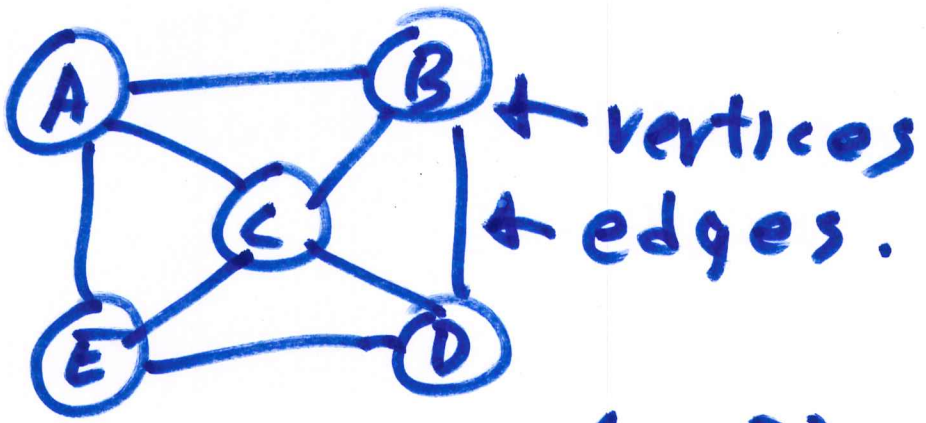
Graphs

They are data structures that are made of Vertices and edges.



Formally a graph $G=(V, E)$ is composed of

V: Set of vertices

E: Set of edges connecting the vertices.

Graphs are important because they are easy to understand and they are able to represent many real problems. There are many algorithms that solve problems with graphs.

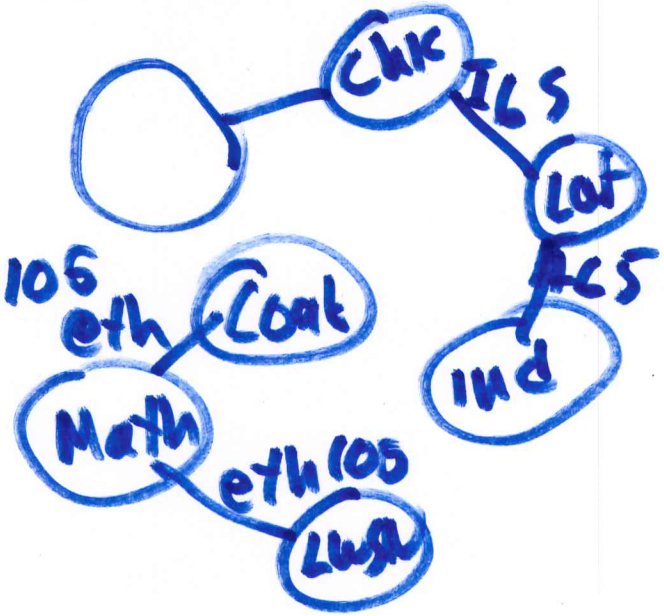
If you are able to represent a problem using graphs, it is likely that there is already a graph algorithm that may solve your problem:

198

Examples of problems that can be represented with graphs.

→ Communication & Transport networks

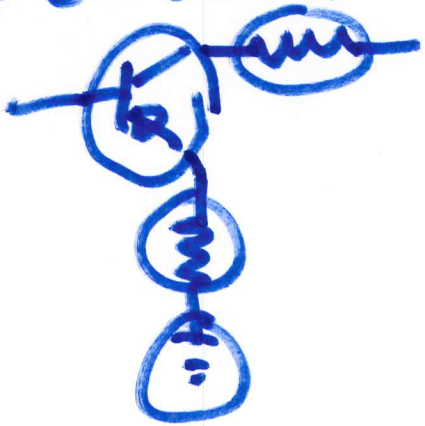
You can represent a street map or road maps as graphs



Transport Network

Communication Net

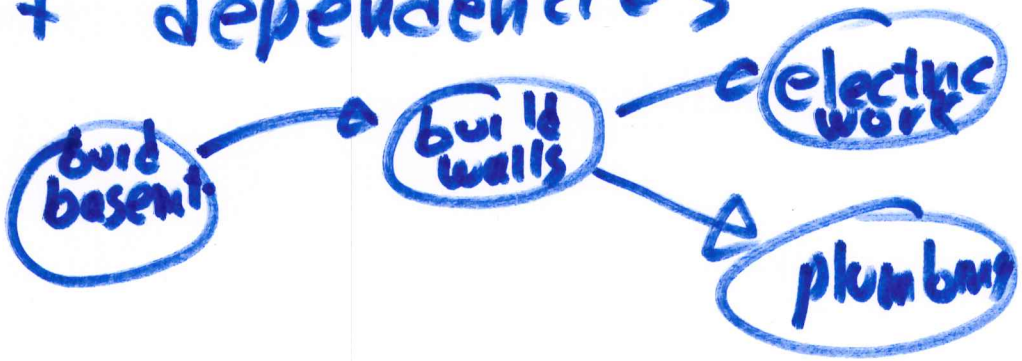
- Electronic circuits



13

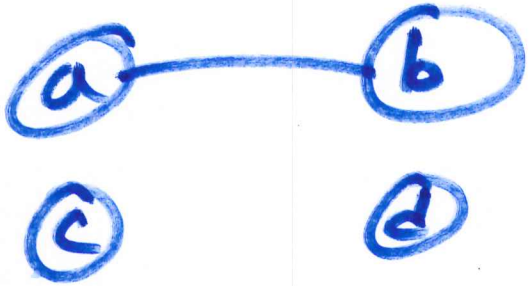
199

- Event dependencies



Graph Terminology

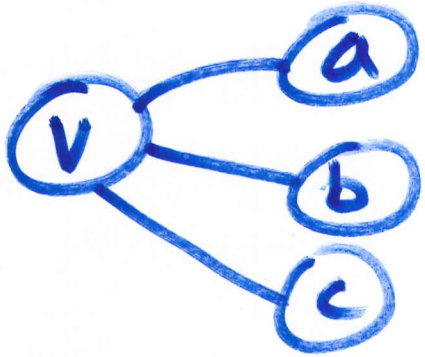
+ Adjacent vertices
  - Vertices connected by an edge



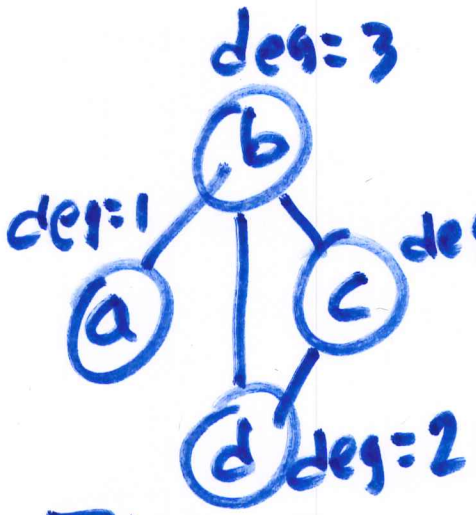
a and b are adjacent

c and d are not adjacent

+ Degree of a vertex
  - # of adjacent vertices to a vertex

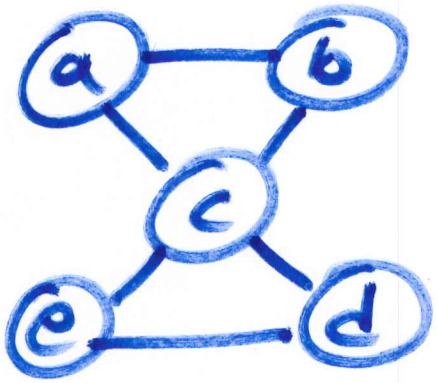


$\deg(v) = 3$



$$\sum_{v \in V} \deg(v) = 2(\#\text{edges})$$

Path: sequence of vertices $v_1, v_2, v_3 \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



a, b, c, e, d is a path

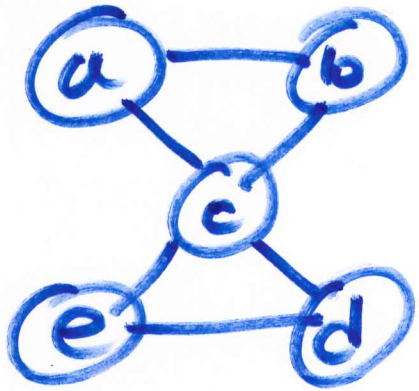
a, b, e, c d is not a path because b, e are not adjacent.

Simple path - It is a path with no repeated vertices

a, b, c, d, e is a simple path

a, b, c, d, e, c is not a simple path because c is repeated

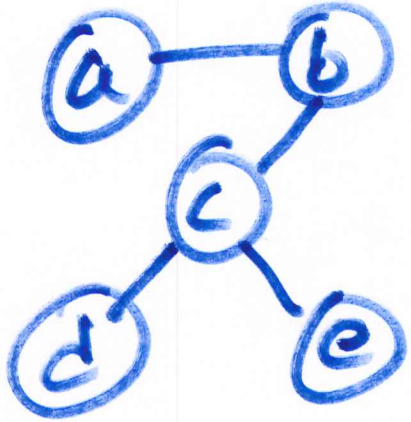
cycle: Similar to the simple path but the first and last vertex in the path are adjacent.



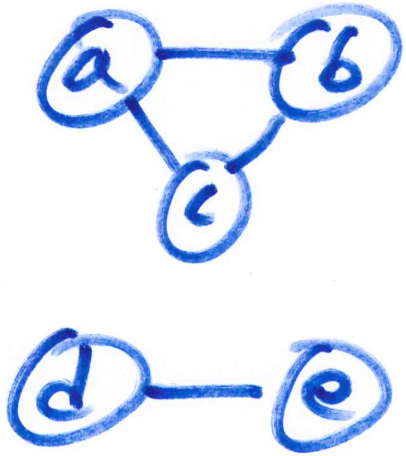
a, b, c is a cycle

a, c, d is not a cycle

connected graph - It is a graph were any two vertices in the graph are connected by some path

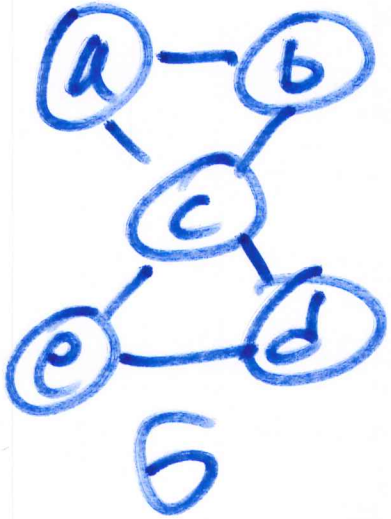


Connected graph

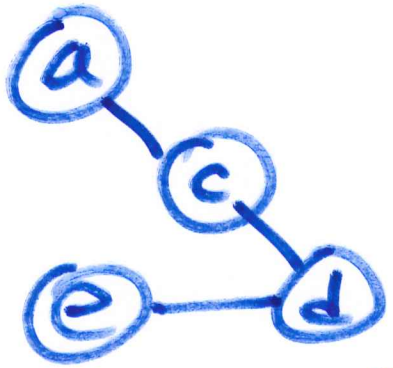


disconnected graph

subgraph - subset of vertices and edges of a graph



G



subgraph of G

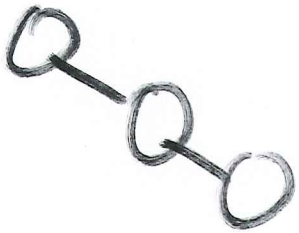
Connected components

A maximal connected subgraph.

this graph has 3 connected components

Tree

It is a connected graph without cycles



tree

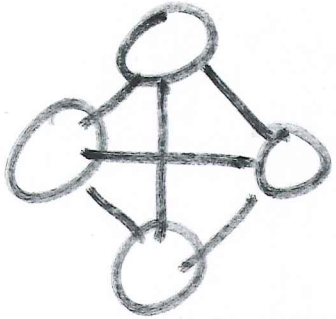
tree

not a tree

Forest - Collection of tree

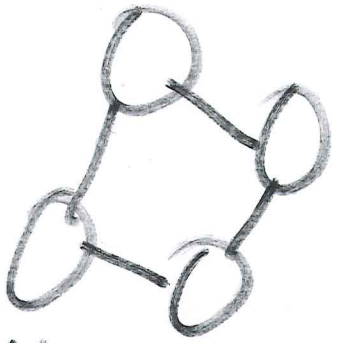
+ complete graph

It is a graph were all pairs of vertices are adjacent.



Complete Graph

Complete graph

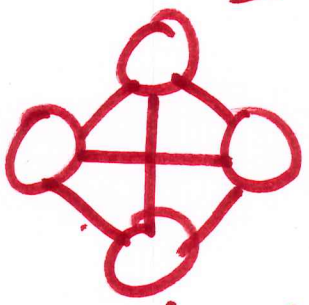


Not a complete graph

In a complete graph

n = # vertices
m = # edges

$$m = \frac{1}{2}\sum_{v \in V} \deg(v)$$ (from before)

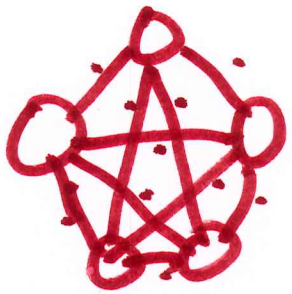


$\deg(v)$ complete graph $= (n-1)$

$$m = \frac{1}{2}\sum_{v \in V} (n-1)$$ (for a complete

$$m = \frac{1}{2}n(n-1)$$ for a complete graph

$m = \frac{1}{2}4(3) = 6$

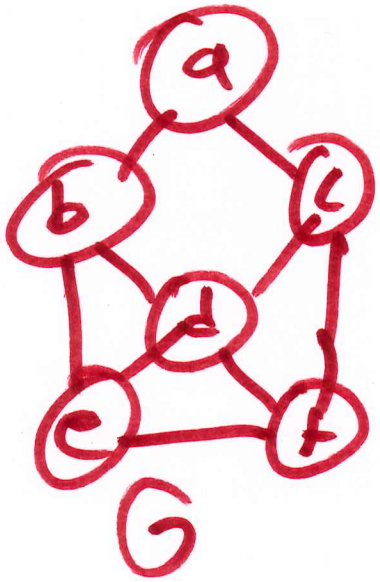


$m = \frac{1}{2}5(4) = 10$

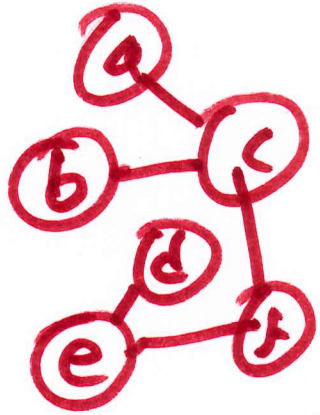
Spanning Tree

A spanning tree of G is a subgraph of G that

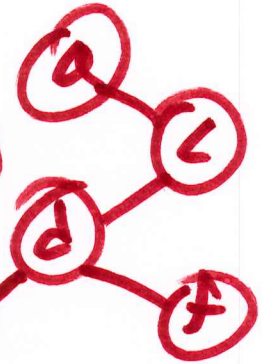
1. — is a tree
2. — contains all vertices of G



G

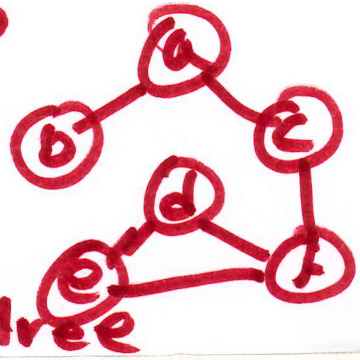


Spanning Tree of G



Not spanning tree

b is missing

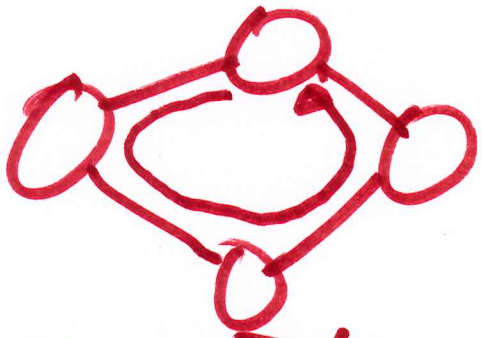


Not a tree

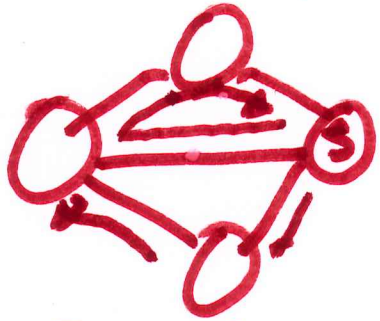
Eulerian Tour

Path that traverses every edge exactly once and returns to the first vertex.

Not all graphs have eulerian tour



It has a Eulerian tour



It does not have Eulerian tour

Euler's theorem

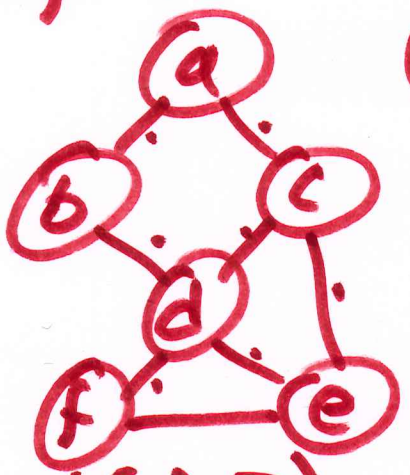
A graph has an Eulerian Tour if and only if all vertices have an even degree.

Data structure for graphs

- Edge List
- Adjacency List
- Adjacency Matrix

Edge List

It simply stores the list of edges of the graph.



G=[(a, b), (a, c), (c, d), (b, d), (d, f), (d, e), (f, e), (e, f)]

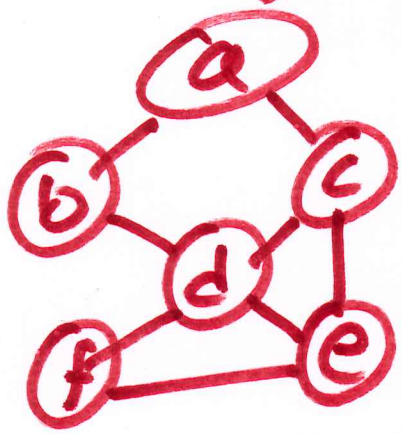
Space: O(m)

n - #vertices
m - #edges

Adjacent(i, j)
may take O(m)

Adjacency list

For very vertex u there is a list of vertices adjacent to u.



G:

a → (b, c)
b → (a, d)
c → (a, d, e)
d → (b, c, e, f)
e → (c, d, f)
f → (d, e)

2*m

n - #vertices
m - #edges

Space: O(n+m)

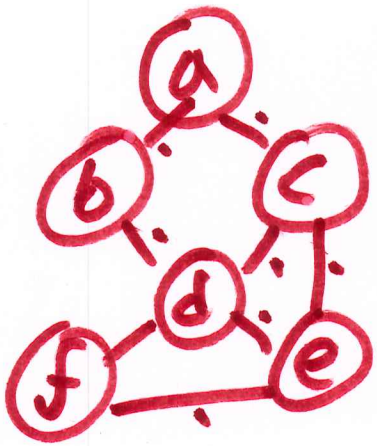
adjacent(i, j)
May take O(n).

Adjacency Matrix

It is a matrix M such that

M[i,j] = True if there is an edge from v_i to v_j.

M[i,j] = False if there is no edge from v_i to v_j.



n = #vertices
m = #edges

space: $O(n^2)$

	a	b	c	d	e	f
a	F	T	T	F	F	F
b	T	F	F	T	F	F
c	T	F	F	T	T	F
d	F	T	T	F	T	T
e	F	F	T	T	F	T
f	F	F	F	T	T	F

The function adjacent(i,j) takes = $O(1)$

Quiz 22

(B) 211

In a complete graph with n vertices, the number of edges is:

a) n^2

b) n

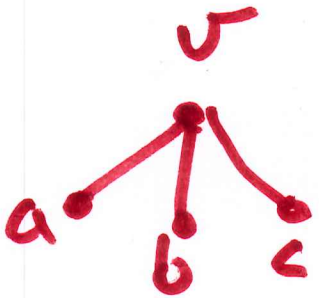
c) $n(n-1)$

d) $\frac{1}{2}n(n-1)$

Depth First Search

- It is a traversal of a graph that starts at a vertex and recursively visits every adjacent vertex.
- Every time a vertex is visited, it is marked. A marked vertex is not visited again.

Algorithm DFS(v)
Input: A vertex v in the graph
Output: A labeling of edges as discovery edges. Edges used to discover an undiscovered vertex.
back edges: Edges that reach an already visited vertex

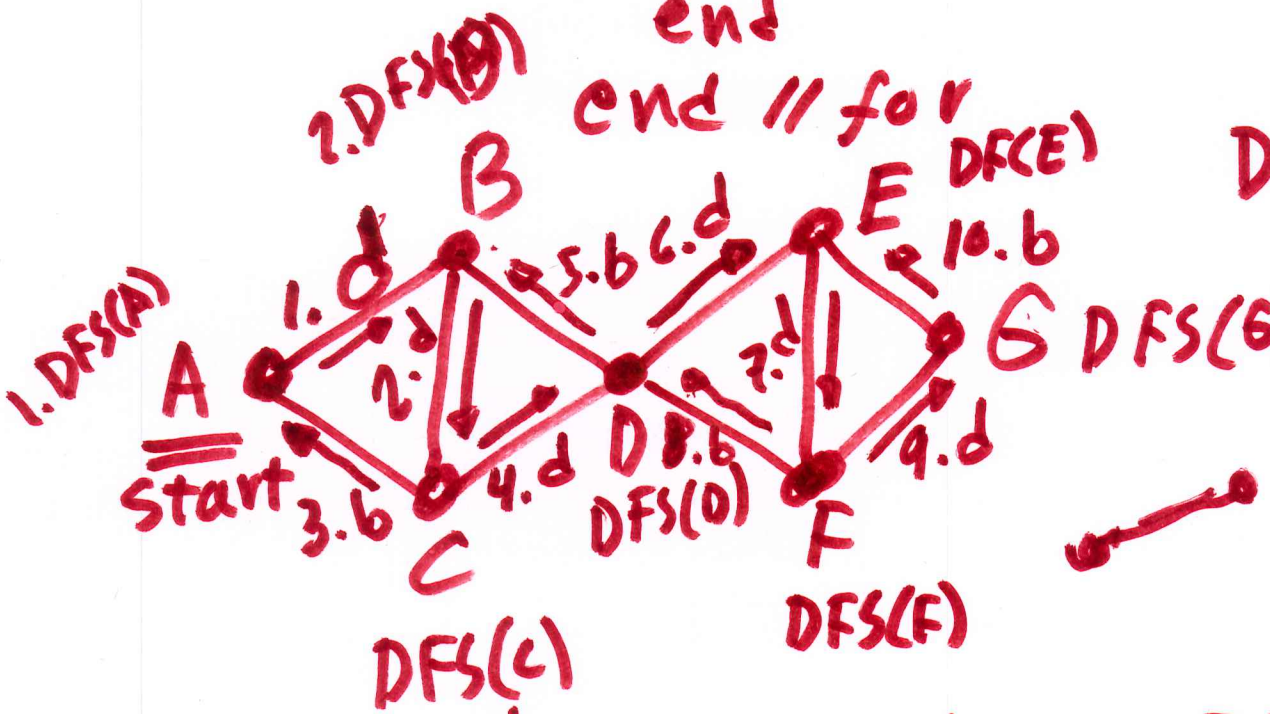


for each edge incident in v do
  if edge $e=(v,w)$ is unexplored then
    if vertex w is unexplored then
      label e as a discovery edge
      recursively call DFS(w)
    else
      label e as a back edge

end

end

end // for



Subgraph of discovery edges: It gives a possible path to visit all edges.

The subgraph of discovery edges is a tree (no cycles).

DFS properties

Let G be an undirected graph with a DFS traversal that starts at vertex s.

1. The traversal visits all nodes in the connected component of s.
2. The discovery edges form a spanning tree of the connected components of s.

A spanning tree of G is a subgraph that:

a) Contains all vertices of G

and b) Does not have a cycle

c) Connects all vertices.

n = # vertices of G

m = # edges

Complexity of DFS

$O(n)$ - DFS is called for each vertex only once

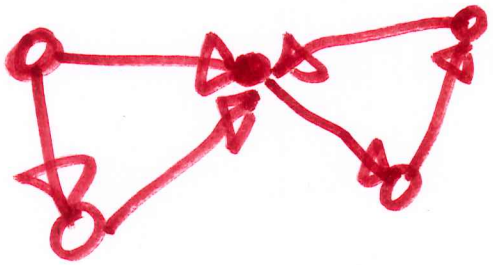
\+ $O(m)$ - Every edge is examined twice, once for each DFS of its vertices

$O(m+n)$ Total = $O(m+n)$ It assumes we use Adjacency Lists.

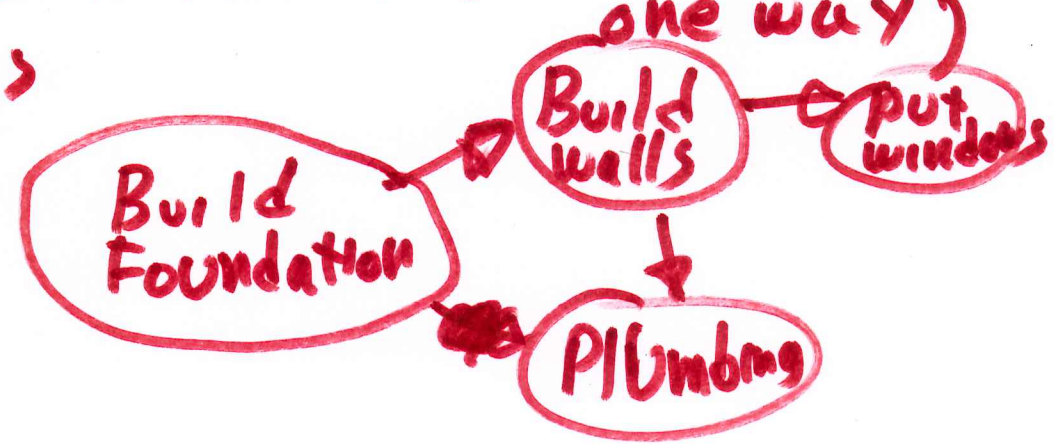
Digraphs

- Directed graphs

  They are graphs where edges go only in one direction



- Useful to represent
  - Transportation networks (streets may have only one way)
  - Dependencies
  - Scheduling



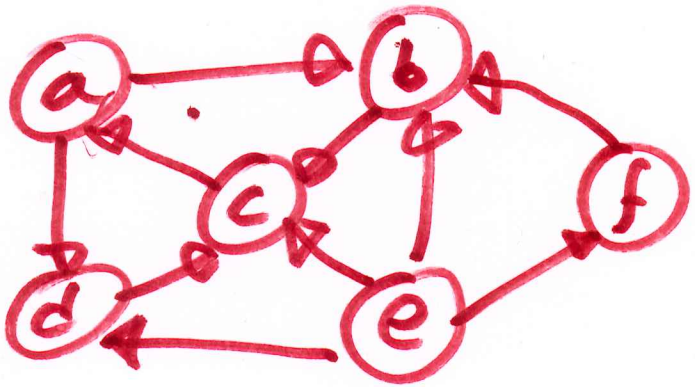
DAG

- Directed Acyclic Graph
- Directed graph with no cycles

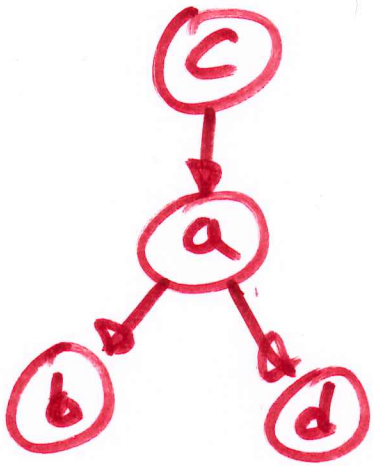
DFS can also be used with directed acyclic graphs

Reachability

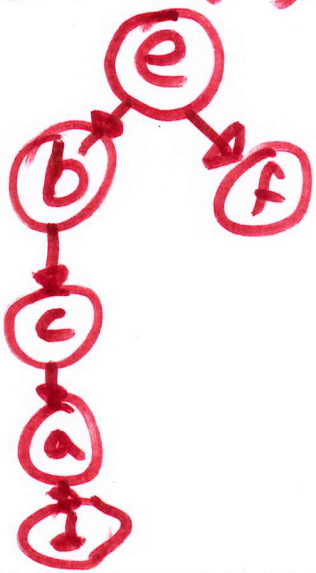
A DFS tree of v gives the vertices that are reachable from v via directed paths



DFS(c)

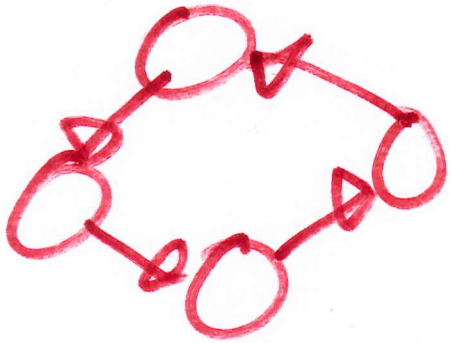


DFS(e)

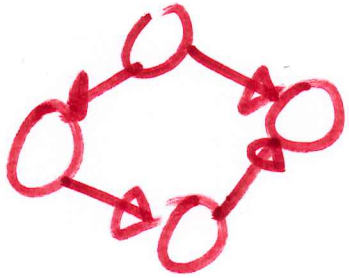


- Strongly connected digraphs

They are directed graphs where each vertex can reach all other vertices



Strongly Connected



Not strongly Connected

To determine if a graph is strongly connected, run DFS for every vertex and if at every DFS all vertices are visited then the graph is strongly connected

n times ⇒ for all vertices v of G
DFS(v) → $O(m+n)$
| and if not all vertices are visited return false
return true

Complexity = $n\,O(m+n) = O(nm+n^2)$

Transitive Closure

Quiz 23

The function getAdjacentVertices(v) returns an array with the vertex numbers of vertices adjacent to v. What is the complexity of this function if we use an edge list data structure to represent the graph.

n = # vertices
m = # edges

a) O(n)
b) $O(n^2)$
c) O(m)
d) O(1)

Dijkstra's Algorithm

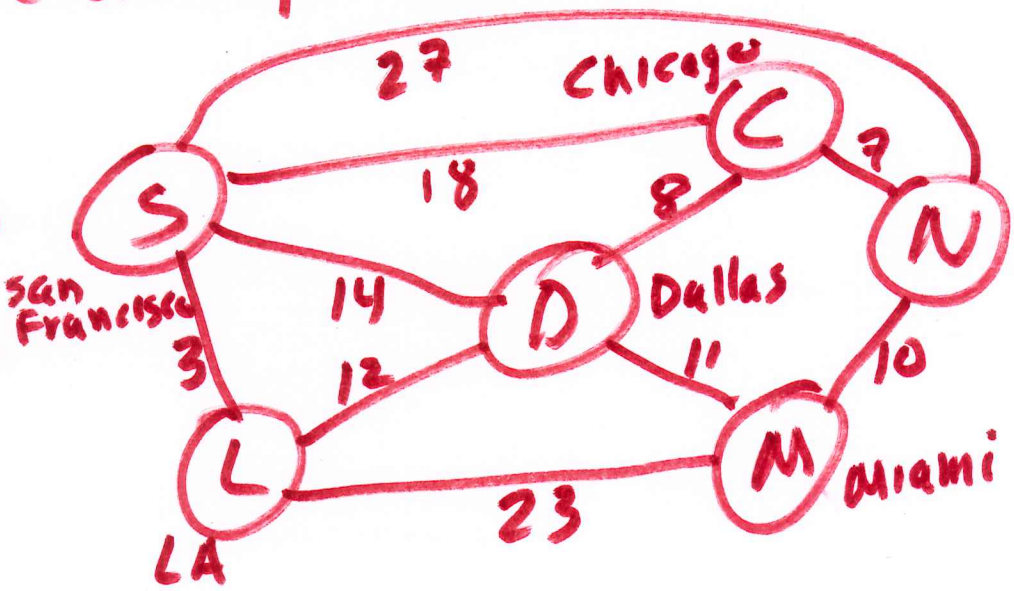
Finds shortest path from a starting vertex to all vertices in graph.

- Undirected edges
- nonnegative edge weights.

Edges have a weight that represent the distance.

weight is distance in 100's miles

27 → 2700 miles



shortestpath(G, v)

v start vertex

returns shortest path to every other vertex

- The algorithm stores in an array $D[v]$ the estimated distance from the starting vertex v to vertex v. The algorithm may modify $D[v]$ at every iteration. At the end of the algorithm $D[v]$ contains the shortest distance from v to v for every v.

- Initially

$$D[v] = 0$$
$$D[v] = \infty \quad \text{for } v \neq v$$

- At each step, we choose vertex v such that $D[v]$ is minimum and v has not been visited.

- For each edge z adjacent to v


```
if (D[v] + w[v,z] < D[z]) {
    D[z] = D[v] + w[v,z]
}
```

Algorithm ShortestPath(G, v)

Input: A weighted graph G and vertex v

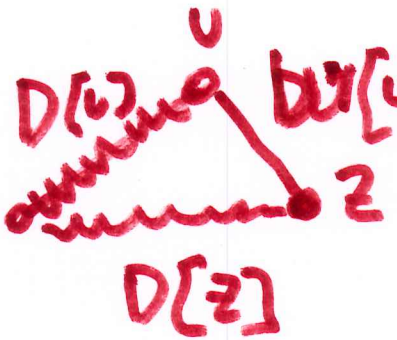
Output: An array D[u] such that D[u] is the length of the shortest path from v to u in G

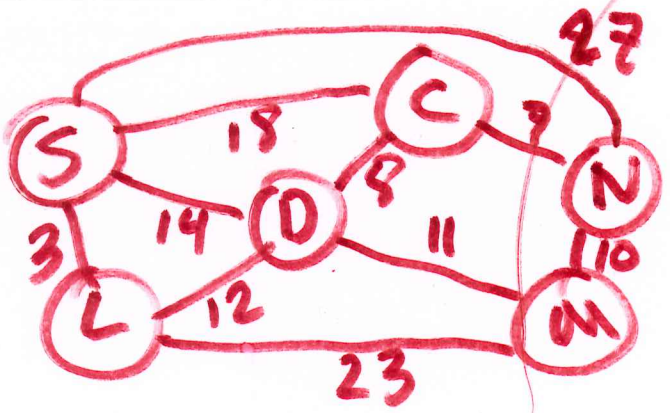
Initialize
$D[v] = \phi$
$D[u] = \infty \quad u \neq v$

Initialize $Q = \{$all vertices in G$\}$

while Q is not empty do
- Get a vertex u from Q such that D[u] is minimum among all the vertices in Q.
- Remove u from Q
- For each vertex z adjacent to u such that z is in Q do
  - if $D[u] + w[u,z] < D[z]$
    - $D[z] = D[u] + w[u,z]$
  - end
- end

end





Shortest (G, C)

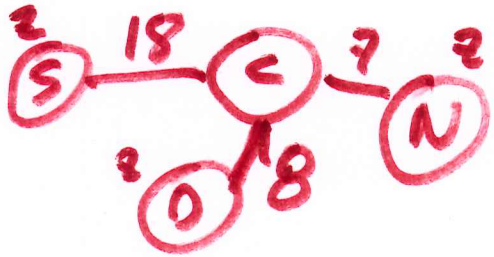
Step 1 D[]

Q = {C, D, L, M, N, S}

	D[]
C	0
D	∞
L	∞
M	∞
N	∞
S	∞

Step 2 ☛ U = min D[v] in Q

U = C ✓ Q = {D, L, M, N, S}



D[]

C 0

- D → min(∞, 0+8) = 8, C
- L ∞
- M ∞
- N → min(∞, 0+7) = 7, C
- S → min(∞, 0+18) = 18, C

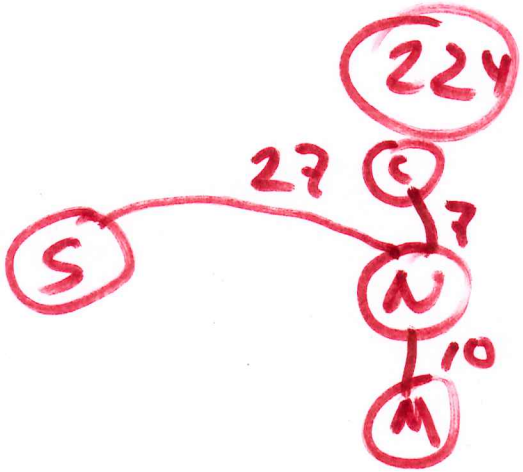
Step 3

$u = N$ (min $D[]$ in Q)

$Q = \{D, L, M, S\}$

	$D[]$
C	0
D	8 ✓
L	∞
M	$\min(\infty, 7+10) = 17$
N	7
S	$\min(18, 7+27) = 18$

$D[N]$



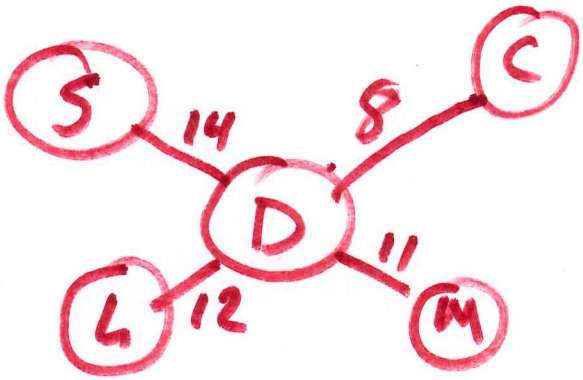
Step 4

$u = D$ (min $D[]$ in Q)

$Q = \{L, M, S\}$

	$D[]$
C	0
D	8
L	$\min(\infty, 8+12) = 20$
M	$\min(17, 8+11) = 17$, D
N	7
S	$\min(18, 8+14) = 18$

(D)



Step 5

U = M (min D[v], v in Q)

Q = {L, S}

D[]

C 0

D 8

L min(20, 17+23) = 20

M 17

N 7

S 18 ✓

275

Step 6 U = S (min D[v] s.t. v in Q)

D[]

Q = {L}

C 0

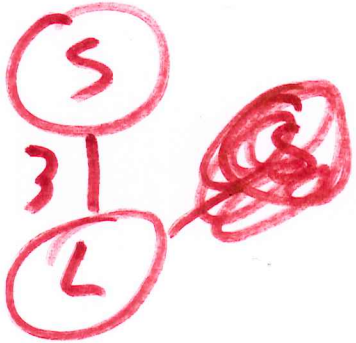
D 8

L min(20, 18+3) = 20

M 17

N 7

S 18



Step 7 D[] U = L (min D[v] s.t. v in Q

Q = {}

C 0

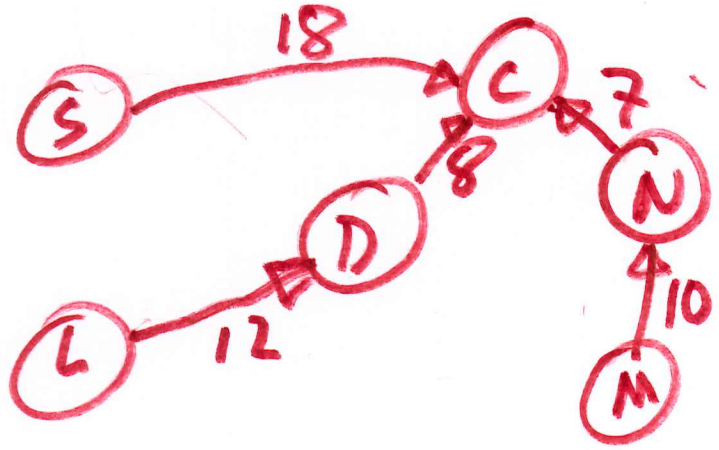
D 8

L 20 N 7

M 17 S 18

End D[] is min path.

P[C] = C
P[D] = C
P[L] = D
P[M] = N
P[N] = C
P[S] = C



Shortest path tree

To obtain the path modify the algorithm to store for every vertex that the minimum is updated with the vertex that allows reaching that vertex with the minimum.

```
if ( D[u]+w[u,z] < D[z] ) {
    D[z] = D[u]+w[u,z]
    //update path
    P[z] = u          // P points to the
                      // parent vertex in
                      // shortest path
}
```

Quiz 24

Can the shortest path graph with all edges used to reach the vertices contain cycles?

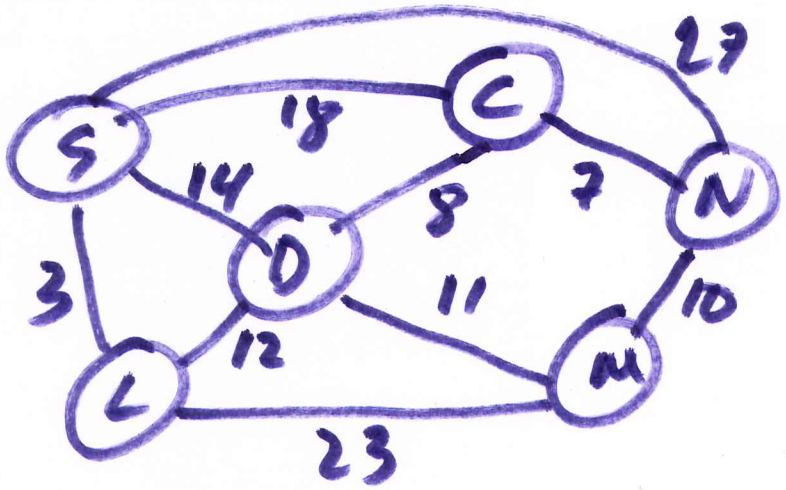
a) Yes

b) No

Minimum Spanning Tree

- It is the spanning Tree with the minimum total weight.
- Example: Connect all computers in a building using the minimum amount of cable.

Example: Connect all the cities in a map using the minimum amount of road miles.

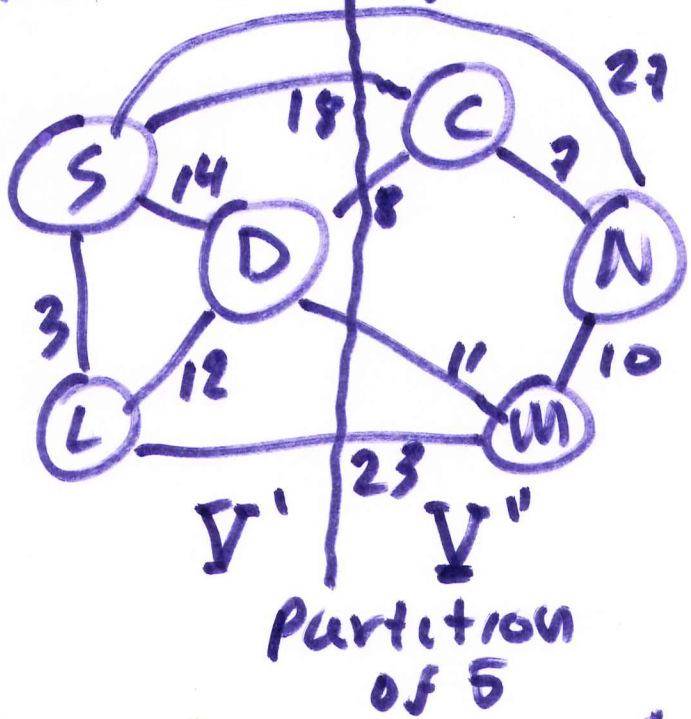


what is the MST?

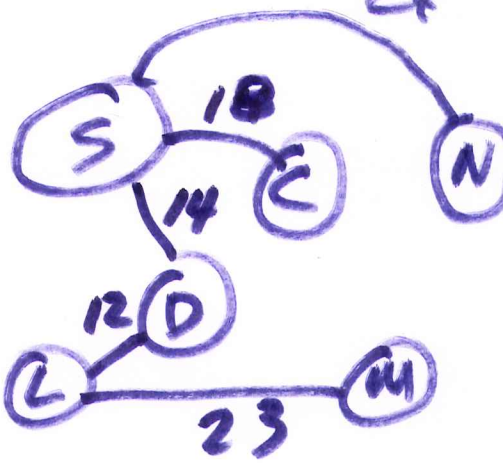
Minimum Spanning Tree Property

Let V' and V'' be a partition of the vertices of G

Example:

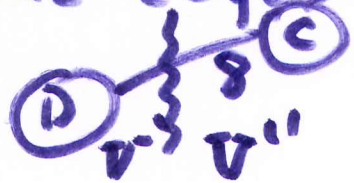


Example of spanning tree that is not minimum



* Let $(v' \text{ and } v'')$ be an edge across the partition s.t. $v' \in V'$ and $v'' \in V''$ and this edge has the minimum weight across the edges in the partition.

Example



then (v', v'') is in the Minimum Spanning Tree.

Proof by contradiction

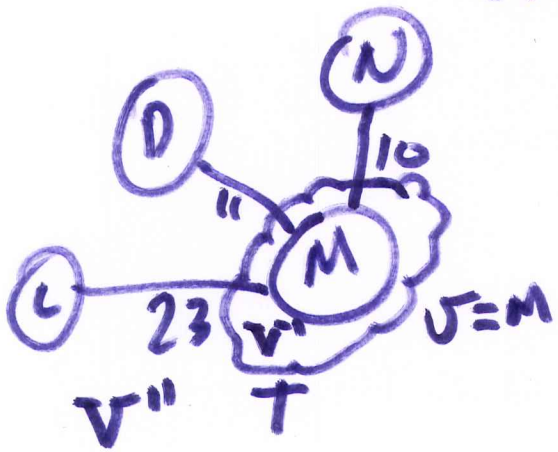
Assume that (v', v'') is not in the Minimum Spanning Tree.

Then there should be another edge e in the MST that joins partitions V' and V''

Since (v', v'') is the minimum edge across the partition then we can substitute e by (v', v'') and obtain another tree that has a smaller sum of weights, so the initial MST was not the minimum that is a contradiction.

Example MST algorithm (Prim-Jarnick)

- Algorithm PrimJarnik(G)
  Input: A weighted graph G
  Output: A minimum spanning tree T of G
  - Pick any vertex v in G
  - $T \leftarrow \{v\}$ // start tree with v
  - $D[v] = \phi$ // D[v] is the distance from vertex v to spanning tree T
  - $D[u] = \infty$ for all $u \neq v$
  - $E[v] =$ none // edge that links v to spanning tree.
  - $Q = \{$all vertices in G$\}$




```
O(n) while (Q is not empty) do
    O(n)  Choose the minimum D[v] s.t.
          v is in Q
        - remove v from Q
        - add E[v] to T
        - for each vertex z adjacent to v
            if weight(v,z) < D[z] then
                D[z] ← w(v,z)
                E[z] ← (v,z)
            end
        end
    end
```

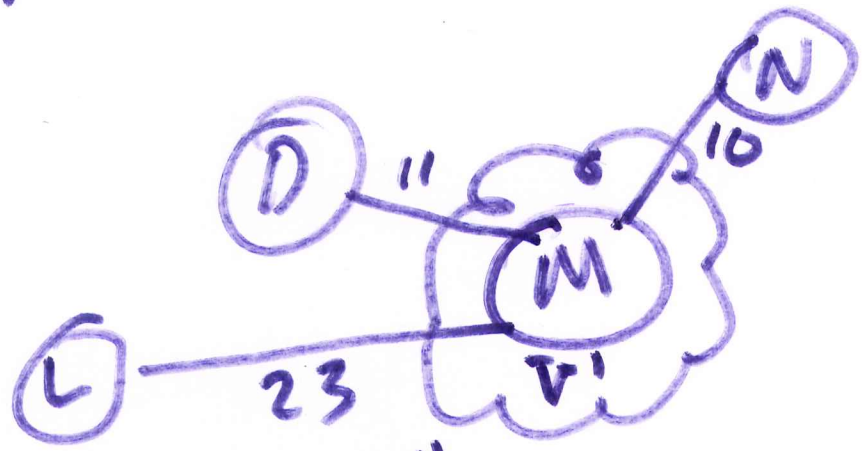
At most once for each edge

$O(m)$

Total time $O(n)O(n) + O(m)$

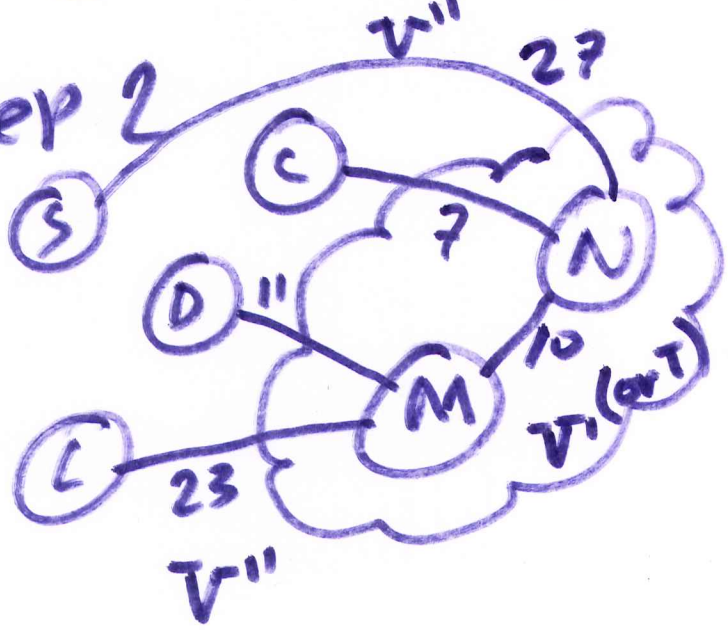
$= O(n^2 + m)$

Step 1 Let V = M starting vertex



should b in MST because is minimum edge across partitions

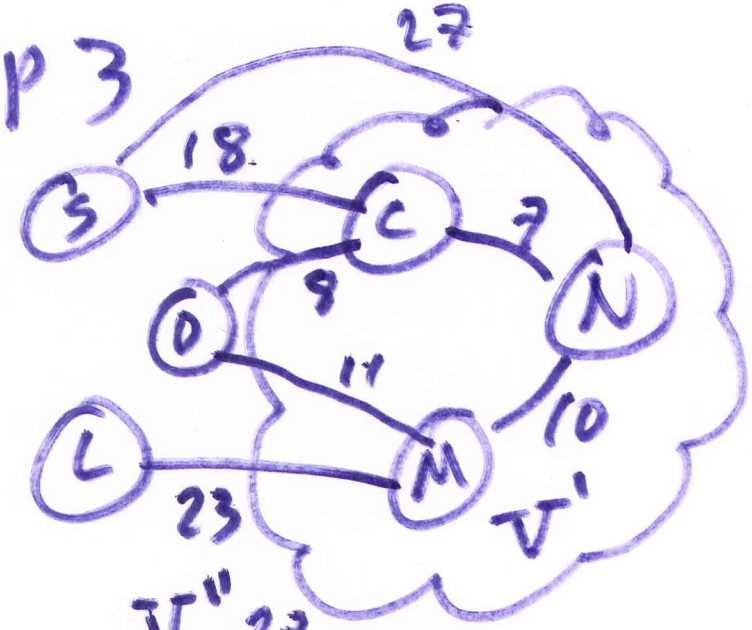
Step 2

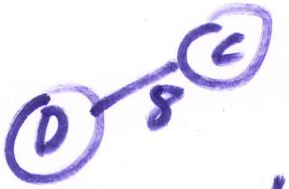


Choose 

to be added to MST because is minimum edge across V' and V''

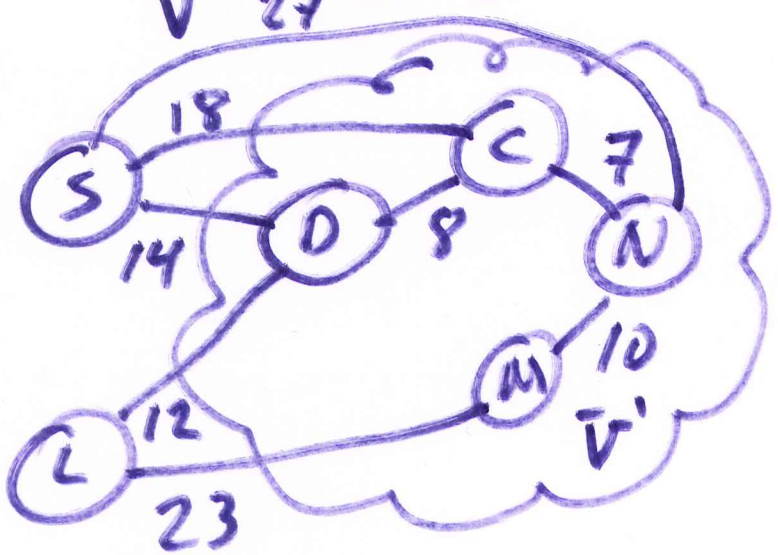
Step 3

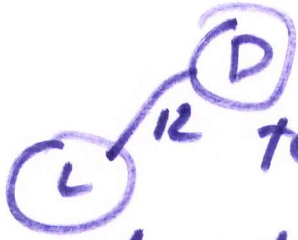


Add  to MST

because is the min edge across partition.

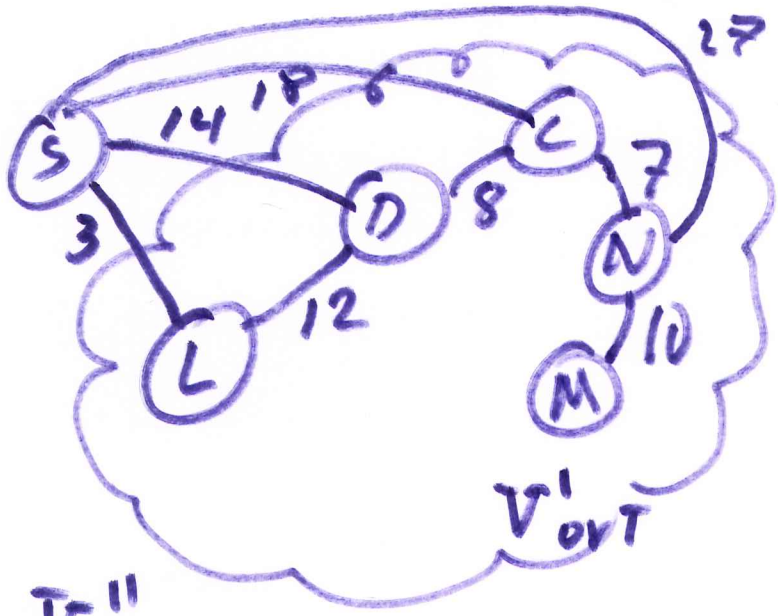
Step 4

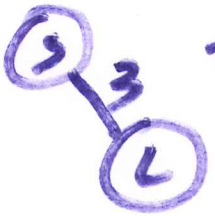


Add  to MST

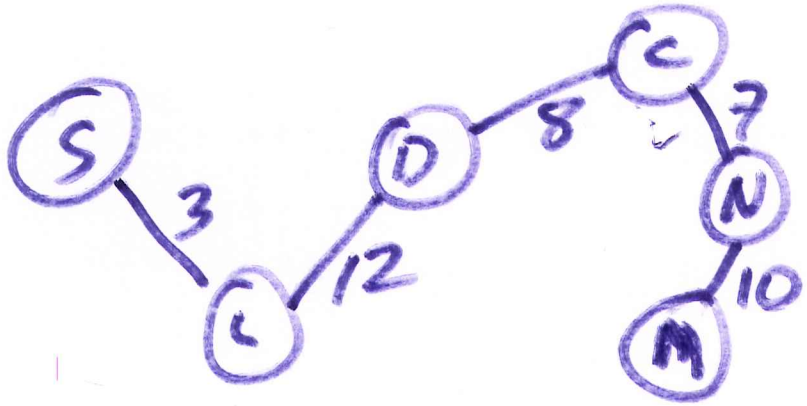
becaus it is the min edge across partition

Step 5



Add  to MST

because it is the min edge across partition

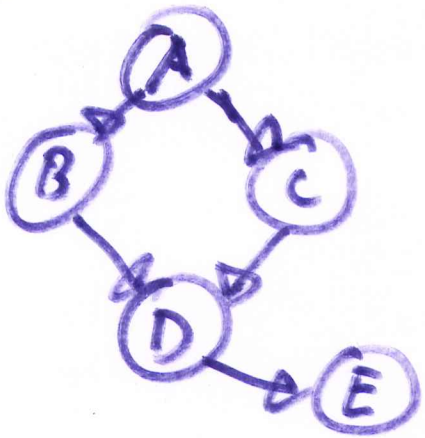


Stop

Minimum Spanning Tree (MST)

Topological Sort

A topological sort of a graph G is an array of vertices such that if there is an edge (i, j)  then i will appear before j in the topological sort.



Topological Sort:

A, B, C, D, E ✓

another topological sort

A, C, B, D, E ✓

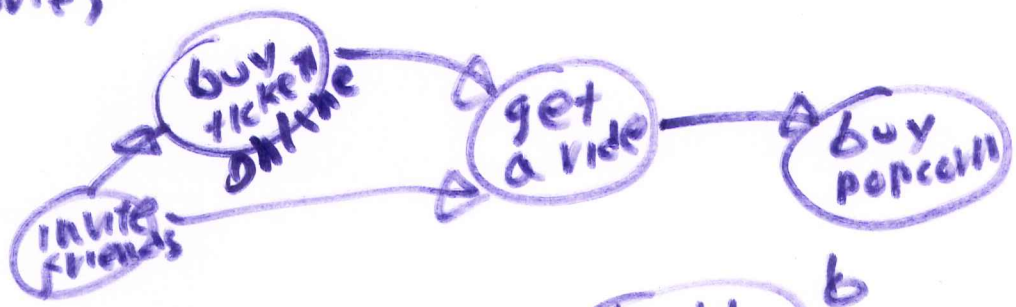
Topological sort may not be unique.

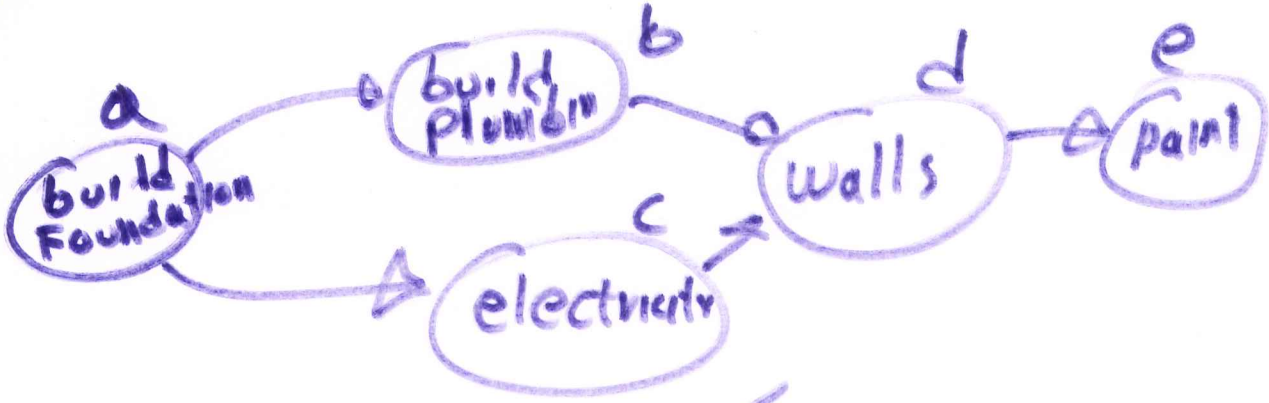
B, A, C, D, E is not a topological sort.

B appears before A and there is edge 

Topological sort can be used to solve problems of scheduling

go Movies





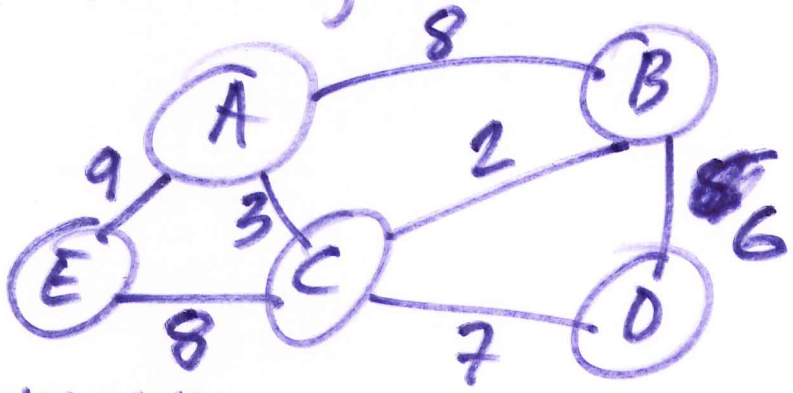
a, b, c, d, e ✓

d, e, a, b, c ✗ not a topological sort

Topological Sort Algorithm

Quiz 24

Assume the graph



Choose right answer

a) The edge (A)–(B) is in the MST

b) The edge (E)–(D) is in the MST

c) The edge (B)–(D) is in the MST

d) The edge (A)–(E) is in the MST

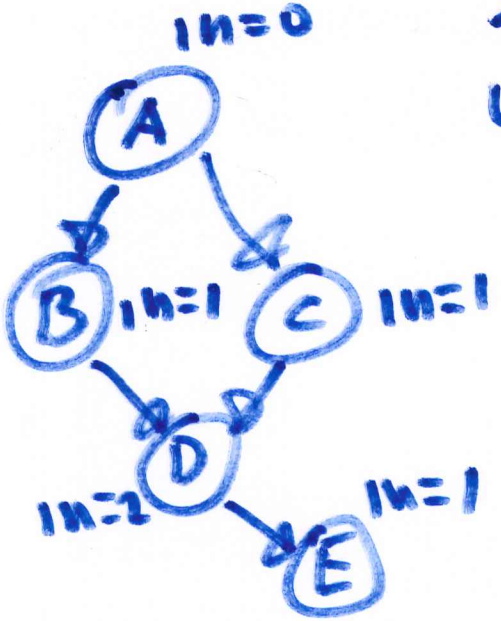
Topological Sorting

Algorithm:

Input: A digraph G with n vertices

Output: A topological ordering $v_1, v_2, \ldots v_n$

```
S ← An empty stack
//Compute indegree
for each edge (v,w) in G {
|  indegree[w]++;
}
//Put in stack vertices with indegree of 0
for each vertex v in G
| if indegree[v]==0 then
| | push v to S
| end
end
```




```
i = φ
while S is not empty do
    v ← S.pop()
    sorted[i] = v        // add v to sorted
                         // array
    i++
    for all outgoing edges (v,w) of v do
        indegree[w]--;   // decrement indegree
                         // of vertex pointed
        if (indegree[w] == φ) then   // by edge from v
            S.push(w)
        end
    end
end
```

Step 1

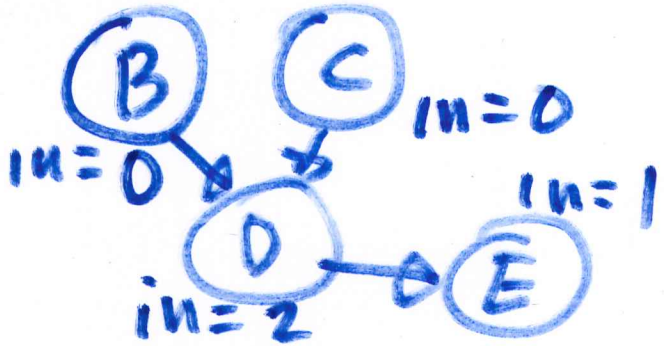
$S = \{A\}$

sorted $= \{\}$

Step 2

$S = \{B, C\}$

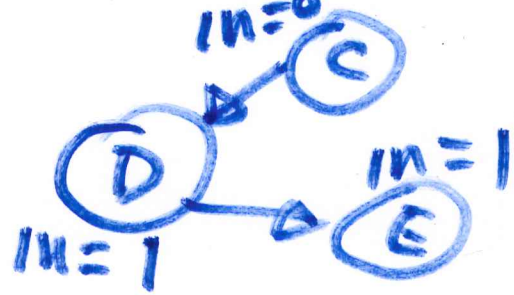
sorted $= \{A\}$



Step 3

$S = \{C, B\}$

sorted $= \{A, B\}$



Step 4

$S = \{D\}$

Sorted $= \{A, B, C\}$

Step 5

$S = \{E\}$

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

Step 6

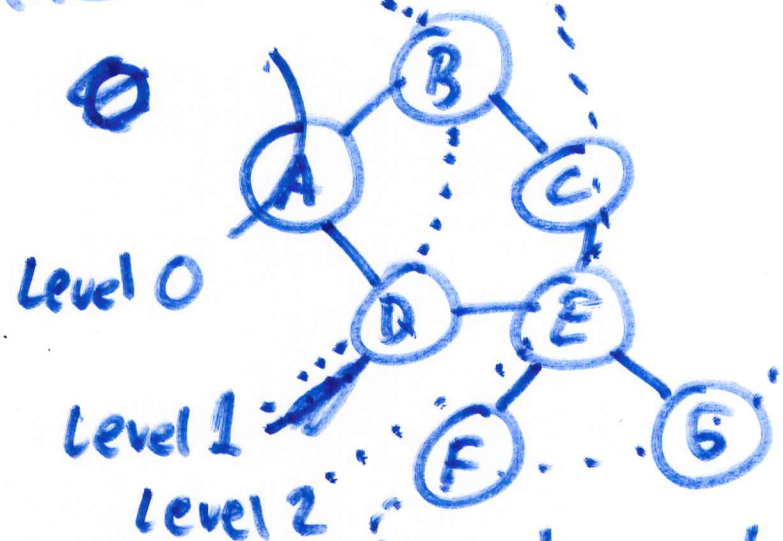
Sorted $= \{A, B, C, D, E\}$

$S = \{\}$

End!

BFS - Breadth First Search

Like DFS, BFS traverses a connected graph and it builds a spanning tree.



BFS(s)

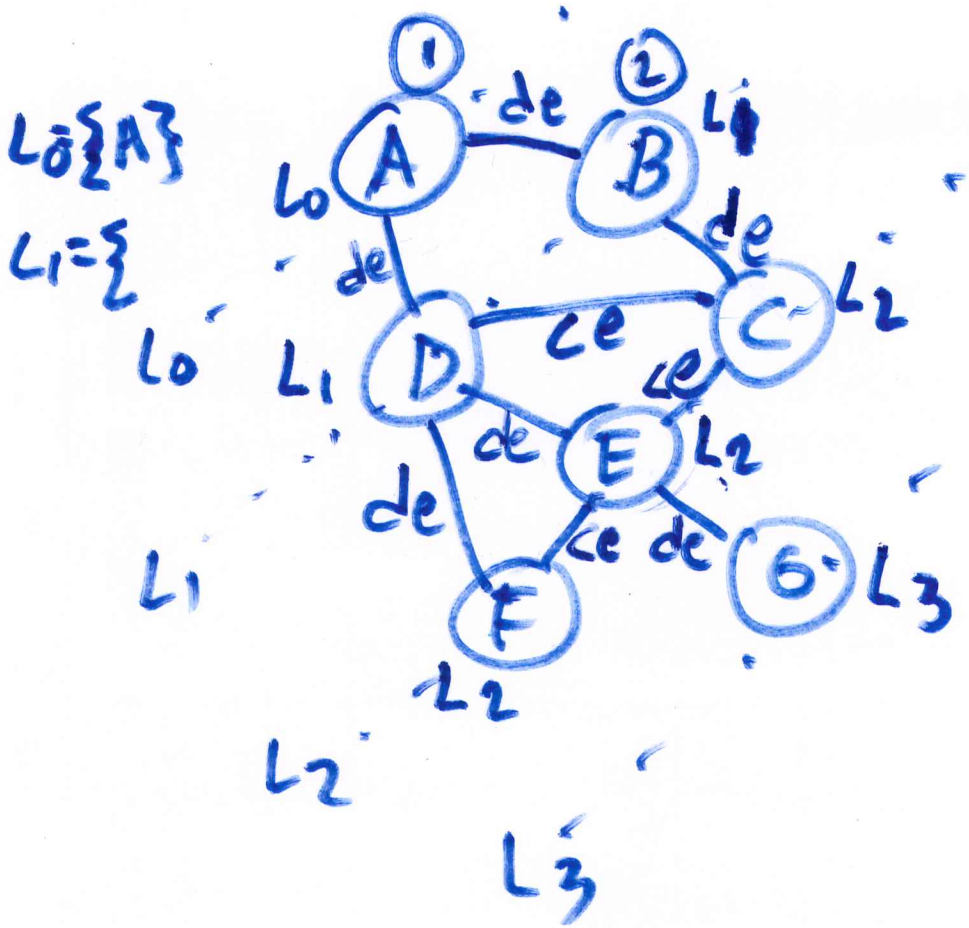
- The starting vertex has level ϕ E.g. BFS(A), A will have a level ϕ
- In the first round, all vertices that are one edge away from s are visited. ~~The edge~~ The vertices will have level 1 E.g. B, D will have level 1.
- In the second round, all new vertices that are two edges away are placed at level 2.
- This continues until all vertices have been assigned to a level

BFS implementation

Algorithm BFS(s)
Input: vertex s
Output: A labeling of the edges as "discovery" edges and "cross edges" and ordering of the vis(t.

Initialize container $L_0 = \{s\}$ ← level 0
$i = 0$
while L_i is not empty do
  create container $L_{i+1} = \{\}$
  for each vertex v in L_i do
    for each edge $e = (v, w)$ do
      if edge $e = (v, w)$ is unexplored then
        If w is unexplored then
          label e as a discovery edge
          insert w into L_{i+1}
        else label e as cross edge
  $i = i + 1$
end while
end for
end for

BFS(A)



$L_0 = \{A\}$

$L_1 = \{$

L_0

L_1

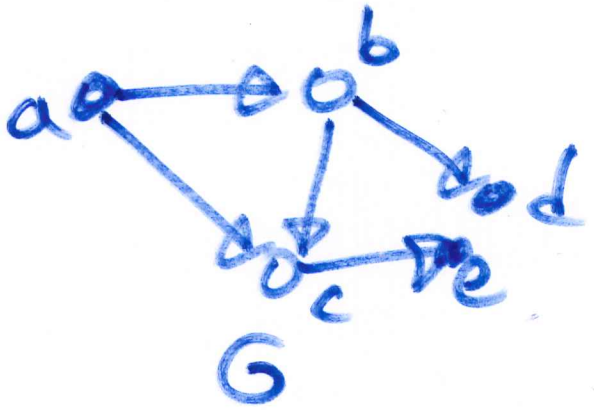
L_2

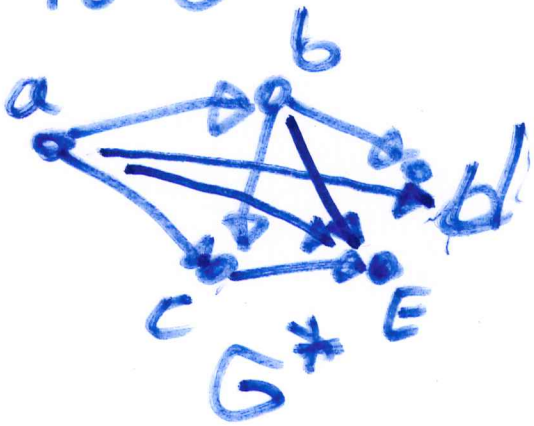
L_3

Transitive Closure

- The transitive closure of a graph G is denoted as G^*

- It is obtained by the following rule:
  If there is a directed path from a to b then add (a, b) to G^*





We can use Floyd Warshall algorithm to compute transitive closure

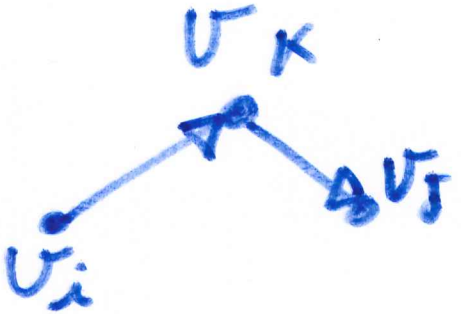
Algorithm TransitiveClosure(G)

Input: Graph G

Output: Transitive closure G^*

$G^* = G$

for (k = 0; k < n; k++) do
  for (j = 0; j < n; j++) do
    for (i = 0; i < n; i++) do
      if (i ≠ j && j ≠ k && k ≠ i
        && (v_i, v_k) is in G^*
        && (v_k, v_j) is in G^*) then
        add (v_i, v_j) to G^*
    end
  end
end



If there is directed path v_i and v_k and v_j then add (v_i, v_j) to G^*

Runs in $O(n^3)$

Final Review

- AVL trees
  - → operations < insert, remove, restructure
  - ← implementation
- Merge Sort (implem.)
  $O(n \log n)$
- Quick Sort (implem)
  $O(n \log n)$ average
- Radix Sort (implem) $O(bn)$ (b = #bits)
- Bucket Sort (implem.) $O(m+n)$ (m = size of range, n = #elems.)
- String matching
  - Brute Force $O(MN)$ (M = length substring, N = length string)
  - Rabin Karp $O(N)$ average
- Data Compression
  - Fixed and variable encoding
  - Huffman Encoding

- Graphs
  - + Graph properties
  - + Shortest Path
  - + Minimum Spanning Tree
  - + Topological Sort
  - + DFS
  - + BFS

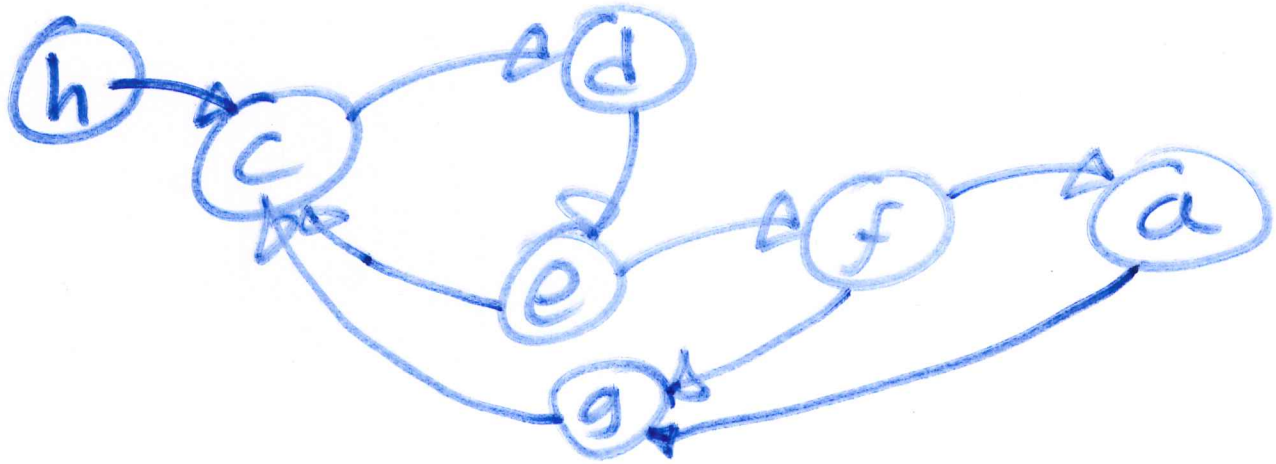
Weight
- 80% 2nd half
- 20% 1st half

To Study
- - Notes
- - Homework
- - Book
- - Projects 2, 3

Quiz 25

(250)

In the topological sort of the following graph:



a) c will appear before a in the topological sord

b) a will appear before c in the topological sort

c) There is no topological sort

d) e appears before f in topological sort