CHAPTER 06
GRAPH ALGORITHMS

Data Structures and Algorithms (17ECSC204)

Prakash Hegade
School of CSE, KLE TECH

2017-18

Graph Traversal Algorithms:

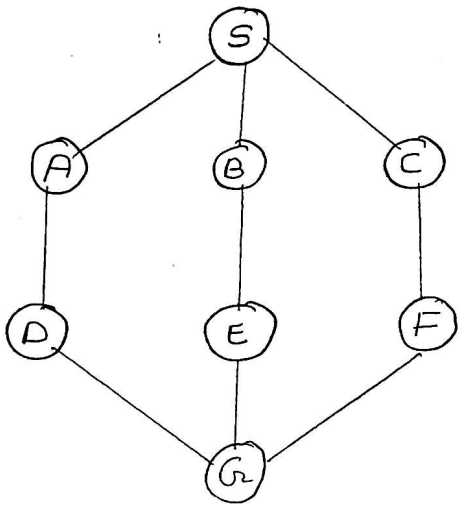
Depth First Search:

is an algorithm for traversing or searching tree or graph data structures.

One starts at root (selecting some arbitrary node as the root in case of a graph) and explores as far as possible along each branch before backtracking.

Examples: Perform DFS on:

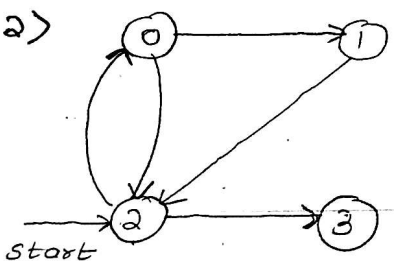
1>



Traversal:

S A D G E B F C

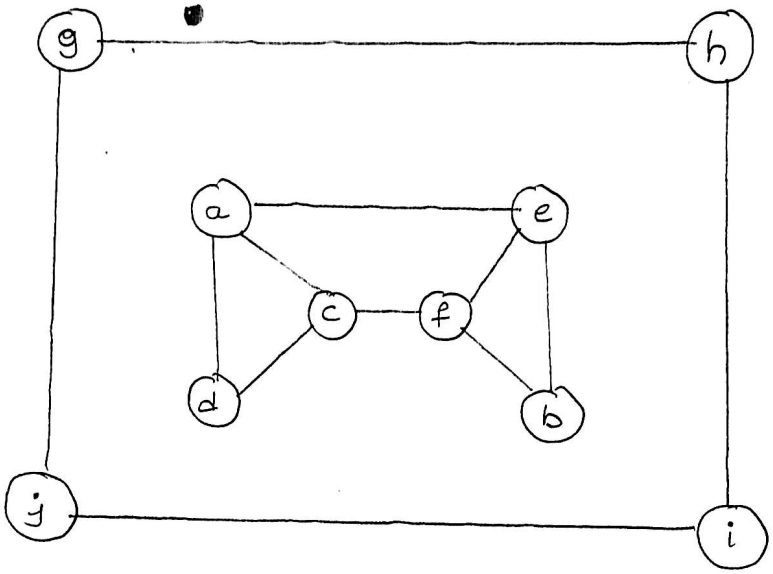
2>



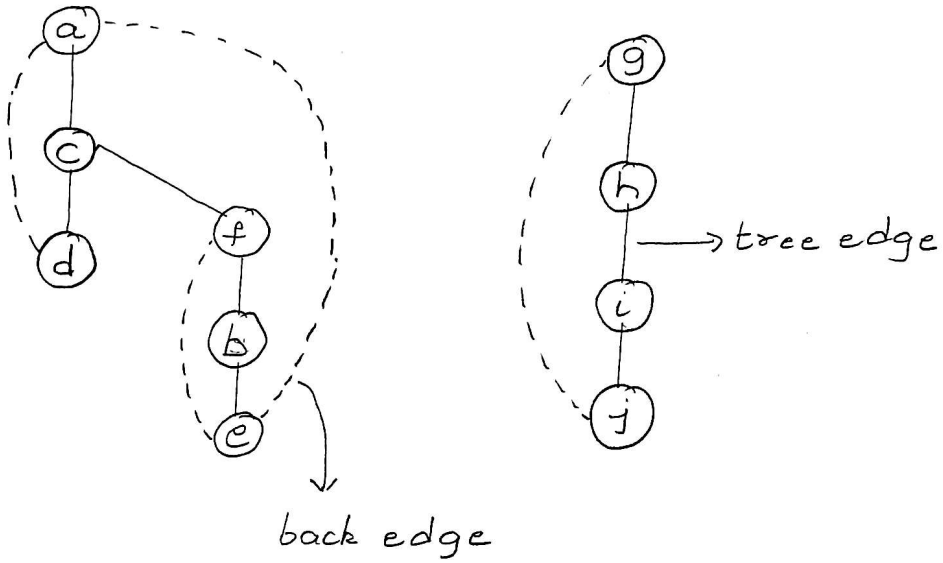
Traversal:

2 0 1 3

3)



DFS forest:



Traversal result:

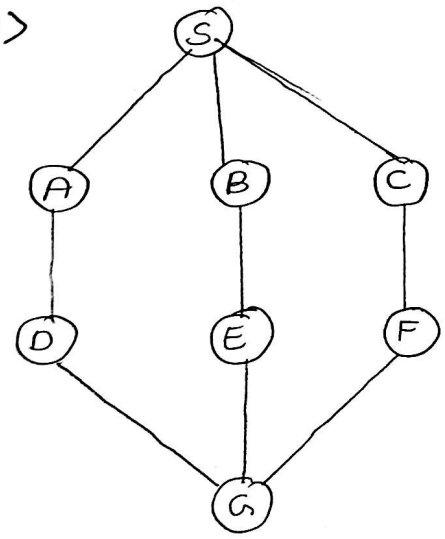
a c d f b e g h i j

Breadth First Search:

is an algorithm for traversing or Searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors.

Examples: Perform BFS on:

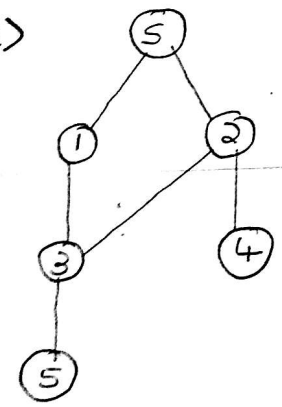
1)



Traversal:

S A B C D E F G

2)

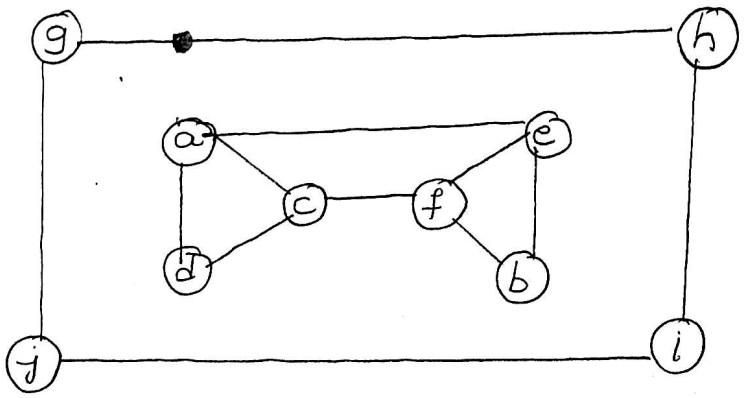


Traversal:

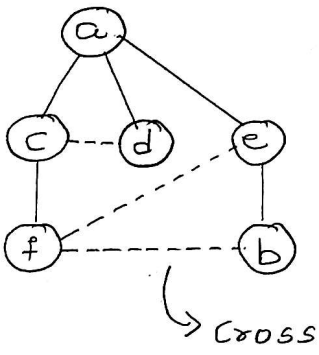
S 1 2 3 4 5

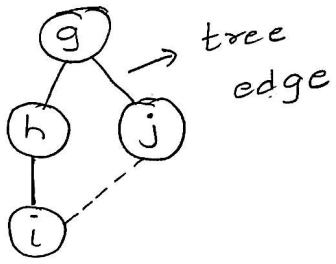
-PH

3)



BFS Forest:





Traversal result:

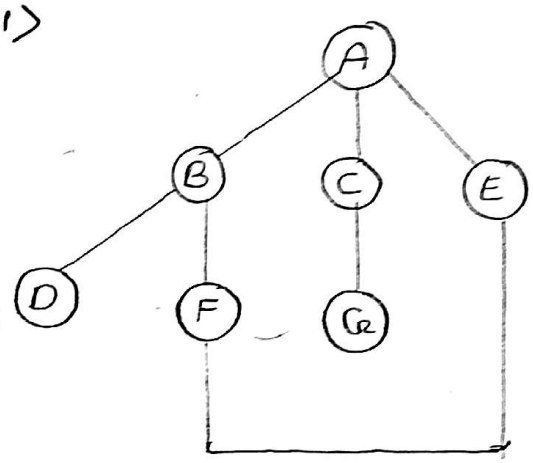
a c d e f b g h j i

DFS and BFS Comparison:

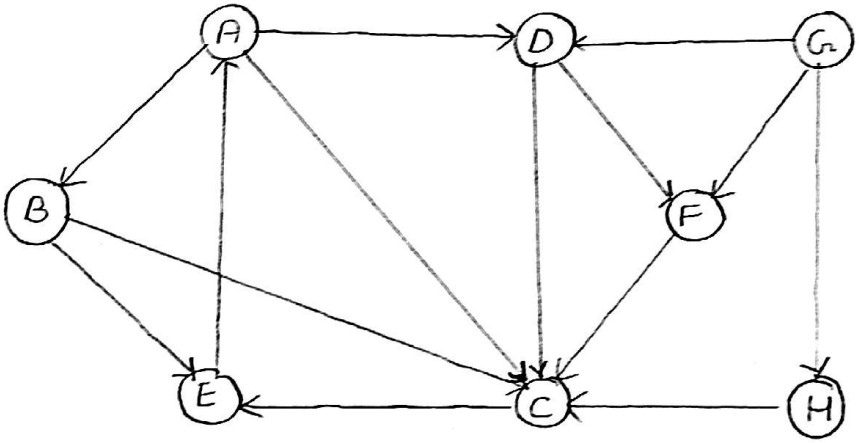
Feature	DFS	BFS
1. For	Brave ones	cautious ones
2. Data structure	Stack	Queue
3. Edge types	tree edges back edges	tree edges cross edges
4. Matrix representation efficiency	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
5. List representation efficiency	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$
6. Vertex Orderings number	2 orderings	1 ordering
7. Applications	-Detecting cycles -Path finding -Bipartite graph -maze with one solution -etc	- Crawlers - Social n/w websites - GPS systems -cycle detection -etc

Exercise: Perform DFS & BFS on:

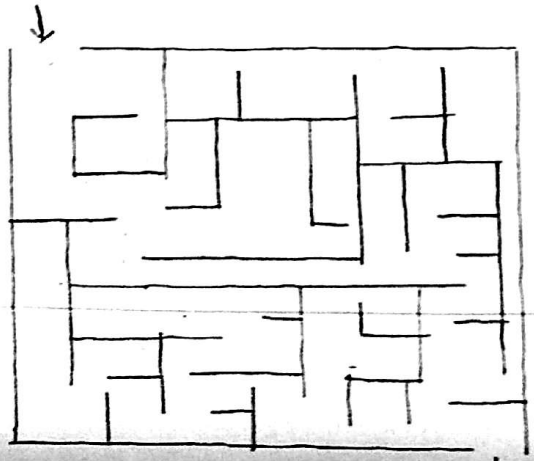
1)



2)



3) Is the maze for brave ones or cautious ones?



Generic Graph Search:

Goals:

a) Find everything reachable from given start vertex

b) Dont explore anything twice

Generic Algorithm: Given graph G, vertex s

- initially s-explored, all other vertices unexplored
- while possible:
  - choose an edge (u, v) with u explored & v unexplored
  - mark v explored

{ if none, halt

Discovering all the Nodes:

Input Graph (s)

DiscoveredNodes ← {s}

while there is an edge e leaving DiscoveredNodes that has not been explored:

add vertex at other end of e to the DiscoveredNodes

return DiscoveredNodes

-PH

Book-keeping:

- How to keep track of which edges / vertices we have dealt with?
- What order do we explore new edges in?
- What if edges have weights?
- Can we sort the edges?
- Is the graph directed or undirected?
- How do we represent the graph?
- Does the graph have more than one component?
- Is the graph sparse or dense?
- Do we need to keep track of discovered edges or discovered vertices?
- How do we make sure that every node or ~~vis~~ is visited only once?
- what other supporting Data Structures would be needed for traversal?

Preorder & Postorder Numbers your DFS:

DFS marks all vertices as visited. We can certainly keep track of other data that can be useful. Design Augment functions to store additional information.

```
Explore(v)
visited(v) ← true
previsit(v)
for (v, w) ∈ E:
    if not visited(w):
        explore(w)
postvisit(v)
```

What can be previsit(v) & postvisit(v)?

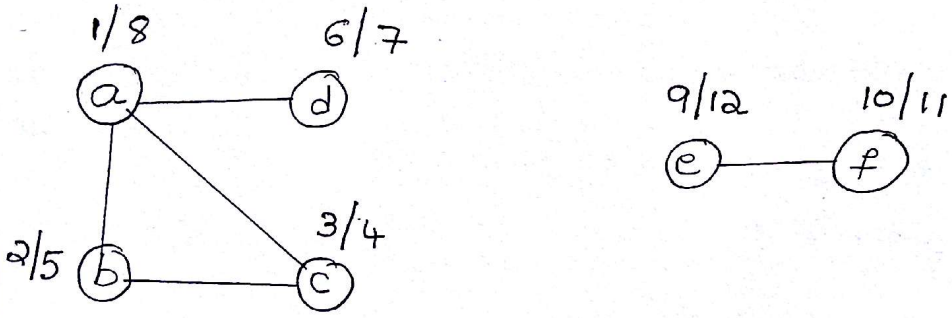
One possible option:

- keep track of order of visits
- Record previsit & postvisit times for each v.
- Its like having ~~some~~ clock ticks

we can certainly design the functions any other way as the application demands.

Example:

Ordering:




```
tick ← 1

previsit (v)
  pre(v) ← tick
  tick ← tick + 1

postvisit (v)
  post(v) ← tick
  tick ← tick + 1
```

Previsit and Postvisit numbers tell us about the execution of DFS.

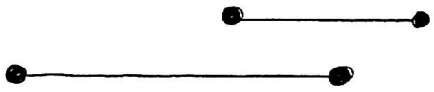
Lemma: For any vertices u, v the intervals [pre(u), post(u)] and [pre(v), post(v)] are either nested

Nested & Disjoint:

Nested:

Disjoint:

The interleaved case is not possible:



Which of the following tables is not a valid set of pre- & post- ordees?

Table 01:

V	Pre	Post
A	1	8
B	9	10
C	3	4
D	2	7
E	5	6

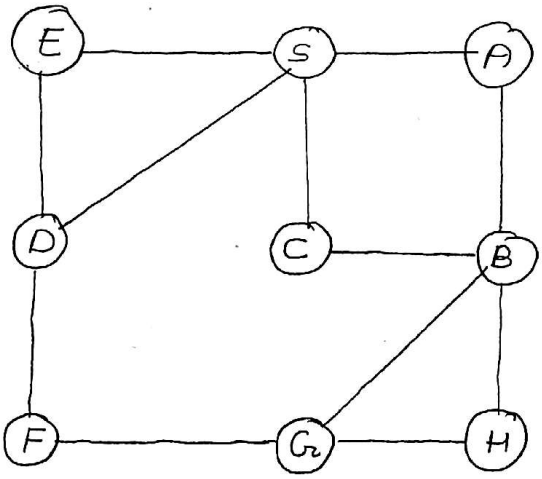
Table 02:

V	Pre	Past
A	1	9
B	8	10
C	2	7
D	3	6
E	4	5

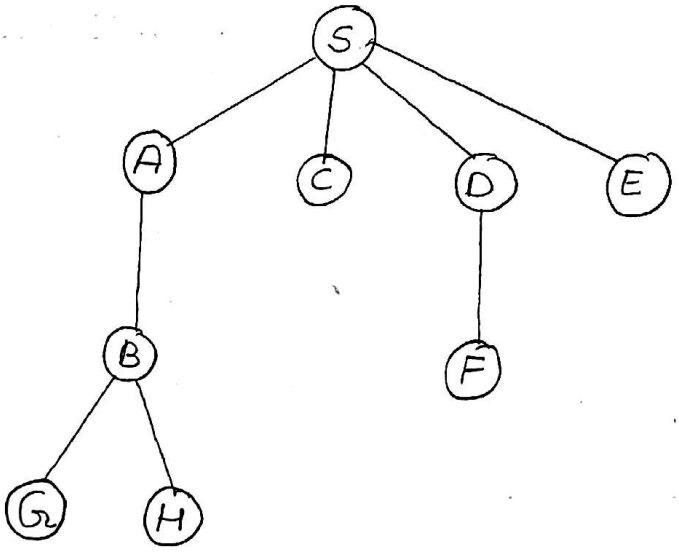
Answer: Table 02.

Can you justify why?

Question: Draw a shortest path tree for the given graph



We can apply BFS traversal to obtain the shortest path tree.

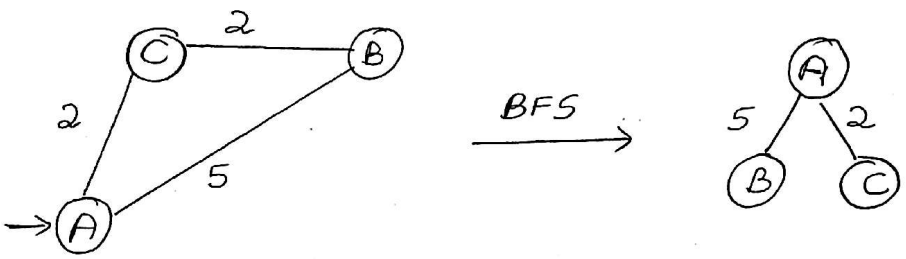


Note how BFS creates a tree with shortest path from given source vertex to every other vertices.

Fastest Route:

Most of the real time problems come with weights. Path can be based on weights rather than based on number of edges.

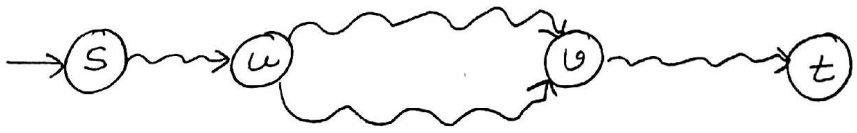
Example: For the graph below:



AB with weight 5 is not the optimal path. ACB with weight 4 is.

Naive Algorithm:

Observation: Any subpath of an optimal path is also optimal.



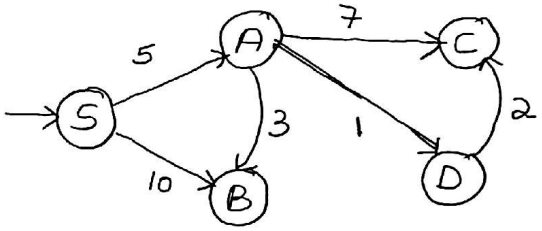
A shorter path from u to v would get a shorter path from s to t.

Let dist[v] be an upper bound on the actual distance from s to v.

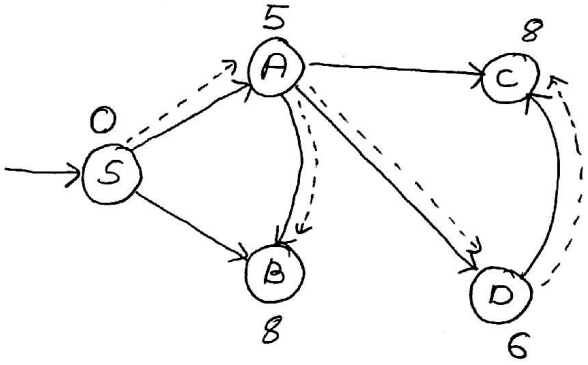
The edge relaxation procedure for an edge (u, v) just checks whether going from s to v through u improves the current value of dist[v].

Dijkstra's Algorithm:

Intuition:

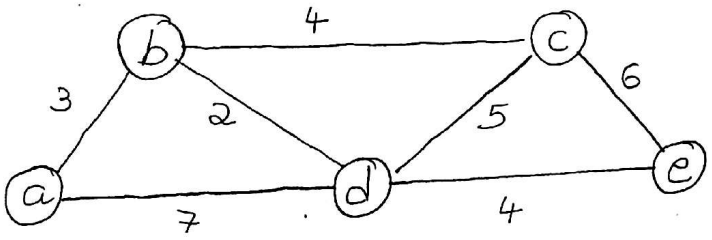


After Edge Relaxation



- We maintain a set R of vertices for which dist is set correctly
- The first vertex added to R is s.
- On each iteration take a vertex outside of R with the minimal dist-value, add it to R & relax all its outgoing edges.

Eg:- Apply dijkstra's algorithm (single source shortest path algorithm) your:



Trce Vertices / Edges	remaining Vertices / Edges
a →	a ↔(3) b a ↔(∞) c a ↔(7) d a ↔(∞) e
a ↔(3) b →	b ↔(7) c b ↔(5) d b ↔(∞) e , a ↔(∞) e
a ↔(3) b, b ↔(2) d →	b ↔(7) c d ↔(9) e
a ↔(3) b ↔(4) c, b ↔(2) d →	d ↔(9) e
a ↔(3) b ↔(4) c, b ↔(2) d ↔(4) e	

- PH

Analysis:

The time is given by:

$$\sum_{i=1}^{n-1} \left[\sum_{j=0}^{n-1} 1 + \sum_{v=0}^{n-1} 1 \right]$$

$$= \sum_{i=1}^{n-1} \left[(n-1-0+1) + (n-1-0+1) \right]$$

$$= \sum_{i=1}^{n-1} \left[n+n \right] = \sum_{i=1}^{n-1} 2n$$

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

$$= 2n \left[n-1-1+1 \right]$$

$$= 2n \left[n-1 \right]$$

$$= 2n^2 - 2n \approx n^2$$

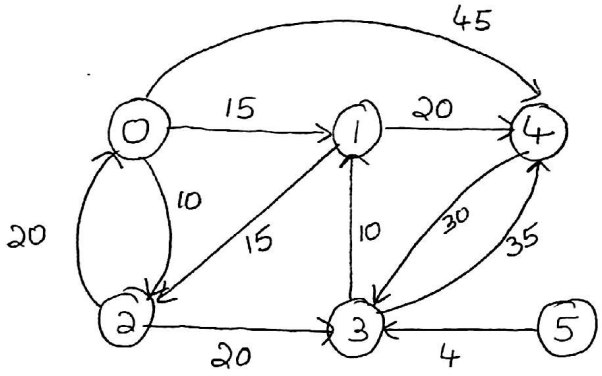
$\therefore T(n) \in \Theta(n^2)$.

as here n corresponds to number of vertices in $G(V, E)$:

Time complexity of Dijkstra's algorithm is given by $\Theta(|V^2|)$.

For adjacency lists: $O((|E|+|V|)\log|V|)$

Trace for Dijkstra's algorithm using data structures. Use 5 as the source vertex.



Cost matrix:

Cost [6] [6] =

	0	1	2	3	4	5
0	0	15	10	∞	45	∞
1	∞	0	15	∞	20	∞
2	20	∞	0	20	∞	∞
3	∞	10	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	4	∞	0

Source = 5

Initializations:

	dist
0	∞
1	∞
2	∞
3	4
4	∞
5	0

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

V-S = {0, 1, 2, 3, 4}

Iteration: 01:

$u = 3$

$dist[u] = 4$

$S = \{5, 3\}$

$V - S = \{0, 1, 2, 4\}$

	dist
0	∞
1	14
2	∞
3	4
4	39
5	0

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

$\min(\infty, 4+\infty) = \infty$

$\min(\infty, 4+10) = 14$

$\min(\infty, 4+\infty) = \infty$

$\min(\infty, 4+35) = 39$

Iteration 02:

$u = 1$

$dist[u] = dist[u] = 14$

$S = \{1, 3, 5\}$

$V - S = \{0, 2, 4\}$

	dist
0	∞
1	14
2	29
3	4
4	34
5	0

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

$\min(\infty, 14+\infty) = \infty$

$\min(\infty, 14+15) = 29$

$\min(39, 14+20) = 34$

Iteration 03:

$u = 2$

$dist[u] = 29$

$S = \{1, 2, 3, 5\}$

$V - S = \{0, 4\}$

	dist
0	49
1	14
2	29
3	4
4	34
5	0

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

$\min(\infty, 29 + 20) = 49$

$\min(34, 29 + \infty) = 34$

Iteration 04:

$u = 4$

$dist[u] = 34$

$S = \{1, 2, 3, 4, 5\}$

$V - S = \{0\}$

	dist
0	49
1	14
2	29
3	4
4	34
5	0

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

$\min(49, 34 + \infty) = 49$

Iteration 05:

$u = 0$

$dist[u] = 49$

$S = \{0, 1, 2, 3, 4, 5\}$

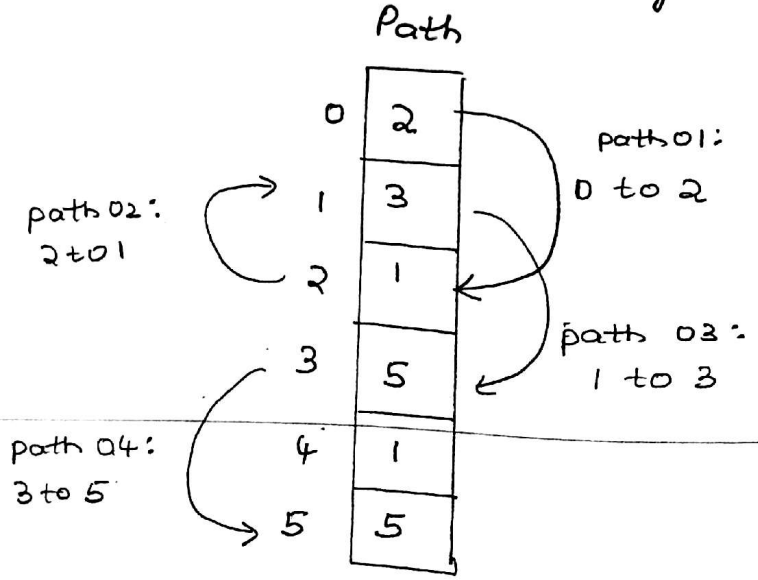
$V - S = \{\}$

	dist
0	49
1	14
2	29
3	4
4	34
5	0

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

How to traverse : Path :

To get the path from S to O, go from reverse order following the vertex path.



Note: For graphs represented by their adjacency lists and priority queue implemented as a min-heap, the efficiency is

$$O(|E| \log |V|).$$

A still better efficiency for dijkstras can be achieved using Fibonacci heap data structure.

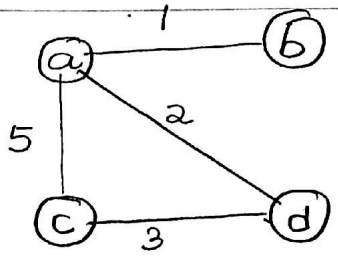
Spanning Trees:

A spanning tree of a connected graph is its connected acyclic subgraph (a tree) that contains all the vertices of the graph.

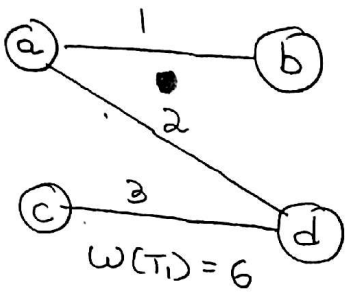
A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where weight of the tree is defined as the sum of the weights on all its edges.

The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

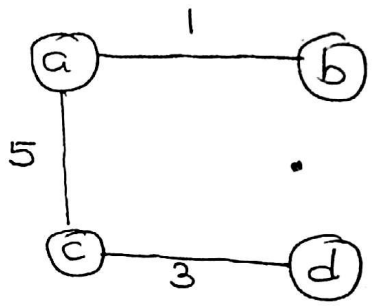
Eg:-



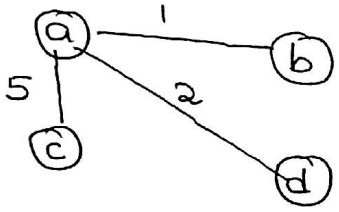
→ Graph.



$w(T_1) = 6$



$w(T_2) = 9$



$w(T_3) = 8$

T_1 is minimum Spanning Tree

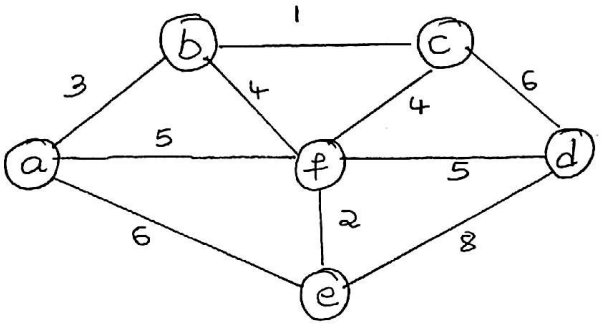
Algorithms to find minimum Spanning Tree:

a) Prims algorithm

b) kruskals algorithm

Prim's Algorithm:

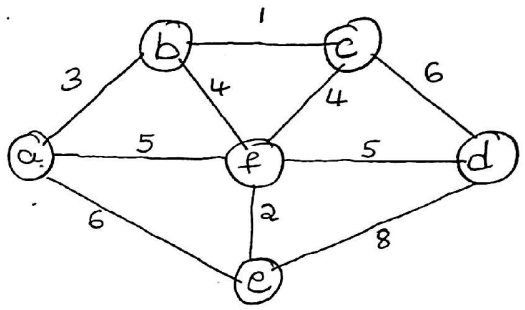
Apply Prim's algorithm on the following graph:



Tree Vertices	Remaining Vertices
a	(a, b) → 3 (a, d) → ∞ (a, f) → 5 (a, c) → ∞ (a, e) → 6
(a, b) → 3	(b, c) → 1 (b, e) → 6 (b, d) → ∞ (b, f) → 4
(b, c) → 1	(c, d) → 6 (b, f) → 4 (a, e) → 6
(b, f) → 4	(f, d) → 5 (f, e) → 2
(f, e) → 2	(f, d) → 5
(f, d) → 5	

Kruskal's Algorithm:

Apply kruskal's algorithm for given graph:



Tree edges

bc
1

ef
2

ab
3

bf
4

df
5

Sorted list of edges

bc	ef	ab	bf	cf	af
1	2	3	4	4	5

df	ae	cd	de
5	6	6	8

Note: We pick edges in such a way that no cycle is formed.

Kruskal's	Prim's
1) Grows with minimum cost edge	1) Grows with minimum cost vertex
2) If we stop algorithm in the middle, kruskal can give a disconnected tree or forest.	2) If we stop algorithm in the middle, prim always generates connected tree.
3) Need to give attention on cycle check	3) Need not give attention on cycle check
4) Can function on the disconnected graphs too	4) Graph must be connected
5) Edge selected is not based on previous step.	5) Spans from one vertex to another
6) Allows new to new & old to old to get connected.	6) Joins new vertex to old vertex
7) With an efficient union-find algorithm, running time will be dominated by time needed for sorting edges, ∴ time efficiency $O(\|E\| \log \|E\|)$	7) Weight matrix & priority queue as unordered array implementation is $\Theta(\|V^2\|)$. Adjacency list & priority queue as min heap running time is $O(\|E\| \log \|V\|)$

Dijkstra's, Prim's & Kruskal's fall under the category of Greedy techniques.

Greedy Technique:

→ Technique mostly applied only to optimiza-tion problems.

The greedy approach suggests constructing a solution through sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be:

a) feasible: has to satisfy the problem constraints

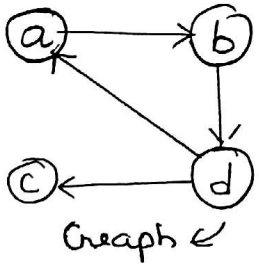
b) locally optimal: has to be best local choice among all feasible choices available on that step

c) irrevocable: one made, it cannot be changed on subsequent steps of the algorithm.

Philosophically, is greedy good or bad?

Warshall's Algorithm:

A transitive closure of a directed graph with n vertices can be defined as the n-by-n boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row $(1 \le i \le n)$ and the jth column $(1 \le j \le n)$ is 1 if there exists a nontrivial directed path from ith vertex to jth vertex; otherwise 0.



Graph

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

→ Adjacency matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

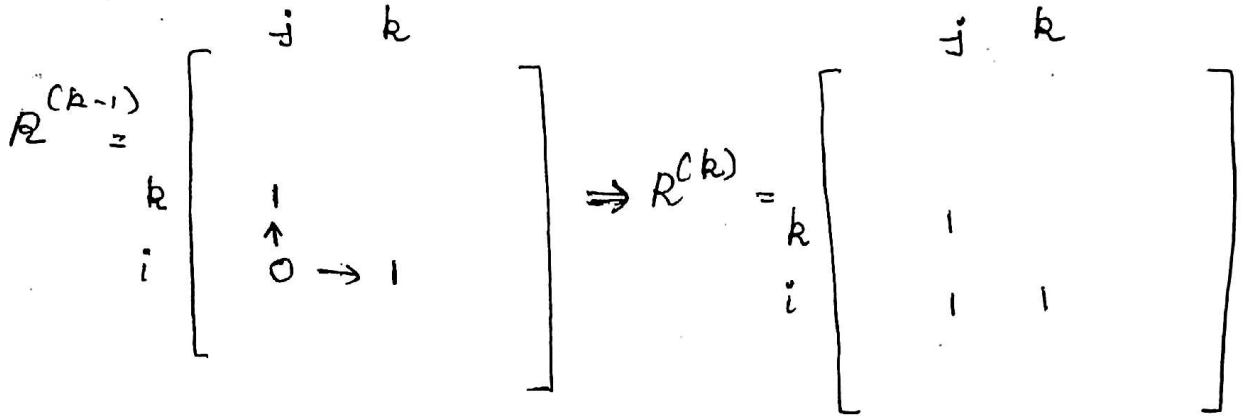
→ Transitive closure

The heart of Warshall's algorithm has the formula:

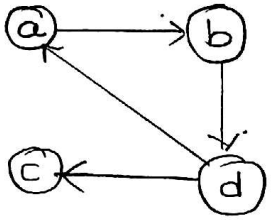
$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } (r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)})$$

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if

the element in its row i and column k & the element in its column j & row k both are 1 in $R^{(k-1)}$.



Tracing:



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

use the boxed one for next iteration

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)}$ =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

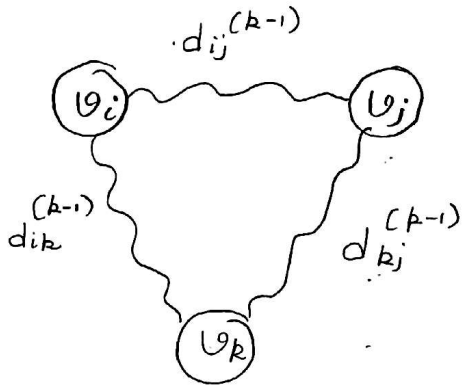
Time efficiency of Warshall is:

$\Theta(n^3)$

Floyd's Algorithm for All-Pairs Shortest-Path Problem:

Given a weighted graph, all-pairs shortest-paths problem asks to find the distances from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n-by-n matrix D called distance matrix: the element d_{ij} in the ith row & jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex $(1 \le i, j \le n)$.

Idea:-

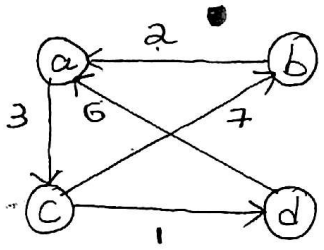


Idea leads to the following recurrence:

$$d_{ij}^{(0)} = w_{ij}$$

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \text{ for } k \ge 1.$$

Apply floyd's algorithm for:



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Time efficiency of Floyd's is:

$\Theta(n^3)$

-PH

Dynamic Programming:

The 'programming' in this technique name stands for 'planning'. Technique invented by Richard Bellman.

It is a technique for solving problems with overlapping sub problems. Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Rather than solving overlapping sub-problems again and again, dynamic programming suggests solving each of the smaller sub-problems only once and recording the results in a table from which we can the obtain a solution to the original problem.

Eg:-
- Warshall's algorithm
- Floyd's algorithm

- PH

Case Study: Efficiency Analysis of Binary Search

Best case: Key found at mid. $\Omega(1)$

Worst case: key found at first or last position.

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

→ Comparison

↳ Search either upper or lower part

$T(n) = T(n/2) + 1$ Assume $n = 2^k$

$T(2^k) = T(2^{k-1}) + 1 \rightarrow (1)$

put $k = k-1$ in (1)

$T(2^{k-1}) = T(2^{k-2}) + 1 \rightarrow (2)$

Substitute (2) in (1)

$T(2^k) = T(2^{k-2}) + 1 + 1$

$= T(2^{k-2}) + 2$

$= T(2^{k-3}) + 3$

$= \ldots$

$= T(2^{k-k}) + k \quad = T(1) + k \quad = 1 + k$

$= 1 + \log_2 n$

$\therefore T(n) \in O(\log_2 n)$

$n = 2^k$

take log on both sides

$\log_2 n = k \log_2 2$

$k = \log_2 n$

Average case: is given by

$$T(n) \in \Theta(\log_2 n)$$

Graph Algorithms

DFS-iterative (G, s)

```
let S be stack
S.push( s )
mark s as visited.
while ( S is not empty):
  //Pop a vertex from stack to visit next
  v = S.top( )
  S.pop( )
  //Push all the neighbours of v in stack that are not visited
  for all neighbours w of v in Graph G:
    if w is not visited :
        S.push( w )
        mark w as visited
```

BFS (G, s)

```
let Q be queue.
Q.enqueue( s )
mark s as visited.
while ( Q is not empty)
  v = Q.dequeue( )

  for all neighbours w of v in Graph G
    if w is not visited
        Q.enqueue( w )
        mark w as visited
```

DFS-recursive(G, s):

```
mark s as visited
for all neighbours w of s in Graph G:
  if w is not visited:
    DFS-recursive(G, w)
```

ALGORITHM Prim(G)

```
// Prim's algorithms for constructing a minimum spanning tree
// Input: A weighted connected graph G = (V, E)
// Output: E_T, the set of edges composing a minimum spanning tree of G
V_T ← {v_0}
E_T ← 0
for i ← 1 to |V| - 1 do
  find a minimum weight edge e* = (v*, u*) among all the edges (v, u)
  such that v is in V_T and u is in V- V_T
  V_T ← V_T U {u*}
  E_T ← E_T U {e*}
return E_T
```

ALGORITHM Kruskal(G)

```
// Kruskal's algorithms for constructing a minimum spanning tree
// Input: A weighted connected graph G = (V, E)
// Output: E_T, the set of edges composing a minimum spanning tree of G
Sort E in increasing order of the edge weights w(e_i1) <= w(e_i2)<= ... w(e_i|E|)
E_T ← 0 ; ecounter ← 0
k ← 0
while ecounter < |V| - 1 do
  k ← k + 1
  if E_T U { e_ik} is acyclic
    E_T ← E_T U { e_ik} ; ecounter ← ecounter + 1
return E_T
```

ALGORITHM Dijkstra (n, cost[ ][ ], src, dest, dist[ ], path[ ])

```
for i from 0 to n-1 do
  s[i] ← 0
  dist[i] ← cost[src, i]
  path[i] ← src

s[src] = 1

for i from 1 to n-1 do
  find u and dist[u] such that dist[u] is minimum and u in V-S
  min ← 99999
  u ← -1
  for j from 0 to n-1 do
    if s[j] = 0 and dist[j]<=min
      min← d[j]
      u← j
  if u = -1 return
  s[u] ← 1
  if u = destination return
  for every v in V-S do
    if dist[u] + cost[u, v] < dist[v]
      dist[v] = dist[u] + cost[u, v]
      path[v] = u
return
```

ALGORITHM Warshall (A[1... n, 1... n])

```
// Implements Warshal's algorithm for computing the transitive closure
// Input: The adjacency matrix A of a digraph with n vertices
// Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      R^(k) [i, j] ← R^(k-1) [i, j] or (R^(k-1) [i, k] and R^(k-1) [k, j])
return R^(n)
```

ALGORITHM Floyd (W[1... n, 1... n])

```
// Implements Floyd's algorithm for the all-pair shortest-paths problem
// Input: The weight matrix with no negative-length cycles
// Output: The distance matrix of the shortest path's lengths
D ← W
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      D[i, j] ← min{ D[i, j], D[i, k] + D[k, j] }
return D
```