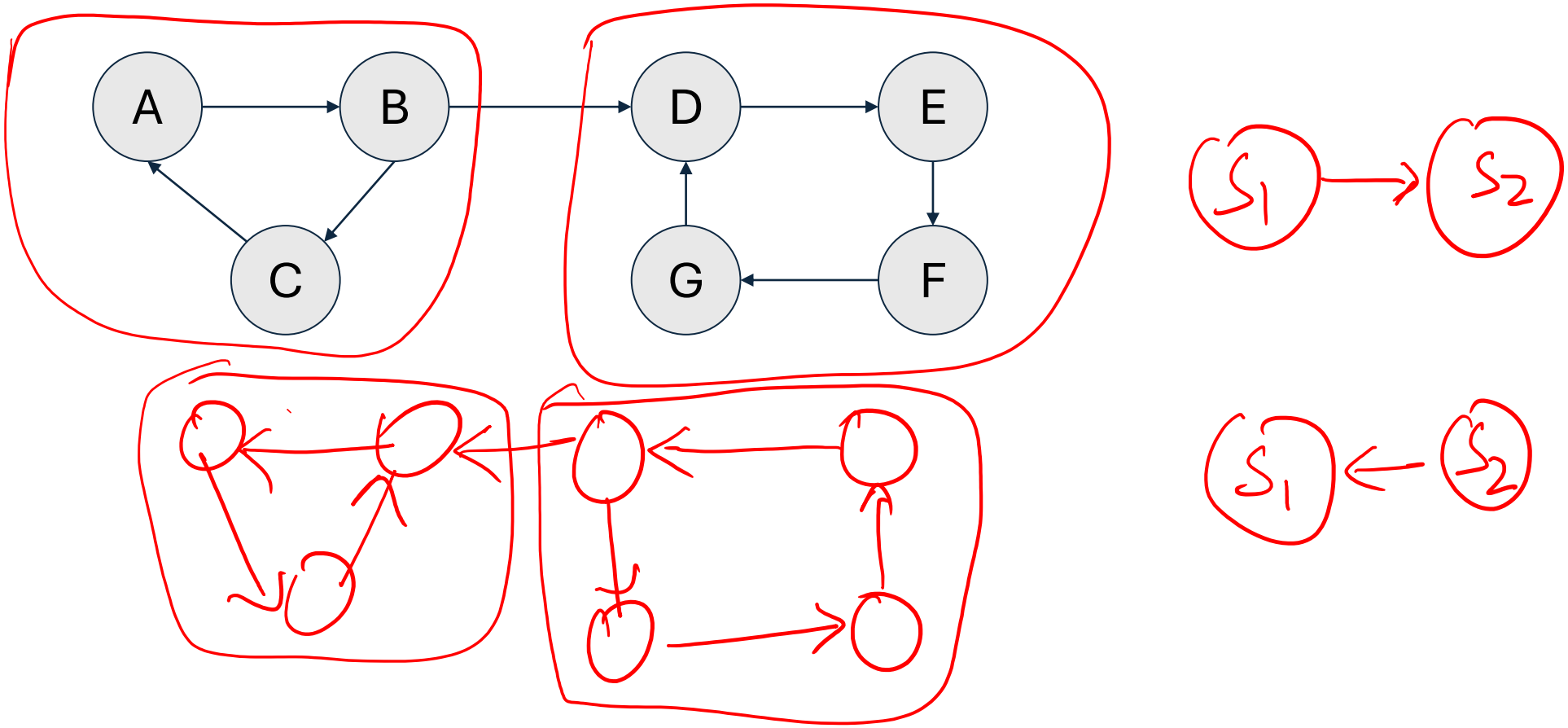
- Let C be an arbitrary sc of G. Consider a path v ∈ C to w ∈ C.
- Every vertex in this path can reach w and it can also reach C.
- Similarly every vertex in this path cab reached from v and can also be reached from any vertex in C.
- So every vertex in this path is also in C.
- Let v ∈ C is the earliest vertex, i.e., start(v) is lowest among all vertices ∈ C.
- So, at the call DFS(v) all vertex in C were not visited, so any w ∈ C is a descendant of v. So except v, parent of every vertex in path v to w is in C.

Algorithm Strong Component

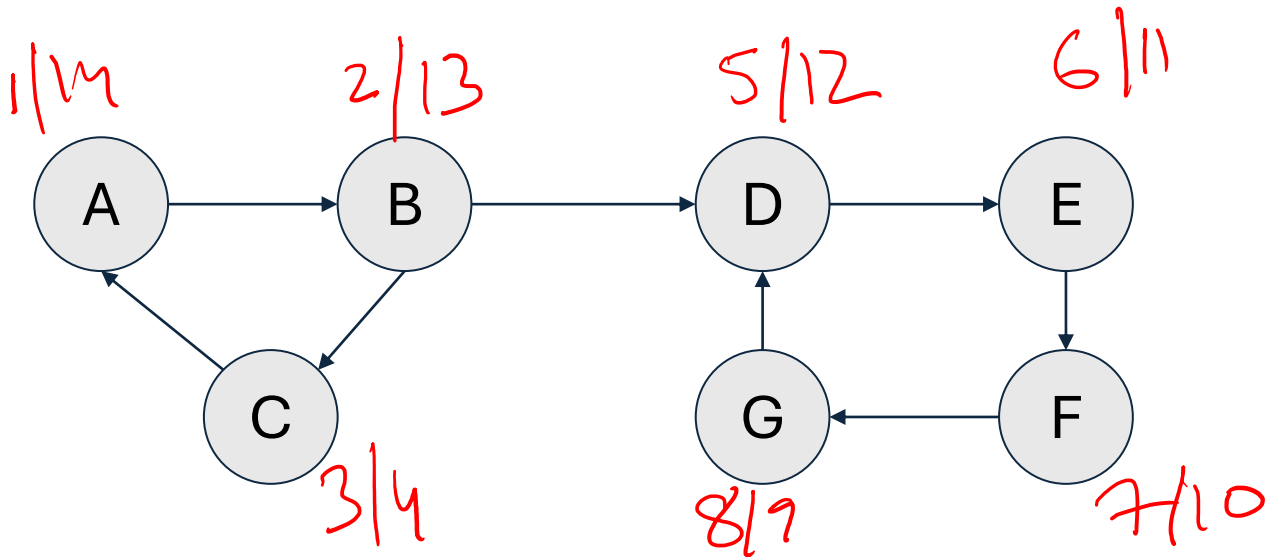
```
StrongComponents(G):
  count ← 0
  while G is non-empty
      C ← ∅
      count ← count + 1
      v ← any vertex in a sink component of G     ⟨⟨Magic!⟩⟩
      for all vertices w in reach(v)
          w.label ← count
          add w to C
      remove C and its incoming edges from G
```

How to find sink component?

Strong Component



Strong Component



SC(G) is equal to SC(reverse(G))

C G F E D B A

Strong Component

- Find source component of G.
- Compute SC(reverse(G)). (why?)

Strong Component

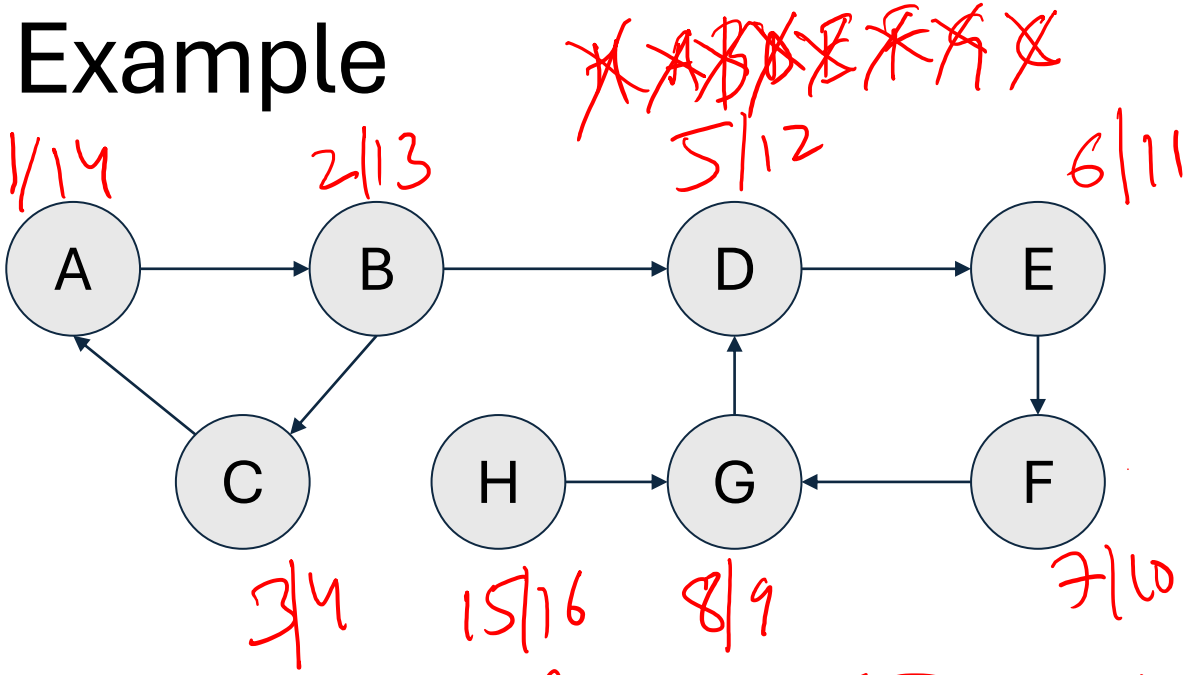
<u>Claim:</u> The last vertex in any post ordering of directed a graph lies in a source component of G.

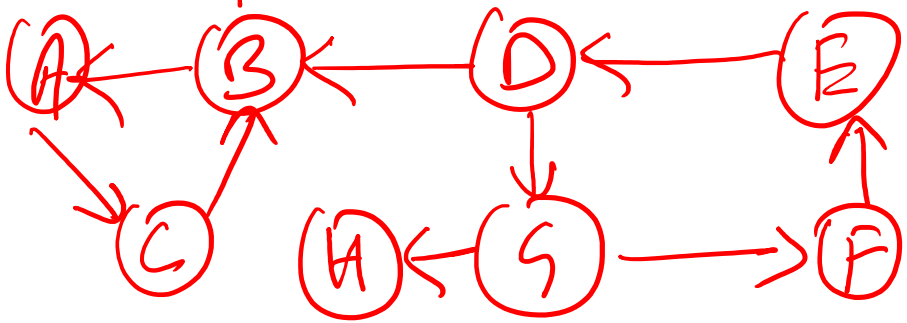
Algorithm

Input: Graph G
Output: Compute strong components.

1. Compute postordering(G) and store it in a stack.
2. Compute G^{-1}.
3. Pop vertex from stack and save nodes reachable from that vertex in G^{-1}.
4. Remove the all the reachable vertices from the stack and repeat 3.

Example





G^{-1}

H

A, C, B

D, G, F, E

Input: Graph G
Output: Compute strong components.

1. Compute postordering(G) and store it in a stack. $O(V+E)$
2. Compute G^{-1}. $O(V+E)$
3. Pop vertex from stack and save nodes reachable from that vertex in G^{-1}. $O(E)$
4. Remove the all the reachable vertices from the stack and repeat 3. $O(V)$

Reference

Slides

Jeff Erickson Chp-6.5 & 6.6

Algorithm Design & Analysis (CSE222)

Lecture-15

Recap

- Strongly Connected
  - Two vertices u and v are strongly connected if there is path from u to v and v to u.
  - If every pairs of vertices are strongly connected then the graph is a strongly connected graph.
  - Strong component (or scc) is the maximal strongly connected subgraph of G.
  - Condensation of a directed graph is always a DAG.
- Fix a DFS traversal, each SC has a vertex whose parent is either in another SC or it has no parents.
  - In C, show that every vertex in a path v to u are in C.
  - v be the first vertex in C, then for every w in C is its descendant & there is a path from v to w.
  - So, parent(w) $\in$ C.
- Number of SC
  - Vertices in condensation graphs are SC.
  - Find sink component, remove it and increment SC count and repeat.
  - $\#SC(G) = \#SC(G^{-1})$

Outline

- Strong Component

- Problems

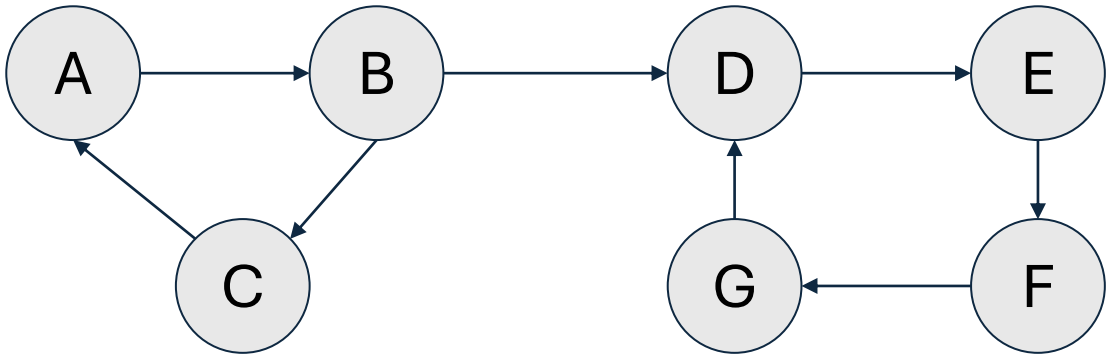
Strong Component

<u>Claim:</u> Fixing a DFS traversal of any directed graph G. Each strong component C of G has exactly one node that does not have a parent in G.

<u>Proof:</u>

- Let C be an arbitrary sc of G. Consider a path v ∈ C to w ∈ C.
- Every vertex in this path can reach w and it can also reach C.
- Similarly every vertex in this path cab reached from v and can also be reached from any vertex in C.
- So every vertex in this path is also in C.
- Let v ∈ C is the earliest vertex, i.e., start(v) is lowest among all vertices ∈ C.
- So, at the call DFS(v) all vertex in C were not visited, so any w ∈ C is a descendant of v. So except v, parent of every vertex in path v to w is in C.

Strong Component



SC(G) is equal to SC(reverse(G))

Strong Component

$s(x) < s(v) < f(x) < f(v)$

<u>Claim:</u> The last vertex in any post ordering of directed a graph lies in a source component of G.

<u>Proof:</u>

- Fix a DFS traversal in G, let v is the last vertex in the postordering. So v is the root of one of the tree in the resultant forest. So, v is the root of sc C.
- For any vertex w, if $s(v) < s(w) < f(w) < f(v)$, then w is a descendant of v.
- Let x be a vertex that is in the source component of G. Consider edge $x \rightarrow y$ such that $x \notin C$ and $y \in C$.
- x can reach y, y to v, so x can reach v. As v is the root, $s(v) < s(y) < f(y) < f(v)$.
- As there is an an edge $x \rightarrow y$, so $s(y) < f(x)$. So, $s(v) < f(x)$.
- But $f(x) < f(v)$, so $s(v) < s(x) < f(x) < f(v)$.
- Hence x is a descendant of v. So v can reach x. So, $x \in C$.

Algorithm

Input: Graph G
Output: Compute strong components.

1. Compute postordering(G) and store it in a stack.
2. Compute G^{-1}.
3. Pop vertex from stack and save nodes reachable from that vertex in G^{-1}.
4. Remove the all the reachable vertices from the stack and repeat 3.

Outline

- Strong Component
- Problems
- Minimum Spanning Tree

Problems

1. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk in G is called a French flag walk if its sequence of edge colors is red, white, blue, red, white, blue, and so on. Describe an algorithm to find all vertices in G that can be reached from a given vertex v through a French flag walk.

Problems

2. Suppose there are n cities connected by highways (not necessarily there is a straight road between every pair of cities). Suppose two of your friends started exploring by cars and in every step each of them can move from city A to city B if the cities have a direct road between them.
   Design an algorithm to compute minimum number of steps needed for them to meet at the same city or to report that no such paths are possible.

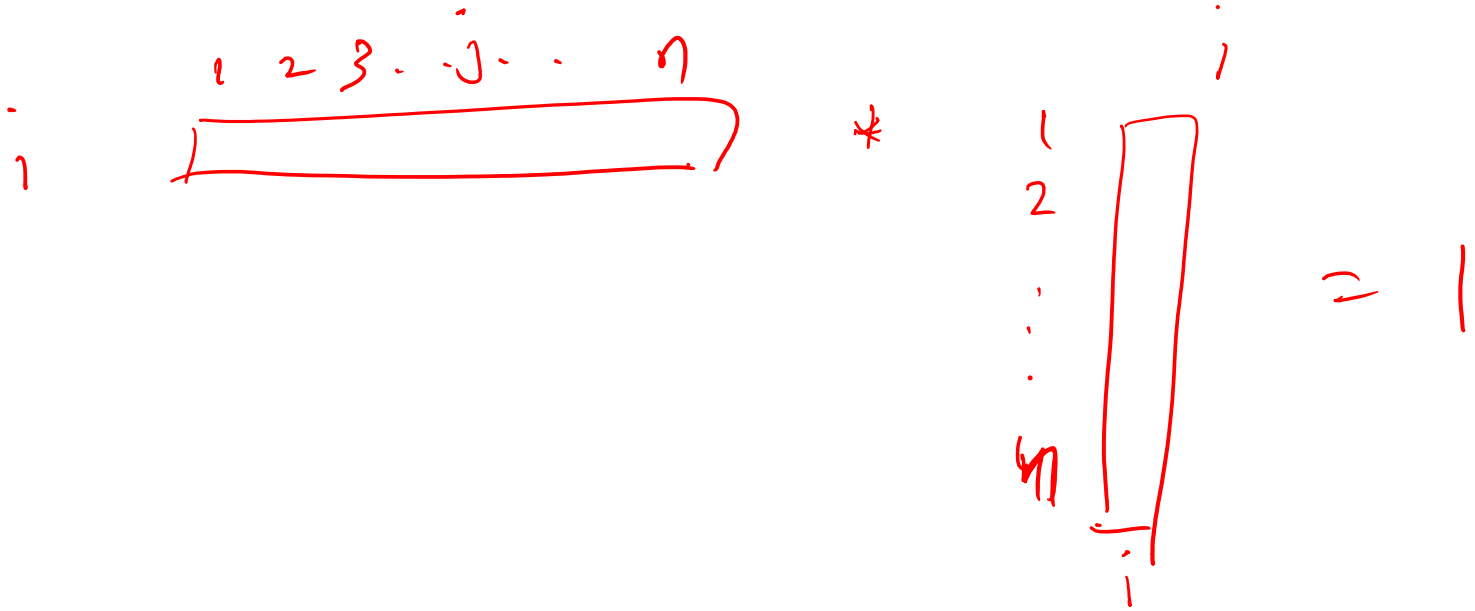
Problems

3. In a city the department of police made every street one way. Despite of complaints and confusion from people the mayor claims that it is possible to legally drive from one intersection to another. Design an algorithm to verify or refute the mayor’s claim.

Problems

4. Given a directed graph G, design an algorithm to compute number of 2k length cycle for 2 < k < n/2.

$$A \longrightarrow A^2 = (A^T A)_{ii}$$



Outline

- Strong Component
- Problems
- Minimum Spanning Tree

Minimum Spanning Tree

Let G = (V, E, w) be a weighted graph, w: E → R, i.e., w(e) is a real value.

Minimum spanning tree of G is defined by a tree T such that it minimizes the tree weight defined as follows.

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

Is minimum spanning tree is always unique for a given G = (V, E, w)?

Reference

Slides

Jeff Erickson Chp-6.5 & 6.6

Algorithm Design & Analysis (CSE222)

Lecture-16

Outline

- Strong Component

- Minimum Spanning Tree

Minimum Spanning Tree

Let G = (V, E, w) be a weighted graph, w: E → R, i.e., w(e) is a real value.

Minimum spanning tree of G is defined by a tree T such that it minimizes the tree weight defined as follows.

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

Outline

- Minimum Spanning Tree
- Jarniks / Prims
- Kruskals

Unique Minimum Spanning Tree

<u>Claim:</u> If all the edge weights in a connected graph G are distinct then G has a unique minimum spanning tree.

<u>Proof:</u>

- Let G has two minimum spanning trees T and T'.
- Let e be the min weight edge in T \ T' and similarly e' in T' \ T. Let w(e) < w(e')
- The subgraph T' U {e} has a cycle that passes through e.
- Let e'' be some edge of the cycle that is not in T. It exists as T is a tree.
- Since, e ∈ T, and e'' ≠ e so e'' ∈ T' \ T. Hence, w(e) < w(e') ≤ w(e'').
- Consider T'' = T' + e \ e''. Then w(T'') = w(T') + w(e) - w(e'') < w(T')!
- But T and T' were the minimum spanning trees. So w(e'') = w(e).
- Even if T'' = T, we will have w(e') = w(e) so T and T' are the same tree.

Minimum Spanning Tree

Given a graph G, maintain an acyclic subgraph F, such it is also a subgraph of the MST of G.

At each step F encounters two types of edges,

- <u>Useless</u>: if the edge not in F but both end points in the same component of F.
- <u>Safe</u>: minimum weighted edge with exactly an endpoint in a component of F.

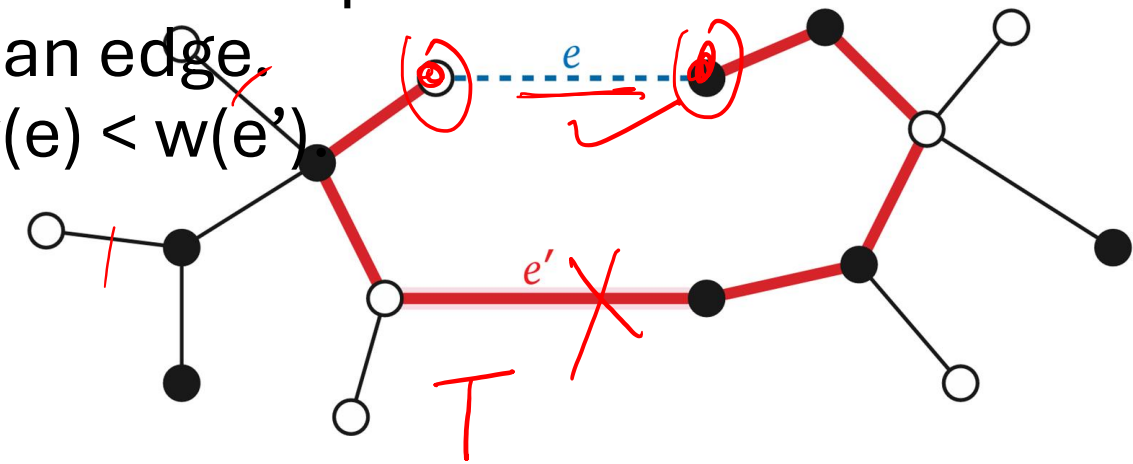
Assume we have unique edge weights, ties are broken arbitrarily.

Minimum Spanning Tree

Claim: Minimum spanning tree of a graph G contains every safe edge.

Proof:

- For any subset S ⊆ V, the MST contains the min weight edge with an end in S
- Let e be the lightest edge with exactly one end in S.
- Let T be the minimum spanning tree that does not contain e.
- Remove an edge e' from T such that the two endpoints of e are in two different components. Let e' be such an edge.
- Consider T' = T + e \ e'. By definition w(e) < w(e')
- So, w(T') < w(T)!

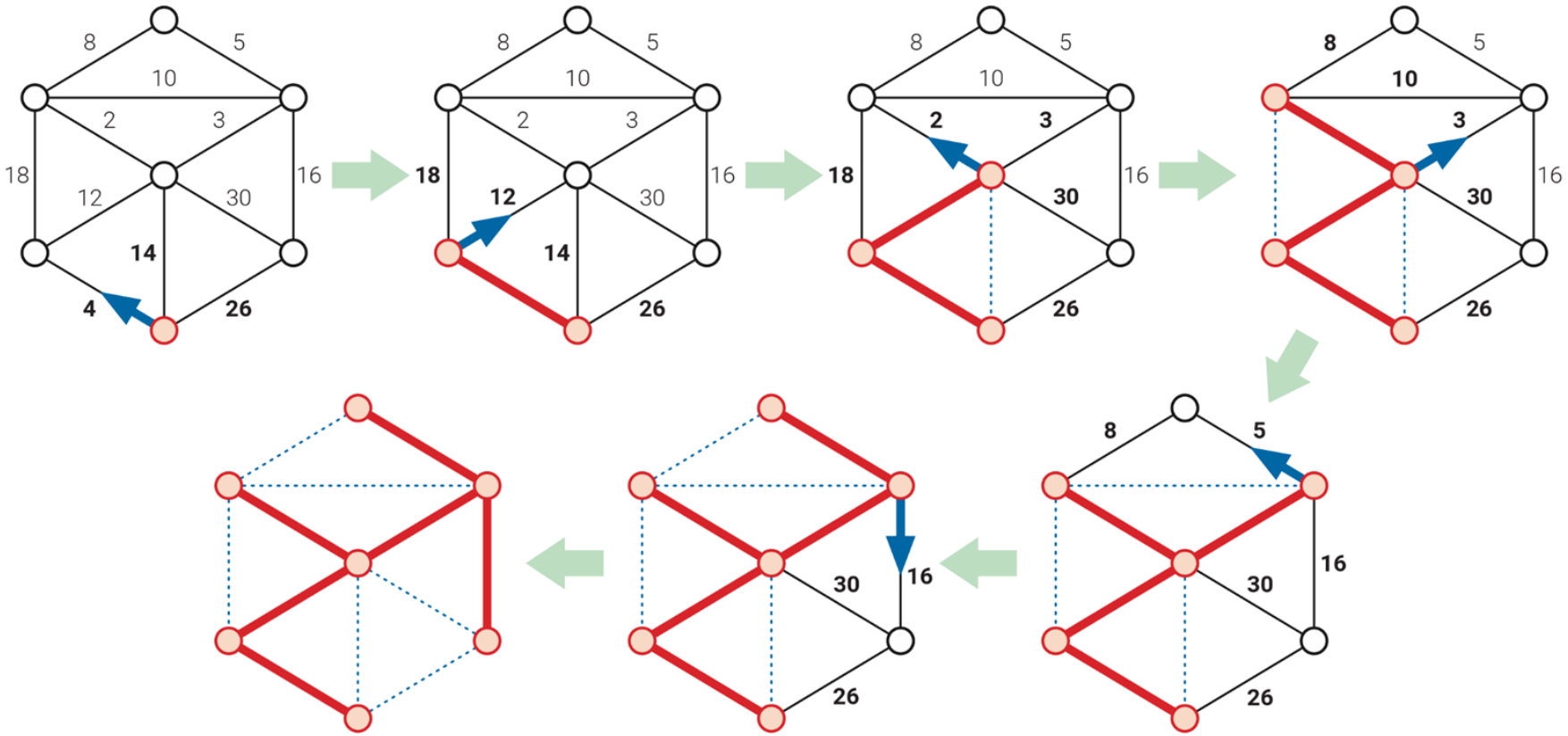


Outline

- Minimum Spanning Tree
- Jarniks / Prims
- Kruskals

Jarnik's / Prim's Algorithm

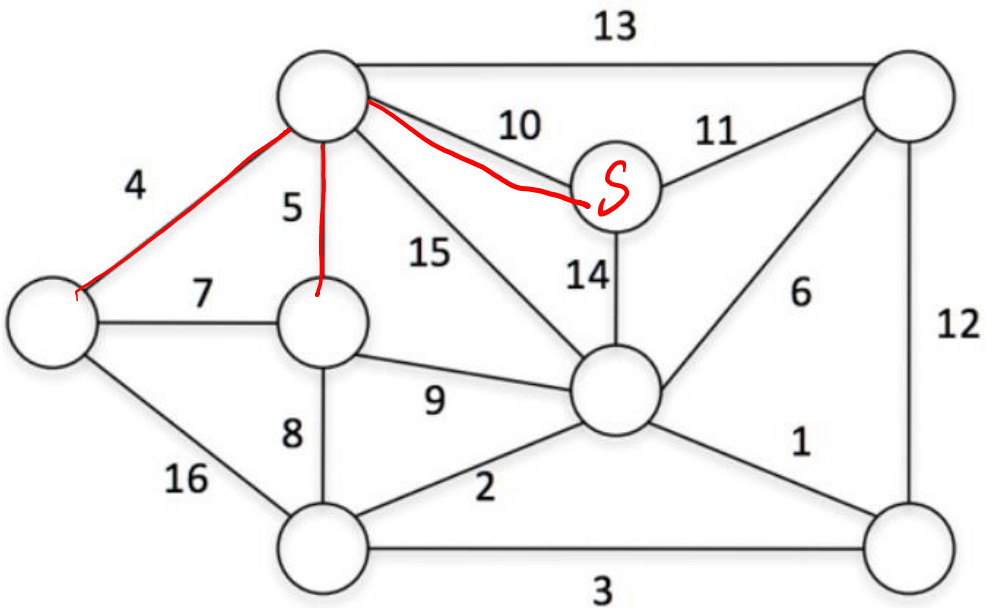
Start from a vertex and repeatedly add safe edge to T.



Jarnik's / Prim's Algorithm

- Maintain priority queue of edges adjacent to T.
- Dequeue minimum weight edge from the queue and check if both end points are in T.
- If not, then add the end point of the edge to T and update the priority queue.

Running time?

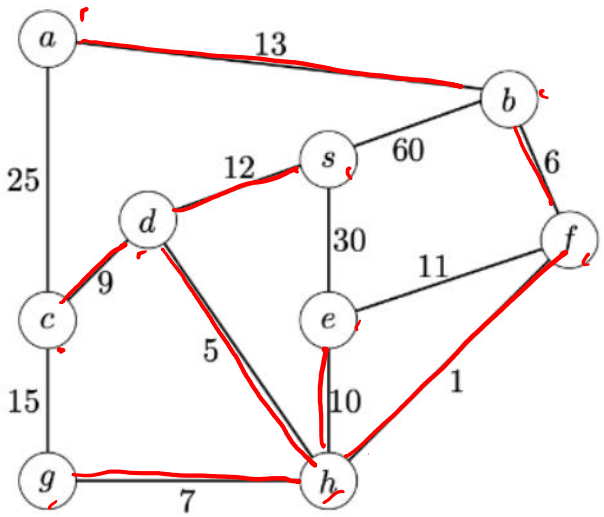


Outline

- Minimum Spanning Tree
- Jarniks / Prims
- Kruskals

Kruskal's

Add end point of edges to T in an increasing weight order, if the edge is safe.



Kruskal's Algorithm

Consider a data structure that supports following operations.

MakeSet(v): Create a set containing only the vertex v.
Find(v): Return unique identifier of the set containing v.
Union(u, v): Replace the sets containing u and v with their union.

```
KRUSKAL(V, E):
    sort E by increasing weight
    F ← (V, ∅)
    for each vertex v ∈ V
        MAKESET(v)
    for i ← 1 to |E|
        uv ← ith lightest edge in E
        if FIND(u) ≠ FIND(v)
            UNION(u, v)
            add uv to F
    return F
```

Kruskal's Algorithm

Consider a data structure that supports following operations.

2|E| Find operations
|V| - 1 Union operations
Amortized running time of Union is O(log |V|)

So, O(E + Vlog V)

```
KRUSKAL(V, E):
    sort E by increasing weight
    F ← (V, ∅)
    for each vertex v ∈ V
        MAKESET(v)
    for i ← 1 to |E|
        uv ← ith lightest edge in E
        if FIND(u) ≠ FIND(v)
            UNION(u, v)
            add uv to F
    return F
```

Reference

Slides

Jeff Erickson Chp-7

Algorithms Design by Kleinberg & Tardos - Chp 4.5

Algorithm Design & Analysis (CSE222)

Lecture-17

Outline

- Jarniks / Prims

- Kruskals

Outline

- Union Find data Structure

- Shortest Paths

Union Find Data Structure

Consider following operations in a data structure.

MakeSet(v): Create a set containing only the vertex v.
Find(v): Return unique identifier of the set containing v.
Union(u, v): Replace the sets containing u and v with their union.

Maintain an array *component* of size |V|, where component[x] is the name of the set that contains $x \in V$.

Running time of union(u, v): O(|V|)

Can we do better?

Union Find Data Structure

Claim: There is a way to perform the Union() operation such that its amortized running time for a sequence of k operations is O(k log k).

Proof: Union of two sets uses the name of the larger set.

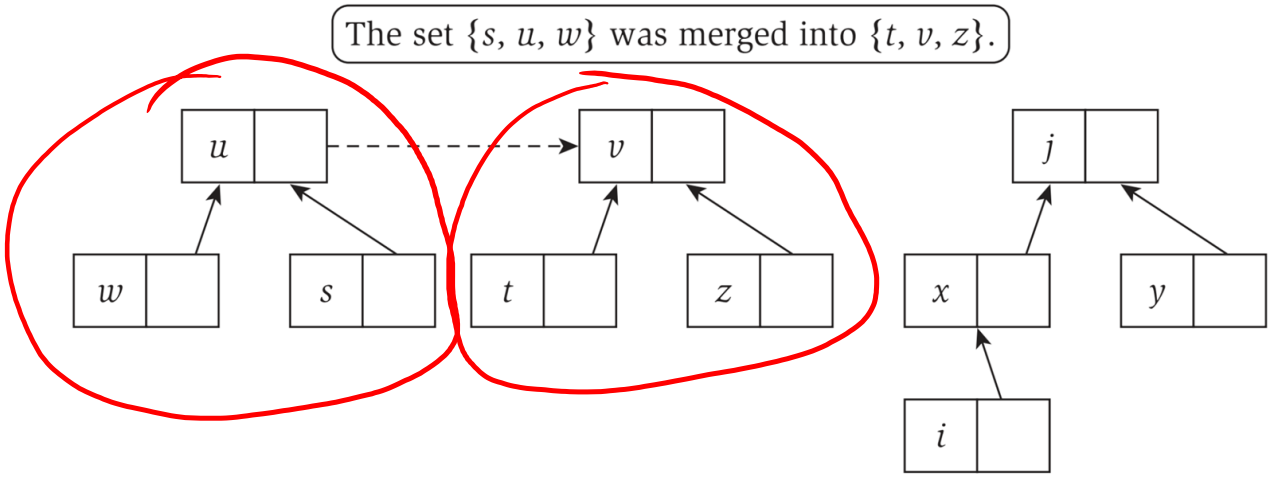
- First we bound the number of times component[u] gets updated.
- The algorithm starts with singleton set for each |V| elements.
- After first Union call the set size increased to 2. After k operations at most |V| - 2k elements are singleton.
- To change the name of the component[u] it size of its union set has to be at least twice the size of set containing u. Since, 2k elements are touched by Union() so name change happens for at most log 2k times.
- As there are 2k elements involved, the amortized running time is O(k log k).

Union Find Data Structure

Can we improve the worst case running time?

Use pointer to represent component names.

- For singleton sets the pointer is null. Hence, the element itself is the component name or root.
- Union(u, v): if size of set containing v is larger than the size of set containing u, then the pointer of u points to v else pointer of v points to u.



Union Find Data Structure

Claim: There is an improved data structure where the Union operation takes O(1) time and worst case running time of Find() is O(log |V|).

Proof:

- Consider an element u.
- The running time of Find(u) is the number of times its name changes during Union() operations.
- As we keep the name of the larger set among the two sets so the name is changed only if the other set is bigger than the set containing u.
- Every time the component name changes the size of the set at least doubles.
- Starting from size 1 to maximum size of |V|, the change can happen at max log|V| times.

Outline

- Union Find data Structure

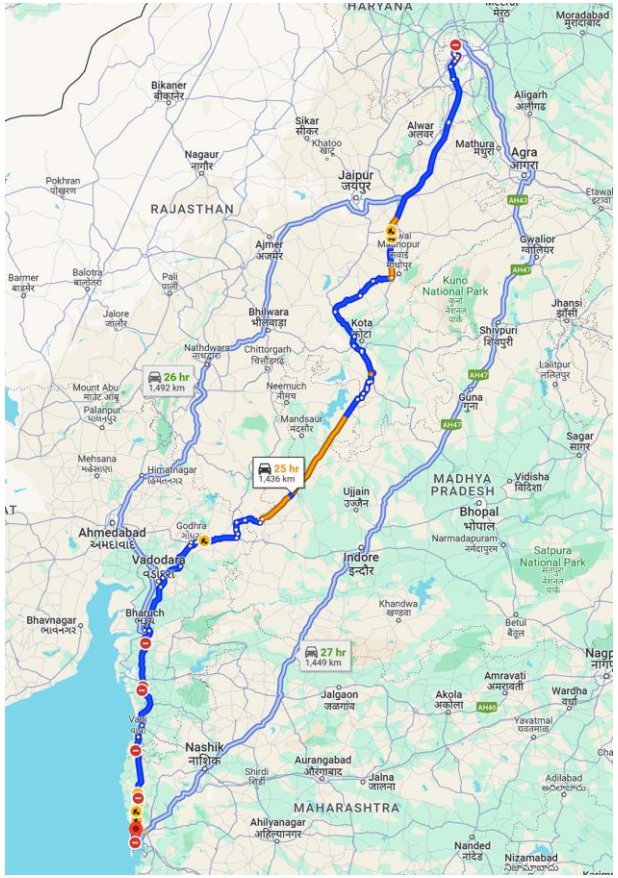
- Shortest Paths

Shortest Path

What is the fastest way to drive from Delhi to Mumbai?

- Cities as vertices
- Roads as edges
- Time to travel between two vertices as edge weights.

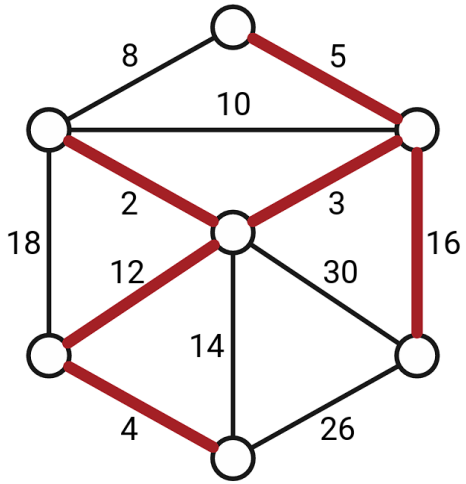
$$w(P) := \sum_{u \to v \in P} w(u \to v)$$

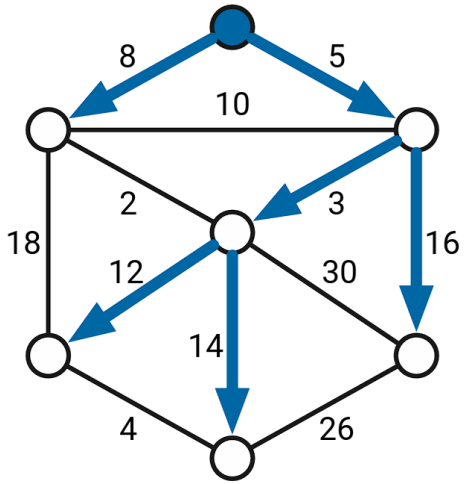


Shortest Path

Single Source Shortest Path: It is the shortest path from source vertex (say s) to every other vertex in the graph.

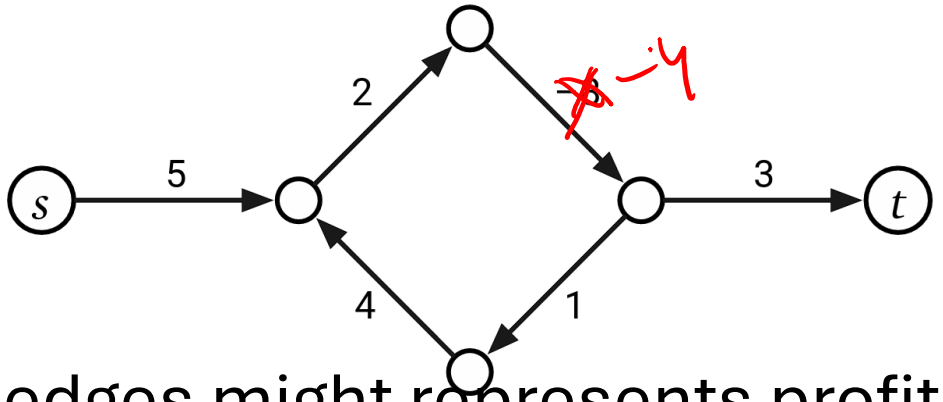
Minimum spanning tree and shortest path tree are not the same!

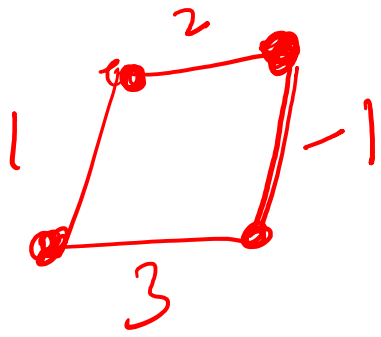




Shortest Path

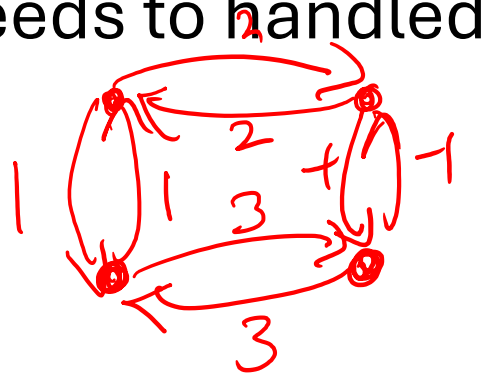
In case of negative cycle shortest path is undefined.





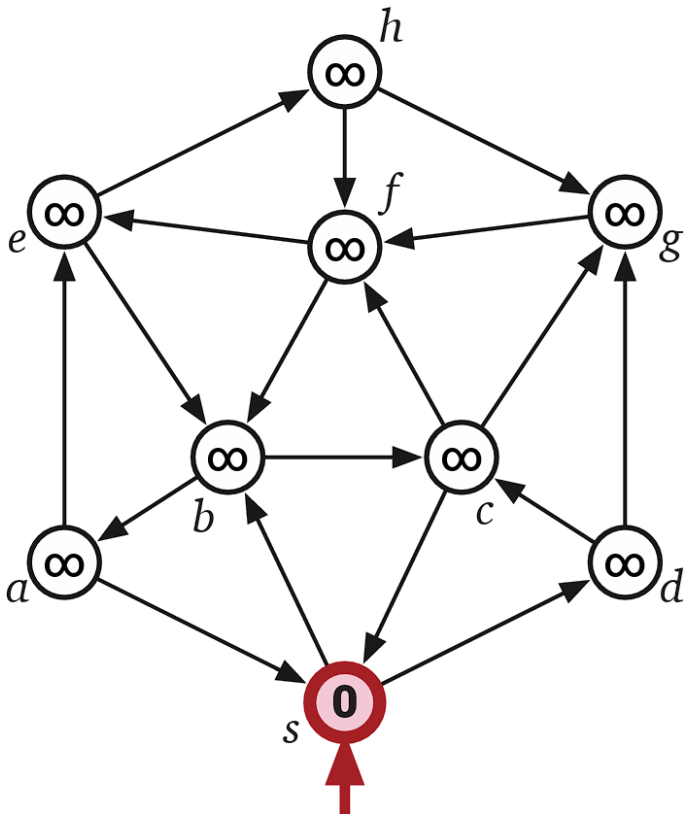
Negative and positive edges might represents profit and loss. Change -8 to -4.

In case of undirected graphs the negative edge needs to handled carefully to while computing shortest path.



Shortest Path

Consider a diagraph having all the edge weights weights as 1.



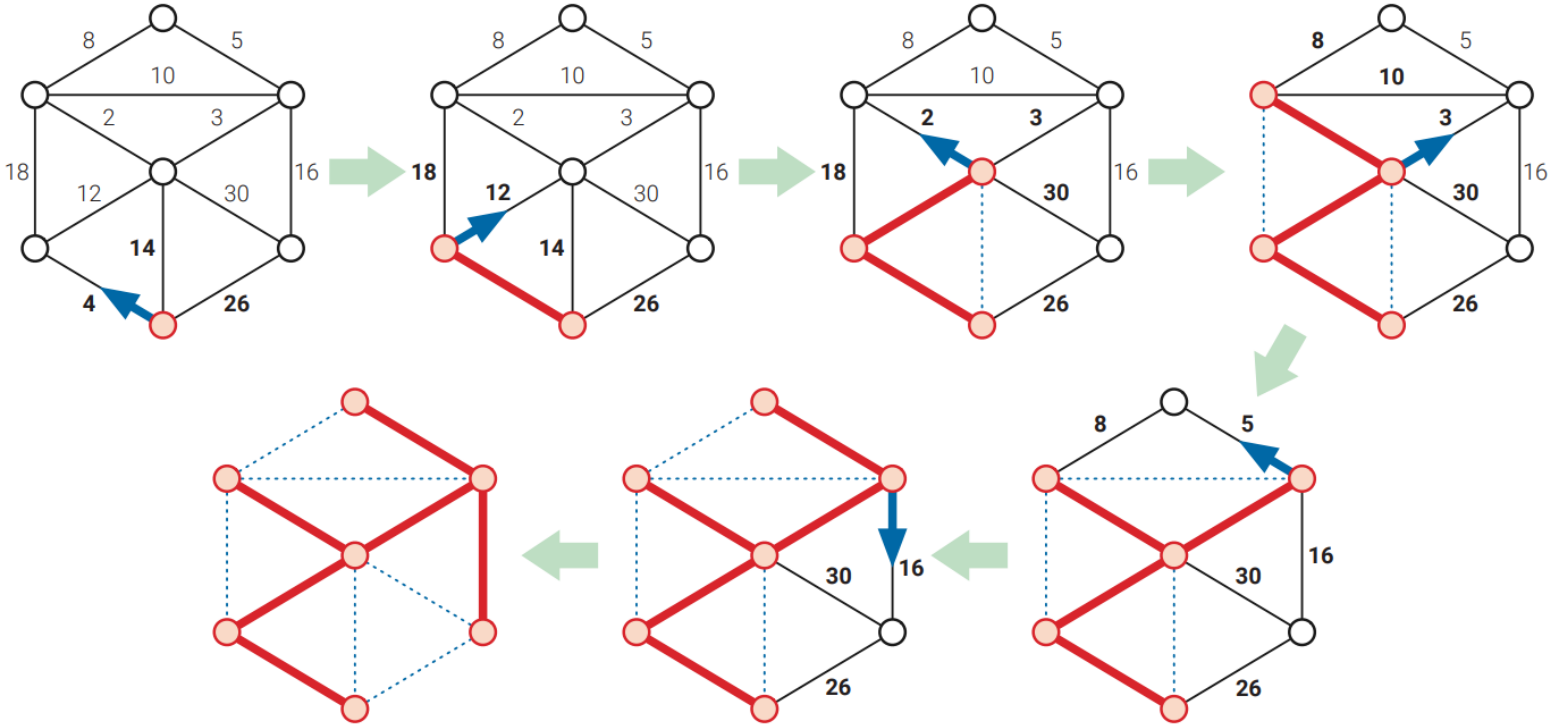
<u>BFS(s):</u>

```
INITSSSP(s)
PUSH(s)
while the queue is not empty
    u ← PULL( )
    for all edges u→v
        if dist(v) > dist(u) + 1
            dist(v) ← dist(u) + 1
            pred(v) ← u
            PUSH(v)
```

BFS Shortest Path

```
BFSWithToken(s):
    InitSSSP(s)
    Push(s)
    Push(✠)                    ⟨⟨start the first phase⟩⟩
    while the queue contains at least one vertex
        u ← Pull( )
        if u = ✠
            Push(✠)            ⟨⟨start the next phase⟩⟩
        else
            for all edges u→v
                if dist(v) > dist(u) + 1        ⟨⟨if u→v is tense⟩⟩
                    dist(v) ← dist(u) + 1
                    pred(v) ← u                 ⟨⟨relax u→v⟩⟩
                    Push(v)
```

BFS Shortest Path




```
BFSWithToken(s):
  InitSSSP(s)
  Push(s)
  Push(✠)                    ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      u ← Pull()
      if u = ✠
          Push(✠)            ⟨⟨start the next phase⟩⟩
      else
          for all edges u→v
              if dist(v) > dist(u) + 1        ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + 1       ⟨⟨relax u→v⟩⟩
                  pred(v) ← u
                  Push(v)
```

BFS Shortest Path

<u>Claim:</u> For every integer i ≥ 0, and every vertex v at the end of ith phase, either dist(v) = ∞ or dist(v) ≤ i and v is the queue if and only if dist(v) = i.

<u>Proof:</u> By induction.
at i = 0, dist(s) = 0 and dist(v) = ∞ for every v ≠ s.
Fix i > 0, the queue has dist(u) = i-1 followed by the token. → ✠ $i-1$ $i-1$ $\cdots$ $i-1$ →
Before we pull out the token we pull out every vertex u s.t. dist(u) = i-1
For every u→v, if dist(v) > ∞ then set dist(v) = i and push v.
So, before we pull the token we have, → i i $\cdots$ i ✠ →
Finally after phase i we have → ✠ i $\cdots$ i

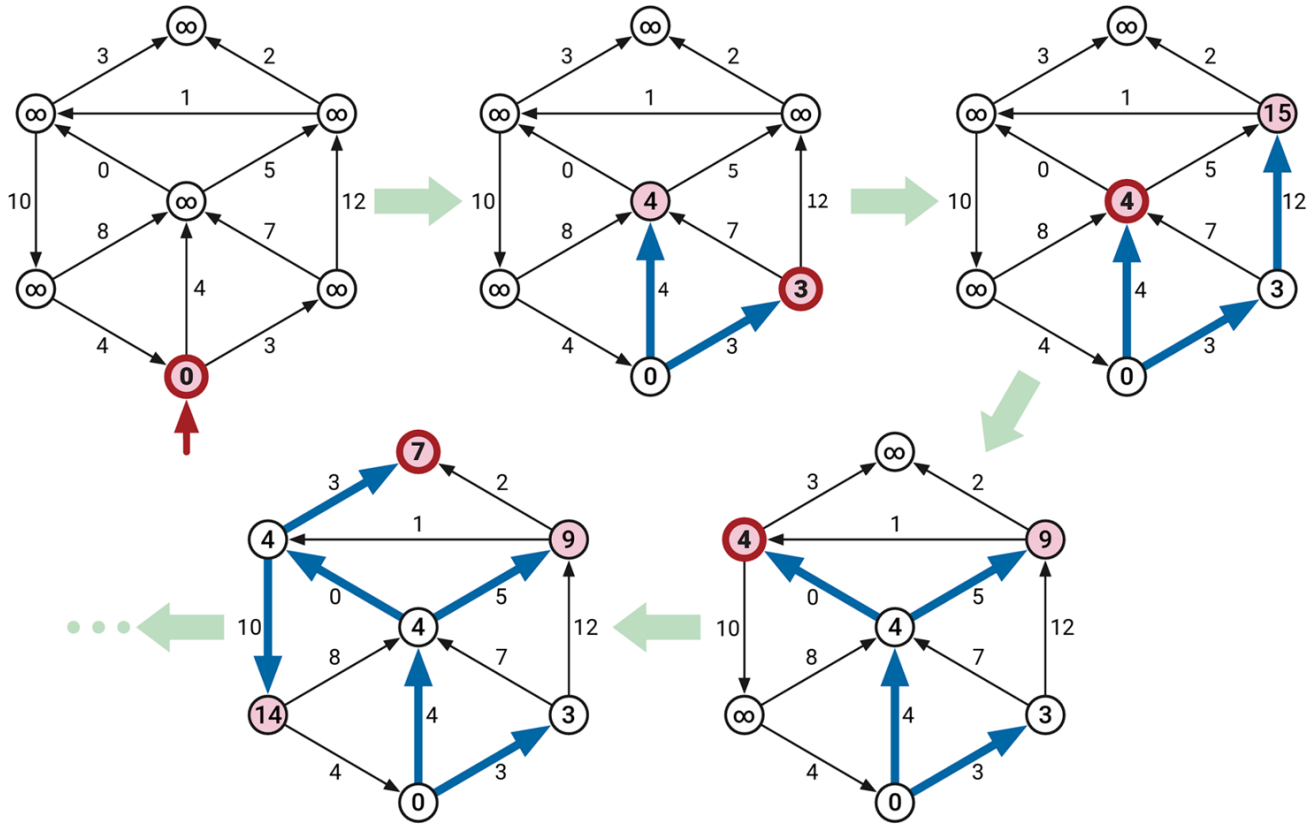
BFS Shortest Path

<u>Claim:</u> The dist(v) is the length of the shortest path from s to v in the graph G.

<u>Proof:</u> Fix an arbitrary path then prove by induction.

Dijkstra’s Algorithm

Replace the FIFO queue in BFS shortest path with a priority queue.



Dijkstra's Algorithm

```
DIJKSTRA(s):
    INITSSSP(s)
    INSERT(s, 0)
    while the priority queue is not empty
        u ← EXTRACTMIN( )
        for all edges u→v
            if u→v is tense
                RELAX(u→v)
                if v is in the priority queue
                    DECREASEKEY(v, dist(v))
                else
                    INSERT(v, dist(v))
```

Reference

Slides

Jeff Erickson Chp-8

Algorithms Design by Kleinberg & Tardos - Chp 4.6

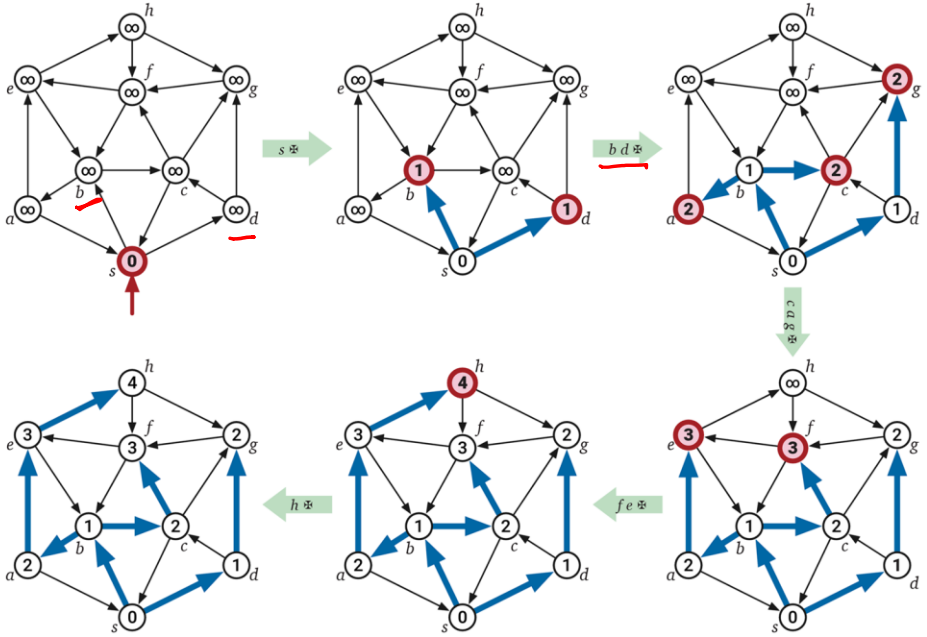
Algorithm Design & Analysis (CSE222)

Lecture-18

Outline

- Union Find data Structure

- Shortest Path
  - Using BFS & token



```
BFSWithToken(s):
  InitSSSP(s)
  Push(s)
  Push(✠)                    ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      u ← Pull()
      if u = ✠
          Push(✠)            ⟨⟨start the next phase⟩⟩
      else
          for all edges u→v
              if dist(v) > dist(u) + 1        ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + 1
                  pred(v) ← u                 ⟨⟨relax u→v⟩⟩
                  Push(v)
```

Outline

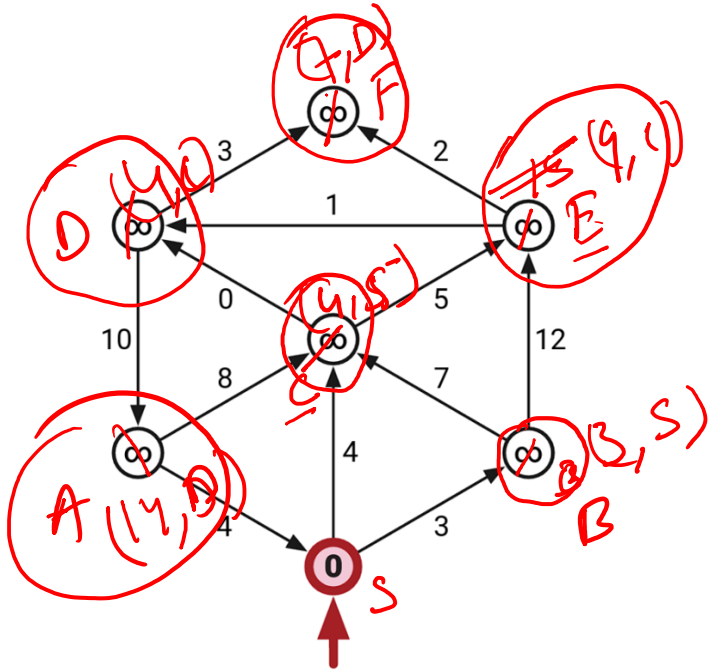
- Shortest Path
  - Dijkstra's algorithm (single source)

Dijkstra’s Algorithm

For an edge $u \rightarrow v$
Tense: If $dist(v) > dist(u) + w(u \rightarrow v)$
Relaxation: $dist(v) = dist(u) + w(u \rightarrow v)$

Dijkstra's Algorithm

1. Start with a source.
2. Check tense edge from visited vertex set.
3. Relax such edge.
4. Include the vertex with smallest distance.
5. Repeat from 2 to 4.



S, C, D, A

Running time: $O(|V|^2)$

Dijkstra's Algorithm

```
DIJKSTRA(s):
  INITSSSP(s)
  INSERT(s, 0)
  while the priority queue is not empty
      u ← EXTRACTMIN( )
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              if v is in the priority queue
                  DECREASEKEY(v, dist(v))
              else
                  INSERT(v, dist(v))
```

Dijkstra's Algorithm

<u>Claim:</u> If G has no negative-weighted edge then for all u_i and u_j, such that $i < j$, we have $dist(u_i) < dist(u_j)$.

<u>Proof:</u> Fixing an arbitrary index i we have following two cases

- If G have the edge $u_i \rightarrow u_{i+1}$, and this edge has been relaxed in the ith iteration. Then we have $dist(u_{i+1}) = dist(u_i) + w(u_i \rightarrow u_{i+1}) > dist(u_i)$.
- Else either G does not have the edge $u_i \rightarrow u_{i+1}$ or the edge has not been relaxed in the ith iteration. In this iteration, the priority queue must have, $dist(u_{i+1}) > dist(u_i)$.

Dijkstra's Algorithm

<u>Claim:</u> If G has no negative-weighted edge then each vertex will be visited at most once.

<u>Proof:</u> For a vertex v, prove that for any i and j iteration, where $i < j$, it is impossible to get dist(v) at ith iteration is smaller than dist(v) in jth iteration.

Dijkstra's Algorithm

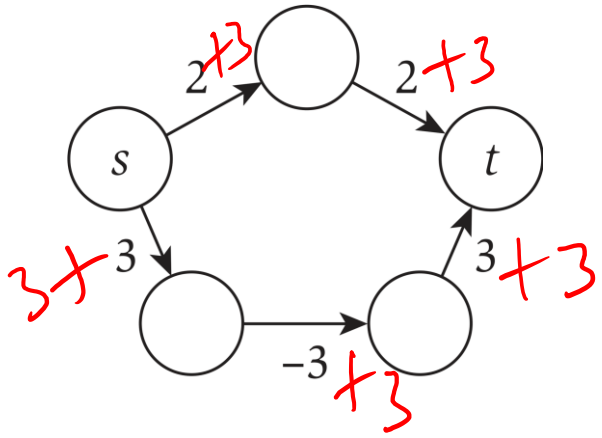
<u>Claim:</u> If G has no negative-weighted edge, then after the end of Dijkstra, dist(u) is the length of the shortest path in G from s to u for every vertex u.

<u>Proof:</u> Consider an arbitrary path in G, $u_0 \to u_1 \to \ldots \to u_t$, where $s=u_0$ and $u = u_t$.

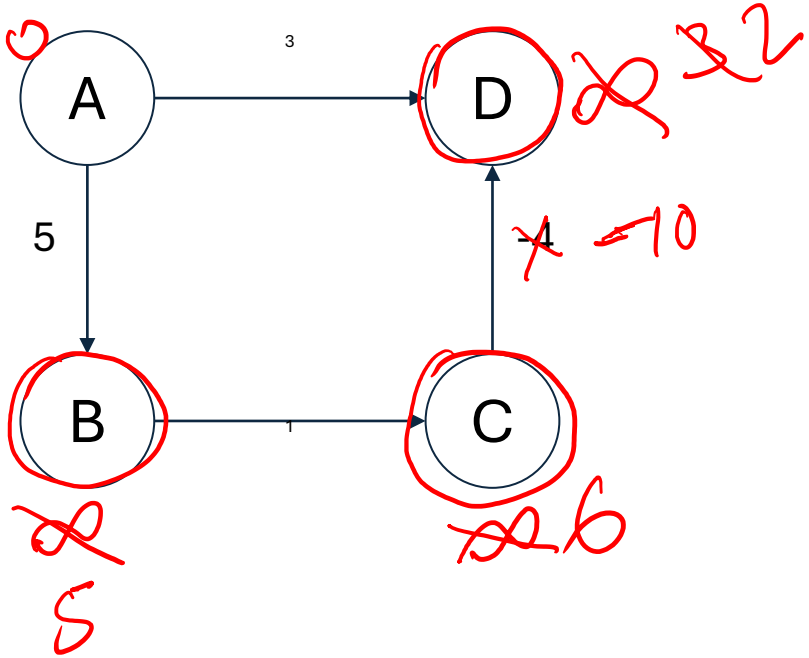
- Base: At $t = 0$, $\text{dist}(u_0) = \text{dist}(s) = 0$.
- For $j > 0$, the induction hypothesis is $\text{dist}(u_j) \leq L_j$
- After visiting u_j either $\text{dist}(u_{j+1}) \leq \text{dist}(u_j)+w(u_j \to u_{j+1})$ or we set $\text{dist}(u_{j+1}) = \text{dist}(u_j)+w(u_j \to u_{j+1})$
- In either way $\text{dist}(u_{j+1}) \leq \text{dist}(u_j)+w(u_j \to u_{j+1}) = L_j + w(u_j \to u_{j+1}) = L_{j+1}$.

Negative Edges

Add smallest weight to all the edges.



Negative Edges



Reference

Slides

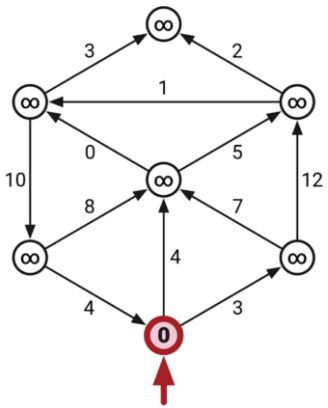
Jeff Erickson Chp-8

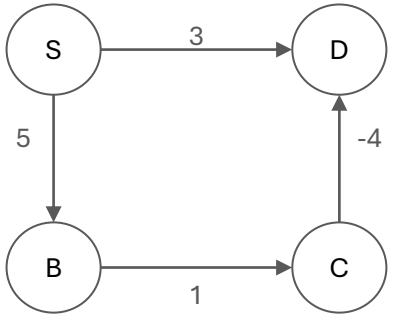
Algorithm Design & Analysis (CSE222)

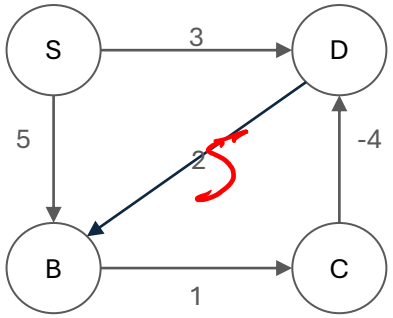
Lecture-19

Recap

- Shortest Path
  - Dijkstra's algorithm (single source)








```
DIJKSTRA(s):
  INITSSSP(s)
  INSERT(s, 0)
  while the priority queue is not empty
      u ← EXTRACTMIN( )
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              if v is in the priority queue
                  DECREASEKEY(v, dist(v))
              else
                  INSERT(v, dist(v))
```

Outline

- Shortest Path
  - Bellman Ford
- Distance Vector Protocols
- Network Flows

Dijkstra’s Algorithm

When there is a negative edge, Dijkstra’s algorithms may or may not return correct answer.

- In the presence of negative cycle, shortest paths are undefined.

- In the presence of negative edge but not a negative cycle then Dijkstra returns correct answer.
  - How to ensure correct answers?

Bellman Ford

<u>Claim:</u> If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most n-1 edges.

<u>Proof:</u>

- Since, every cycle (if present) has a nonnegative cost then no path from s to t would not repeat any vertex v.
- If a vertex repeats in a path then we can remove the portion of the path. This will not increase the path cost from s to t.
- So at max n-1 edges are traversed to reach t from s.

Using dynamic programming by taking advantage of the above claim.

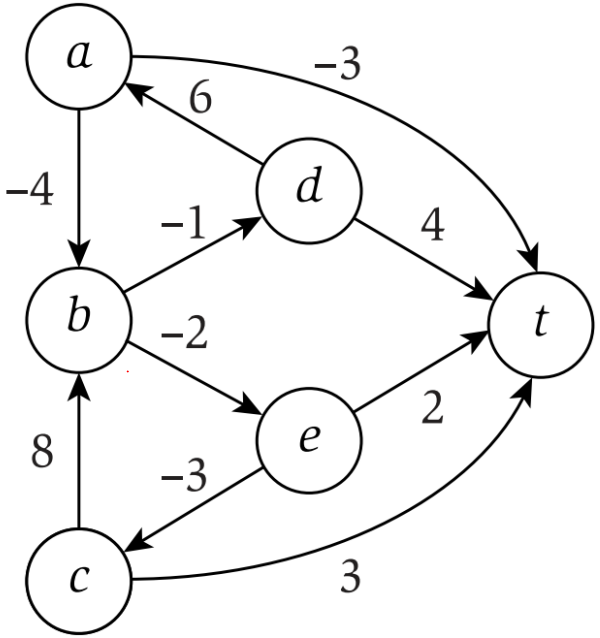
Bellman Ford

Using dynamic programming.
Subproblem: opt(i, v) is the minimum cost of v-t path using at most i edges.

- If path uses at most i-1 edge then opt(i, v) = opt(i-1, v)
- If path uses i edges and there is an edge (v, w) then opt(i, v) = opt(i-1,w) + c_{vw}

opt(i, v) = $\min\{OPT(i-1, v), \underbrace{\min_{w \in V}\{OPT(i-1, w) + c_{vw}\}}\}$

Bellman Ford



opt	0	1	2	3	4	5
t	0	0	0	0	0	0
a	inf	$\boxed{-3}$	−3	−4	−6	−6
b	inf	∞	$\boxed{0}$	−2	−2	−2
c	inf	$\boxed{3}$	3	3	3	3
d	inf	$\boxed{4}$	$\boxed{3}$	3	2	0
e	inf	2	0	0	0	0

Bellman Ford

Running time: Depends on the table size and time compute value at every index.
$O(n^3)$

Can we improve its running time?

$$OPT(i,v) = \min\left(OPT(i-1,v),\ \min_{w \in N_v} OPT(i-1,w) + c_{vw}\right)$$

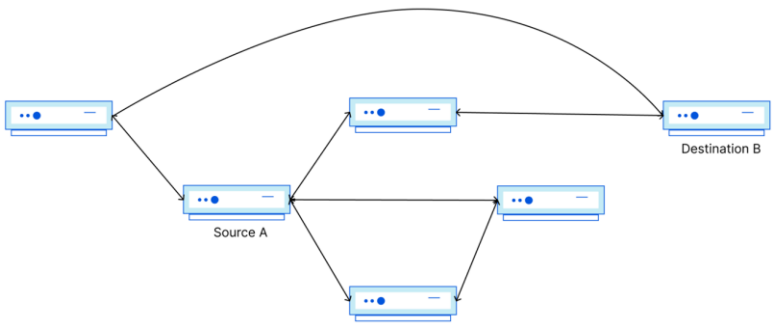
$$n \cdot \sum_{v \in V} N_v = O(n \cdot |E|)$$

Outline

- Shortest Path
  - Bellman Ford
- Distance Vector Protocols
- Network Flows

Application: Distance Vector Protocols

- Nodes are routers.
- Edges are direct links between routers.
- Edge weights are the cost represents dela
  the link.



Problem: Find a path from s to t with a minimum delay.
Delays are non-negative values. So, we can use Dijkstra.
In case of large network global knowledge is difficult.

Use Bellman Ford algorithm.

Problems

Design an algorithm that detects if a graph contains negative cycles.

Outline

- Shortest Path
  - Bellman Ford
- Distance Vector Protocols
- Network Flows

Network Flow

An (s, t)-flow is a function f: E → R that satisfies the following conservation constraints for every vertex v (possibly) except s and t.

$$\sum_u f(u \to v) = \sum_w f(v \to w)$$

Total flow into v is equal to the total flow out of v.

Flow is defined as |f|. It is the total flow out of the source s.

$$|f| := \sum_w f(s \to w) - \sum_u f(u \to s)$$

Prove that the net flow into t (the sink) is |f|.

Capacity is a function c: E → $R_{\geq 0}$.

Network Flow

An (s, t)-flow is a function f: E → R that satisfies the following conservation constraints for every vertex v (possibly) except s and t.

$$\sum_{u} f(u \to v) = \sum_{w} f(v \to w)$$

Capacity is a function c: E → $R_{\geq 0}$.

- A flow f is feasible if f(e) ≤ c(e) for every edge e in G.
- A flow f saturates an edge e if f(e) = c(e).
- A flow f avoids an edge e if f(e) = 0.

Problem: Given a graph with capacity find a maximum feasible flow.

Cut

An (s, t)-cut is a partition of vertices into <u>disjoint</u> set S and T, where s ∈ S and t ∈ T

The capacity of a cut is defined as

$$\|S, T\| := \sum_{v \in S} \sum_{w \in T} c(v \to w)$$

Problem: Given a graph with capacity find the minimum cut.

Reference

Slides

Jeff Erickson Chp-10

Algorithms Design by Kleinberg & Tardos - Chp 6.8 - 6.10

Algorithm Design & Analysis (CSE222)

Lecture-20

Recap

- Bellman Ford
  - Uses dynamic paradigm
  - Distance vector protocols

- Network Flows
  - Flows & Cut

Outline

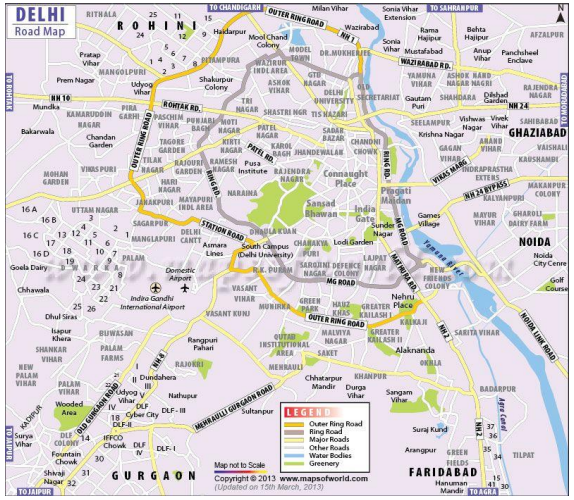
- Network Flows
  - Maximum Flow (Ford-Fulkerson)

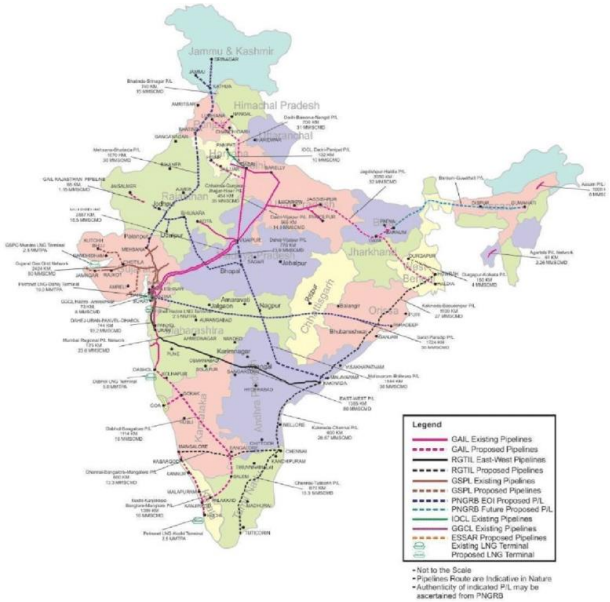
Flow function

Conservation constraint: In flow = out flow at every vertex, but s and t.
Capacity constraint: flow is less or equal to capacity at each edge.

Cut is a partition of vertex set into two disjoint sets.
Capacity of cut: total capacity of all edges whose nodes are in two disjoint sets.

Network Flow



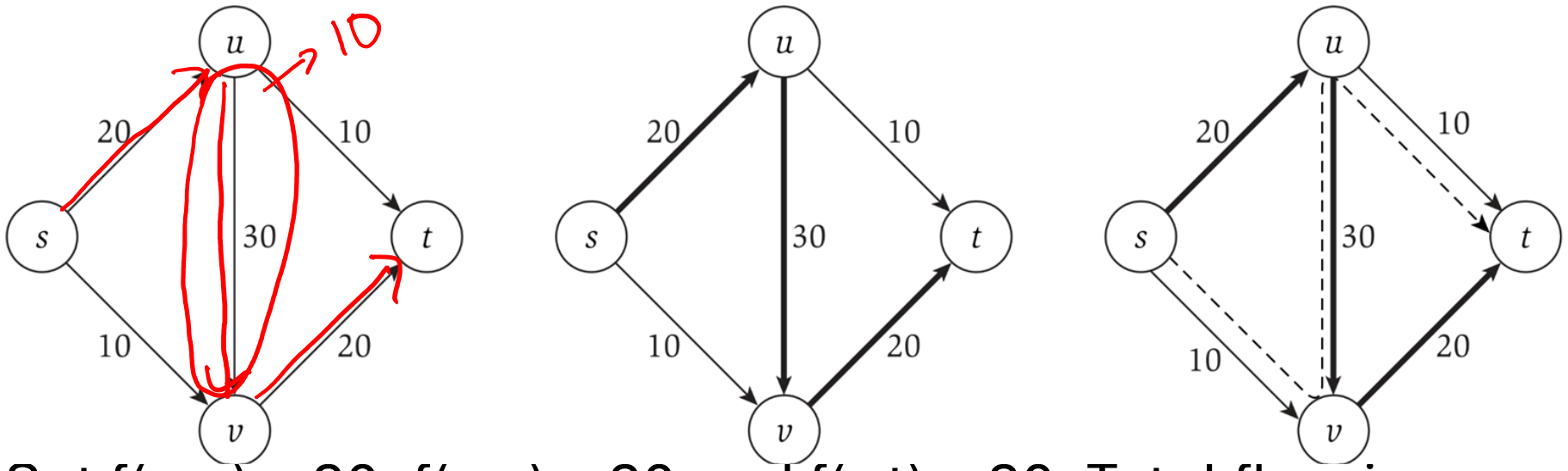


Flow network: single source and single sink.

Here we will only consider integer valued capacities.

Network Flow

<u>Flow network</u>: single source and single sink.

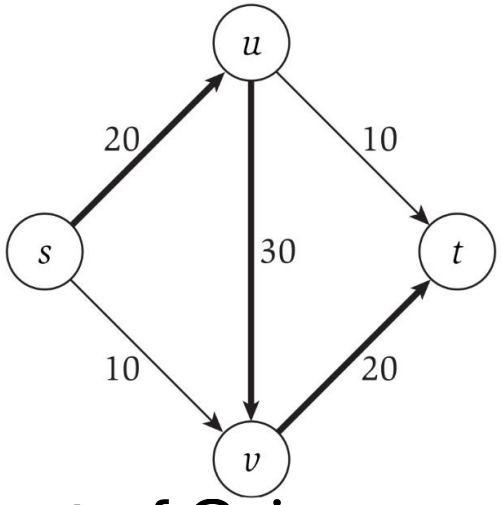


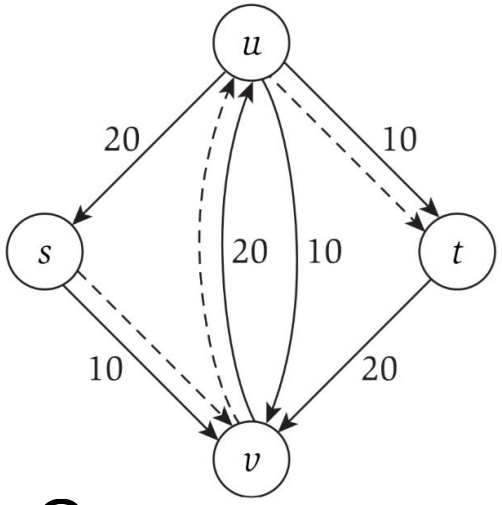
Set f(s,u) = 20, f(u,v) = 20 and f(v,t) = 20. Total flow increased to 20!
Is it the max flow of the graph?

Push forward on edge based on remaining capacity.
Push backward on edge based on flow.

Graph Residual

Given flow f on G (flow network), the residual graph G_f is defined as follows.

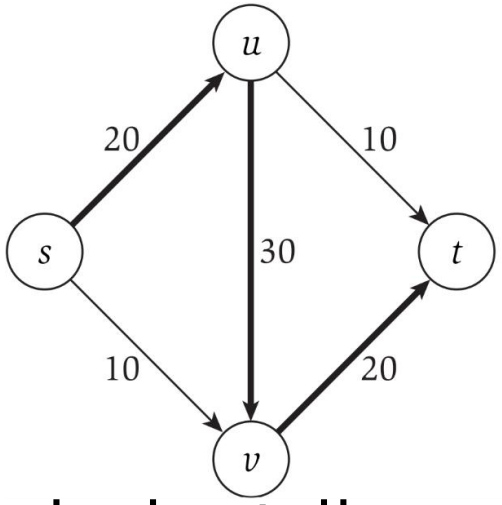


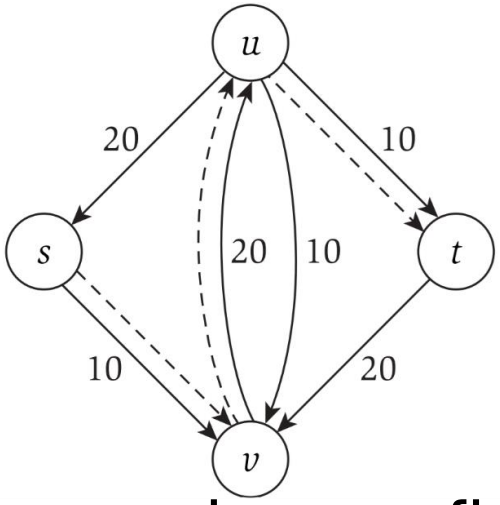


- Node set of G_f is same as G.
- For any e ∈ E, if f(e)<c(e) then include edge e in G_f with a capacity of c(e)-f(e). The edge is called forward edge with left over capacity c(e)-f(e).
- For any e = (u,v), if f(e)>0, then include e' = (v,u) with a capacity f(e). This edge is called backward edge.

Graph Residual

Given flow f on G (flow network), the residual graph G_f is defined as follows.





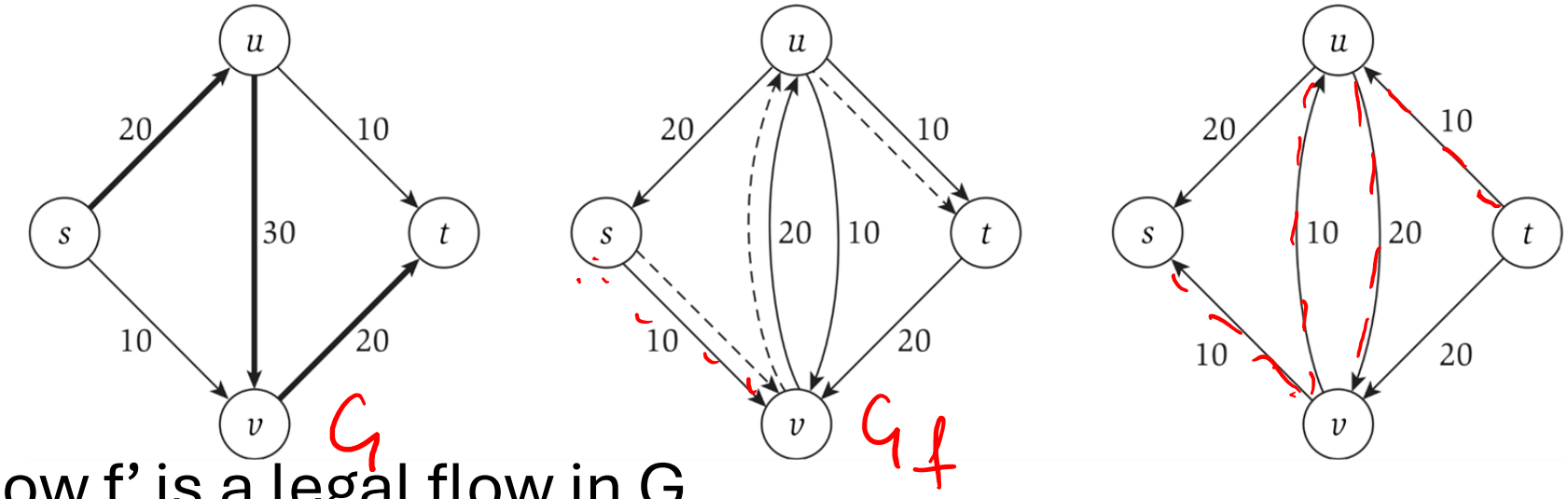
- Forward edge tells us how much more flow can be passed through the edge.
- Backward edge tells us how much flow can be undone through the edge.

Claim: G_f has most twice as many edges as G.

Augmenting Path

- Let P be a simple path in G_f.
- Let b is bottleneck of P in G_f.
- For every edge (u,v) in G_f,
  - If (u,v) is a forward edge then increase f(e) by b in G.
  - Else decrease f(e) by b in G.
- Return f.

(s, v, u, t) → 10



Claim: Returned flow f' is a legal flow in G.

Augmenting Path

<u>Claim</u>: Returned flow f' is a legal flow in G.

<u>Proof</u>: *Capacity Condition*

- The change between f and f' is only due to edge of P.
- If e = (u,v) is forward edge then f' ensures f'(e) ≤ c(e)

$0 \leq f(e) \leq f'(e) = f(e) + \text{bottleneck}(P, f) = f(e) + c(e) - f(e) = c(e).$

- If e' = (u,v) is backward edge then f' ensures f'(e) ≥ 0, where e = (v,u)

$c(e) \geq f(e) \geq f'(e) = f(e) - \text{bottleneck}(P, f) \geq f(e) - f(e) = 0.$

Augmenting Path

<u>Claim</u>: Returned flow f’ is a legal flow in G.

<u>Proof</u>: *Capacity Condition*

- The change between f and f’ is only due to edge of P.
- If e = (u,v) is forward edge then f’ ensures f’(e) ≤ c(e)
- If e’ = (u,v) is backward edge then f’ ensures f’(e) ≥ 0, where e = (v,u)

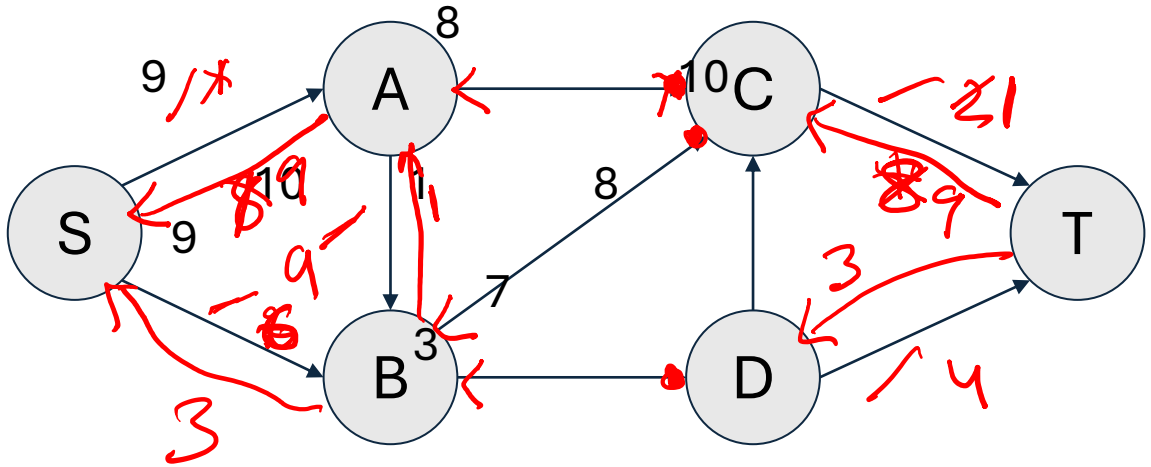
Conservation condition

- f followed conservation condition in G.
- The flow on the augmenting path ensures conservation condition in G.
- So, f’ follows conservation condition in G.

Maximum FLow (Ford Fulkerson)

```
Max-Flow
  Initially f(e) = 0 for all e in G
  While there is an s-t path in the residual graph G_f
    Let P be a simple s-t path in G_f
    f' = augment(f, P)
    Update f to be f'
    Update the residual graph G_f to be G_f'
  Endwhile
  Return f
```

S, A, C, T — 8
S A B C T — 1
S B D T — 3
12



Ford Fulkerson

<u>Claim</u>: At every intermediate step the flow values f(e) and the residual capacities in G_f are integers.

Does Ford Fulkerson algorithm always terminates?

While applying augmentations we have the following guarantee.

Claim: If f be a flow in G & P be a simple path s-t in G_f, |f'| = |f| + bottleneck(P,f). Since, bottleneck(P,f) > 0, hence |f'| > |f|.

Ford Fulkerson

<u>Claim</u>: If f be a flow in G & P be a simple path s-t in G_f, |f'| = |f| + bottleneck(P,f). Since, bottleneck(P,f) > 0, hence |f'| > |f|.

<u>Proof</u>:

- The first edge of P must be an edge out of s in G_f. It doesn't visit s again as P is a simple path.
- So, the first edge must be a forward edge, to which we increase the flow value by bottleneck(P,f).
- Since we do not change flow value of any edge going in s, |f'| > |f|.

Ford Fulkerson

<u>Claim</u>: The Ford-Fulkerson algorithm terminates in at most C iterations, where C is the total capacity from s.

<u>Proof</u>: We know the following statements to be true at each iteration

- $|f'|$ is an integer.
- $|f'| > |f|$
- For every edge $f'(e) \leq c(e)$.

So, the number of iterations of the while loop is bounded by C.

Ford Fulkerson

Running time:

We know that at most C times the algorithm computes augmenting path.

Let T be the running time of augmenting path.

The running time of Ford-Fulkerson is O(TC).

What is the most efficient value of T?

Can you find maximum flow in a flow graph that rational valued capacities?

Can you find maximum flow in a flow graph that rational valued capacities?

Reference

Slides

Algorithms Design by Kleinberg & Tardos - Chp 7.1

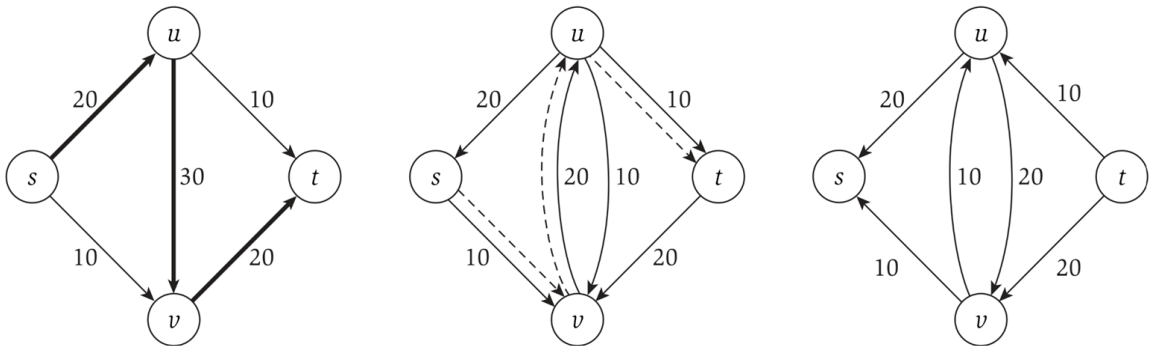
Algorithm Design & Analysis (CSE222)

Lecture-21

the maximum flow problem asks for the maximum rate
at which some resource can be moved from s to t; the minimum cut problem
asks for the minimum damage needed to separate s from t.

Recap

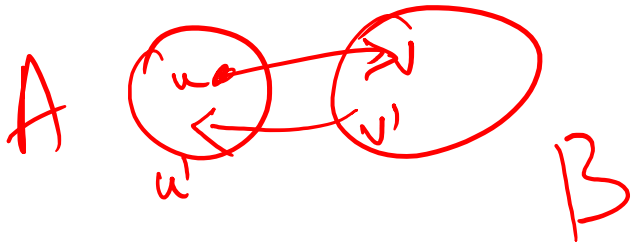
- Network Flow
  - Maximum flow (Ford-Fulkerson).
  - Augmenting path on G_f (increases flow).
  - Augmenting path insures capacity constraints & algorithm ensures conservation constraints.
  - Tractable running time for integer and rational valued capacities.



Outline

- Network Flows

Network Flow



Does Ford-Fulkerson returns the maximum possible flow in G?

- Relation between flow f and total capacity C at source: $|f| \leq C$.
- A <u>cut</u> is a partition of vertices into <u>disjoint</u> set A and B, where $s \in A$ and $t \in B$. The capacity of a cut is $c(A,B) = \sum_{e:A \to B} c(e)$.

Claim: Given a flow network, let f be a s-t flow and (A,B) be a s-t cut. Then the total flow $|f| = f^{out}(A) - f^{in}(A)$.

Network Flow

<u>Claim</u>: Given a flow network, let f be a s-t flow and (A,B) be a s-t cut. Then the total flow $|f| = f^{out}(A) - f^{in}(A)$.

$f^{out}(s) - f^{in}(s) + \sum_{v \in A \setminus s} f^{out}(v) - f^{in}(v)$

<u>Proof</u>:

- Def: $|f| = f^{out}(s)$. It implies $|f| = f^{out}(s) - f^{in}(s)$.
- Further, $|f| = \sum_{v \in A} f^{out}(v) - f^{in}(v) = \sum_{e \text{ from } A} f(e) - \sum_{e \text{ to } A} f(e) = f^{out}(A) - f^{in}(A)$.

Similar claim: Given a flow network, let f be a s-t flow and (A,B) be a s-t cut. Then the total flow $|f| = f^{in}(B) - f^{out}(B)$.

Network Flow

<u>Claim</u>: Given a flow network, let f be a s-t flow and (A,B) be a s-t cut. Then the total flow $|f| \leq c(A,B)$.

<u>Proof</u>: We know, $|f| = f^{out}(A) - f^{in}(A) \leq f^{out}(A) = \sum_{e \text{ from } A} f(e) \leq \sum_{e \text{ from } A} c(e) = c(A,B)$.

In terms of bounding |f| the above claim is weaker than previous claim. Why?

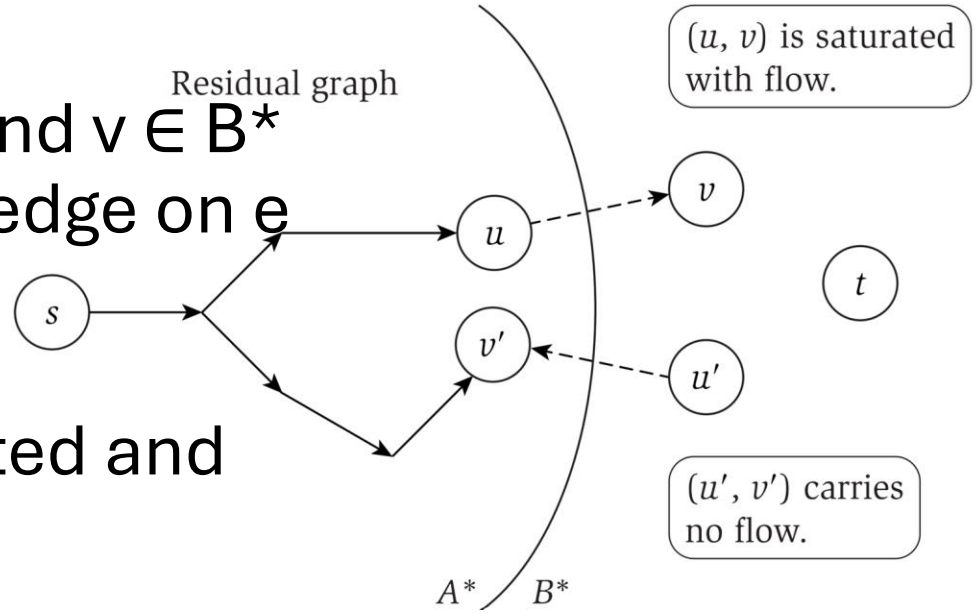
However the above claim is stronger in general. Why?

Max-flow & Min-cut

<u>Claim</u>: For a flow f in a flow graph there is a cut (A*,B*) such that |f| = c(A*,B*).

<u>Proof</u>: Let A* be the set of all nodes such that there is path s-v in G_f & B* = V \ A*.
(A*,B*) is a cut.

- Let, e = (u,v) be an edge such that u ∈ A* and v ∈ B*
- Then f(e) = c(e). Since there is no forward edge on e
- Let e' = (u',v') edge in G, u'∈B* & v'∈A*.
- Then f(e') = 0. As no backward edge in G_f.
- So all edge from A* are completely saturated and all edge into A* are completely unused.



So, $|f| = f^{out}(A^*) - f^{in}(A^*) = \sum_{e \text{ from } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) = \sum_{e \text{ from } A^*} c(e) - 0 = c(A^*,B^*)$.

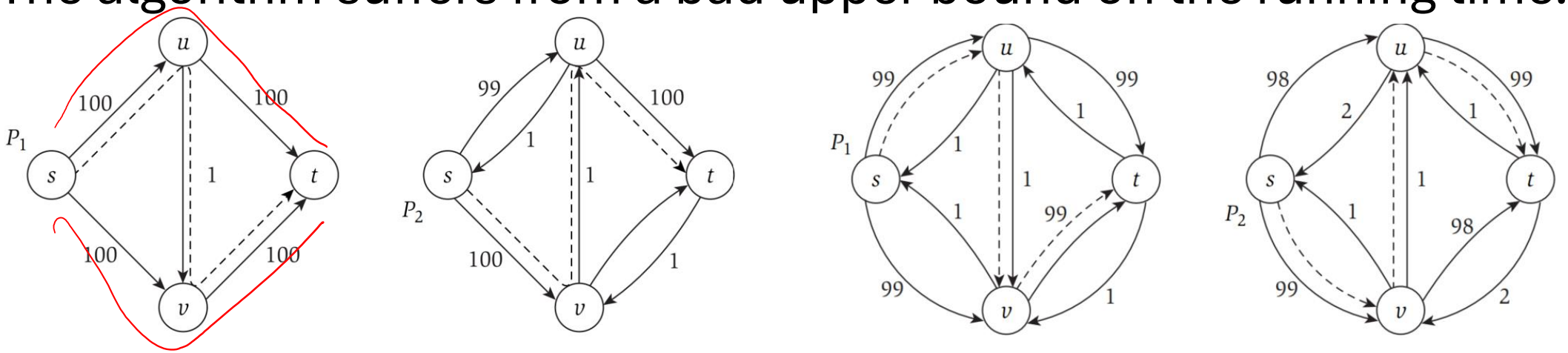
Network Flow

<u>Claim</u>: Given a max flow f, one can compute s-t min cut in O(m) time.

<u>Proof</u>: Based on the previous claim.

Ford-Fulkerson

The algorithm suffers from a bad upper bound on the running time.



How to improve the running time: Choose a random path s-t in G_f.

- Guarantees, that over expectation the augmenting path with worst bottleneck not selected. Bounding running time not easy.
- Augmenting path with highest bottleneck capacity in G_f. How to find?
- Large enough bottleneck capacity. How to decide?

Scaling Max Flow

CAPACITY-SCALING(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$\Delta \leftarrow$ largest power of 2 $\leq C$.

WHILE ($\Delta \geq 1$)

$G_f(\Delta) \leftarrow \Delta$-residual network of G with respect to flow f.

WHILE (there exists an $s \rightsquigarrow t$ path P in $G_f(\Delta)$)

$f \leftarrow$ AUGMENT(f, c, P).

Update $G_f(\Delta)$.

Δ-scaling phase

$\Delta \leftarrow \Delta / 2$.

RETURN f.

The residual graph $G_f(\Delta)$ contains only those edges whose capacity $\geq \Delta$.

Every augmenting path in this residual graph has a bottleneck at least Δ.

Reference

Slides

Jeff Erickson Chp-10

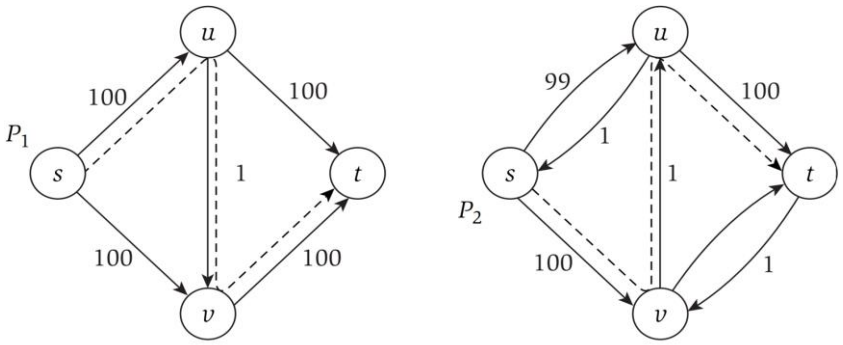
Algorithms Design by Kleinberg & Tardos - Chp 7.2 & 7.3

Algorithm Design & Analysis (CSE222)

Lecture-22

Recap

- Network Flow
  - Min Cut
  - Max-Flow Min-Cut Theorem



- Scaling Max-Flow

CAPACITY-SCALING(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$\Delta \leftarrow$ largest power of 2 $\leq C$.

WHILE ($\Delta \geq 1$)

$G_f(\Delta) \leftarrow \Delta$-residual network of G with respect to flow f.

WHILE (there exists an $s \rightsquigarrow t$ path P in $G_f(\Delta)$)

$f \leftarrow$ AUGMENT(f, c, P).

Update $G_f(\Delta)$.

$\Delta \leftarrow \Delta / 2$.

Δ-scaling phase

RETURN f.

Outline

- Scaling Max-Flow

- Shortest Path

Scaling Max Flow

Observation

- If the capacities are integers then the flow and the residual capacities are also integers.
- At $\Delta = 1$ we get $G_f(\Delta) = G_f$.
- When it terminates, the flow f is the maximum flow.

It is important to analyze the number of augmentation done in each phase, fixed Δ

Claim: There are at most $1+\log C$ scaling phases in the algorithm.

Scaling Max Flow

<u>Claim</u>: There are at most 1+log C scaling phases in the algorithm.

<u>Proof</u>:

- The total capacity is C.
- Starting with $\Delta' = C$, in every subsequent phase we update $\Delta = \Delta'/2$.
- So at max the the algorithm goes over 1 + log C scaling phases.

Scaling Max Flow

<u>Claim</u>: Let f be a flow at the end of Δ-scaling phase. There is an s-t cut (A,B) such that $c(A,B) \leq |f| + m\Delta$. So possible max flow is $|f| + m\Delta$.

<u>Proof</u>: Let A be the set of all nodes such that there is path s-v in $G_f(\Delta)$ & $B = V \setminus A$.

(A,B) is a cut: $s \in A$ as there is always a path s-s.
As, no path s-t in $G_f(\Delta)$ so, $t \notin A$ hence, $t \in B$.

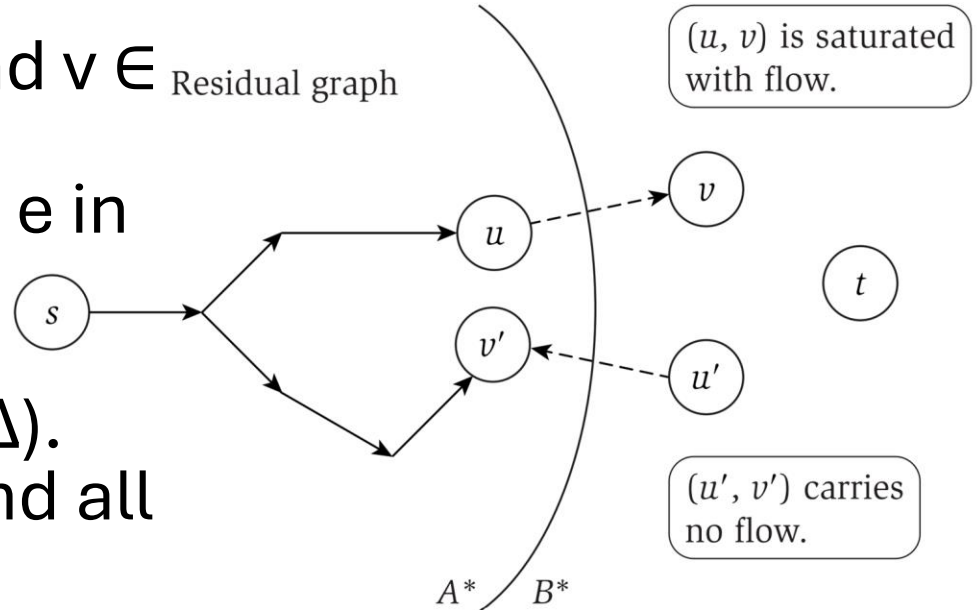
Scaling Max Flow

<u>Claim</u>: Let f be a flow at the end of Δ-scaling phase. There is an s-t cut (A,B) such that $c(A,B) \le |f| + m\Delta$. So possible max flow is $|f| + m\Delta$.

<u>Proof</u>: Let A be the set of all nodes such that there is path s-v in $G_f(\Delta)$ & $B = V \setminus A$.

(A,B) is a cut.

- Let, e = (u,v) be an edge such that $u \in A$ and $v \in B$
- Then $c(e) < f(e) + \Delta$. As no forward edge on e in $G_f(\Delta)$.
- Let e' = (u',v') edge in G, $u' \in B$ & $v' \in A$.
- Then $f(e') < \Delta$. As no backward edge in $G_f(\Delta)$.
- So all edge from A are almost saturated and all edge into A are almost unused.



So, $|f| = f^{out}(A) - f^{in}(A) > \sum_{e \text{ from } A^*}[c(e)-\Delta] - \sum_{e \text{ into } A^*} \Delta > \sum_{e \text{ from } A^*} c(e) - m\Delta = c(A,B) - m\Delta$.

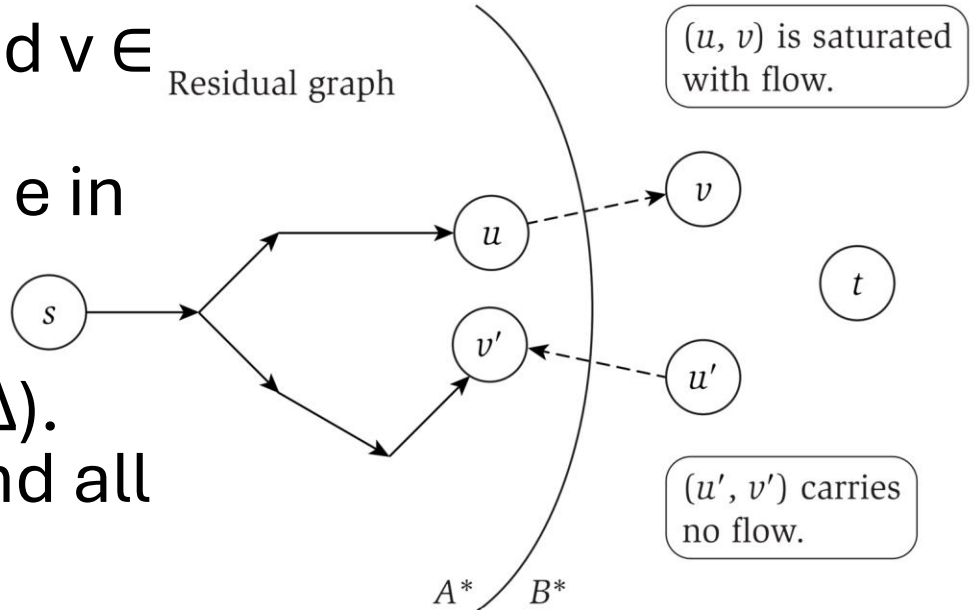
Scaling Max Flow

<u>Claim</u>: Let f be a flow at the end of Δ-scaling phase. There is an s-t cut (A,B) such that c(A,B) ≤ |f| + mΔ. So possible max flow is |f| + mΔ.

<u>Proof</u>: Let A be the set of all nodes such that there is path s-v in $G_f(\Delta)$ & B = V \ A.

(A,B) is a cut.

- Let, e = (u,v) be an edge such that u ∈ A and v ∈ B
- Then c(e) < f(e) + Δ. As no forward edge on e in $G_f(\Delta)$.
- Let e' = (u',v') edge in G, u'∈B & v'∈A.
- Then f(e') < Δ. As no backward edge in $G_f(\Delta)$.
- So all edge from A are almost saturated and all edge into A are almost unused.



So, c(A,B) < |f| + mΔ. Hence max flow is at most |f| + mΔ.

Scaling Max Flow

<u>Claim</u>: Number of augmentations in a scaling phase is at most 2m.

<u>Proof</u>:

- At first phase it is trivially true as every edge is used for only one path s-t.
- At any intermediate scaling phase Δ, let f_p be the flow by the end of the previous phase.
- In the previous phase we used $\Delta' = 2\Delta$.
- By previous claim $|f^*| \leq |f_p| + m\Delta' = |f_p| + 2m\Delta$
- In the Δ-scaling phase each augmentation increases the flow by at least Δ
- So, there can be at most 2m augmentations.

Scaling Max Flow

- There are $1 + \log_2 C$ scaling phases.
- In each phase there are at most 2m computations of augmenting paths.
- Every augmentation takes O(m) time.

<u>Claim</u>: Total running time of Scaling Max Flow is $O(m^2 \cdot \log_2 C)$.

Is it better than Ford-Fulkerson algorithm?

Outline

- Scaling Max-Flow

- Shortest Path

Shortest Augmenting Path (Edmonds-Karp)

- In the residual graph, choose the shortest (#edges) path s-t.
- Let f_i be the flow after i^{th} augmentation step. So f_0: E → 0.
- Let G_i be the corresponding residual graph. So, $G_0 = G$.
- For each node v $level_i(v)$ be the simple shortest path s-v or level of v in BFS(s).
- If v is unreachable from s in G_i then $level_i(v) = \infty$.

Claim: The level of a vertex can only increase over time.

Shortest Augmenting Path (Edmonds-Karp)

<u>Claim</u>: For every node v and integer $i > 0$, $level_i(v) \geq level_{i-1}(v)$.

<u>Proof</u>: Fix a node v and prove by induction.
Inductive hypothesis: assume for every node u, $level_i(u) \geq level_{i-1}(u)$.

- If v=s then $level_i(v) = level_i(s) = 0$.
- If no path s-v then $level_i(v) = \infty \geq level_{i-1}(v)$.
- Let $s \longrightarrow \ldots \longrightarrow u \longrightarrow v$ be simple shortest path s-v in G_i. So, $level_i(v) = level_i(u) + 1 \geq level_{i-1}(u) + 1$.
  - If $u \rightarrow v$ is an edge in G_{i-1}, then $level_{i-1}(v) \leq level_{i-1}(u) + 1$. As levels are defined by BFS(s).
  - If $u \rightarrow v$ is not an edge in G_{i-1} then its reverse $v \rightarrow u$ must be an edge in i^{th} augmenting path.
  - This is simple shortest path s-t in G_{i-1}. So, $level_{i-1}(v) = level_{i-1}(u) - 1 \leq level_{i-1}(u) + 1$.
- So we conclude, $level_i(v) \geq level_{i-1}(u) + 1 \geq level_{i-1}(v) -1 + 1 = level_{i-1}(v)$.

Shortest Augmenting Path (Edmonds-Karp)

<u>Claim</u>: Each edge u→v disappears from the residual graphs G_i at most |V|/2 times.

<u>Proof</u>:

- Let u→v edge is in two residual graph G_i and G_{j+1} but not in any intermediate G_l for $i < l < j$, for some $i < j+1$.
- So, u→v must be in the ith augmenting path. So, $level_i(v) = level_i(u) + 1$.
- So, v→u must be in the jth augmenting path. So, $level_j(u) = level_j(v) + 1$ or $level_j(v) = level_j(u) - 1$.

We know that,

$level_j(u) = level_j(v) + 1 \geq level_i(v) + 1 = level_i(u) + 2$.

- The level difference between appearance and disappearance is 2.
- There are at most |V| possible labels, so #disappearance is at most |V|/2.

Shortest Augmenting Path (Edmonds-Karp)

Running time

- Each edge disappears at most V/2 times.
- There are EV/2 many number of disappearance.
- Each iteration takes O(E) time to compute the simple shortest s-t path.
- Finally the running time is $O(VE^2)$.

Reference

Slides

Jeff Erickson Chp-10

Algorithms Design by Kleinberg & Tardos - Chp 7.3

Algorithm Design & Analysis (CSE222)

Lecture-23

Recap

- Scaling Max-Flow
  - In the residual graph, choose the s-t path with high bottleneck.
  - Running time: $O(E^2 \cdot \log_2 C)$

- Shortest Path
  - In the residual graph, choose the shortest (#edges) path s-t.
  - Running time: $O(E^2 \cdot V)$

Outline

- Network Flow: Applications

- Computational Intractability

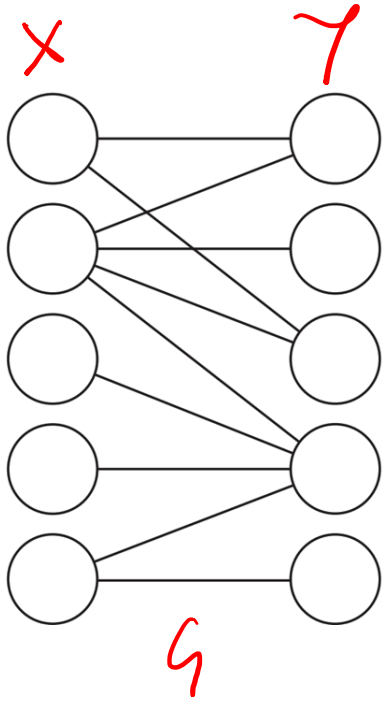
Bipartite Matching

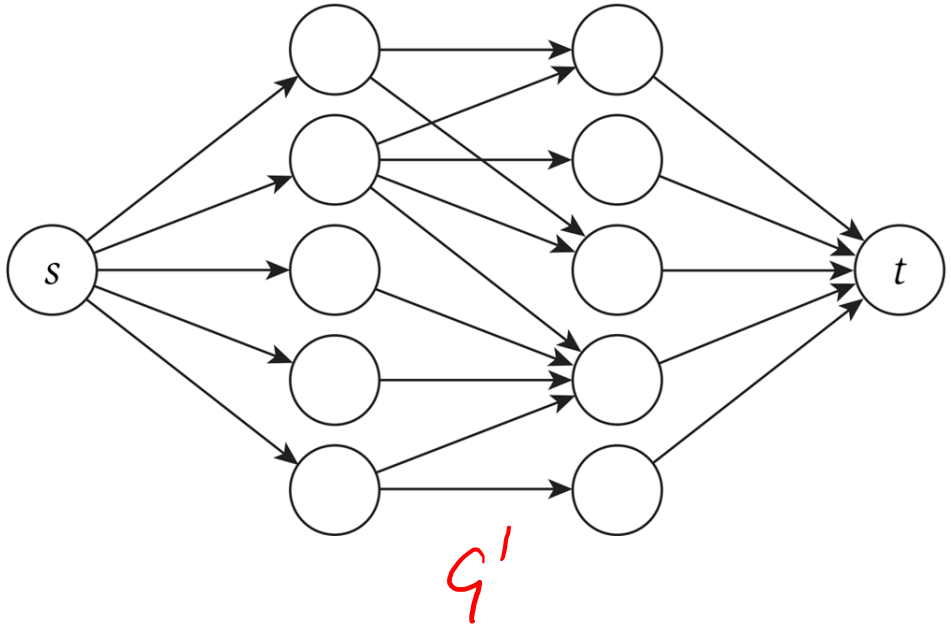
Let G = (V, E) be an undirected graph such that V = X U Y and for every e ∈ E, one end point is in X and another is in Y. A matching M ⊆ E such that each node appears in at most one edge in M.

Problem: Compute maximum matching of G.

Given an instance of bipartite matching create an instance of maximum flow.

- Give directions.
- Add s and t.
- Assign capacity 1.

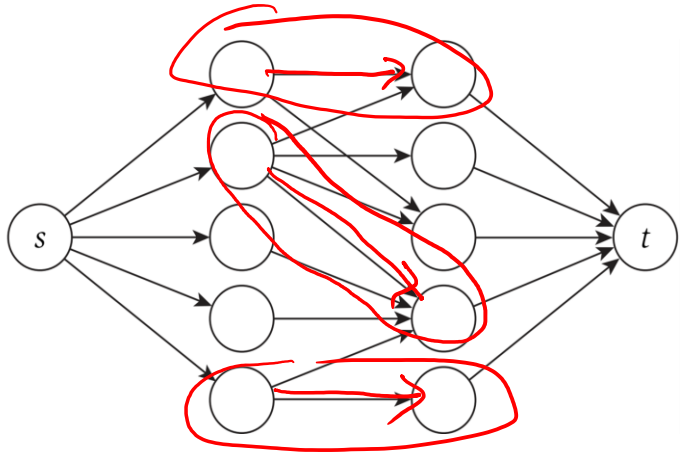




Bipartite Matching

Compute maximum flow.

Let there is a matching of k edges (x_1,y_1), (x_2,y_2),...(x_k,y_k). Each edge (x_i,y_j) will pass 1 unit of ~~flow s-x_i-y_j-t, $f(x_i,y_j)$~~=1. Since, capacity of each edge is 1, so due to both constraints the total flow is k.



Again, let f' is a flow of value k. Then by integrality property and all capacities being 1, for every edge f(e) is 1 or 0. Let, M' be the set of edges (x,y) such that f(x,y) = 1.

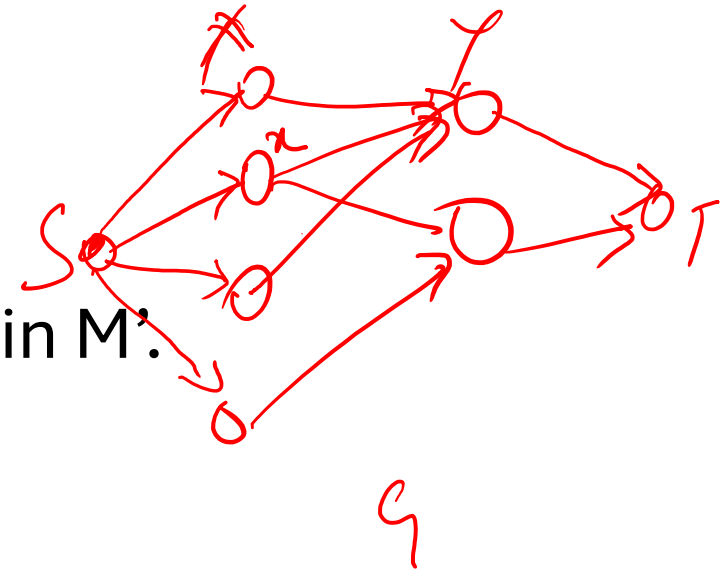
Bipartite Matching

<u>Claim</u>: M’ has k edges.

<u>Proof</u>: Let A = s U X and B = t U Y.

- Value of flow is total flow leaving A minus total flow entering A.
- The first terms is of |M’|, as all edge carries 1 unit of flow.
- The second term is 0 as there is no edge that goes into A in the graph.
- So, M’ contains k edges.

Bipartite Matching



<u>Claim</u>: Each node in X is the tail of at most one edge in M'.

<u>Proof</u>: Prove by contradiction.

- Let x ∈ X be tail of at least two edges in M'.
- As flow is integer valued, so at least two unit of flow leaves x.
- By conservation constraints, at least two unit of flow would have come to x.
- This is impossible as only one edge with capacity 1 comes into x from s.
- So, x is the tail of at most one edge in M'.

<u>Claim</u>: Each node in Y is the head of at most one edge in M'.

Bipartite Matching

Combining all the claims we get,

<u>Claim</u>: The size of maximum matching is equal to the value of maximum flow in G’ and the edges in such a matching G are the ones that carries flow X to Y in G’.

Outline

- Network Flow: Applications

- Computational Intractability

Computational Intractability

- Informal categorization of problems

- Polynomial time algorithms

<u>NP-Completeness</u>

- A large class of problems are equivalent. A polynomial time algorithm to any one of them implies existence of polynomial time algorithm for all of them.
- Exists thousands of such problems.
- All these problems are just one single open problem.

NP-Completeness

- Computationally hard, though we cannot prove it.

- Identifying a problem to be NP-Complete is a good enough reason to stop looking for an efficient algorithm for it.

- Compare relative difficulty of different problems.

Reductions

To prove some problems are computationally difficult we show,

- ‘Problem X is at least as hard as problem Y’

To claim this we reduce problem Y to problem X: If there is a black box that solves instances of X. How to solve any instance of Y in polynomial number of steps and polynomial number of calls to the black box that solves X?

If yes then denote $Y \leq_p X$.
Read as: ‘Y is polynomial time reducible to X’ or ‘X is as hard as Y with respect to polynomial running time’

Reduction

If $Y \leq_p X$, and X is polynomial time solvable then we can replace the black box call to X with the polynomial time algorithm for X. Recall Y was solved in polynomial number of steps and polynomial number of calls to the black box.

Now solution of Y involves polynomial number of steps and polynomial number of calls to a subroutine (algorithm for X) that runs in polynomial time. Hence, solving Y takes polynomial time.

<u>Claim</u>: Let $Y \leq_p X$. If X can be solved in polynomial time then Y can be solved in polynomial time.

Reduction

Y is polynomial time reducible to X: $Y \leq_p X$

Although we use X to solve instances of Y, it does not mean that X is less harder than Y or Y is at least as hard as X.

Problem: (Y) Smallest number in an array of size n.
It takes O(n).
(X) Sort the array in ascending order.
It takes O(nlog n).

$Y \leq_p X$. Solving Y by reducing to X takes O(nlog n + n) = O(nlog n).

Reduction

<u>Claim</u>: Let $Y \leq_p X$. If Y cannot be solved in polynomial time then X cannot be solved in polynomial time.

<u>Proof</u>:

- If we have a problem Y that is known to be hard and $Y \leq_p X$ then the hardness extends to problem X.
- Else we could use X to solve Y.

So, if we can find one hard problem (Y) then we can show that another problem (X) is hard by reducing Y to X.

Reduction Example: Max Bipartite Matching $\leq_p$ Max Network Flow.

Vertex Cover

A vertex cover of a graph G = (V, E) is a set of nodes S such that every edge has at least one endpoints in S.

Cover each edge by choosing at least one of its vertices.

<u>Problem</u>: Given a graph G and a number k, does G contains a vertex cover of size at most k.

Independent Set

An independent set of a graph G = (V, E) is a set of nodes S such that no two nodes are adjacent to each other.

<u>Problem</u>: Given a graph G and a number k, does G contains an independent set of size at least k.

Relation b/w Independent Set and Vertex Cover

<u>Claim</u>: Given a graph G = (V, E), iff S is an independent set of G then V \ S is a vertex cover of G.

<u>Proof</u>: Both independent set and vertex cover are defined using edges.

- Suppose S is an independent set and let e = (u, v) be some edge.
- At most one of u or v can be in S. Hence, at least one of u or v is in V-S.
- So V \ S is a vertex cover.

- Let V \ S is a vertex cover and u, v ∈ S.
- Then there could not be an edge between u and v, else edge would be covered in V \ S.
- So, S is an independent set.

Optimization Vs Decision

Independent Set

Optimization Problem: Compute maximum independent set in given graph G.

Decision Problem: Given a graph G and a number k check if there exists an independent set of size at least k.

Both problems are same: Solving one would solve other.

How solving optimization problem solves decision problem? (trivial)

How solving decision problem solves optimization problem? (non-trivial)

If decision problem is hard then its optimization is also hard.

Independent Set $\leq_p$ Vertex Cover

We need to show any instance of independent set into an instance of vertex cover

$|V| = n$

- Given an instance of independent set {G, k},
- Query vertex cover black box if there is a vertex cover of size ≤ n-k.

- If yes from black box then there is an independent set of size ≥ k
- If no from black box, there is no VC of size ≤ n-k so there is no IS of size ≥ k

Vertex Cover $\leq_p$ Independent Set

<u>Claim</u>: Vertex cover $\leq_p$ Independent set

<u>Proof</u>: To verify if G has a vertex cover of size at most k, we ask independent set black box to check independent set of size at least n-k.

So, both the problems are equivalently difficult.

Reference

Slides

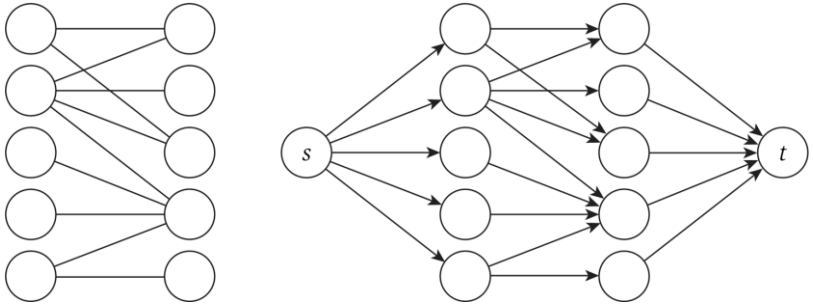
Algorithms Design by Kleinberg & Tardos - Chp 8.1

Algorithm Design & Analysis (CSE222)

Lecture-24

Recap

- Network Flow: Bipartite Matching
  - Give directions.
  - Add s and t.
  - Assign capacity 1.



- Computational Intractability
  - Decision version is equivalent to optimization version.
  - Vertex cover and independent set problems are equivalent.
    - Vertex Cover $\leq_p$ Independent Set & Independent Set $\leq_p$ Vertex Cover

Outline

- P and NP
- Reductions

Problem Class

- P: Problems solvable in polynomial time

- NP: Decision problems solvable in <u>nondeterministic</u> polynomial time.
  - Guess a solution in polynomial time.
  - In polynomial time the solution can be verified if it outputs yes to the decision problem.

Satisfiability

3SAT Problem: Verify if the following boolean expression can be satisfied or not?

$(u_1 \vee \bar{u}_2 \vee u_4) \wedge (u_1 \vee u_4 \vee \bar{u}_3) \wedge (u_2 \vee u_3 \vee \bar{u}_4) \ldots$

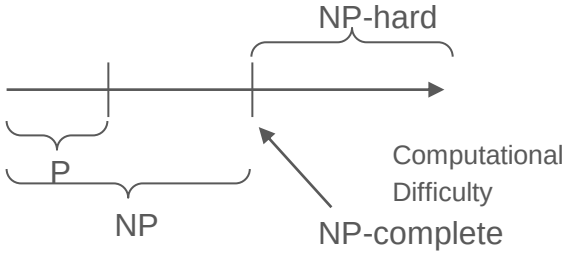
3SAT $\in$ NP.

Given a boolean expression can your friend convince you in polynomial time that the expression is satisfiable?

Given a boolean expression can your friend convince you in polynomial time that the expression is not satisfiable?

NP

- P: Problems solvable in polynomial time.
  - Problems are efficiently solvable.
- NP: Decision problems solvable in <u>nondeterministic</u> polynomial time.
  - These are difficult problems, for which there are no known efficient algorithm to solve them.
- NP-complete: A problem X is NP-complete if $X \in NP$ and it is possible to reduce $Y \leq_p X$ for any NP problem Y.
- NP-hard: A problem X is NP-hard, if there is an NP-complete problem Y such that $Y \leq_p X$.
  - Problem X is at least as hard as problems in NP.



NP

<u>Claim</u>: $P \subseteq NP$

<u>Proof</u>: To show, a problem in P is also in NP.

- Let X be a decision problem such that, $X \in P$. So there is an polynomial time algorithm A that solves X.
- Let B be a certifier for X that verifies X in polynomial time.
- Define B: Let s be an input instance. If A(s) solves X then return Yes, else No.
- Due to existence of nondeterministic polynomial time solver, $X \in NP$.

<u>Claim</u>: (Transitivity) If $Z \leq_p Y$ and $Y \leq_p X$, then $Z \leq_p X$.

<u>Proof</u>:

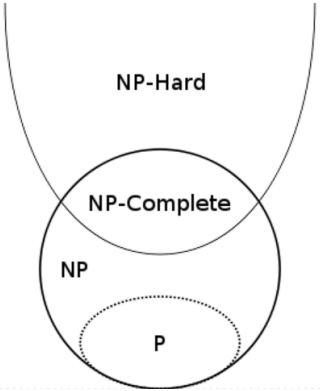
- Solve Z by running a black box on Y. But black box of Y is a polynomial steps and polynomial black box call of X. So, $Z \leq_p X$.

NP

Question: Is there a problem NP that does not belong P?

Alternate: Is P = NP?

<u>Claim</u>: X is an NP-complete problem. X is solvable in poly time if and only if P = NP.



<u>Proof</u>:

- (⇒) P = NP then X can be solved in poly time as X ∈ NP.
- (⇐) If X can be solved in poly time and we know for every Y ∈ NP, $Y \leq_p X$, then every Y can also be solved in poly time.

NP-Complete

How to prove problem X is NP-Complete?

- Prove $X \in NP$
  - Show, given a certificate and verifier the problem can be solved in polynomial time.
- Prove for every problem $Y \in NP$ can be reduced to X.
  - Nontrivial!
  - Reduce from a known NP-complete problem Y to X.
  - Cook Levin ‘70: 3SAT is complete.

NP-Complete

<u>Claim</u>: If Y is NP-complete and $X \in NP$ such that $Y \leq_p X$, then X is NP-complete.

<u>Proof</u>: We know $X \in NP$.

- Let $Z \in NP$, then by NP-completeness we know $Z \leq_p Y$.
- We are given that $Y \leq_p X$, then by transitive $Z \leq_p X$. So X is NP-complete.

Outline

- Network Flow: Applications

- Reductions

3SAT & Independent Sets

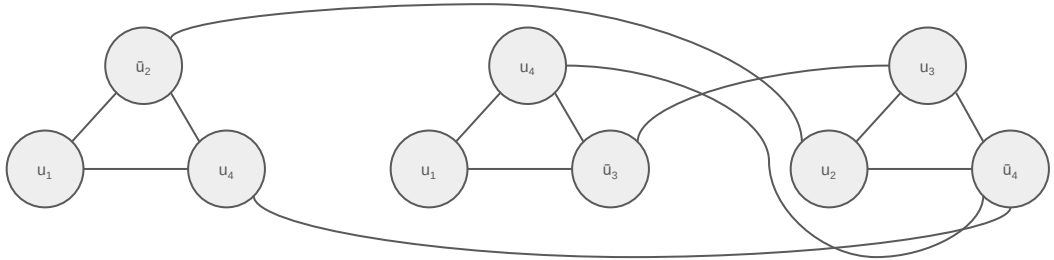
3SAT $\leq_p$ Independent Set: Boolean expression to a graph problem.

The boolean expression will be satisfiable if and only if we get an independent set in our special graph.

<u>Input:</u> $(u_1 \vee \bar{u}_2 \vee u_4) \wedge (u_1 \vee u_4 \vee \bar{u}_3) \wedge (u_2 \vee u_3 \vee \bar{u}_4)$

<u>Graph construction</u>

- Make a clique from every clause.
  - The independent set picks at most one vertex from every clause.
- Join every u_i and $\bar{u}_i$ with an edge.
  - The independent set picks only one from u_i and $\bar{u}_i$.
- Take k number of clauses.



3SAT & Independent Sets

<u>Claim</u>: The boolean expression F is satisfiable iff G_F has an independent set of size k (number of clause in F).

<u>Proof</u>:

- (⇒) If F is satisfiable then each triangle in the graph has at least one which evaluates to 1. Select one such node from each triangles and construct S.
- The set S has to be an independent set.
- (⇐) If S is an independent set, then assign the variables of these vertices to 1
- Graph ensures no conflict and touches every clause.
- So F is satisfiable.

Reference

Slides

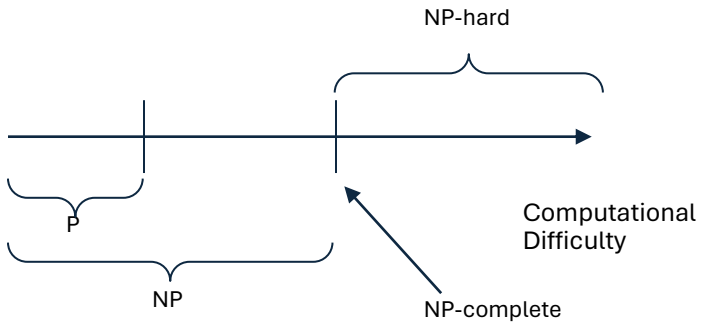
Algorithms Design by Kleinberg & Tardos - Chp 8.2 & 8.3

Algorithm Design & Analysis (CSE222)

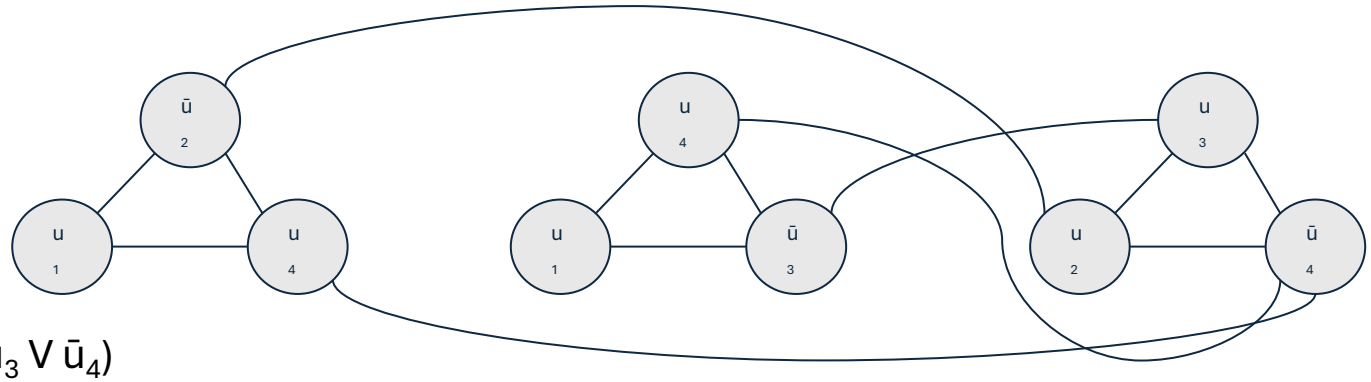
Last-Lecture

Recap

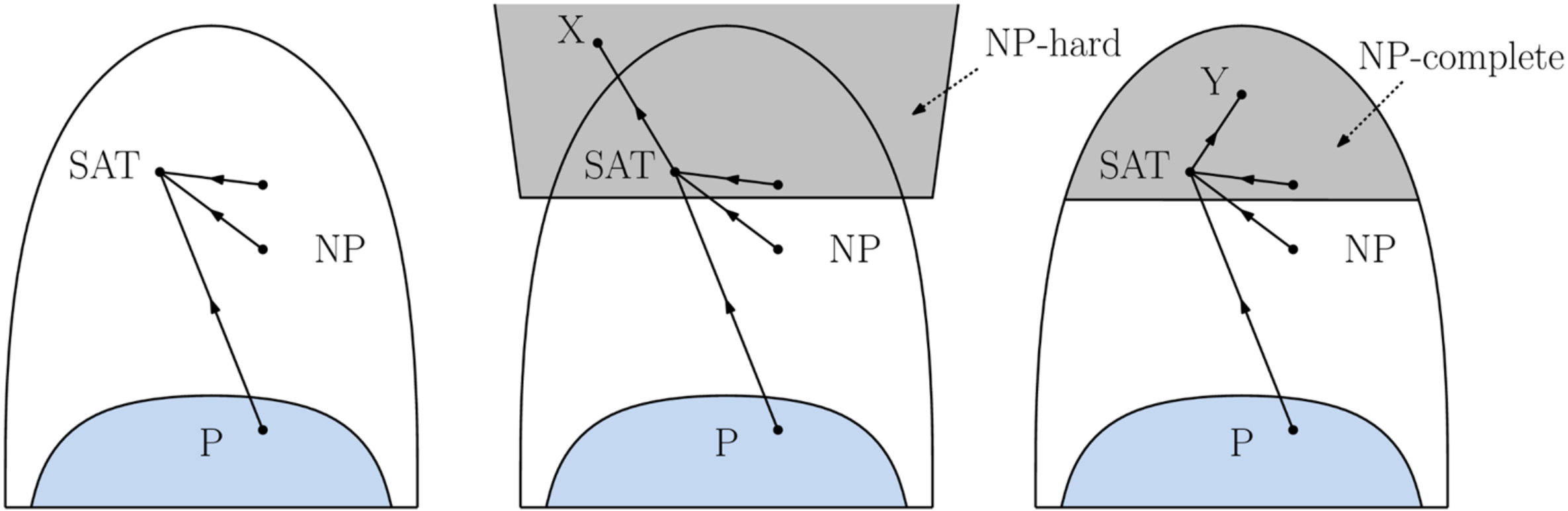
- P vs NP
  - $P \subseteq NP$
  - If $Z \leq_p Y$ and $Y \leq_p X$, then $Z \leq_p X$
  - For an NP-complete problem X, it is polynomial time solvable if and only if P = NP.
  - 3SAT is NP-complete.



- Reductions
  - 3SAT $\leq_p$ Independent Set.
    - $(u_1 \vee \bar{u}_2 \vee u_4) \wedge (u_1 \vee u_4 \vee \bar{u}_3) \wedge (u_2 \vee u_3 \vee \bar{u}_4)$



Problem Class



All NP problem reduces to SAT.

Outline

- Reduction: 3SAT $\leq_p$ 3 Colours

3 Colour

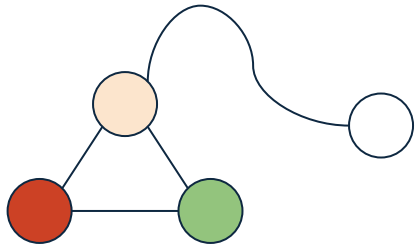
Graph colouring: Given a graph G = (V, E), colour each vertex with a colour such that no two vertices joined by an edge will have same colours.

Application:

- Sudoku puzzles
- Scheduling taxis (Ride hailing apps)
- Lecture time table

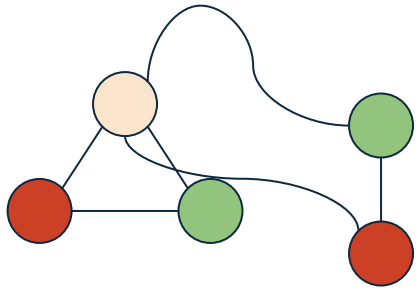
3 Colour: Given a graph G = (V, E), is it possible to colour each vertex with one of the 3 colours such that no two vertices joined by an edge will have same colours.

3SAT to Colouring



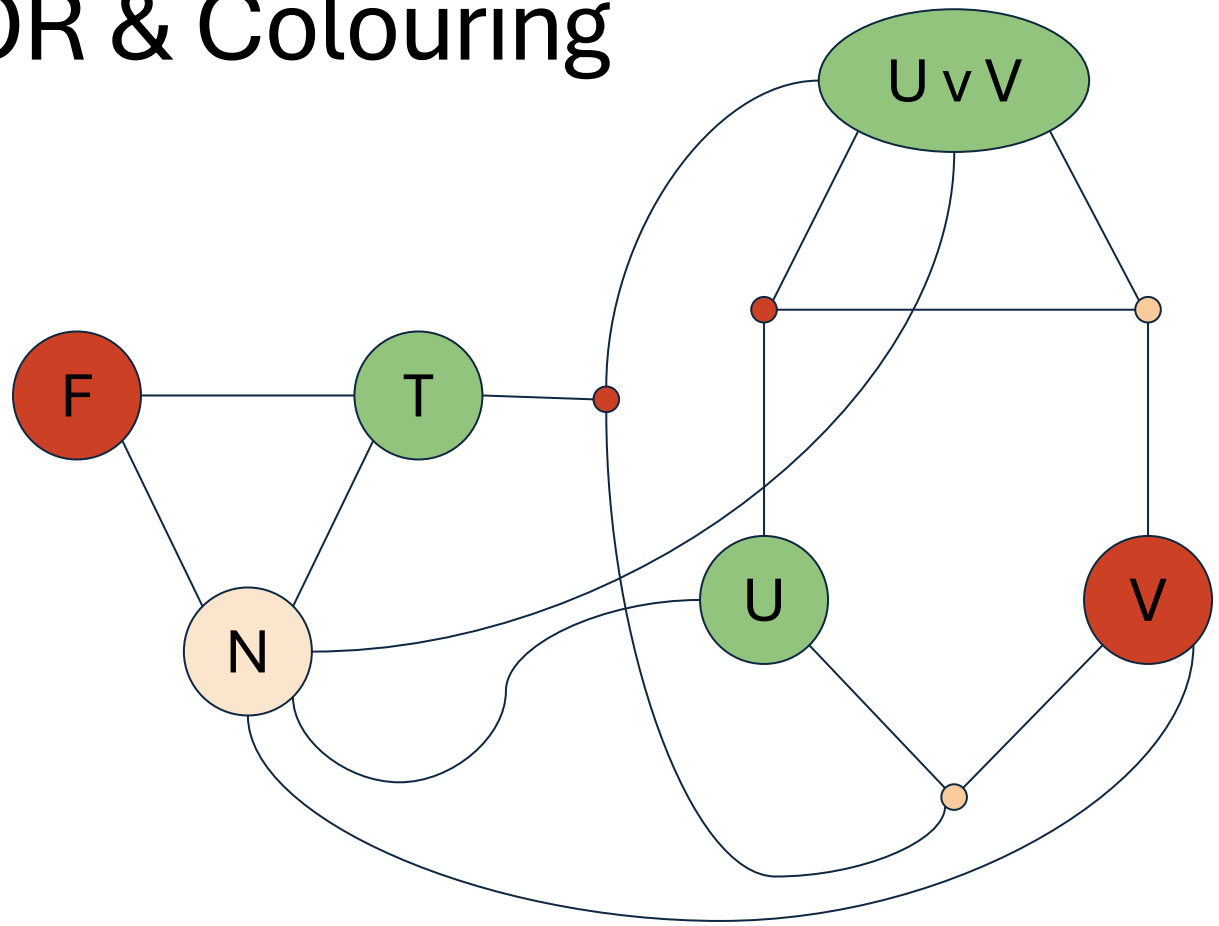
How to force the new vertex (v) with a neutral colour?

3SAT to Colouring

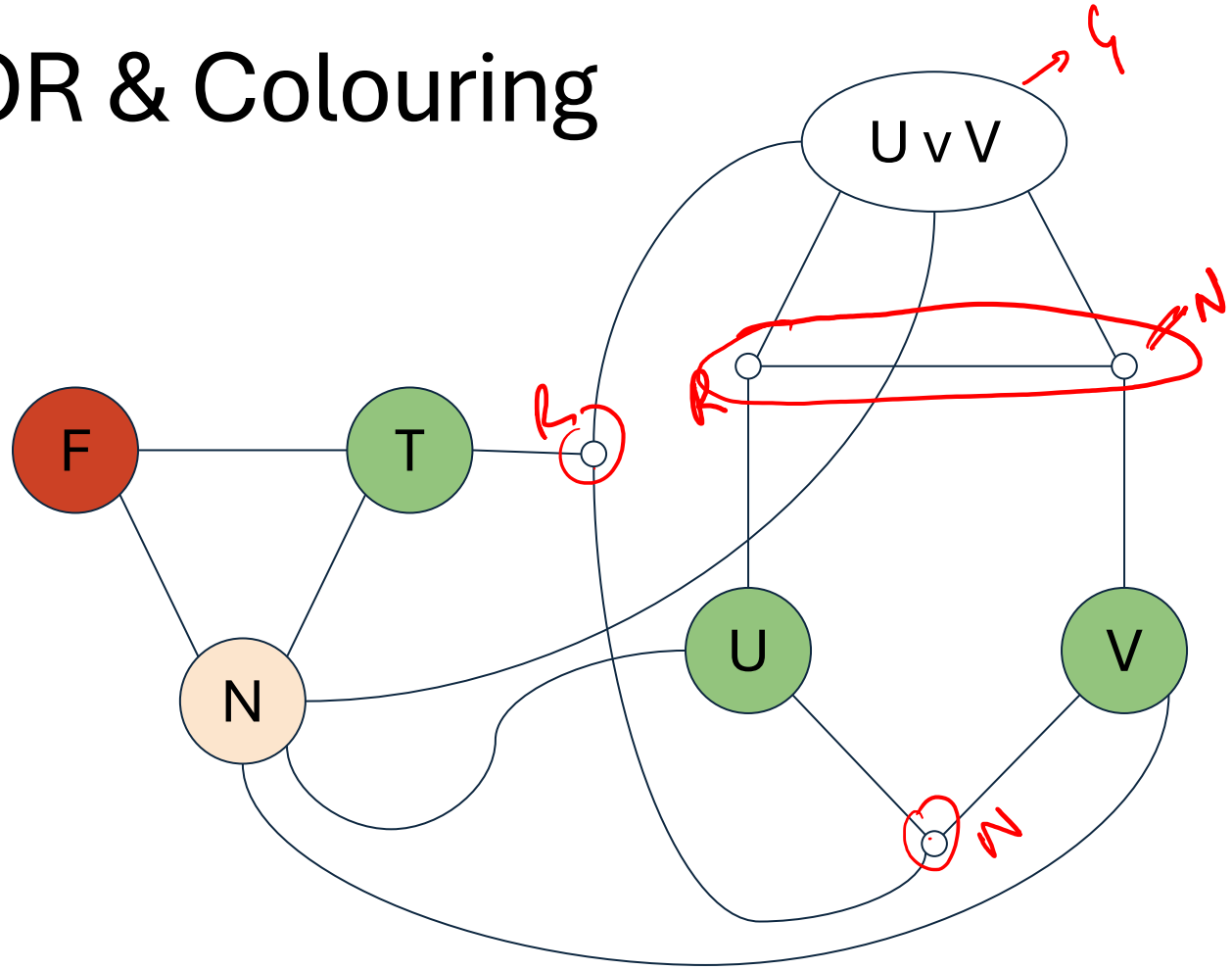


How to colour new vertex (not v)?

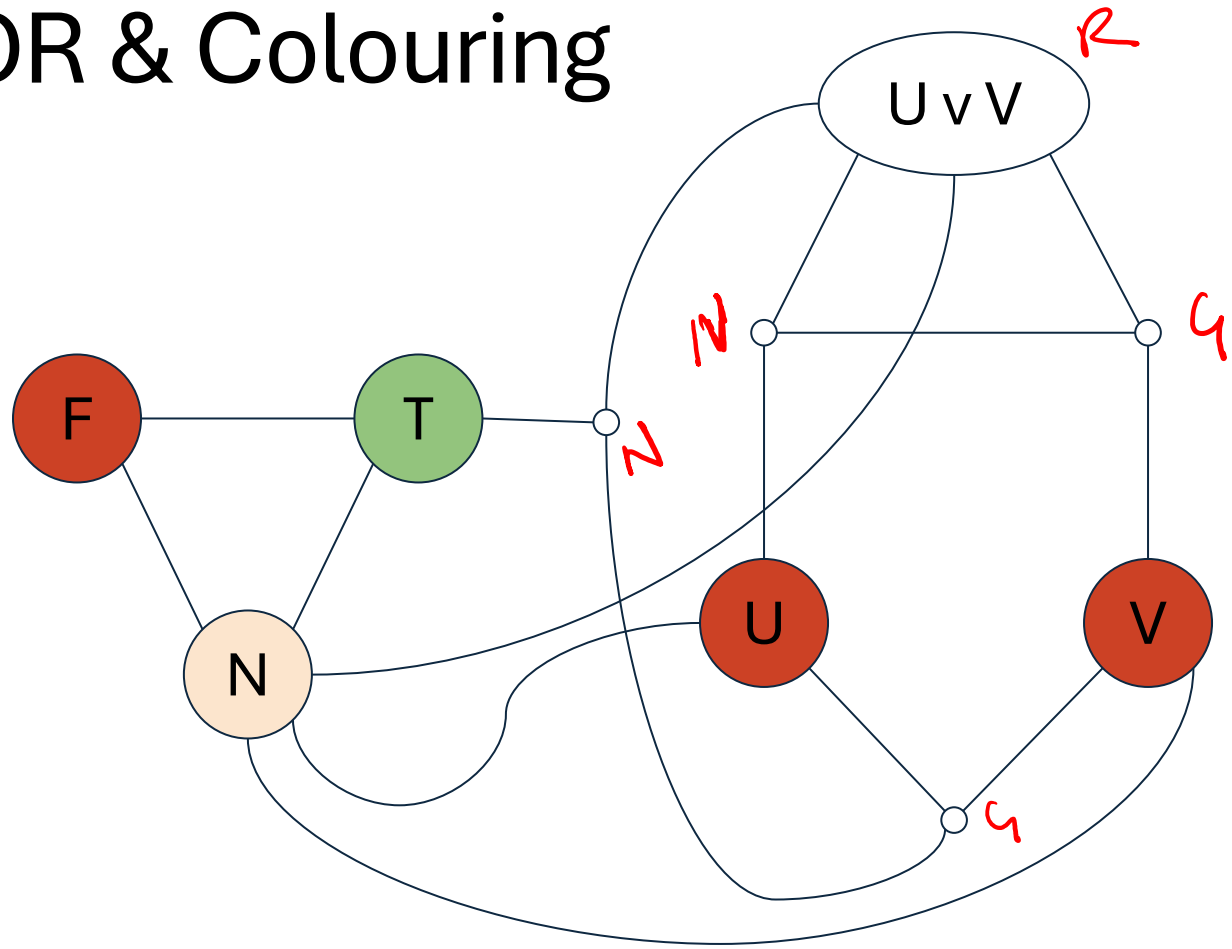
OR & Colouring



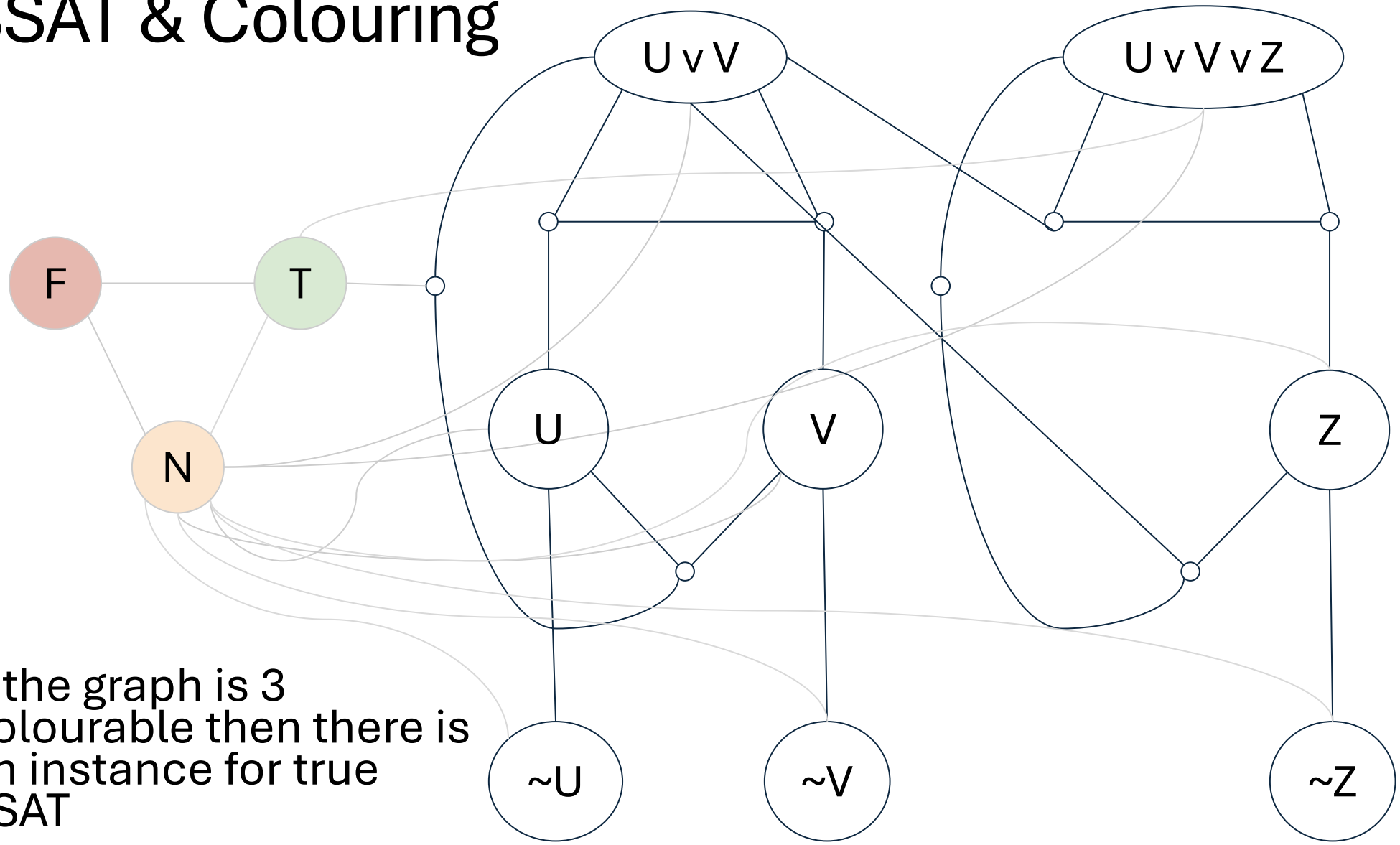
OR & Colouring



OR & Colouring



3SAT & Colouring



If the graph is 3 colourable then there is an instance for true 3SAT

Reference

Slides