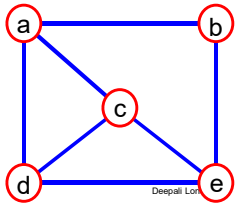
Graph

Deepali Londhe

1

What is a Graph?

- A graph G = (V,E) is composed of:
  - V: set of vertices
  - E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:



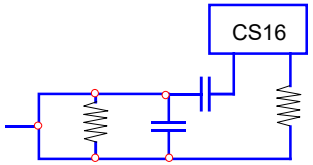
V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),
(d,e)}

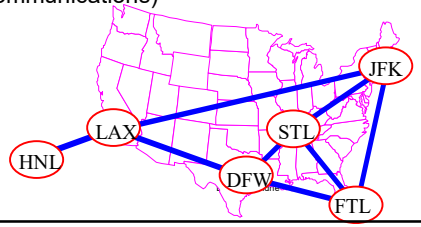
2

Applications

- electronic circuits



- networks (roads, flights, communications)



3

Terminology: Adjacent and Incident

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If <v_0, v_1> is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge <v_0, v_1> is incident on v_0 and v_1

4

Terminology: Degree of a Vertex

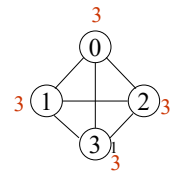
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
- if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i) / 2$$

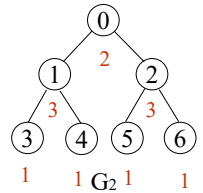
Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

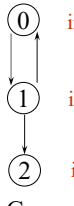
5

Examples



directed graph
in-degree
out-degree





in:1, out: 1

in: 1, out: 2

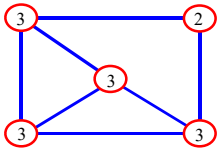
in: 1, out: 0

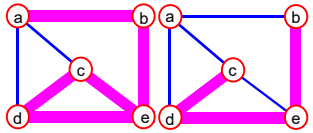
G3

6

Terminology: Path

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



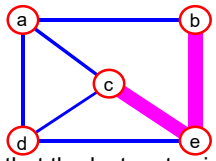


a b e d c e b e d c

7

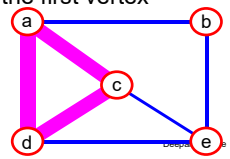
More Terminology

- simple path: no repeated vertices

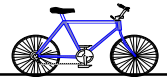


b e c

- cycle: simple path, except that the last vertex is the same as the first vertex

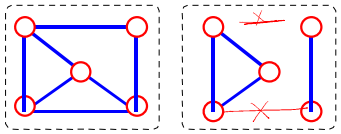


a c d a



Even More Terminology

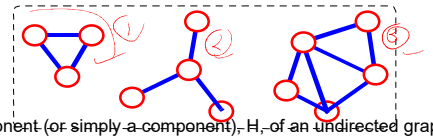
- In an undirected graph, G, two **vertices** u & v are said to be **connected** iff there is a path in G from u to v(& v to u , as undirected)
- An undirected **graph** is said to be **connected** iff for every pair of distinct verices u & v in V(G) there is a path from u to v in G.
- connected graph: any two vertices are connected by some path



connected not connected

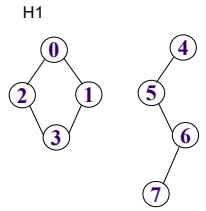
Even More Terminology

- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



- A connected component (or simply a component), H, of an undirected graph is maximal connected subgraph.
  - Maximal means G contains no other subgraph that is both connected and properly contains H.
  - G4 has two connected components H1 & H2 (see next slide)

Connected components

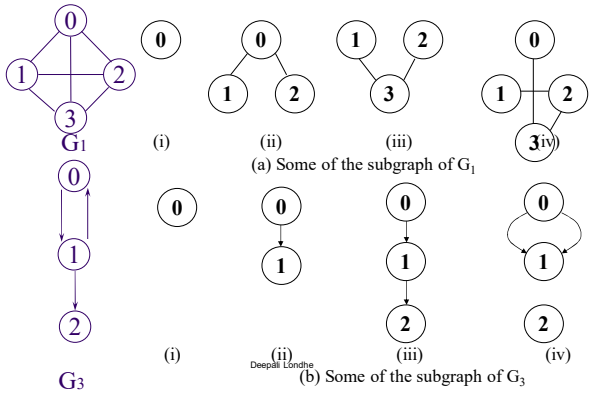


Graph G4 with two connected components H1 & H2

Even More Terminology

- **A directed graph** G is said to be **strongly connected** iff for every pair of for every pair of distinct verices u & v in V(G) there is directed path from u to v and also from v to u.
- Graph G3 is not strongly connected as there is no path from 2 to 1.
- A strongly connected component is a maximal subgraph that is strongly connected. G3 has two strongly connected components. (see iv of G3 on next slide)

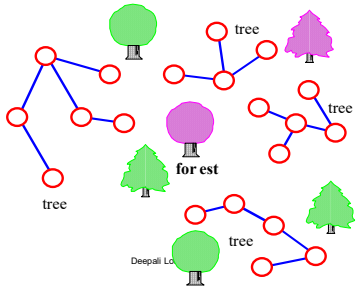
Subgraphs Examples



13

More…

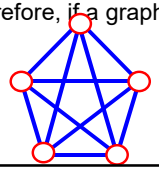
- tree - connected graph without cycles
- forest - collection of trees



14

Connectivity

- Let n = #vertices, and m = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to n-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = n(n -1)/2.
- Therefore, if a graph is not complete, m < n(n -1)/2

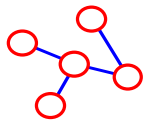


n = 5
m = (5 ∗ 4)/2 = 10

15

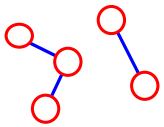
More Connectivity

n = #vertices
m = #edges
- For a tree m = n - 1



n = 5
m = 4

If m < n - 1, G is not connected

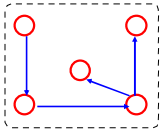


n = 5
m = 3

16

Oriented (Directed) Graph

- A graph where edges are directed



17

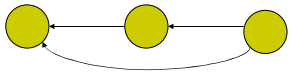
Directed vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1> != <v_1, v_0>$

18

Directed Acyclic Graph

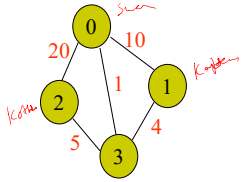
- Directed Acyclic Graph (DAG) : directed graph without cycle



19

Weighted Graph

- Weighted graph: a graph with numbers assigned to its edges
- Weight: cost, distance, travel time, hop, etc.



20

ADT for Graph

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*,*v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

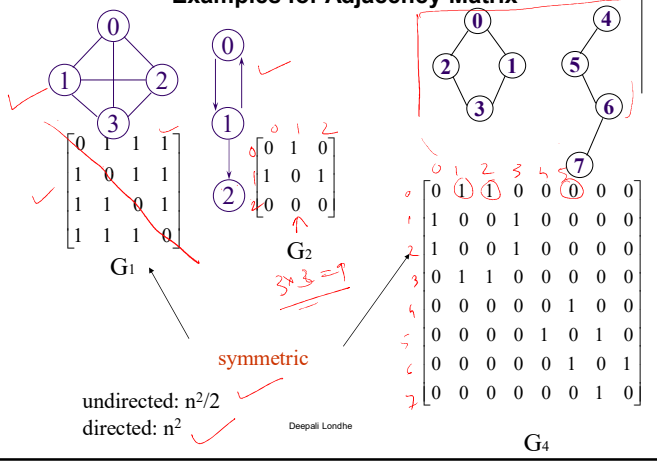
Graph Representations

- Adjacency Matrix
- Adjacency Lists

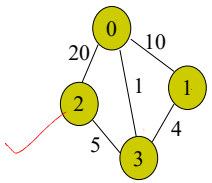
Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



Weighted Graph



A[i][j]	0	1	2	3
0	0	20	10	1
1	20	0	0	5
2	10	0	0	4
3	1	5	4	0

So, Matrix A =

$$\begin{pmatrix} 0 & 20 & 10 & 1 \\ 20 & 0 & 0 & 5 \\ 10 & 0 & 0 & 4 \\ 1 & 5 & 4 & 0 \end{pmatrix}$$

25

Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph (= directed graph), the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

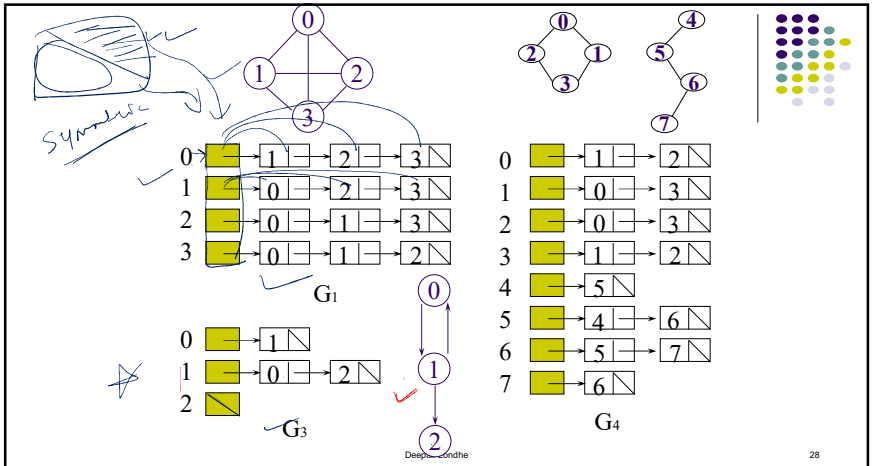
26

Adjacency Lists (data structures)

Each row in adjacency matrix is represented as an adjacency list.

```
#define MAX_VERTICES 50
struct node *node_pointer;
struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```


27

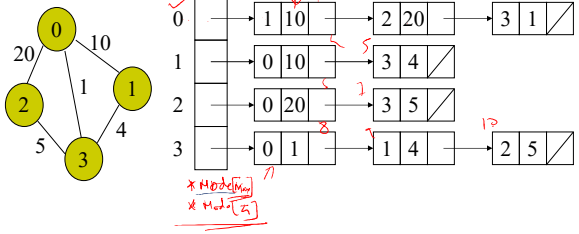


An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

28

Weighted Graph

- Weighted graph: extend each node with an addition field: weight



29

Some Operations

- degree of a vertex in an undirected graph
  - # of nodes in adjacency list
- # of edges in a graph
  - determined in O(n+e)
- out-degree of a vertex in a directed graph
  - # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - traverse the whole data structure

30

Graph Traversal

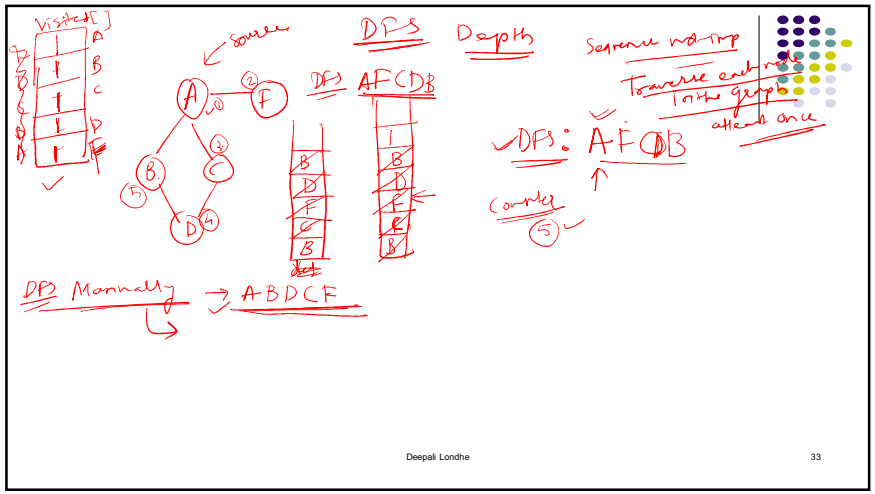
- Problem: Search for a certain node or traverse all nodes in the graph
- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time

31

Applications: Finding a Path

- Find path from source vertex s to destination vertex d
- Use graph search starting at s and terminating as soon as we reach d
  - Need to remember edges traversed
- Use depth – first search ?
- Use breath – first search?

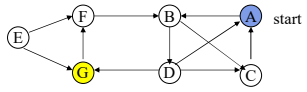
32



33

DFS vs. BFS

DFS Process

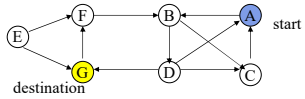


A	DFS on A	
B A	DFS on B	
C B A	DFS on C	
D B A	Call DFS on D Return to call on B	
G D B A	Call DFS on G	found destination - done! Path is implicitly stored in DFS recursion Path is: A, B, D, G

34

DFS vs. BFS

BFS Process



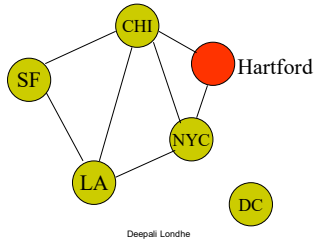
rear front	rear front	rear front	rear front
A	B	D C	D
Initial call to BFS on A Add A to queue	Dequeue A Add B	Dequeue B Add C, D	Dequeue C Nothing to add

rear front	
G	found destination - done! Path must be stored separately
Dequeue D Add G	

35

Graph Traversal

- List out all cities that United Airline can reach from Hartford Airport



36

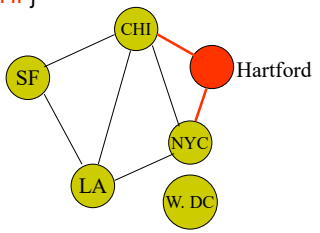
Graph Traversal

- From vertex u, list out all vertices that can be reached in graph G
- Set of nodes to expand
- Each node has a flag to indicate visited or not

37

Traversal Algorithm

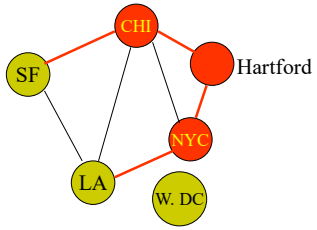
- Step 1: { Hartford }
  - find neighbors of Hartford
  - { Hartford, NYC, CHI }



38

Traversal Algorithm

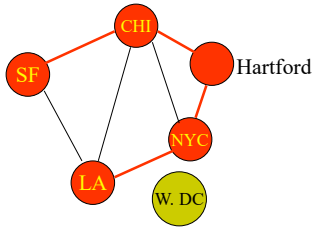
- Step 2: { Hartford, NYC, CHI }
  - find neighbors of NYC, CHI
  - { Hartford, NYC, CHI, LA, SF }



39

Traversal Algorithm

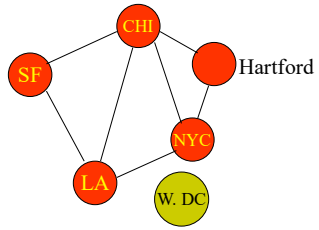
- Step 3: {Hartford, NYC, CHI, LA, SF }
  - find neighbors of LA, SF
  - no other new neighbors



40

Traversal Algorithm

- Finally we get all cities that United Airline can reach from Hartford Airport
  - {Hartford, NYC, CHI, LA, SF }



41

Algorithm of Graph Traversal

1. Mark all nodes as unvisited
2. Pick a starting vertex u, add u to probing list
3. While (probing list is not empty)

{

Remove a node v from probing list

Mark node v as visited

For each neighbor w of v, if w is unvisited, add w to the probing list

}

42

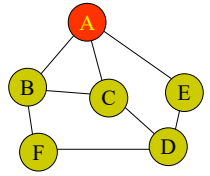
Graph Traversal Algorithms

- Two algorithms
  - Depth First Traversal
  - Breadth First Traversal

43

Depth First Traversal

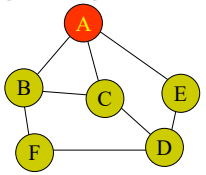
- Probing List is implemented as stack (LIFO)
- Example
  - A's neighbor: B, C, E
  - B's neighbor: A, C, F
  - C's neighbor: A, B, D
  - D's neighbor: E, C, F
  - E's neighbor: A, D
  - F's neighbor: B, D
  - start from vertex A



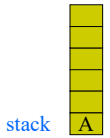
44

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



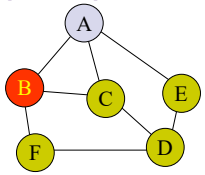
- Initial State
  - Visited Vertices { }
  - Probing Vertices { A }
  - Unvisited Vertices { A, B, C, D, E, F }



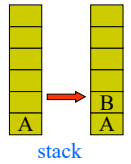
45

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



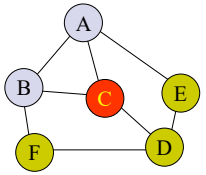
- Peek a vertex from stack, it is A, mark it as visited
- Find A's first unvisited neighbor, push it into stack
  - Visited Vertices { A }
  - Probing vertices { A, B }
  - Unvisited Vertices { B, C, D, E, F }



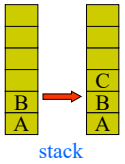
46

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



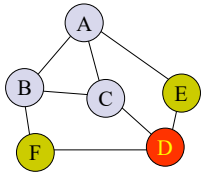
- Peek a vertex from stack, it is B, mark it as visited
- Find B's first unvisited neighbor, push it in stack
  - Visited Vertices { A, B }
  - Probing Vertices { A, B, C }
  - Unvisited Vertices { C, D, E, F }



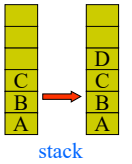
47

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



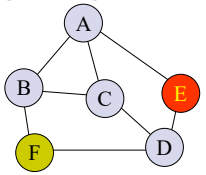
- Peek a vertex from stack, it is C, mark it as visited
- Find C's first unvisited neighbor, push it in stack
  - Visited Vertices { A, B, C }
  - Probing Vertices { A, B, C, D }
  - Unvisited Vertices { D, E, F }



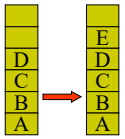
48

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is D, mark it as visited
- Find D's first unvisited neighbor, push it in stack
  - Visited Vertices { A, B, C, D }
  - Probing Vertices { A, B, C, D, E }
  - Unvisited Vertices { E, F }

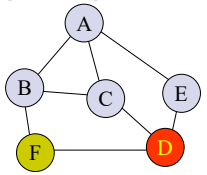


stack

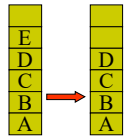
49

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is E, mark it as visited
- Find E's first unvisited neighbor, no vertex found, Pop E
  - Visited Vertices { A, B, C, D, E }
  - Probing Vertices { A, B, C, D }
  - Unvisited Vertices { F }

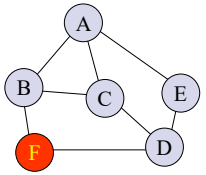


stack

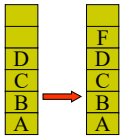
50

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is D, mark it as visited
- Find D's first unvisited neighbor, push it in stack
  - Visited Vertices { A, B, C, D, E }
  - Probing Vertices { A, B, C, D, F}
  - Unvisited Vertices { F }

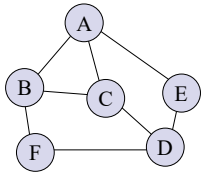


stack

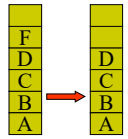
51

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is F, mark it as visited
- Find F's first unvisited neighbor, no vertex found, Pop F
  - Visited Vertices { A, B, C, D, E, F }
  - Probing Vertices { A, B, C, D}
  - Unvisited Vertices { }

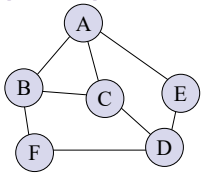


stack

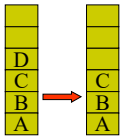
52

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is D, mark it as visited
- Find D's first unvisited neighbor, no vertex found, Pop D
  - Visited Vertices { A, B, C, D, E, F }
  - Probing Vertices { A, B, C }
  - Unvisited Vertices { }

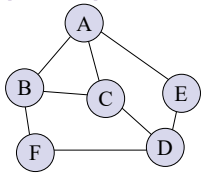


stack

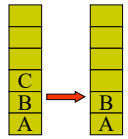
53

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is C, mark it as visited
- Find C's first unvisited neighbor, no vertex found, Pop C
  - Visited Vertices { A, B, C, D, E, F }
  - Probing Vertices { A, B }
  - Unvisited Vertices { }

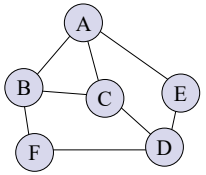


stack

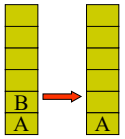
54

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is B, mark it as visited
- Find B's first unvisited neighbor, no vertex found, Pop B
  - Visited Vertices { A, B, C, D, E, F }
  - Probing Vertices { A }
  - Unvisited Vertices { }

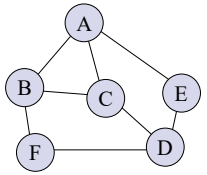


stack

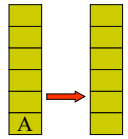
55

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Peek a vertex from stack, it is A, mark it as visited
- Find A's first unvisited neighbor, no vertex found, Pop A
  - Visited Vertices { A, B, C, D, E, F }
  - Probing Vertices { }
  - Unvisited Vertices { }

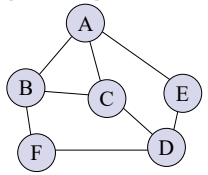


stack

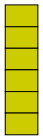
56

Depth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Now probing list is empty
- End of Depth First Traversal
  - Visited Vertices { A, B, C, D, E, F }
  - Probing Vertices { }
  - Unvisited Vertices { }



stack

57

Pseudocode DFS

- Initialize visited array
  for all vertices i till n in G
  make visited[i]=FALSE
- Call DFS(0)

Procedure DFS(int v)

1. Visited [v]=TRUE
2. For each vertex w adjacent to v
   If (!visited[w])
   Call DFS(w)
3. end

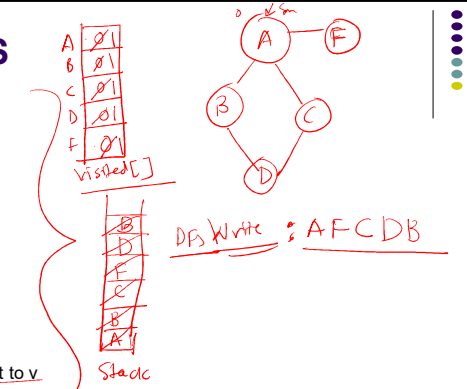
58

Pseudocode DFS

- Initialize visited array
  for all vertices i till n in G
  make visited[i]=FALSE
- Call nr_DFS(0)

Procedure nr_dfs(int v)

1. Push(S,v)
2. Repeat while (stack ! Empty)
   v = Pop(S)
   if(visited[V]==0)
   write (v)
   visited[v]=TRUE;
   For each vertex w adjacent to v
   if (!visited[w])
   Push(S,w)
3. End



DFS Write : A F C D B

59

Pseudocode BFS

- Initialize visited array
  for all vertices i till n in G
  make visited[i]=FALSE
- Call BFS(0)

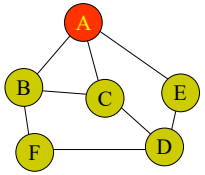
Procedure bfs(int v)

1. enqueue(Q,v)
   visited[v]=TRUE
2. Repeat while (Queue ! Empty)
   v = dequeue(Q)
   For each vertex w adjacent to v
   if (!visited[w])
   enqueue(Q,w)
   visited[w]=TRUE
3. End

60

Breadth First Traversal

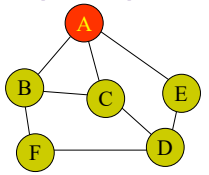
- Probing List is implemented as queue (FIFO)
- Example
  - A's neighbor: B C E
  - B's neighbor: A C F
  - C's neighbor: A B D
  - D's neighbor: E C F
  - E's neighbor: A D
  - F's neighbor: B D
  - start from vertex A



61

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



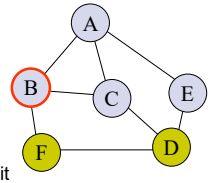
- Initial State
  - Visited Vertices { }
  - Probing Vertices { A }
  - Unvisited Vertices { A, B, C, D, E, F }

queue

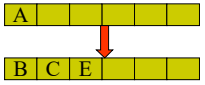
62

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Delete first vertex from queue, it is A, mark it as visited
- Find A's all unvisited neighbors, mark them as visited, put them into queue
  - Visited Vertices { A, B, C, E }
  - Probing Vertices { B, C, E }
  - Unvisited Vertices { D, F }

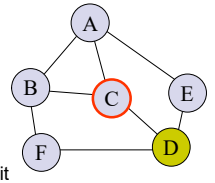


queue

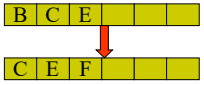
63

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Delete first vertex from queue, it is B, mark it as visited
- Find B's all unvisited neighbors, mark them as visited, put them into queue
  - Visited Vertices { A, B, C, E, F }
  - Probing Vertices { C, E, F }
  - Unvisited Vertices { D }

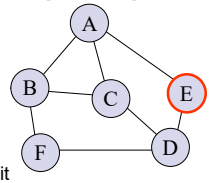


queue

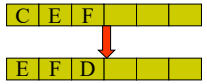
64

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Delete first vertex from queue, it is C, mark it as visited
- Find C's all unvisited neighbors, mark them as visited, put them into queue
  - Visited Vertices { A, B, C, E, F, D }
  - Probing Vertices { E, F, D }
  - Unvisited Vertices { }

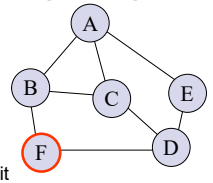


queue

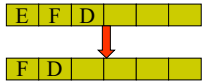
65

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Delete first vertex from queue, it is E, mark it as visited
- Find E's all unvisited neighbors, no vertex found
  - Visited Vertices { A, B, C, E, F, D }
  - Probing Vertices { F, D }
  - Unvisited Vertices { }

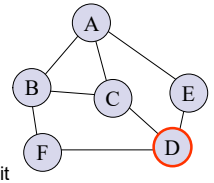


queue

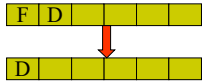
66

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Delete first vertex from queue, it is F, mark it as visited
- Find F's all unvisited neighbors, no vertex found
  - Visited Vertices { A, B, C, E, F, D }
  - Probing Vertices { D }
  - Unvisited Vertices { }

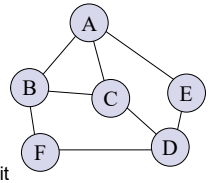


queue

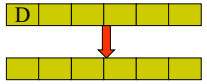
67

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Delete first vertex from queue, it is D, mark it as visited
- Find D's all unvisited neighbors, no vertex found
  - Visited Vertices { A, B, C, E, F, D }
  - Probing Vertices { }
  - Unvisited Vertices { }

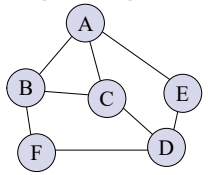


queue

68

Breadth First Traversal (Cont)

- A's neighbor: B C E
- B's neighbor: A C F
- C's neighbor: A B D
- D's neighbor: E C F
- E's neighbor: A D
- F's neighbor: B D



- Now the queue is empty
- End of Breadth First Traversal
  - Visited Vertices { A, B, C, E, F, D }
  - Probing Vertices { }
  - Unvisited Vertices { }

queue

Pseudocode BFS

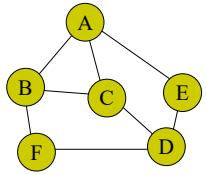
- Initialize visited array
  for all vertices i till n in G
  make visited[i]=FALSE
- Call BFS(0)

Procedure bfs(int v)

1. enqueue(Q,v)
   visited[v]=TRUE
2. Repeat while (Queue ! Empty)
   v = dequeue(Q)
   For each vertex w adjacent to v
   if (!visited[w])
   enqueue(Q,w)
   visited[w]=TRUE
3. End

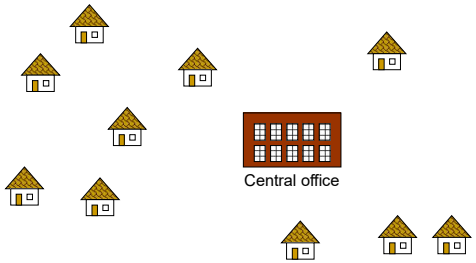
Difference Between DFT & BFT

- Depth First Traversal (DFT)
  - order of visited: A, B, C, D, E, F
- Breadth First Traversal (BFT)
  - order of visited: A, B, C, E, F, D



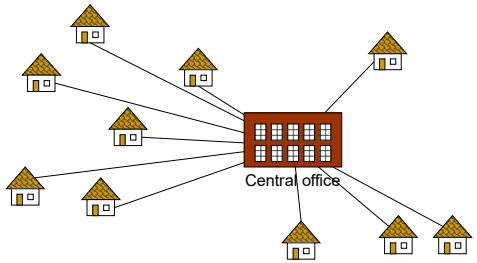
Minimum Spanning Tree

Problem: Laying Telephone Wire



73

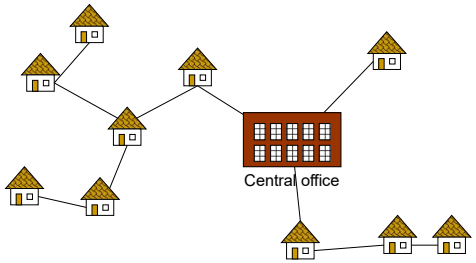
Wiring: Naïve Approach



Expensive!

74

Wiring: Better Approach



Minimize the total length of wire connecting the customers

75

Minimum Spanning Tree (MST)

A **minimum spanning tree** is a subgraph of an undirected weighted graph G, such that

- it is a tree (i.e., it is acyclic)
- it covers all the vertices V
  - contains $|V| - 1$ edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique

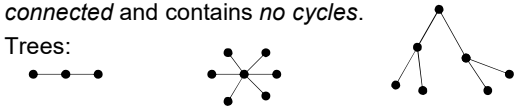
76

Applications of MST

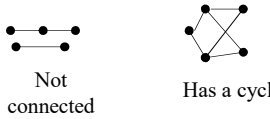
- Any time you want to visit all vertices in a graph at minimum cost (e.g., wire routing on printed circuit boards, sewer pipe layout, road planning…)
- Internet content distribution
  - Idea: publisher produces web pages, content distribution network replicates web pages to many locations so consumers can access at higher speed
  - MST may not be good enough!
    - content distribution on minimum cost tree may take a long time!
- Provides a heuristic for traveling salesman problems.

Tree

- We call an undirected graph a **tree** if the graph is *connected* and contains *no cycles*.
- Trees:



- Not Trees:

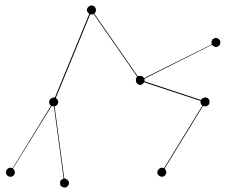


Number of Vertices

- If a graph is a tree, then the number of edges in the graph is one less than the number of vertices.
- **A tree with *n* vertices has *n* – 1 edges.**
  - Each node has one parent except for the root.
    - Note: Any node can be the root here, as we are not dealing with rooted trees.

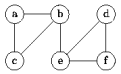
Connected Graph

- A **connected graph** is one in which there is **at least one path** between each pair of vertices.

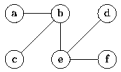


Spanning Tree

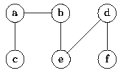
- In a tree there is always **exactly one path** from each vertex in the graph to any other vertex in the graph.
- A **spanning tree** for a graph is a subgraph that includes every vertex of the original, and is a tree.



(a) Graph G



(b) Breadth-first spanning tree of G rooted at b



(c) Depth-first spanning tree of G rooted at c

81

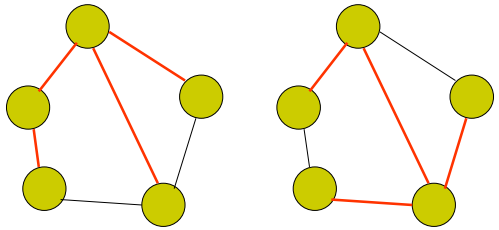
Spanning Tree

- Connected subgraph that includes all vertices of the original connected graph
- Subgraph is a tree
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.
  - No circle in this subgraph

82

Spanning Tree

- Minimum number of edges to keep it connected
- If N vertices, spanning tree has N-1 edges



83

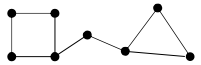
Non-Connected Graphs

- If the graph is not connected, we get a spanning tree for each **connected component** of the graph.
  - That is we get a forest.

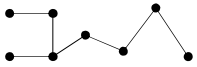
84

Finding a Spanning Tree

Find a spanning tree for the graph below.



We could break the two cycles by removing a single edge from each. One of several possible ways to do this is shown below.



Was breadth-first or depth-first search (or neither) used to create this?

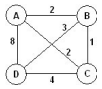
Minimum Spanning Tree

- A spanning tree that has minimum total weight is called a **minimum spanning tree** for the graph.
  - Technically it is a minimum-weight spanning tree.
- If all edges have the same weight, breadth-first search or depth-first search will yield minimum spanning trees.
  - For the rest of this discussion, we assume the edges have weights associated with them.

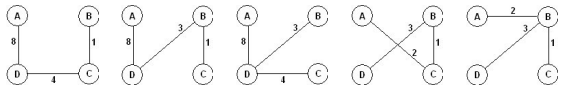
Note, we are strictly dealing with undirected graphs here, for directed graphs we would want to find the optimum branching or arborescence of the directed graph.

Minimum Spanning Tree

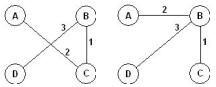
- Consider this graph.



- It has 16 spanning trees. Some are:

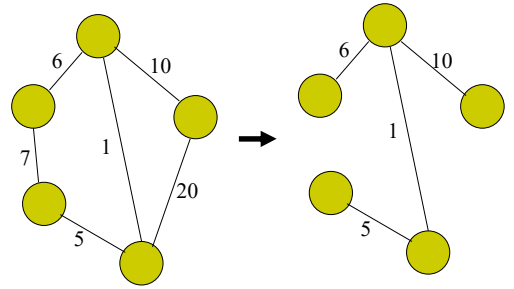


- There are two minimum-cost spanning trees, each with a cost of 6:



Minimum Spanning Tree (MST)

Spanning tree with minimum weight



Minimum Spanning Tree

- There are many approaches to computing a minimum spanning tree. We could try to detect cycles and remove edges, but the two algorithms we will study build them from the bottom-up in a *greedy* fashion.

1. **Prim's Algorithm – *starts with a single vertex*** and then adds the minimum edge to extend the spanning tree.
2. **Kruskal's Algorithm – *starts with a forest of single node trees*** and then adds the edge with the minimum weight to connect two components.

Prim's Algorithm

- Prim's algorithm finds a minimum cost spanning tree by selecting edges from the graph one-by-one as follows:

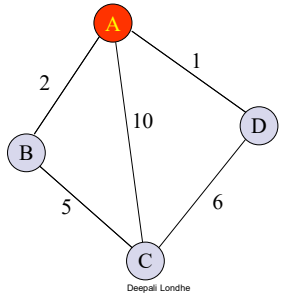
1. It starts with a tree, T, consisting of a single starting vertex, x.
2. Then, it finds the shortest edge emanating from x that connects T to the rest of the graph (i.e., a vertex not in the tree T).
3. It adds this edge and the new vertex to the tree T.
4. It then picks the shortest edge emanating from the revised tree T that also connects T to the rest of the graph and repeats the process.

Prim's Algorithm For Finding MST

```
// Assume that G has atleast one vertex.
TV={0};// start with vertex 0 and no edge selected
For(T=ø; T contains fewer than n-1 edges; add(u,v) to T)
{
      Let (u,v) be the least cost edge such that u Є TV and v ∉ TV;
       if ( there is no such edge ) break;
      Add v to TV
}
if (T contains fewer than n-1 edges) then print spanning tree
```

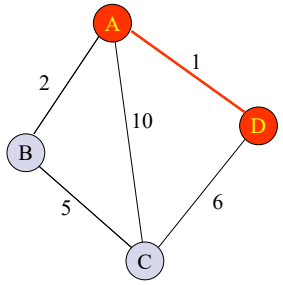

Demos For Finding MST

- Step 1: mark vertex A as selected



Demos For Finding MST

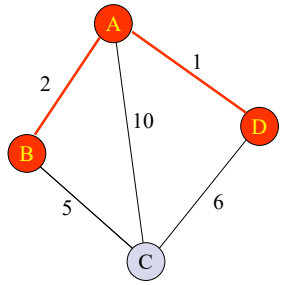
- Step 2: find the minimum weighted edge connected to vertex A, and mark the other vertex on this edge as selected.



93

Demos For Finding MST

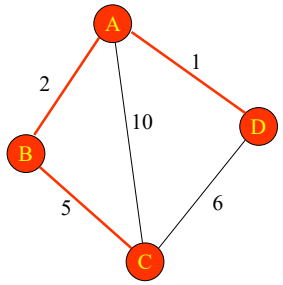
- Step 3: find the minimum weighted edge connected to vertices set { A, D } , and mark the other vertex on this edge as selected.



94

Demos For Finding MST

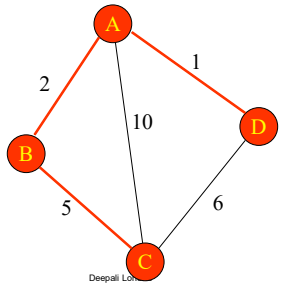
- Step 4: find the minimum weighted edge connected to vertices set { A, D, B} , and mark the other vertex on this edge as selected.



95

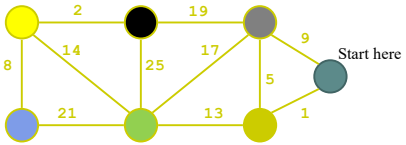
Demos For Finding MST

- Step 5: All vertex are marked as selected, So we find the minimum spanning tree



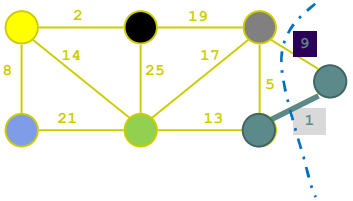
96

Prim's Algorithm



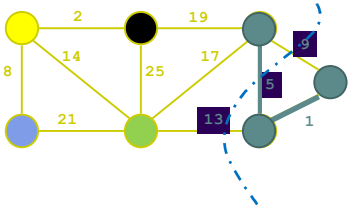
97

Prim's Algorithm



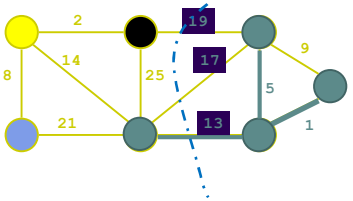
98

Prim's Algorithm



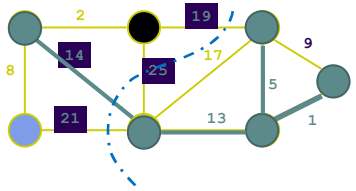
99

Prim's Algorithm



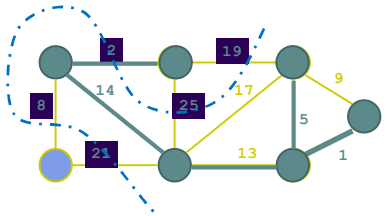
100

Prim's Algorithm



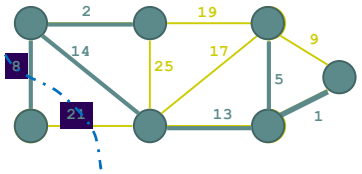
101

Prim's Algorithm



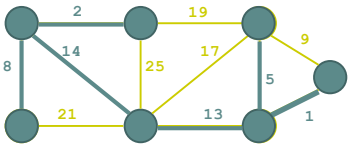
102

Prim's Algorithm



103

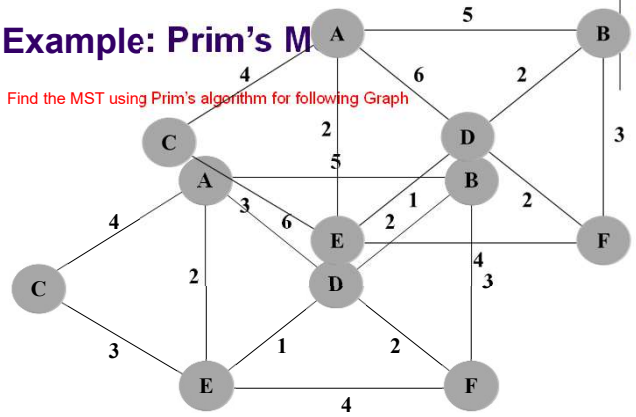
Prim's Algorithm



104

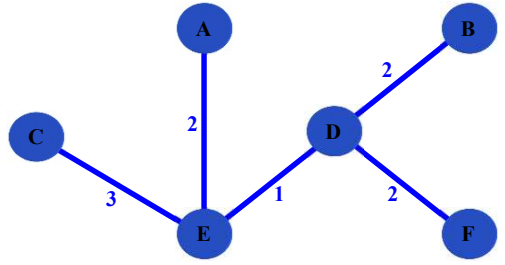
Example: Prim's M

Find the MST using Prim's algorithm for following Graph



105

minimum- spanning tree



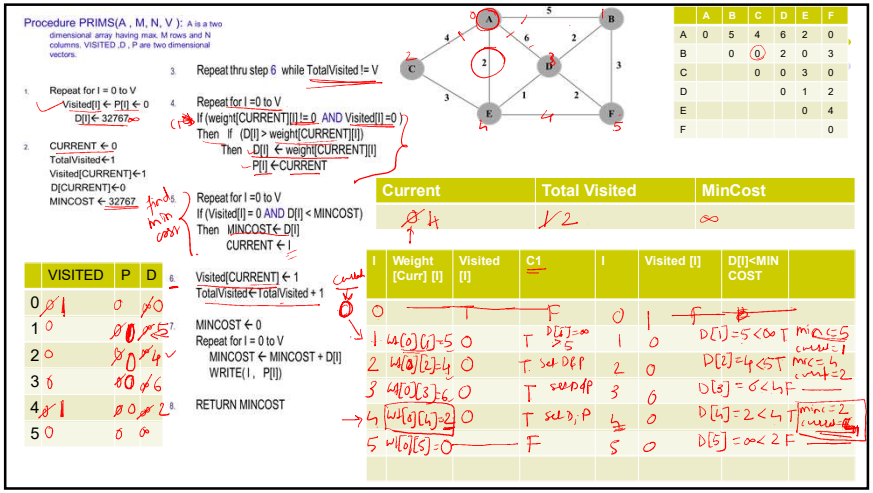
106

Prims Algorithm With adjacency Matrix

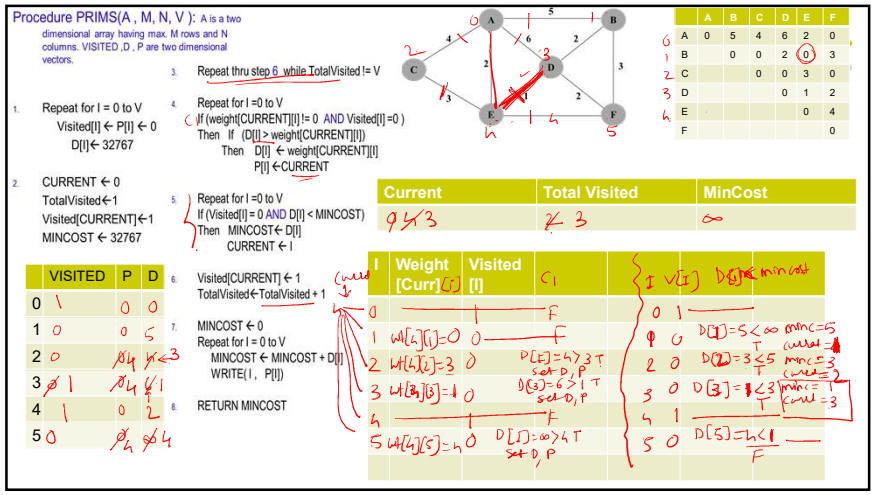
Procedure PRIMS(A , M, N, V): A is a two dimensional array having max. M rows and N columns. VISITED ,D , P are two dimensional vectors.

1. Repeat for I = 0 to V
   Visited[I] ← P[I] ← 0
   D[I]← 32767
2. CURRENT ← 0
   TotalVisited←1
   Visited[CURRENT]←1
   D[CURRENT]←0
   MINCOST ← 32767
3. Repeat thru step 6 while TotalVisited != V
4. Repeat for I =0 to V
   If (weight[CURRENT][I] !=0 AND Visited[I] =0)
   Then If (D[I] > weight[CURRENT][I])
   Then D[I] ← weight[CURRENT][I]
   P[I] ←CURRENT
5. Repeat for I =0 to V
   If (Visited[I] = 0 AND D[I] < MINCOST)
   Then MINCOST← D[I]
   CURRENT ← I
6. Visited[CURRENT] ← 1
   TotalVisited←TotalVisited + 1
7. MINCOST ← 0
   Repeat for I = 0 to V
   MINCOST ← MINCOST + D[I]
   WRITE(I , P[I])
8. RETURN MINCOST

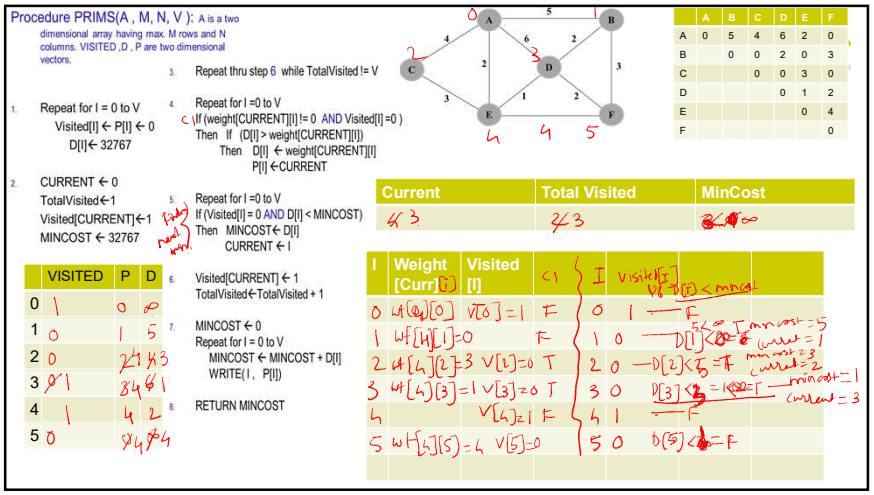
107



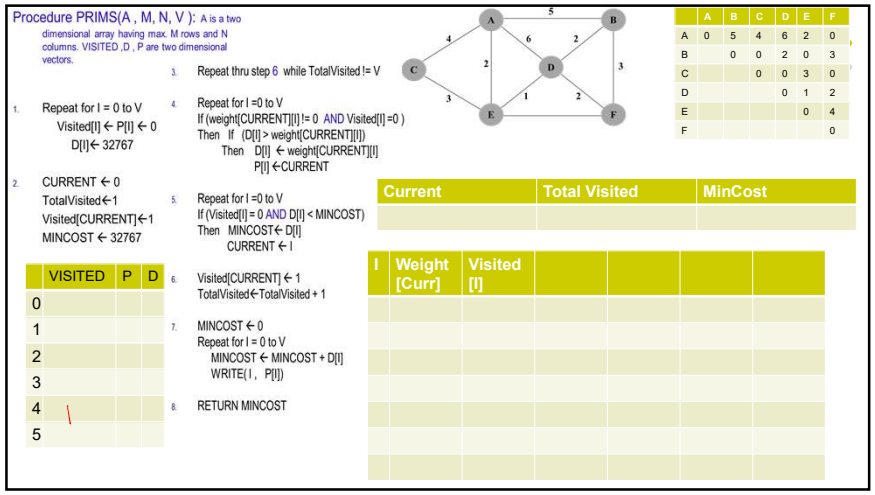
108



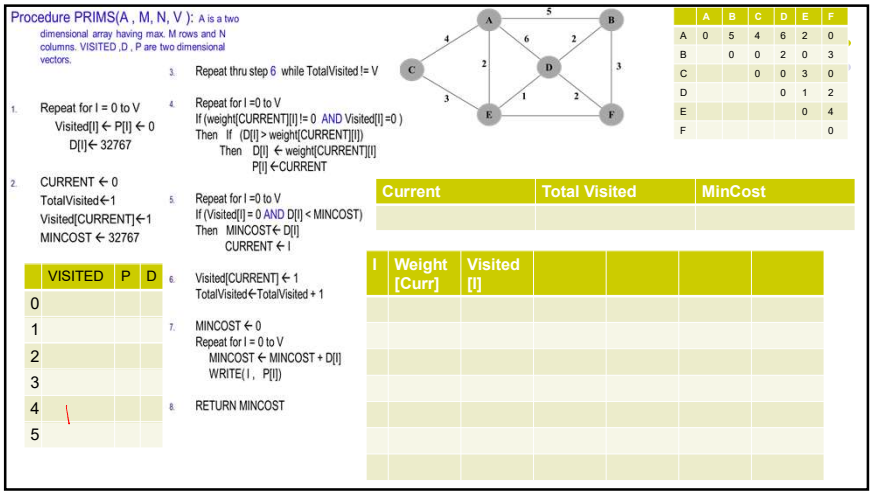
109



110



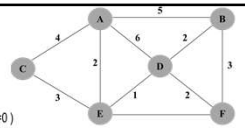
111



112

Procedure PRIMS(A , M, N, V): A is a two dimensional array having max. M rows and N columns. VISITED ,D , P are two dimensional vectors.

1. Repeat for I = 0 to V
   Visited[I] ← P[I] ← 0
   D[I]← 32767
2. CURRENT ← 0
   TotalVisited←1
   Visited[CURRENT]←1
   MINCOST ← 32767
3. Repeat thru step 6 while TotalVisited != V
4. Repeat for I =0 to V
   If (weight[CURRENT][I] != 0 AND Visited[I] =0)
   Then If (D[I] > weight[CURRENT][I])
   Then D[I] ← weight[CURRENT][I]
   P[I] ←CURRENT
5. Repeat for I =0 to V
   If (Visited[I] = 0 AND D[I] < MINCOST)
   Then MINCOST← D[I]
   CURRENT ← I
6. Visited[CURRENT] ← 1
   TotalVisited←TotalVisited + 1
7. MINCOST ← 0
   Repeat for I = 0 to V
   MINCOST ← MINCOST + D[I]
   WRITE(I , P[I])
8. RETURN MINCOST



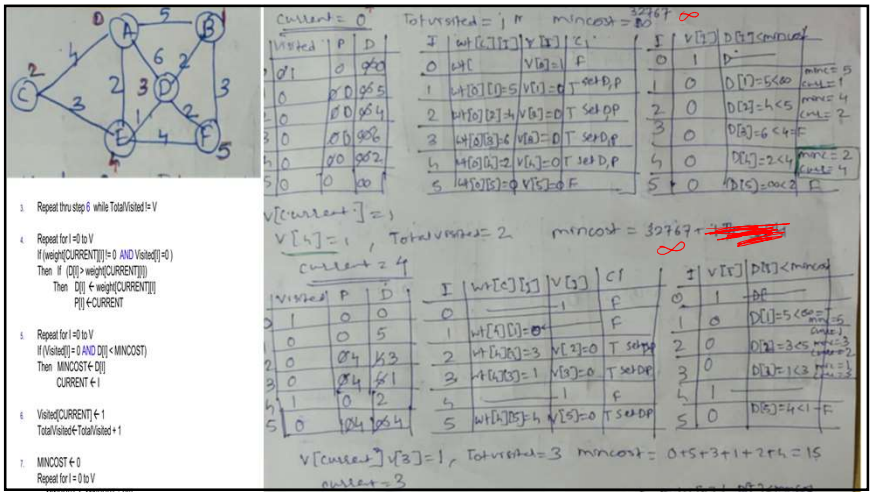
	A	B	C	D	E	F
A	0	5	4	6	2	0
B		0	0	2	0	3
C			0	0	3	0
D				0	1	2
E					0	4
F						0

Current	Total Visited	MinCost

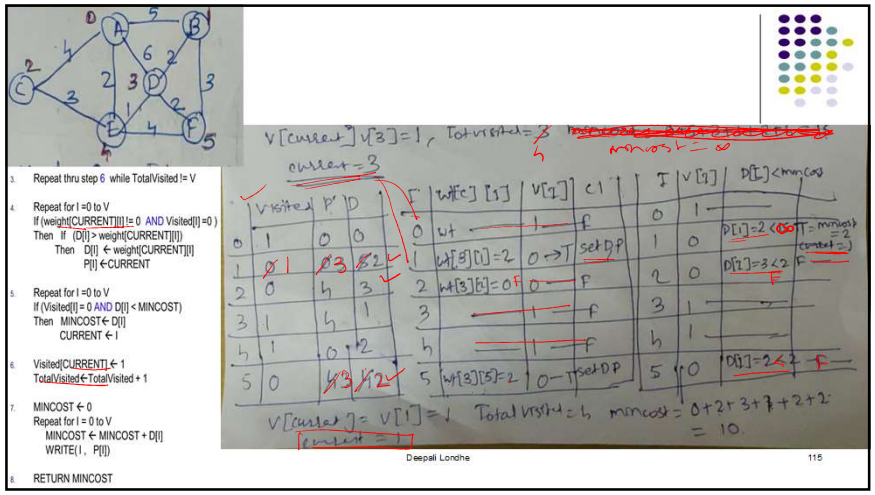
	VISITED	P	D
0			
1			
2			
3			
4			
5			

I	Weight [Curr]	Visited [I]				

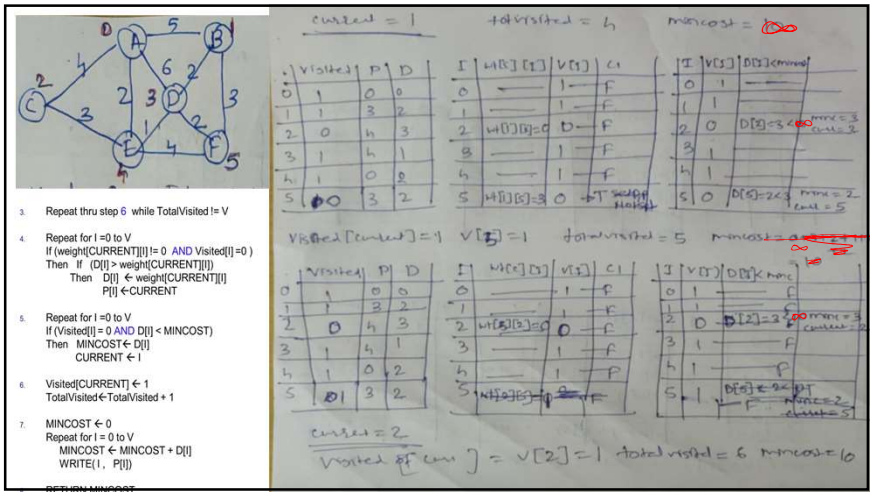
113



114



115



116

Kruskals's Algorithm

```
T= ø;
While((T contains less than n-1 edges ) && (E not empty))
{
  choose an edge (v,w) from E of lowest cost;
  delete (v,w) from E;
  if ((v,w) does not create cycle in T) add (v,w) to T;
  else discard (v,w);
}
if (T contains fewer than n-1 edges) then print spanning tree
```

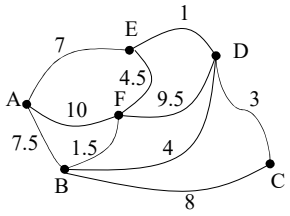

Kruskal's Algorithm

- Greedy algorithm to choose the edges as follows.

Step 1	**First edge: choose any edge with the minimum weight.**
Step 2	**Next edge: choose any edge with minimum weight from *those not yet selected.* (The subgraph can look disconnected at this stage.)**
Step 3	**Continue to choose edges of minimum weight from those not yet selected, except *do not select any edge that creates a cycle* in the subgraph.**
Step 4	**Repeat step 3 until the subgraph connects all vertices of the original graph.**

Kruskal's Algorithm

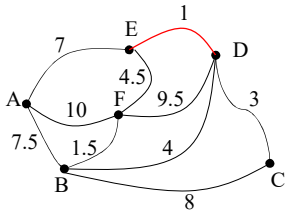
Use Kruskal's algorithm to find a minimum spanning tree for the graph.



Kruskal's Algorithm

Solution

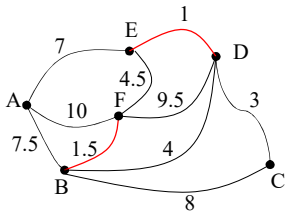
First, choose ED (the smallest weight).



Kruskal's Algorithm

Solution

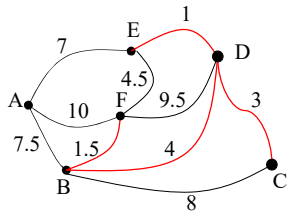
Now choose BF (the smallest remaining weight).



Kruskal's Algorithm

Solution

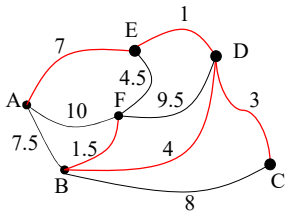
Now CD and then BD.



Kruskal's Algorithm

Solution

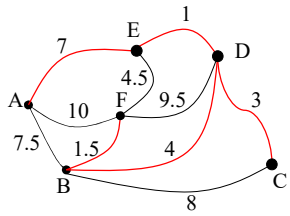
Note EF is the smallest remaining, but that would create a cycle. Choose AE and we are done.



Kruskal's Algorithm

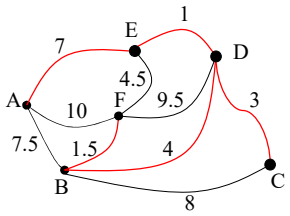
Solution

The total weight of the tree is 16.5.



Kruskal's Algorithm

- Some questions:
  1. How do we know we are finished?
  2. How do we check for cycles?

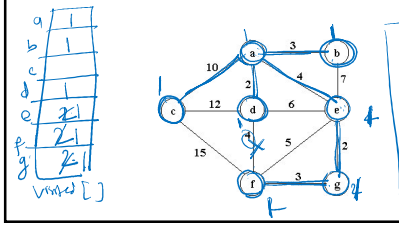


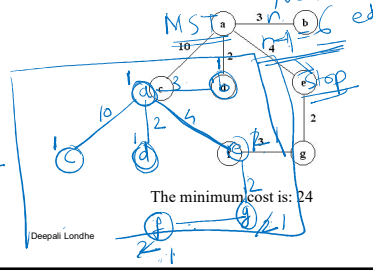
125

Kruskal's Algorithm

edge	ad	eg	ab	fg	ae	df	ef	de	be	ac	cd	cf
weight	2	2	3	3	4	4	5	6	7	10	12	15
insertion status	√	√	√	√	√	x	x	x	x	√	x	x
insertion order	1	2	3	4	5					6		

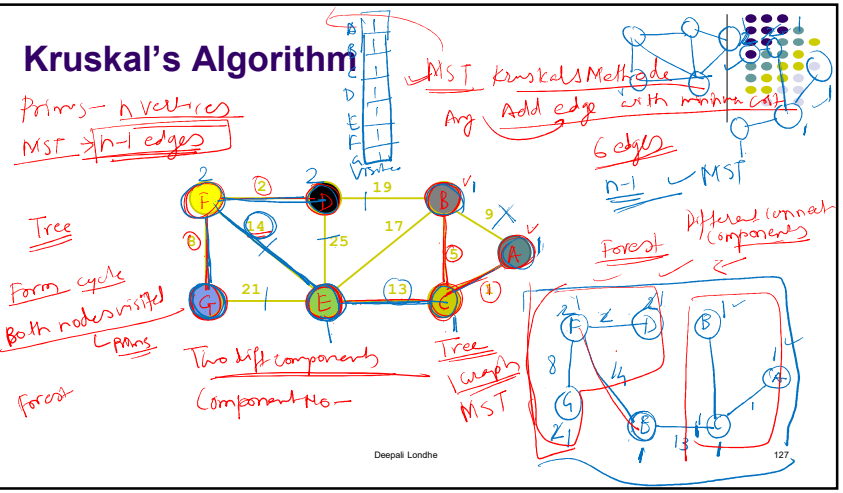
- Trace of Kruskal's algorithm for the undirected, weighted graph:





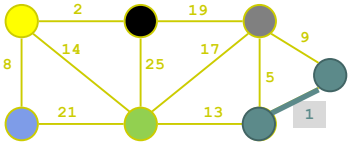
126

Kruskal's Algorithm



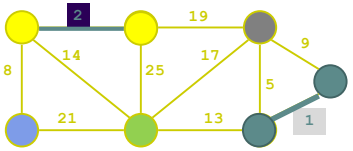
127

Kruskal's Algorithm



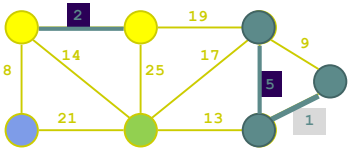
128

Kruskal's Algorithm



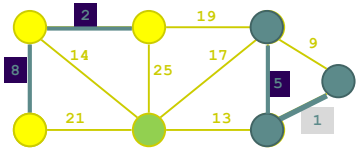
129

Kruskal's Algorithm



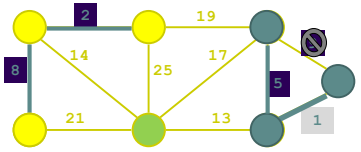
130

Kruskal's Algorithm



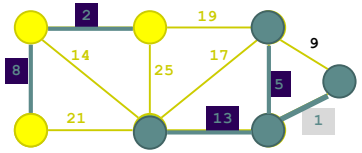
131

Kruskal's Algorithm



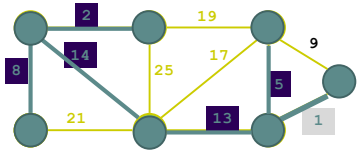
132

Kruskal's Algorithm



133

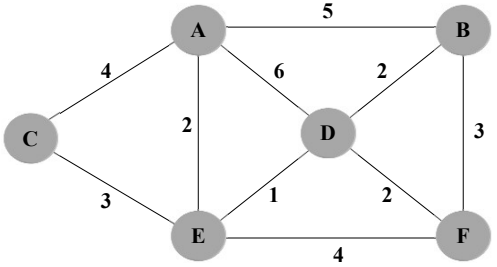
Kruskal's Algorithm



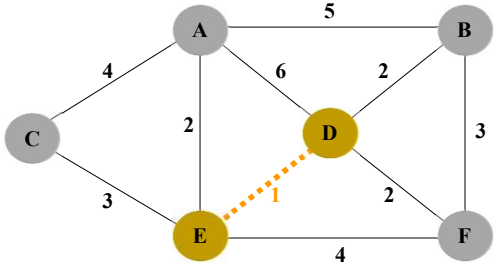
134

Example: Kruskal's Algorithm

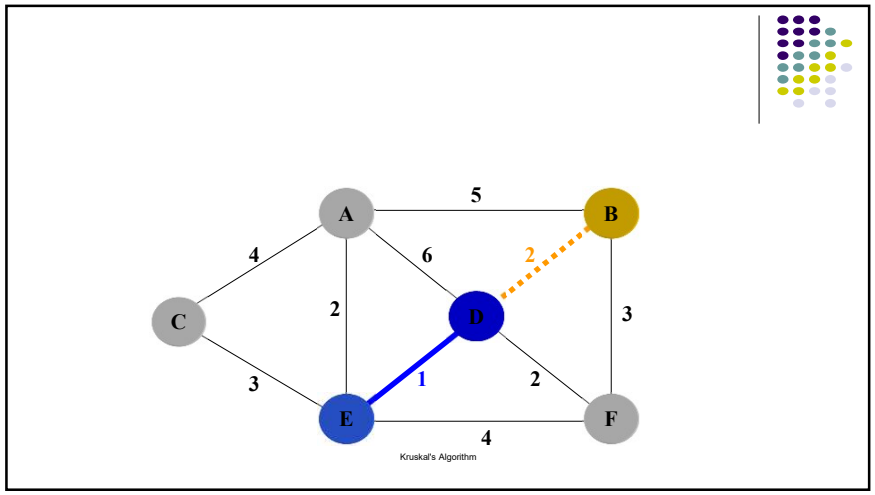
Find MST using Kruskal's algorithm for the following graph.



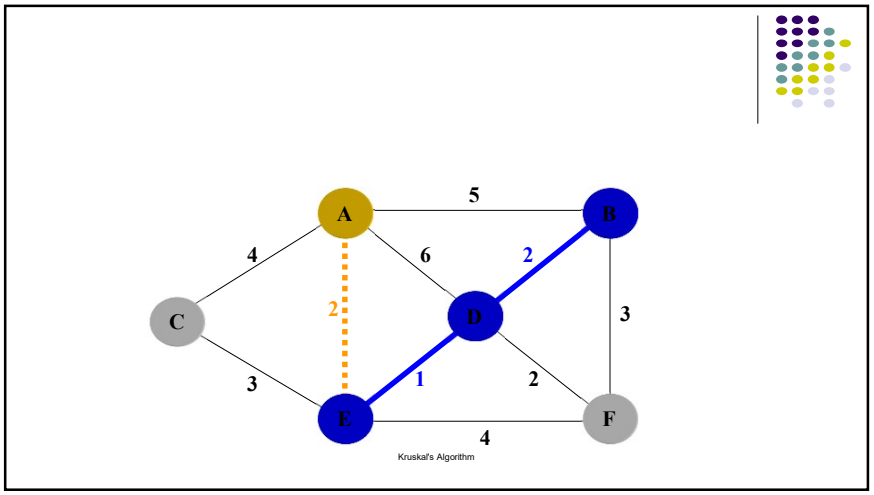
135



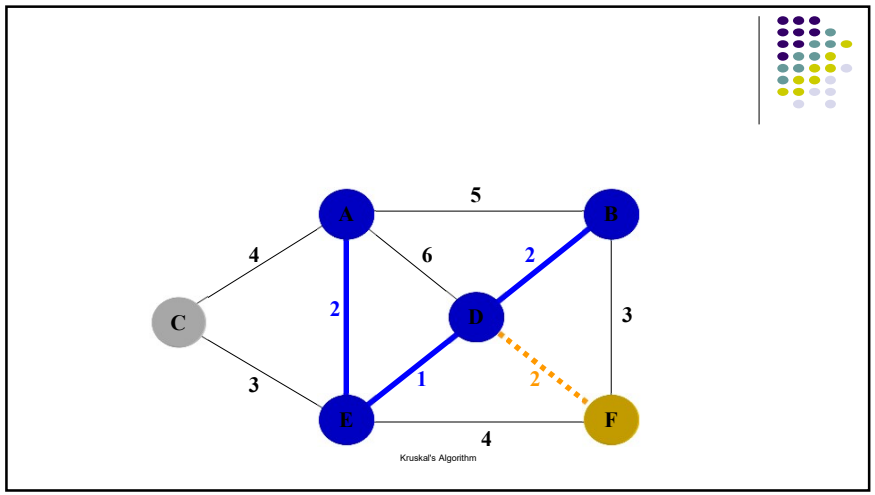
136



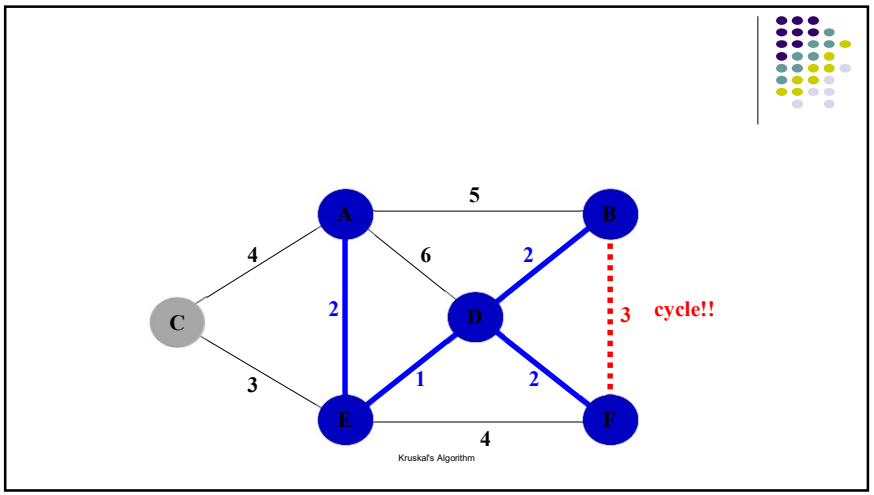
137



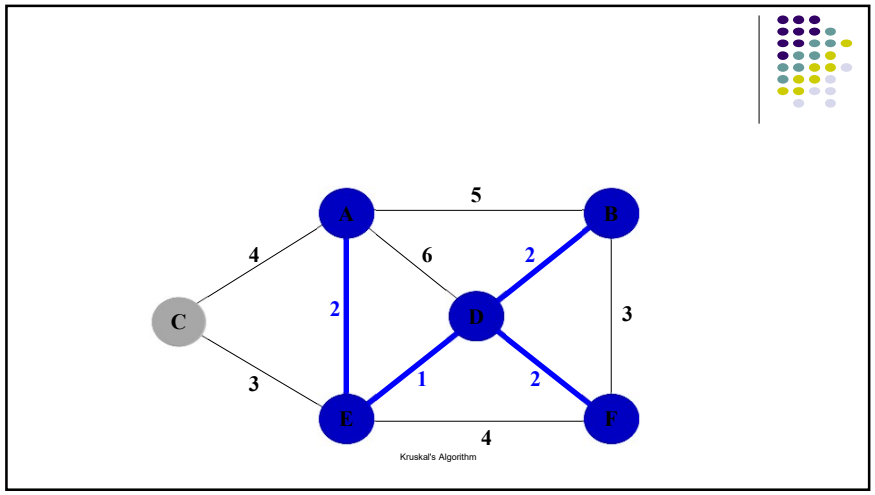
138



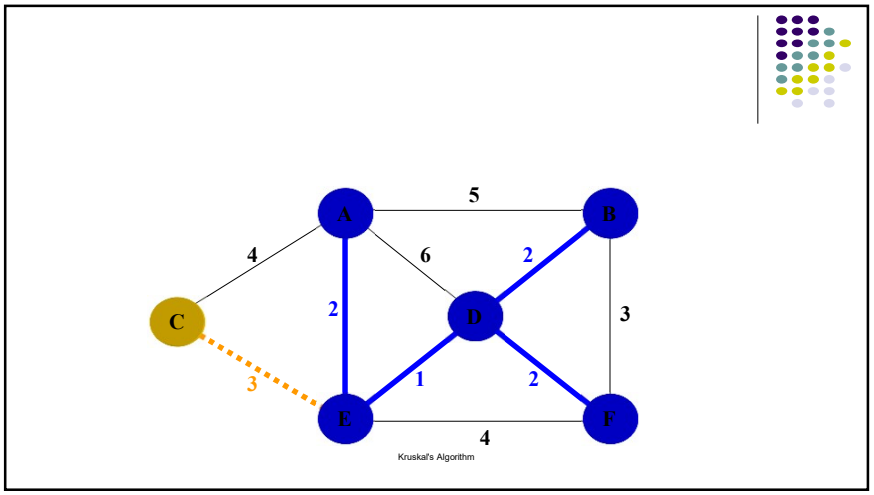
139



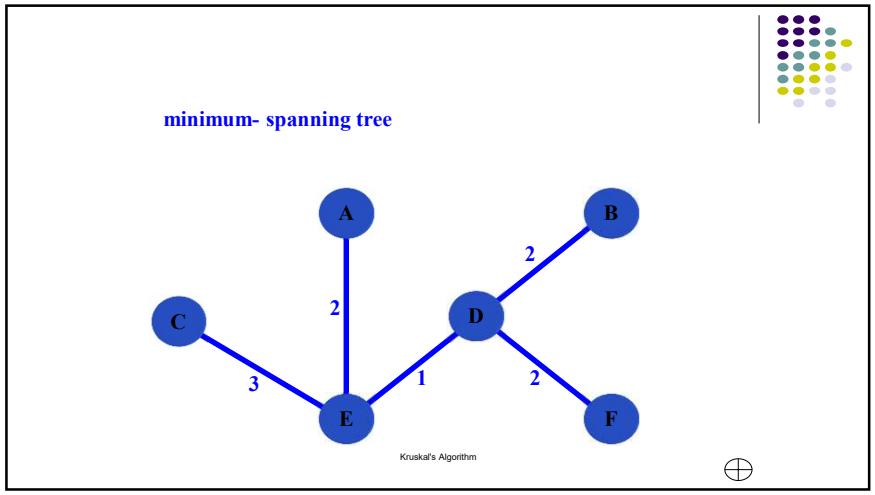
140



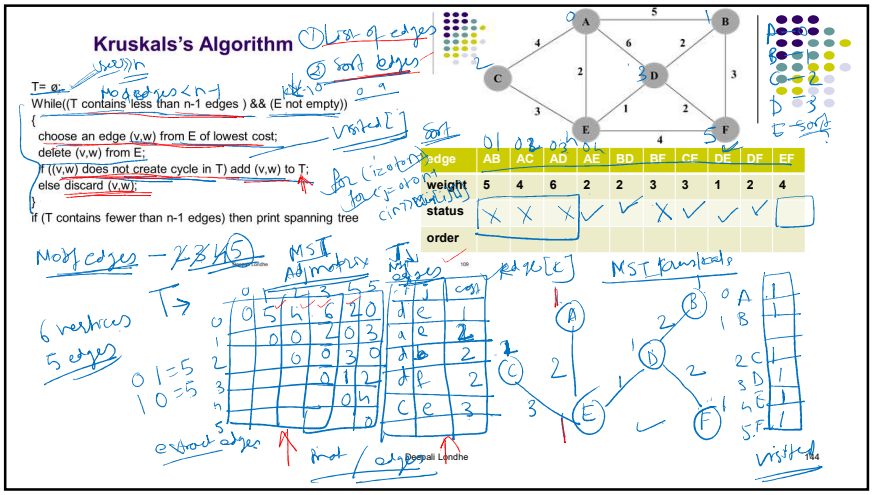
141



142

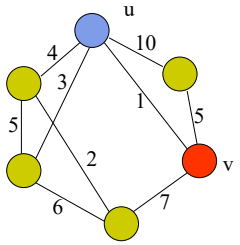


143



144

Shortest Path Problem



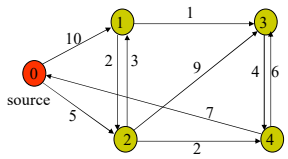
- Weight: cost, distance, travel time, hop …

Single Source Shortest Path Problem

- Single source shortest path problem
  - Find the shortest path to all other nodes
- Dijkstra's shortest path algorithm
  - Find shortest path greedily by updating distance to all other nodes

Example – Dijkstra Algorithm

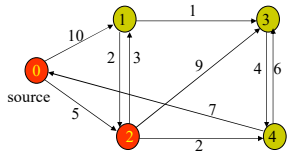
- Greedy Algorithm
- Assume all weight of edge >0



node	from node V_0 to other nodes			
V_1	10			
V_2	5			
V_3	∞			
V_4	∞			
best				

Example – Dijkstra Algorithm

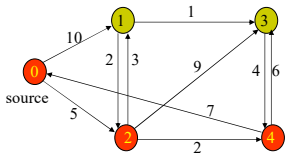
- step 1: find the shortest path to node 0
  - node 2 is selected



node	from node V_0 to other nodes			
V_1	10			
V_2	5			
V_3	∞			
V_4	∞			
best	V_2			

Example – Dijkstra Algorithm

- step 2: recalculate the path to all other nodes
  - find the shortest path to node 0. Node 4 is selected

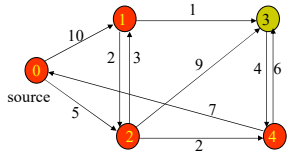


node	from node V_0 to other nodes			
V_1	10	8		
V_2	5	5		
V_3	∞	14		
V_4	∞	7		
best	V_2	V_4		

149

Example – Dijkstra Algorithm

- step 3: recalculate the path to all other nodes
  - find the shortest path to node 0. node 1 is selected

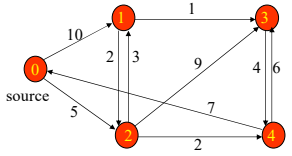


node	from node V_0 to other nodes			
V_1	10	8	8	
V_2	5	5	5	
V_3	∞	14	13	
V_4	∞	7	7	
best	V_2	V_4	V_1	

150

Example – Dijkstra Algorithm

- step 3: recalculate the path to all other nodes
  - find the shortest path to node 0. node 2 is selected

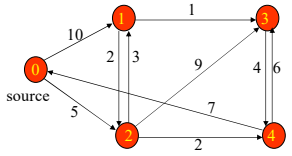


node	from node V_0 to other nodes			
V_1	10	8	8	8
V_2	5	5	5	5
V_3	∞	14	13	9
V_4	∞	7	7	7
best	V_2	V_4	V_1	V_3

151

Example – Dijkstra Algorithm

- Now we get all shortest paths to each node



node	from node V_0 to other nodes			
V_1	10	8 (0,2)	8 (0,2)	8
V_2	5 (0,2)	5	5	5
V_3	∞	14 (0,2,3)	13 (0,2,4,3)	9 (0,2,1,3)
V_4	∞	7 (0,2,4)	7	7
best	V_2	V_4	V_1	V_3

152

Dijkstra Algorithm

Mark source node selected
Initialize all distances to Infinite, source node distance to 0.
Make source node the current node.
While (there is unselected node)
{
    Expand on current node
    Update distance for neighbors of current node
    Find an unselected node with smallest distance,
    and make it current node and mark this node selected
}

Pseudo-code For Dijkstra's Algorithm

Procedure Dijkstra(G, s)

1. Repeat for I = 0 to V
   Visited[I] ← 0
   P[I]← s
   D[I]← 32767

2. CURRENT ← s
   Visited[CURRENT]←1
   D[S] ← 0
3. Repeat thru step 6 for I= 1 to V-2

4. Min=32767
   Repeat for J =0 to V
   If (Visited[J] = 0 AND D[J] < MIN)
   Then MIN← D[J]
   CURRENT ← J

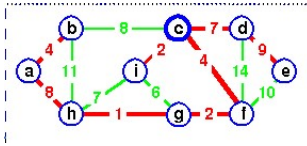
5. Visited[CURRENT] ← 1

6. Repeat for k=0 to V
   res=(min+weight[CURRENT][k]);
   if(visited[k]==0 AND res<d[k])
   then d[k]=res;
   p[k]=CURRENT;

7. [Shortest distance from source s]
   Repeat for I = 0 to V
   if(I!=S)
   WRITE(I , D[I])
   WRITE(Path = I,)
   j=I
   do
   j=P[J]
   WRITE('<-', J)
   while(j!=S)

Graphs - Definitions

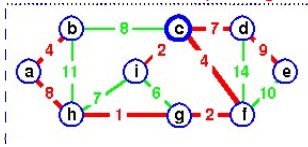
- Spanning Tree
  - A spanning tree is a set of $|V|-1$ edges that connect all the vertices of a graph



The red path connects all vertices, so it's a spanning tree

Graphs - Definitions

- Minimum Spanning Tree
  - Generally there is more than one spanning tree
  - If a cost c_{ij} is associated with edge $e_{ij} = (v_i, v_j)$ then the minimum spanning tree is the set of edges E_{span}



Other ST's can be formed ..
- **Replace 2 with 7**
- **Replace 4 with 11**

The red tree is the Min ST

157

Graphs - Kruskal's Algorithm

- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we've constructed a spanning tree
    - *Extract cheapest edge*
    - If it forms a cycle, ignore it
      else add it to the forest of trees
      (it will join two trees into a larger tree)
  - Return the spanning tree

158

Graphs - Kruskal's Algorithm

- Calculate the minimum spanning tree
  - Put all the vertices into single node trees by themselves
  - Put all the edges in a priority queue
  - Repeat until we've constructed a spanning tree
    - *Extract cheapest edge*

Note that this algorithm makes no attempt
- **to be clever**
- **to make any sophisticated choice of the next edge**
- **it just tries the cheapest one!**

159

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
           e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

Initial Forest: single vertex trees

P Queue of edges

160

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
           e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

We need n-1 edges to fully connect (span) n vertices

Graphs - Kruskal's Algorithm in C

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
           e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

Try the cheapest edge

Until we find one that doesn't form a cycle

... and add it to the forest

Kruskal's Algorithm

- Priority Queue
  - We already know about this!!

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
           e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

Add to a heap here

Extract from a heap here

Kruskal's Algorithm

- Cycle detection

```
Forest MinimumSpanningTree( Graph g, int n,
                            double **costs ) {
    Forest T;
    Queue q;
    Edge e;
    T = ConsForest( g );
    q = ConsEdgeQueue( g, costs );
    for(i=0;i<(n-1);i++) {
        do {
           e = ExtractCheapestEdge( q );
        } while ( !Cycle( e, T ) );
        AddEdge( T, e );
    }
    return T;
}
```

But how do we detect a cycle?

Kruskal's Algorithm

- Cycle detection
  - Uses a Union-find structure
  - For which we need to understand a partition of a set
- Partition
  - A set of sets of elements of a set
    - Every element belongs
    - No element belongs to more than one sub-set
  - Formally:
    - Set,
    - Partition(S) = $\{ P_i \}$, where $P_i = \{ s_i \}$

P_i are subsets of S

All s_i belong to one of the P_j

None of the P_i have common elements

S is the union of all the P_i

Kruskal's Algorithm

- Partition
  - The elements of each set of partition
    - are related by an equivalence
    - equivalence relations are
      - reflexive
      - transitive
      - symmetric
  - The sets of a partition are equivalence classes
    - Each element of the set is related to every other element

x ~ x

if x ~ y and y ~ z, then x ~ z

if x ~ y, then y ~ x

Kruskal's Algorithm

- Partitions
  - In the MST algorithm, the connected vertices form equivalence classes
    - "Being connected" is the equivalence relation
  - Initially, each vertex is in a class by itself
  - As edges are added, more vertices become related and the equivalence classes grow
  - Until finally all the vertices are in a single equivalence class

Kruskal's Algorithm

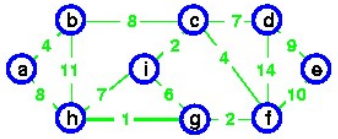
- Representatives
  - One vertex in each class may be chosen as the representative of that class
  - We arrange the vertices in lists that lead to the representative
    - This is the union-find structure
- Cycle determination

Kruskal's Algorithm

- Cycle determination
  - If two vertices have the same representative, they're already connected and adding a further connection between them is pointless
  - Procedure:
    - For each end-point of the edge that you're going to add
    - follow the lists and find its representative
    - if the two representatives are equal, then the edge will form a cycle

Kruskal's Algorithm in operation

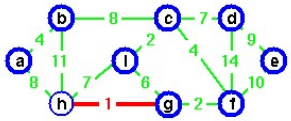
All the vertices are in single element trees



Each vertex is its own representative

Kruskal's Algorithm in operation

All the vertices are in single element trees

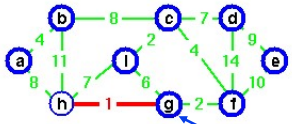


The cheapest edge is h-g

Add it to the forest, joining h and g into a 2-element tree

Kruskal's Algorithm in operation

The cheapest edge is h-g



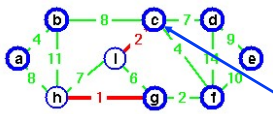
Add it to the forest, joining h and g into a 2-element tree

Choose g as its representative

Kruskal's [illegible]peration

The next cheapest edge is c-i

Add it to the forest, joining c and i into a 2-element tree



Choose c as its representative

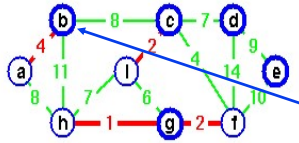
Our forest now has 2 two-element trees and 5 single vertex ones

173

Kruskal's [illegible]peration

The next cheapest edge is a-b

Add it to the forest, joining a and b into a 2-element tree



Choose b as its representative

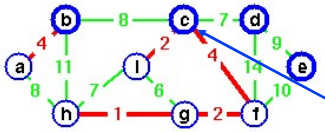
Our forest now has 3 two-element trees and 4 single vertex ones

174

Kruskal's [illegible]peration

The next cheapest edge is c-f

Add it to the forest, *merging* two 2-element trees



Choose the rep of one as its representative

175

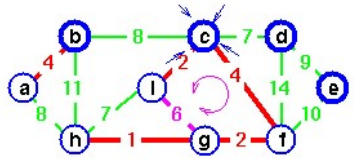
Kruskal's Algorithm in operation

The next cheapest edge is g-i

The rep of g is c

The rep of i is also c

∴ g-i forms a cycle

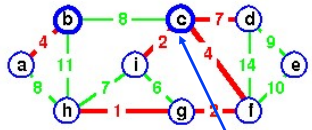


It's clearly not needed!

176

Kruskal's Algorithm in operation

The next cheapest edge is c-d



The rep of c is c

The rep of d is d

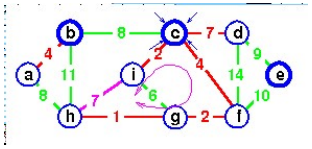
∴ c-d joins two trees, so we add it

.. and keep c as the representative

177

Kruskal's Algorithm in operation

The next cheapest edge is h-i



The rep of h is c

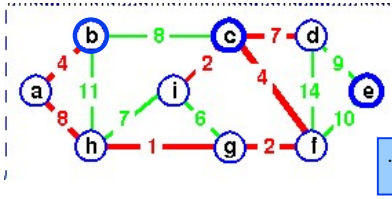
The rep of i is c

∴ h-i forms a cycle, so we skip it

178

Kruskal's Algorithm in operation

The next cheapest edge is a-h



The rep of a is b

The rep of h is c

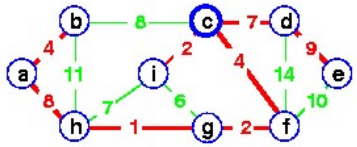
∴ a-h joins two trees, and we add it

179

Kruskal's Algorithm in operation

The next cheapest edge is b-c

But b-c forms a cycle



So add d-e instead

... and we now have a spanning tree

180

Greedy Algorithms

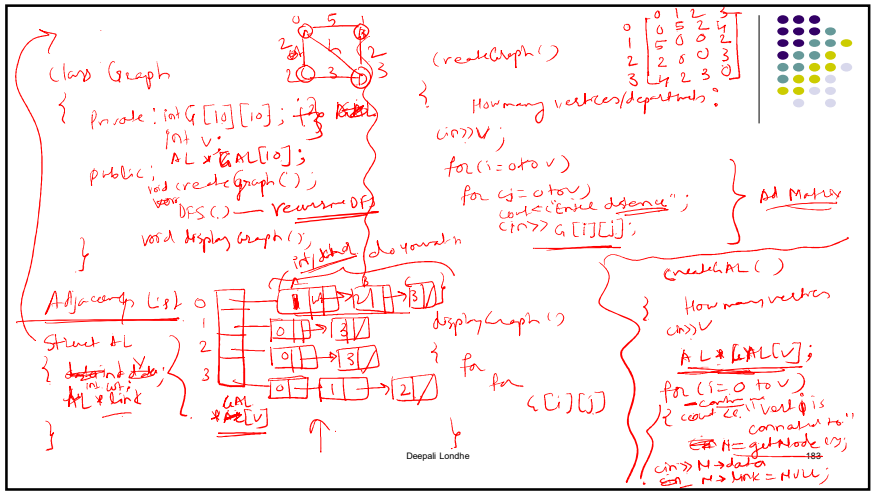
- *At no stage did we attempt to "look ahead"*
- We simply made the naïve choice
  - Choose the cheapest edge!
- MST is an example of a greedy algorithm
- Greedy algorithms
  - Take the "best" choice at each step
  - Don't look ahead and try alternatives
  - *Don't work in many situations*
    - Try playing chess with a greedy approach!
  - *Are often difficult to prove*

181

Kruskal's Algorithm – Time complexity

- Steps
  - Initialize forest $O(|V|)$
  - Sort edges $O(|E|\log|E|)$
    - Check edge for cycles $O(|V|)$ x
    - Number of edges $O(|V|)$ $O(|V|^2)$
  - Total $O(|V|+|E|\log|E|+|V|^2)$
  - Since $|E| = O(|V|^2)$ $O(|V|^2\log|V|)$
- Thus we would class MST as $O(n^2 \log n)$ for a graph with n vertices
- This is an *upper bound*, some improvements on this are known.

182



183