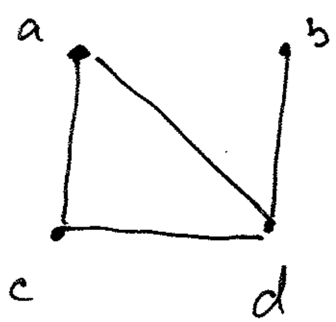
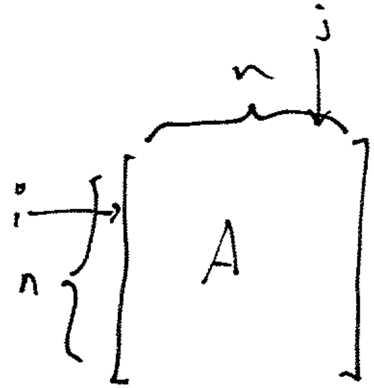
Graph Algorithms

Review:

<u>Representations</u> $\longrightarrow$ Adjacency Matrix

Tree Traversal Adjacency List





$$a_{ij} = \begin{cases} 0 & \text{otherwise } (v_i, v_j) \notin E \\ 1 & (v_i, v_j) \in E \end{cases}$$

$$\begin{array}{c|c} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency List:

- each vertex has a linked list: elements of which are vertices connected by an edge.

a → [c|•] → [d|]

b → [d|]

c → [d|•] → [a|]

d → [b|•] → [a|•] → [c|]

Problem: Given a graph $G = (V, E)$, and 2 vertices $x, y \in V$

Output: true if $(x, y) \in E$, false otherwise

A) Adjacency Matrix

~~output~~ $i \leftarrow$ index of x

$j \leftarrow$ index of y

if $(a_{ij} = 1)$

 L output true

else

 L output false

$O(1)$

B) Adj List:

$L \leftarrow$ list corresponding to x

for each $z \in L$: $\leftarrow$ $O(n)$

 if $(z = y)$

 L output true

output false

Given: A graph $G = (V, E)$ and a vertex $x \in V$

Output: $|N(x)|$ (size of neighborhood of x)

A) Adj. Matrix

$i \leftarrow$ index of x

count $\leftarrow 0$

for ($j = 0, \dots n-1$) $\quad O(n)$

if ($a_{ij} = 1$)

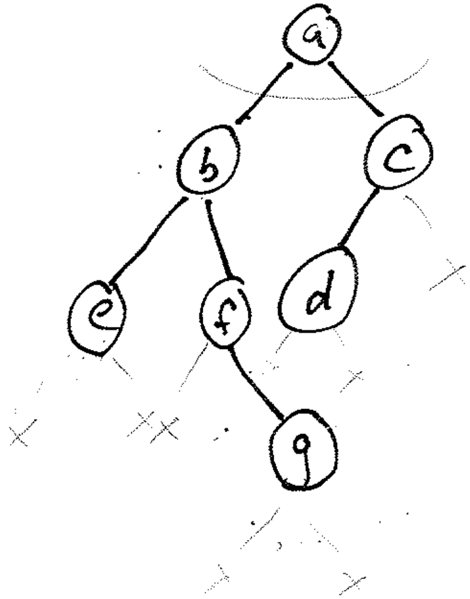
count++

output count

B) Adj List

$L \leftarrow$ list correspondy to x

output $|x|$ $\quad O(1)$



Pre Order: root - left - right

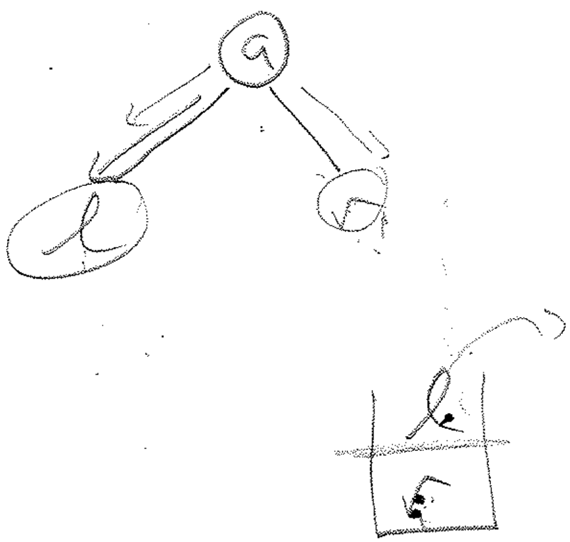
a b e f g c d

Given: node u

process node u

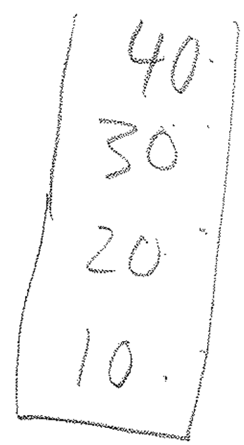
recursively process u.leftChild

recursively process u.rightChild



Stack

LIFO



Input: A binary tree $T = (V, E)$

Output: A preorder processing of T

ⓤ

S ← init stack
cw ← ∅
S.push(T.root)

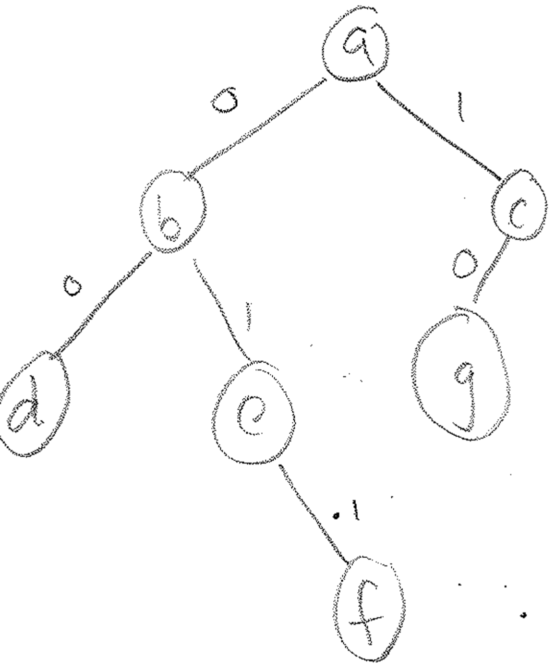
while (!S.isEmpty())

    u ← S.pop()
    ~~process u~~ output ~~cw~~ (associated with u's letter) string associated with ⓤ
    S.push(u.rightChild)   // assuming they exist
    S.push(u.leftChild)

~~cw~~ + '1'

~~cw~~ + '0'

(node, string)



00 → d

d → 00

f → 011

g → 10

<u>curr str</u>

~~00~~

~~011~~

10

(a) (b) $\boxed{d}$ (e) $\boxed{f}$

0 0 1

output 00

output 011

```
codeWordMap = [ ]
S ← init stack
s.push(T.root, "")      empty str
while (!s.isEmpty())
    (u, cw) ← s.pop()
     node   str
    codewordMap[u.letter] = cw
    s.push(u.rightChild, cw + "1")
    s.push(u.leftChild, cw + "0")
```

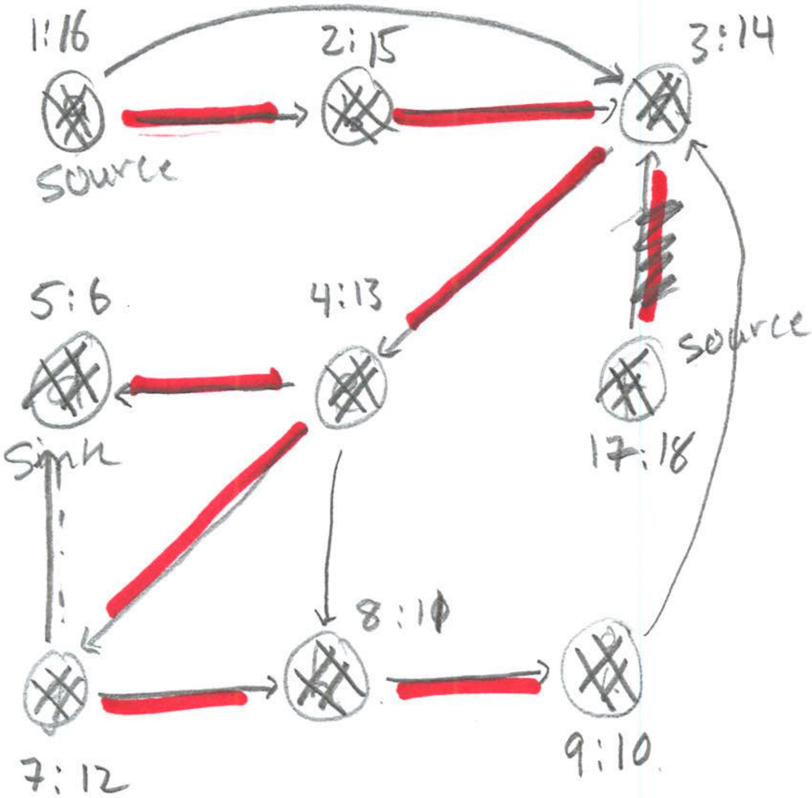
Graph types: $\boxed{V} \to u$ $v \to u$

- Directed vs undirected
- nodes: labeled, unlabeled, weighted
- edges: weighted, unweighted (weighted with wt = 1)

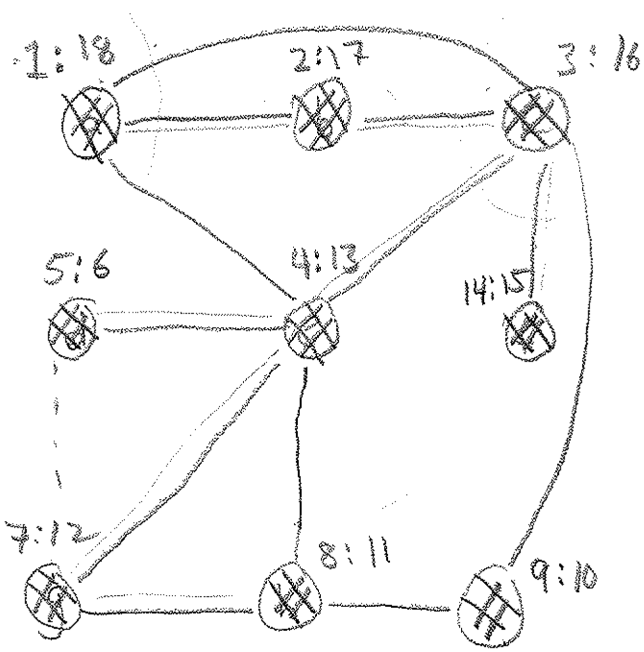
Graph Traversal Algorithms

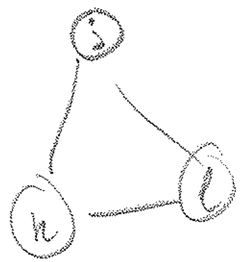
Depth First Search (DFS)

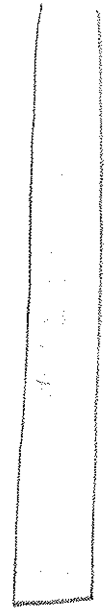
- preorder / inorder / post order: all DFS on binary trees
- Brute Force Hamiltonian Path Problem
- You explore the graph as deep as possible before back tracking



vertex	discovry	finish
a	1	16
b	2	15
c	3	14
d	5	6
e	4	13
g	7	12
f	$\boxed{17}$	$\boxed{18}$
h	8	11
i	9	10







vertex	discovy	finish
a	1	18
b	2	17
c	3	16
d	5	6
e	4	13
f	14	15
g	7	12
h	8	11
i	9	10

DFS Tree:

You can label edges as a result of DFS:

- Tree Edge: edges directly traversed with DFS
- Forward Edges: Connects an ancestor in the DFS tree to a descendant
- Back Edges: descendant to an ancestor
- Cross Edges: Connect "cousins" or siblings

- For undirected graphs:
  - → Forward = Back edges are the same in the DFS Tree
  - → Cross edges are not possible
  - → Forward / Back edge ⇒ a cycle exists
- For directed graphs:
  - → Back edges imply a cycle
  - → All 4 are possible
  - → cross edge may connect vertices in the same or different components
  - → cross edges <u>may</u> or may not imply a cycle

Analysis:

- Each node is processed at most once $O(n)$
- Each node is pushed exactly once, popped exactly once
- Each node is colored exactly 3 times
- Each node is stamped exactly 2 times
- $O(n) \to$ linear with respect to the number of nodes
- However: choosing the next vertex may be an $O(n)$ or $O(m)$ operation
- Overall: $O(n+m)$

$$O(n+m) \to O(n+n^2) = O(n^2)$$

$$O(n+m) \to O(m)$$ linear w.r.t. the input size

Stack-Based DFS

Input: An graph $G = (V, E)$

Output: DFS traversal

```
for each vertex v ∈ V:                        ]— initialization
    color v white                              ]
count ← 1                                      ]
S ← stack
push the start vertex v onto S                 ]— start of the search
stamp v with count (discovery time)            ]
color v gray
while( S is not empty ):
    count ++
    x ← S.peek    // x is the current vertex
    y ← next white vertex in N(x)   // y is the next white (undiscovered) vertex in N(x)
    if(y = ∅)       // no white (undiscovered) vertex in N(x)
        S.pop   //remove/backtrack
        process x
        color x black
        stamp x with count (finish stamp)
    ⋮
```

else

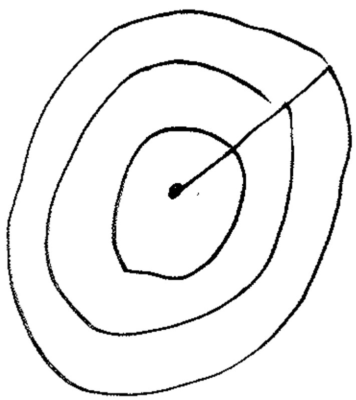
s.push(y)

color y gray

stamp y with count (discovery)

you <u>may</u> have to "start over" in a "main loop"

Breadth First Search

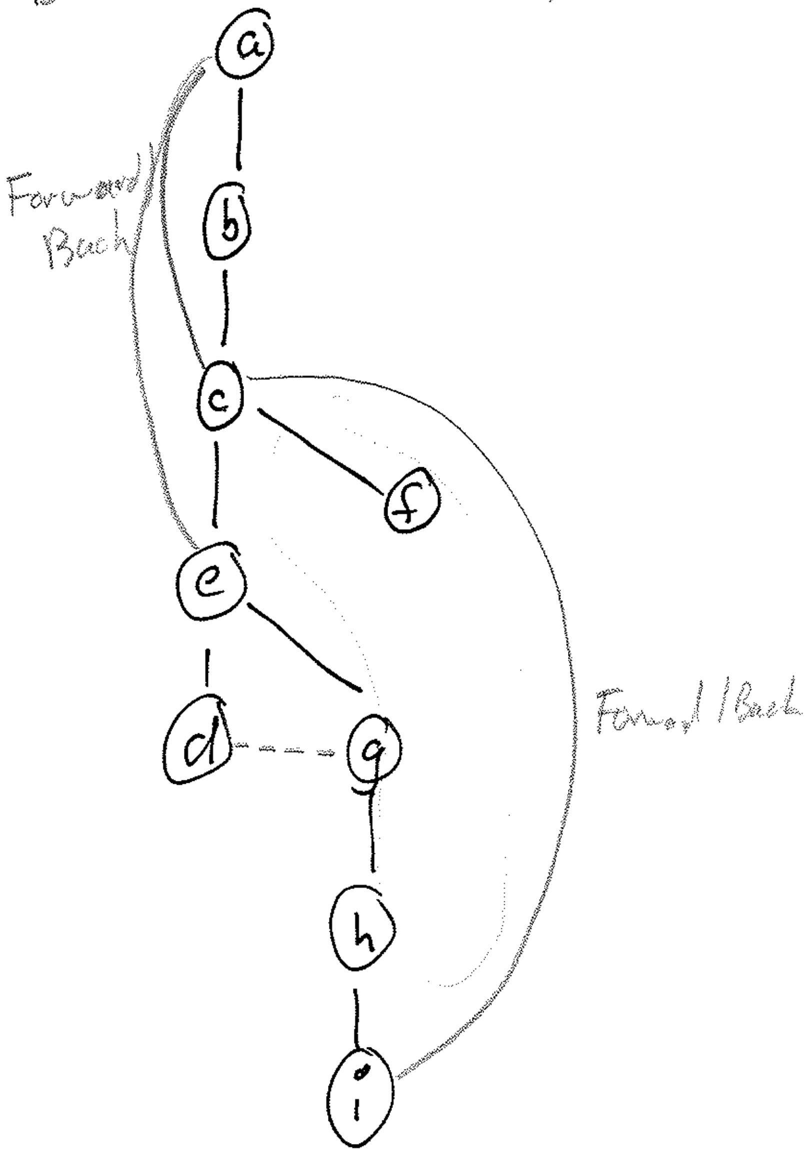


$A = \pi r^2$ $\quad r \to \infty \quad$ A grows quadratically

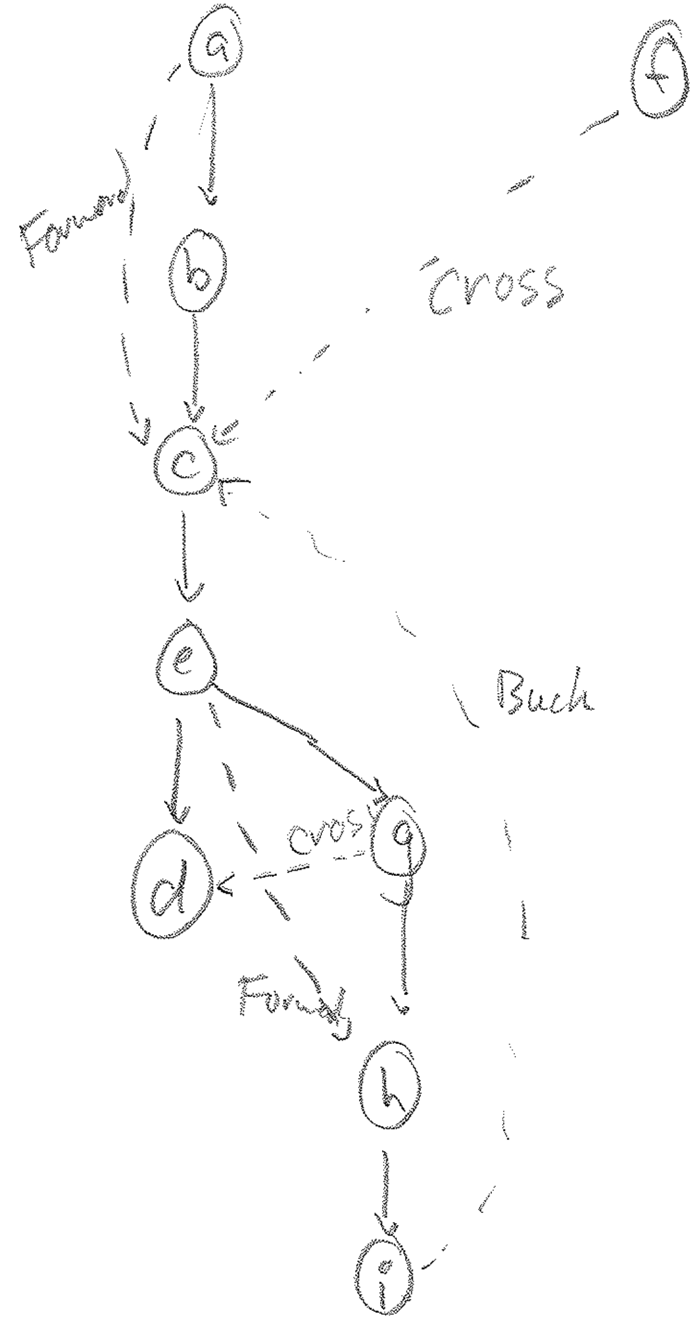
$r \to 2r \Rightarrow$ quadruple area

- search all nodes a distance 1 from the start node
- then distance 2, 3, 4... n-1
- Queue

DFS Tree (undirected)



DFS Forest



DFS data:

- Vertex colors:

  white: unvisited, unprocessed vertices

  gray: visited, but unfinished vertex

  black: finished vertex

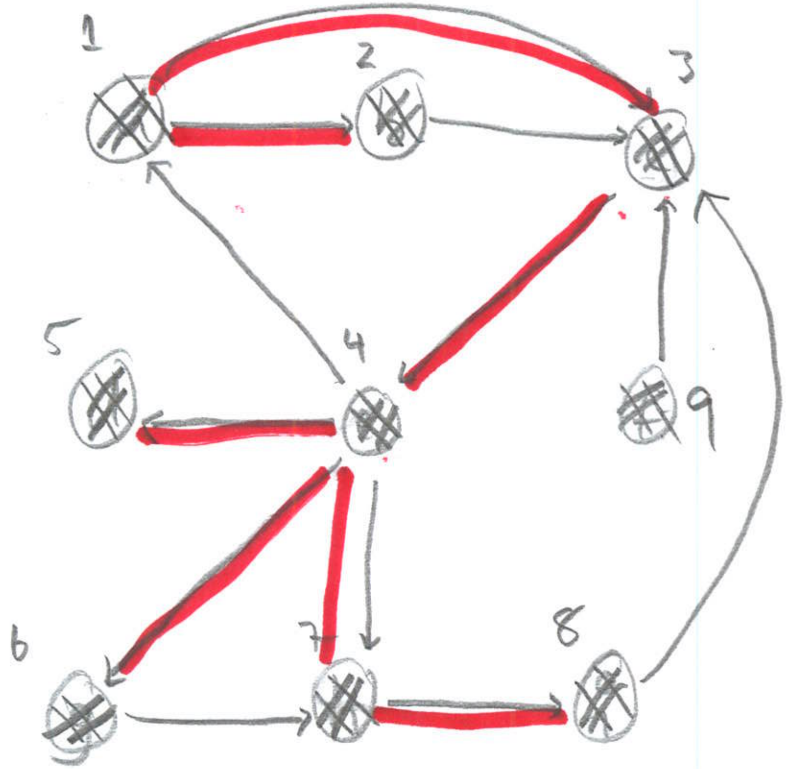
- Discover "time"
- Finish "time"

→ keep a counter: start at 1, increment evry time a color changes.

- Next vertex choice:
  - visited the least weight edge ~~next~~
  - random
  - lexicographically

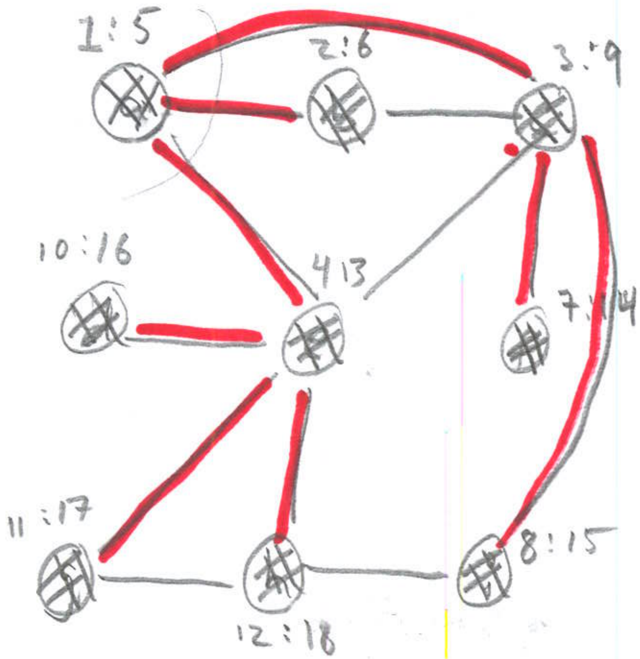
BFS = Breadth First Search

- explore all neighbors first
- use a queue instead of a stack
- explore "close" vertices first
- keep track of similar artifacts:
  - vertex color: white → gray → black
    unvisited → visited → done
  - discovery times
  - finish times(?)



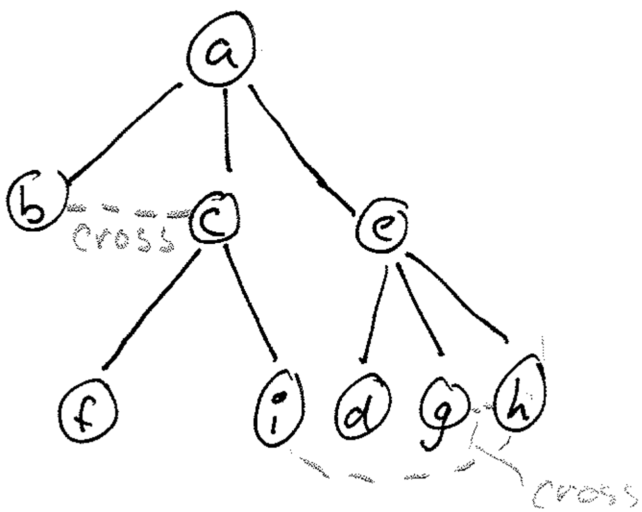
a b c e d g h f

vertex	discovery
a	1
b	2
c	3
d	5
e	4
f	9
g	6
h	5
i	8



vertex	discovery	~~finish~~
a	1	5
b	2	6
c	3	9
d	10	16
e	4	13
f	7	14
g	11	17
h	12	18
i	8	15

BFS Tree

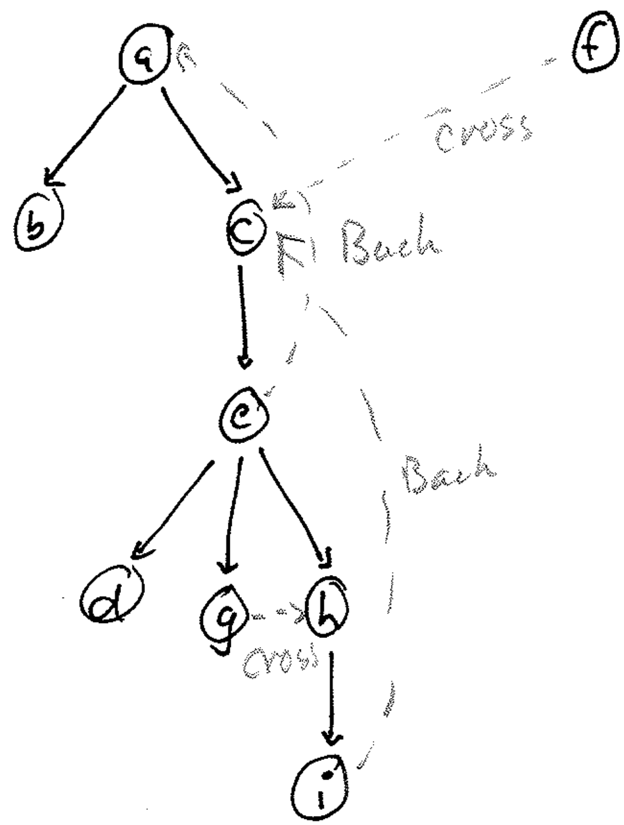


- No forward nor back edges.

- cross edges: cycle!

- For undirected graphs, BFS tree provides the single source shortest path.
(From the start to all other vertices)

BFS Tree : Directed.

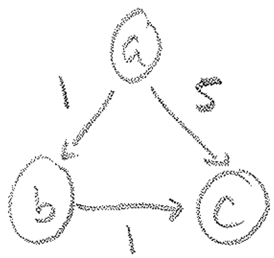


- Back edges are possible : directed cycle

- Forward edges : still not possible

- Cross edges : may or may not imply a cycle

BFS fails to find the shortest path for weighted graphs

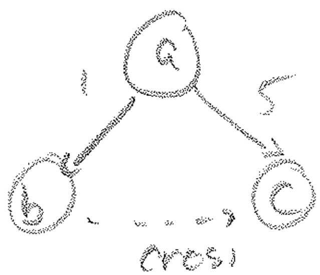


$a \rightarrow b \quad 1$

$a \rightarrow c \quad 1$

$\downarrow$

BFS from a:



$a \rightarrow b : 1$

$a \rightarrow c : 2$ vs 5

Breadth First Search

Input: A graph $G=(V,E)$, an intial vertex $v \in V$

```
count ← 1
Q ← queue
mark v with count
v.color ← gray  (discovered)
Q.enqueue(v)
while( Q is not empty)
|   x ← Q.peek()
|   for each y ∈ N(x)
|   |   if (y.color = white)
|   |   |   count ++
|   |   |   mark y with count (discover time)
|   |   |   Q.enqueue(y)
|   z ← Q.dequeue
|   ~~count ++~~
|   z.color = black   ← process z
```

More Applications

- Connectivity Properties:

Decision Problem (version): Yes/No

. Given 2 vertices $x, y \in V$: determine if there exist a path $x \rightsquigarrow y$

a path (directed/undirected)

· programming: true/false (boolean)

Optimization Version: value (~~length~~ of the best solution)

Given G, $x, y \in V$: output the length of the shortest path

program: int

Functional version: outputs the best solution

Given G, $x, y \in V$: Output the shortest path $p: x \rightsquigarrow y$

program: list

def: existsPath(G, x, y) : boolean

...

DRY = Don't repeat yourself

def shortestPathLength(G, x, y) : int

...

def getShortestPath(G, x, y) : list, None if no such path
or
an empty list

```
def existsPath(G, x, y):
    return (getShortestPath(G, x, y) != None)
```

$S = \{a, b, c\}$

$P(S)$ = powerset of S

A related B iff $A \subseteq B$

Hasse Diagram

reflexive

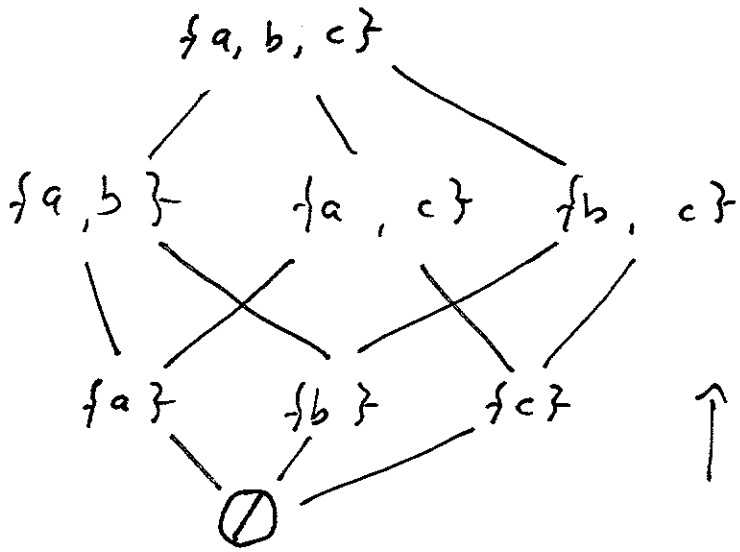
$A \subseteq A$

transitive

$A \subseteq B,\ B \subseteq C \Rightarrow A \subseteq C$

antisymmetric

$A \subseteq B \wedge B \subseteq A \Rightarrow A = B$



$\uparrow$ relation

$\{b, c\}$

$\emptyset$, $\{b\}$, $\{c\}$, $\{a\}$ $\{a,b\}$ $\{a,c\}$ $\{a,b,c\}$

Poset → Total order

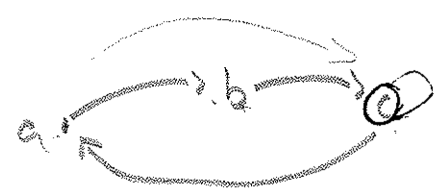
- impose a total order on the poset
- consistent with the original relation
  - incomparable pairs may appear in any order
  - comparable pairs will never be out of order

Topological Sort

Topologial sort:

- Perform a DFS on a DAG = <u>Directed Acyclic Graph</u> $O(n)$
- <u>note</u> the finish time stamps
- sort in descending order of finish time stamps

total order



$a = b = c$

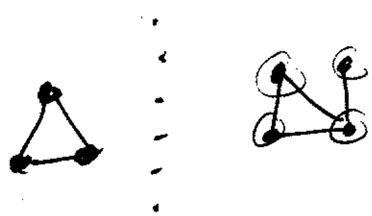
Cycle detection:

Give a graph $G=(V,E)$ determine if it contains a cycle or not

- Run DFS/BFS: if you encounter a gray or black vertex previously encountered: stop report a cycle.

Disconnectivity:

Given G, is it connected or not

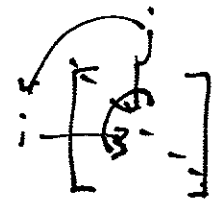


→ Perform a single DFS, if any white (unvisited) vertices remain ⇒ disconnected.

DFS Application: Condensation Graphs

- Large graphs may be made more simple
- Faster algorithms on smaller graphs
- Preserve <u>some</u> topology
- Ex: given a directed graph, <u>condense</u> it into
  <u>strongly</u> <u>connected</u> <u>components</u>
- $x, y \in V$ are <u>strongly</u> connected if
  $\exists p : x \rightsquigarrow y$ and $\exists p : y \rightsquigarrow x$

1) Run DFS, keep track of finishing time stamps $O(n)$

2) Compute the transpose graph G^T $O(1)$  $O(n^2)$

orientation of edge is reversed

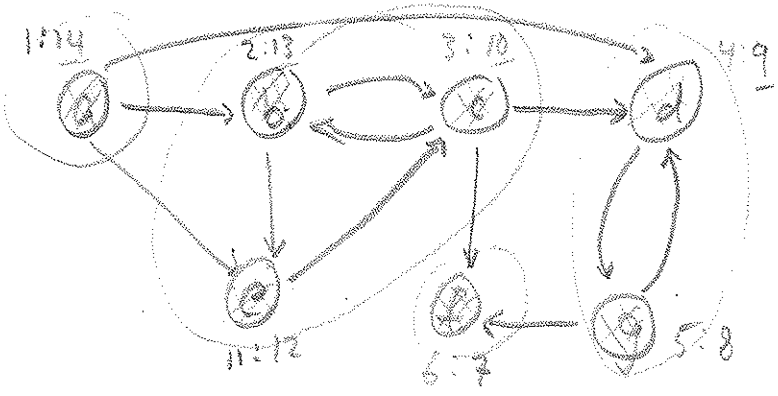
3) Run DFS again on G^T but in the finish order of step 1 $O(n)$

- any time you restart DFS is a new strongly connected component.

- Build a new graph:

  vertices = strongly connected components

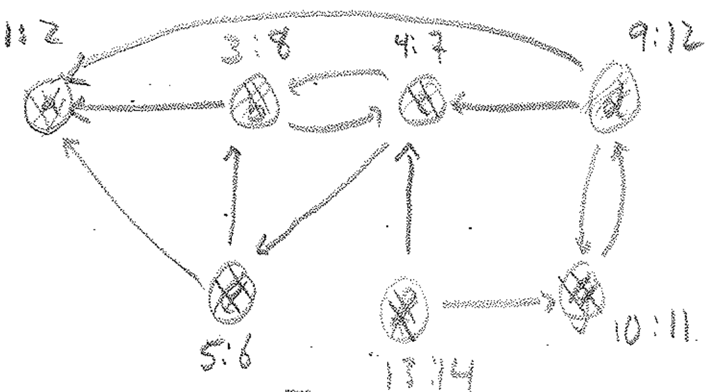
  edges are between components



1) DFS

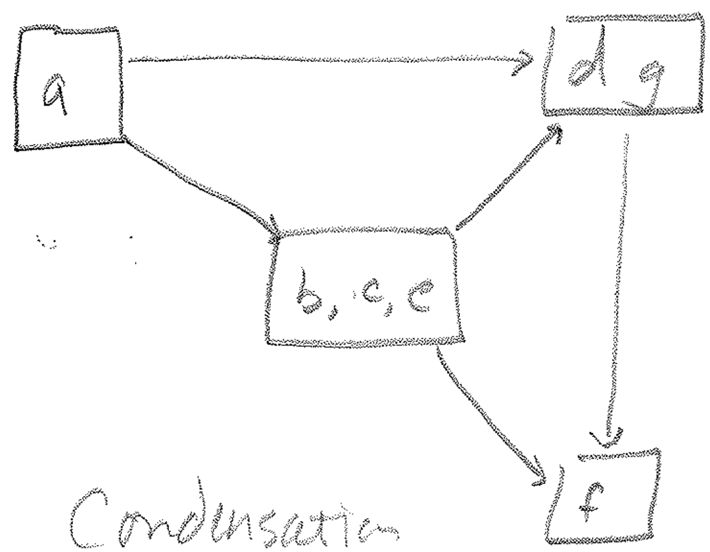
2) Compute G^T

3) DFS on G^T in descending order wrt finish time stamps from DFS on G



G^T

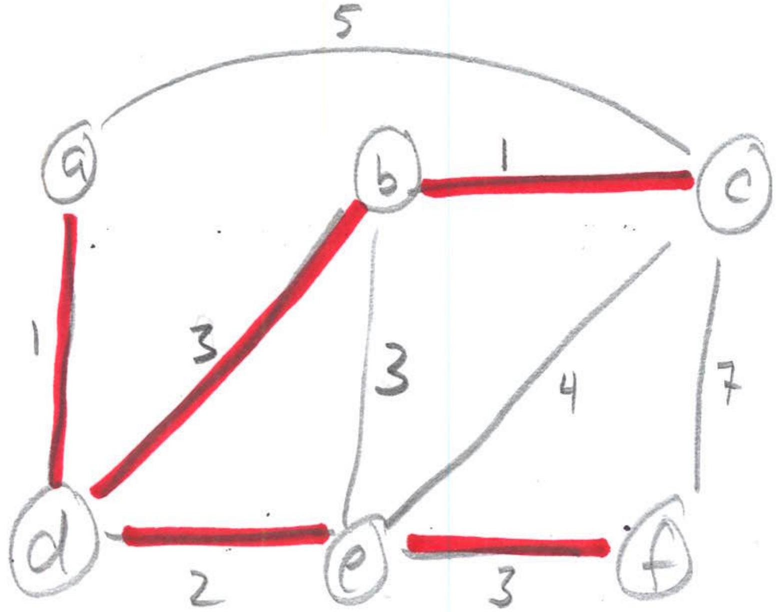
DAG = Directed Acyclic Graph

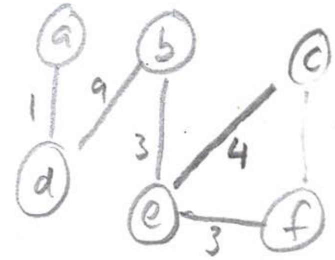


Condensation graph

Minimum Spanning Tree

- Given: A weighted, undirected graph $G=(V,E)$ (assume connected)
- want to build a network "back bone"
- Spanning tree: a tree T (a subset of edges of G) that spans G: any 2 vertices that were connected in G are still connected in T
- Minimize the overall tree weight (sum of all edge weights in T)

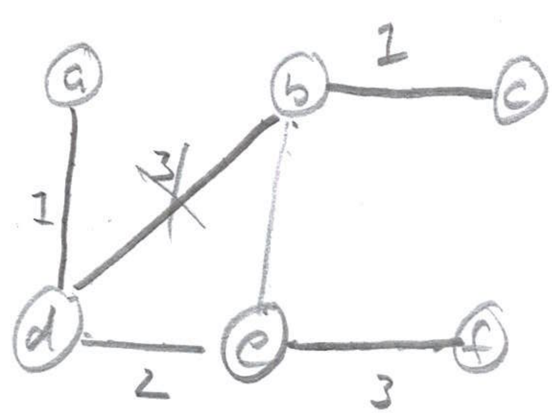




$1 + 9 + 3 + 3 + \cancel{7} = \cancel{23}$

$4 = 20$

edge	wt	result
(a,d)	1	added
(b,c)	1	added
(d,e)	2	added
(b,d)	3	added
(b,e)	3	omit
(e,f)	3	added



total weight

10

Kruskal's Algorithm:

1. Sort the edges by weight in increasing order
2. For each edge: if it does not induce (create) a cycle add it to the tree (initially, the tree has no edges)
3. ... until you have $n-1$ edges

$m \leq \binom{n}{2} = O(n^2)$

Analysis:

1. Sort edges: $O(m \log(m)) = O(n^2 \log(n^2))$

$= O(n^2 \log(n))$

2. $O(m)$ (for each edge) $O(n^2)$

a) add the edge $O(1)$

b) DFS to test if a cycle exists $O(n+m) = O(\cancel{n^2})$

$O(n)$

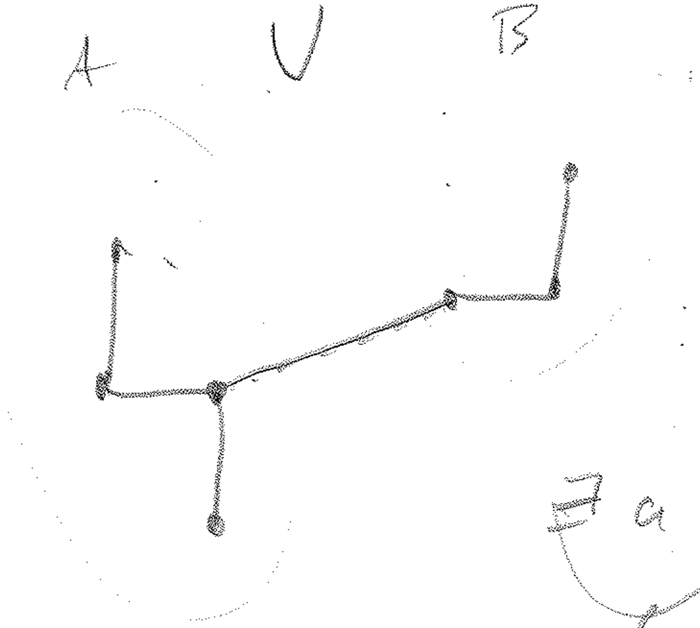
$\rightarrow \cancel{O(n^4)}$

iteration	n	m
1	n	1
2		2
3		3
4		4
⋮		
$n-1$		$n-1$

$\square$

$O(m) = O(n)$

for "sparse" graphs



$\exists$ a cycle iff ~~the edge~~ (x,y) ind

adding the edge (x,y) induces a cycle iff

x and y are in the same connected component.

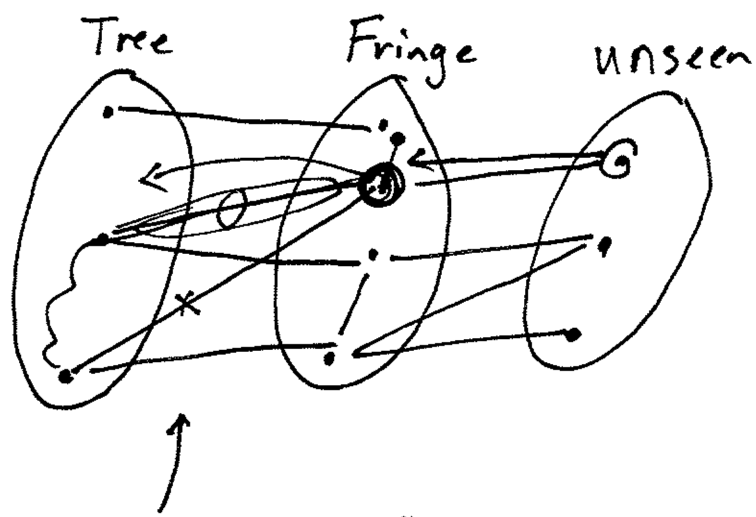
Minimum Spanning Trees: Prim's Algorithm

- Idea: work locally and build the tree "out"
- Similar to BFS: consider least weighted edge next
- 3 regions:

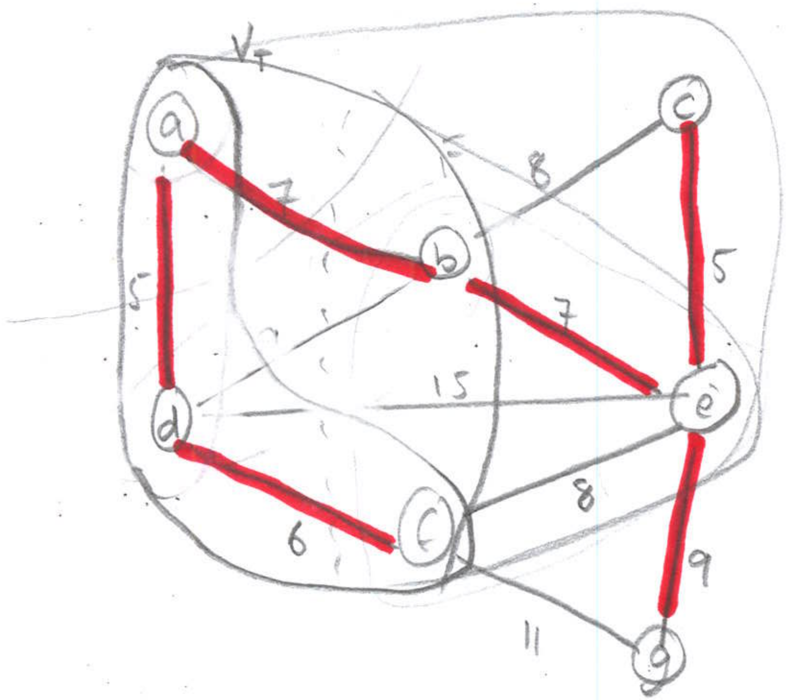
  Tree

  Fringe

  unseen



Edges on the "Fringe" to consider next



$5+6+7+7+5+9$

$=39$

V_T = tree vertex set = a, d, f
b e, c
g

V_F = ~~b 7~~
~~d 5~~
~~b 9~~
e 15
~~f 6~~
~~c 8~~
g 11
~~e 8~~
~~e 7~~
~~c 5~~
g 9

Prim's

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

Output: A MST of G

$E_T \leftarrow \emptyset$

$V_T \leftarrow \{v_1\}$

$E^* \leftarrow N(v_1)$ // Fringe vertex / edge set

For $i = 1 \ldots n-1$

  $e \leftarrow$ min. weighted edge in E^*

  $(u,v) \leftarrow$ endpoints of e: u is in V_T, v is not

  Add v to V_T

  add e to E_T

  update E^*:

    1) add all vertices / edges in $N(v)$

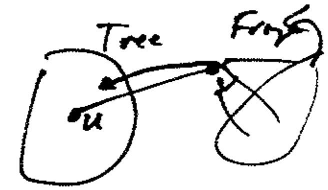
    2) remove (or ignore) all edges in $N(v)$ connected to V_T

Output $T = (V_T, E_T)$

insertion of an edge into a min heap:

$O(\log(m))$

in total: $O(m\log(m))$



Dijkstra's Single Source Shortest Path

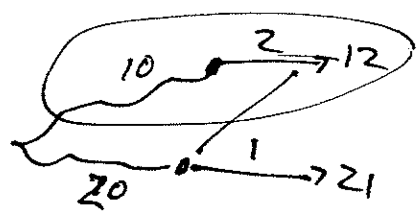
Given: Directed graph $G=(V,E)$ and a start vertex s

Output: The ~~~~ the shortest path from s to all other vertices

Tree vertices

Fringe vertices / edges

Consider edges on the fringe: add the least weighted total edge



From e, how do you get to i?

cost = 20

$e \xrightarrow{5} d \xrightarrow{10} g \xrightarrow{5} i$

e, 0, –
b, 5, e
d, 5, e
a, 10, d
g, 15, d
i, <u>20</u>, g
c, 25, d
f, 25, d
h, ∞, –
j, ∞, –

Input: A weighted $G = (V, E)$, a source vertex $s \in V$

$s \rightsquigarrow u \to v$

Output: the min. weighted path from s to all other vertices

wt + a predecessor vertex: d_v, p_v

min distance $s \rightsquigarrow u$
+ wt of edge $u \to v$

$Q \leftarrow$ min-heap

init: for each $v \in V \setminus \{s\}$

$\quad d_v \leftarrow \infty$

$\quad p_v \leftarrow \emptyset$

$\quad Q.\text{enqueue}(v, d_v)$

$d_s \leftarrow 0$

$p_v \leftarrow \emptyset$

$V_T \leftarrow \emptyset$

$Q.\text{enque}(s, d_s)$

for $i = 1 \ldots n$ $\quad O(n)$

$\quad u \leftarrow Q.\text{dequeue}$

$\quad V_T \leftarrow V_T \cup \{u\}$

$\quad$ For each $v \in N(u) \setminus V_T$ $\quad O(m)$

min. dist found so far to v

$\quad\quad$ if $(d_u + wt(u,v) < d_v)$:

$\quad\quad\quad d_v \leftarrow d_u + wt(u,v)$

$\quad\quad\quad p_v \leftarrow u$

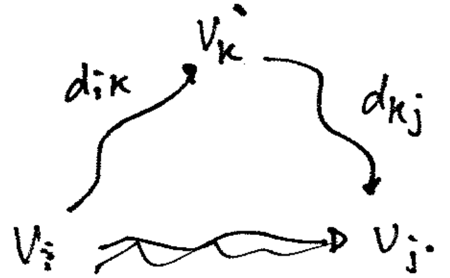
$\quad\quad\quad Q.\text{decreasePriority}(v, d_v)$ $\quad \underline{O(\log(n))}$

output (d_v, p_v) for each vertex

Floyd-Warshall

for each intermediate vertex v_k:

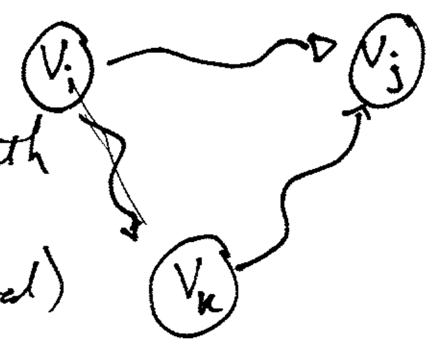
for each pair v_i v_j:



d_{ij} = min dist. $v_i \rightsquigarrow v_j$

$$\begin{tikzcd} a \arrow[r] \arrow[rr, bend right] & b \arrow[r] & c \end{tikzcd}$$

if $d_{ik} + d_{kj} < d_{ij}$: found a better path
(shorter)
(less weighted)



$V_i \rightsquigarrow v_j$ $\qquad$ $V_i \rightsquigarrow V_k$

Floyd-Warshall

Input: a weighted adj matrix representy a directed graph

Output: A shortest distance matrix + a successor matrix

$D \leftarrow (n \times n)$ matrix

for $1 \leq i\, j \leq n$:

$d_{ij} \leftarrow wt(v_i\, v_j)$, ∞ if no such edge

$s_{ij} \leftarrow j$ for all edges, $(v_i\, v_j)$, $\emptyset$ otherwise

for $k = 1 \ldots n$

for $i = 1 \ldots n$

for $j = 1 \ldots n$ — $O(n^3)$

$d_{ij} \leftarrow \min\{d_{ij},\ d_{ik} + d_{kj}\}$

// if d_{ij} is updated, update s_{ij}:

$s_{ij} \leftarrow s_{ik}$

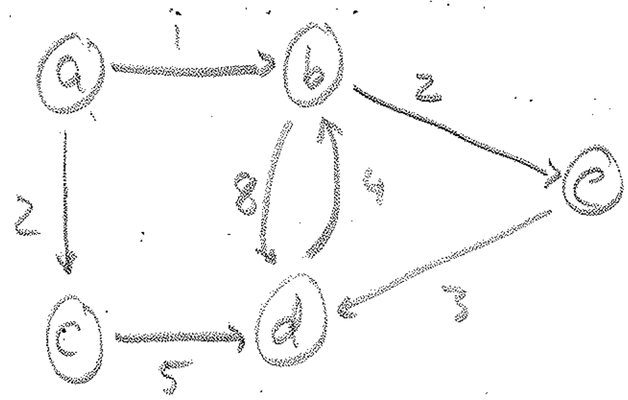
output D, S

input: matrix

size: $n^2 = \underline{N}$ $\quad n = \sqrt{N} = N^{.5}$

$O(n^3) = O(N^{1.5})$

$(N^{.5})^3 = N^{1.5}$



initial distance matrix:

$$\begin{array}{c|c} & \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} & \begin{bmatrix} 0 & 1 & 2 & \cancel{\infty}\,9 & \cancel{\infty}\,3 \\ \infty & 0 & \infty & 8 & 2 \\ \infty & \infty & 0 & 5 & \infty \\ \infty & 4 & \infty & 0 & \cancel{\infty}\,6 \\ \infty & \cancel{\infty}\,6 & \infty & 3 & 0 \end{bmatrix} \end{array}$$

$a \to b = 1$

$b \to e = 2$

$\overline{\quad 3 \quad}$

$$D \quad \begin{array}{c|c} & \begin{matrix} & & & & e \end{matrix} \\ \begin{matrix} a \\ \\ \\ \\ \\ \end{matrix} & \begin{bmatrix} 0 & 1 & 2 & 6 & 3 \\ \infty & 0 & \infty & 5 & 2 \\ \infty & 9 & 0 & 5 & 11 \\ \infty & 4 & \infty & 0 & 6 \\ \infty & 7 & \infty & 3 & 0 \end{bmatrix} \end{array}$$

$$S \quad \begin{array}{c|c} & \begin{matrix} & & & & e \end{matrix} \\ \begin{matrix} a \\ b \\ \\ \\ \\ \end{matrix} & \begin{bmatrix} - & b & c & b & b \\ - & - & - & e & e \\ - & d & - & d & d \\ - & b & - & - & b \\ - & a & - & d & - \end{bmatrix} \end{array}$$

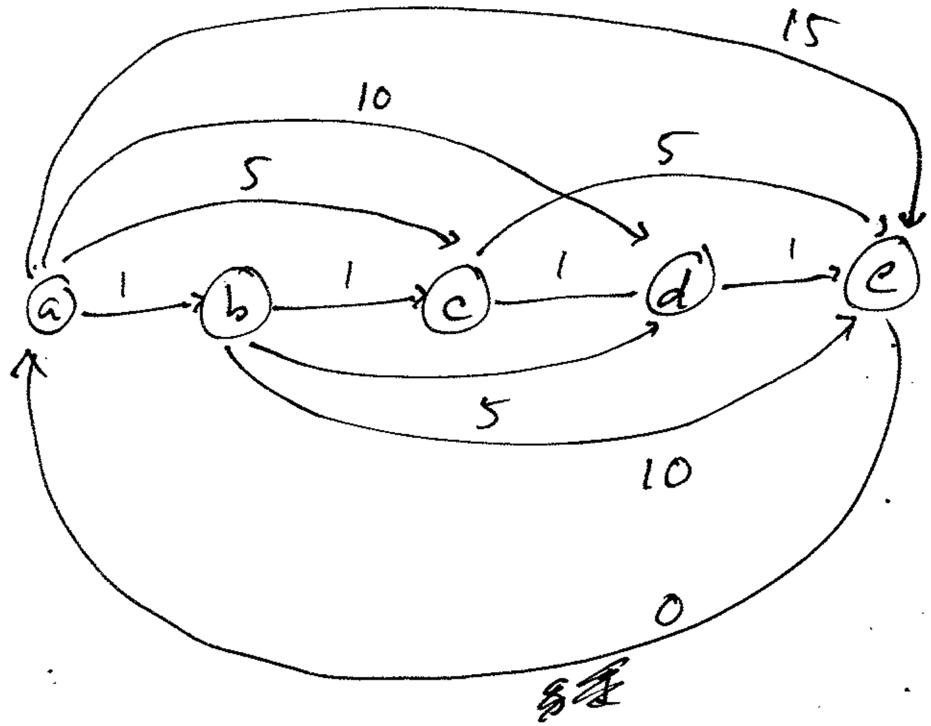
$a \to e$: how?

cost : 3

$\downarrow$

you go to b first

$a \to b \longrightarrow e$



Complete DAG

4

$a \overset{?}{\rightsquigarrow} e$

first go to b

$a \to b \overset{?}{\rightsquigarrow} e$

first go to c

$a \to b \to c \overset{?}{\rightsquigarrow} e$

$a \to b \to c \to d \overset{?}{\rightsquigarrow} e$

$a \to b \to c \to d \to e$

Input: Successor Matrix S from Floyd-Warshall

two vertices v_i v_j

Output: min. weighted path $p: v_i \rightsquigarrow v_j$

$p \leftarrow v_i$

$x \leftarrow v_i$ // current vertex

while ($x \neq v_j$)

$\quad x \leftarrow S_{xj}$

$\quad p \leftarrow p + x$ // add x to the end of the path p

output p.