Chapter 4: Graph Algorithms

4.1 Introduction to Graphs and Terminology

4.2 Depth First Search Tree and
Breadth First Search Tree algorithms

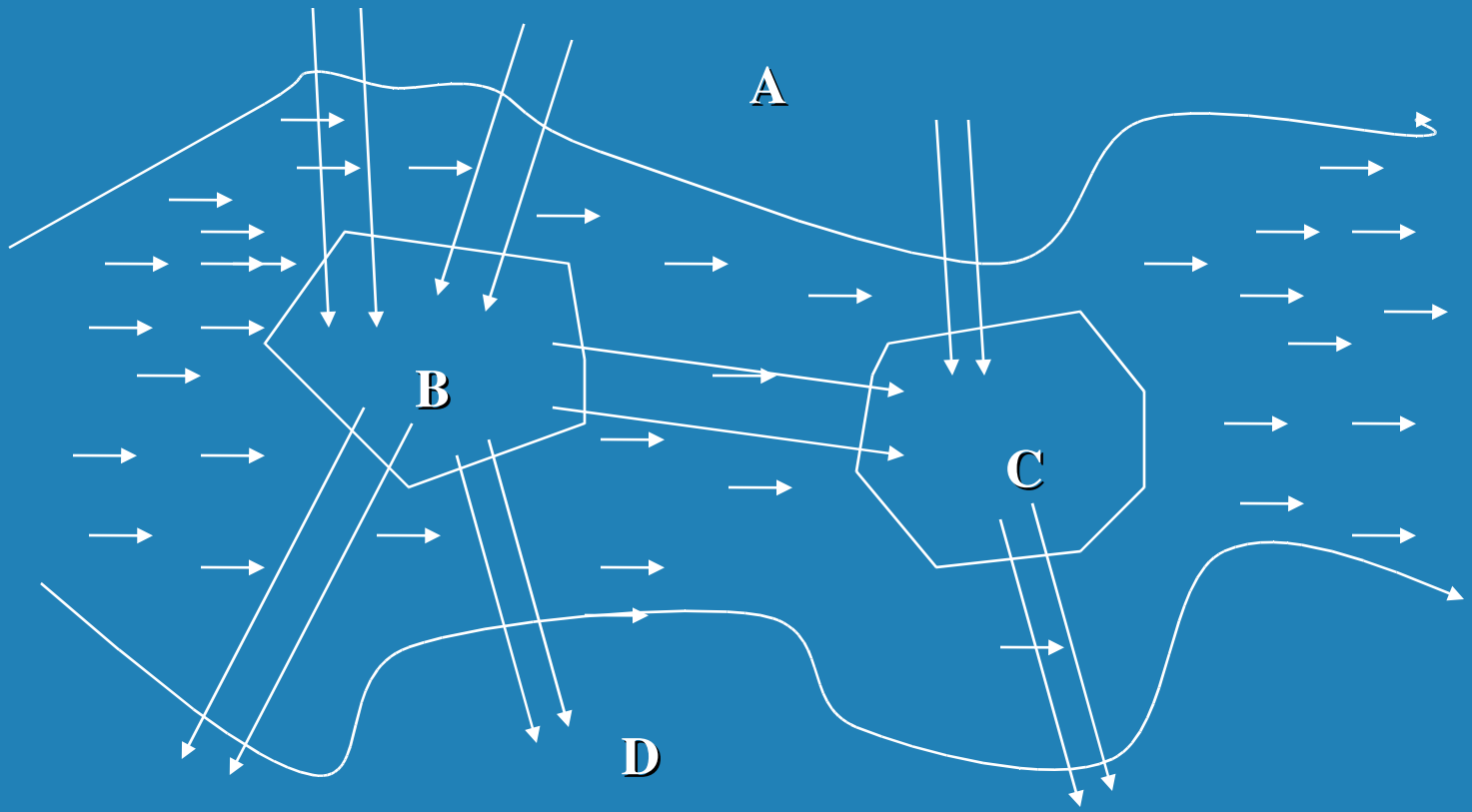
4.3 Weighted Graphs
Minimum Spanning Tree Algorithms
Prim-Dijkstra and Kruskal

4.4 Directed Graphs
Transitive closure- Warshall's algorithm

4.5 Weighted Directed Graphs
All pairs shortest path – Floyd

4.6 Directed Acyclic Graphs
Topological Sorting

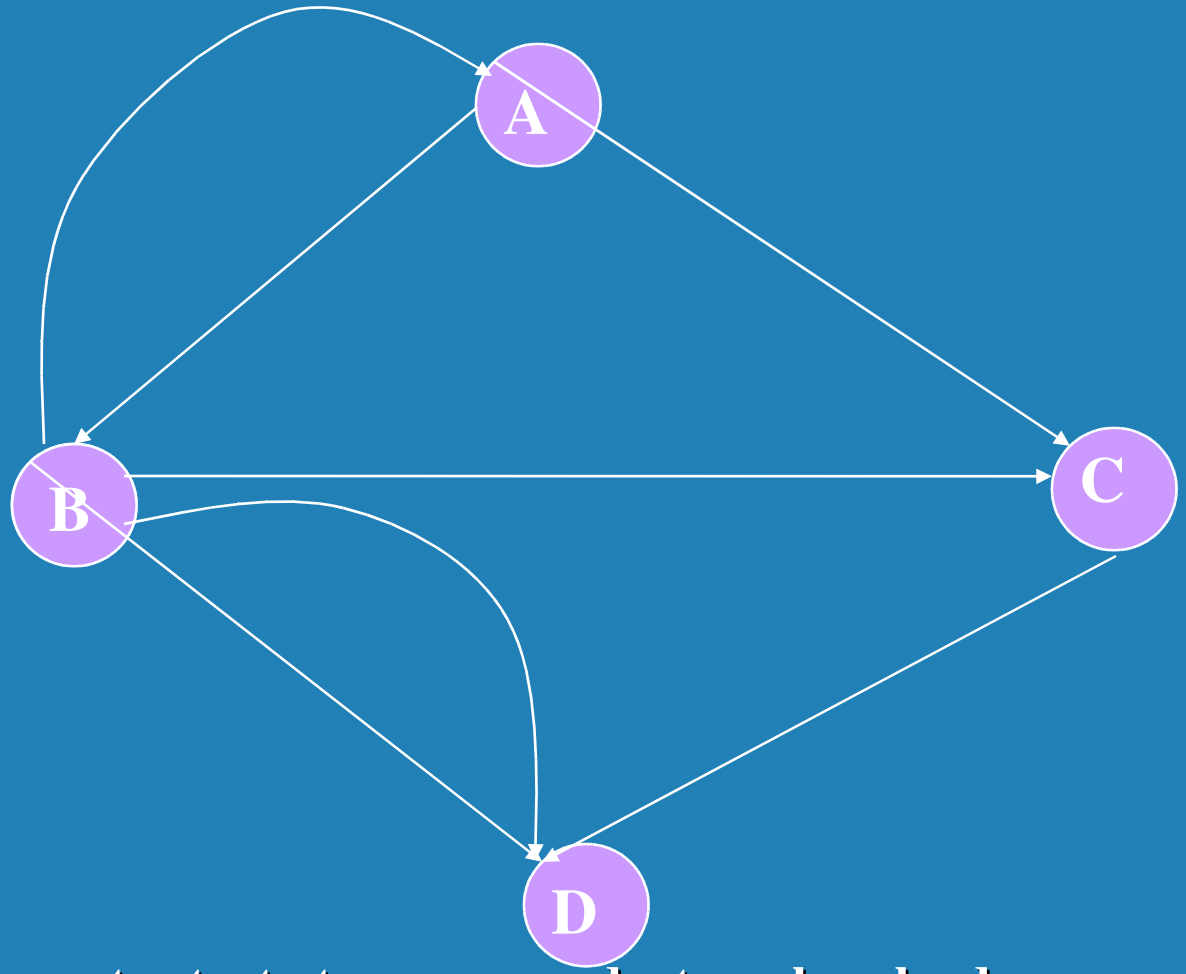
4.1 Introduction to Graphs



Konisberg Bridge Problem : 1736

Is there a way to start at any one city, travel each bridge once, and get back to the same city?

4.1 Introduction to Graphs

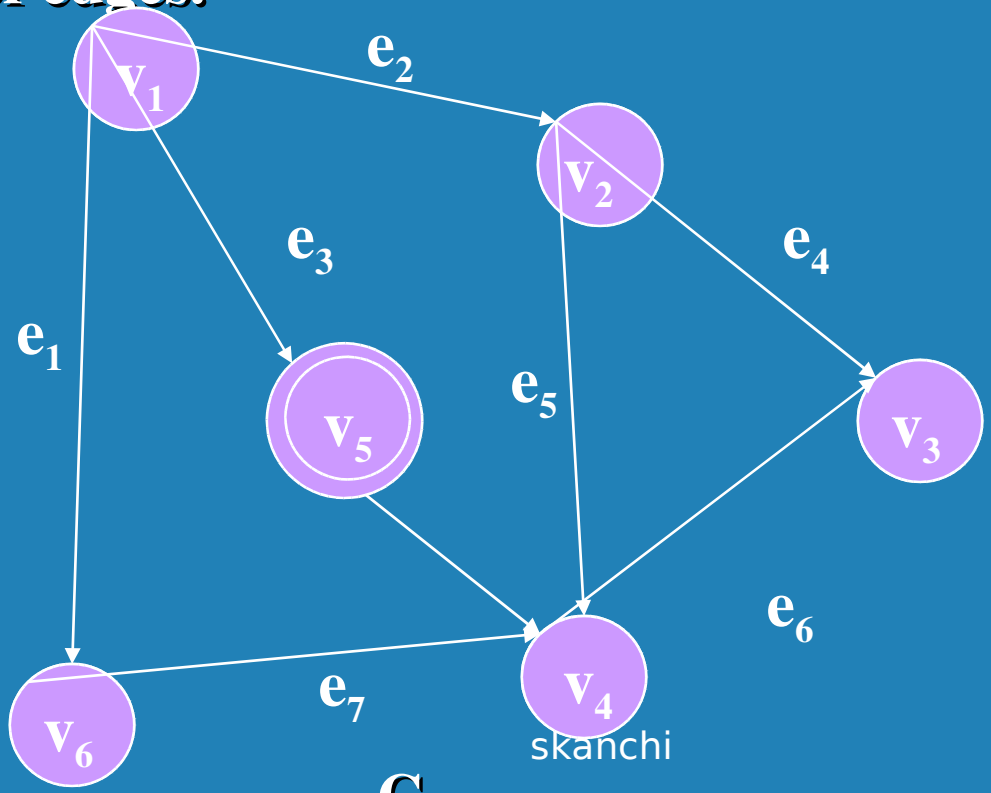


Is there a way to start at any one node, travel each edge once, and get back to the same node?

4.1 Introduction to Graphs

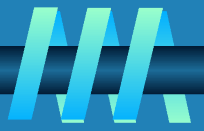
A graph contains vertices denoted by the set V, and edges connecting pairs of vertices. There can be at most one edge between any two vertices.

A graph is written as G = (V,E) where V is the set of vertices and E is the set of edges.



G

4.1 Introduction to Graphs



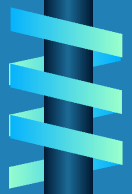
Vertex: is sometimes called nodes or points.

Edges: lines or connections. Edges can be named explicitly as done above, or edge could be written (v_i, v_j) where v_i and v_j are end points of the edge.

Adjacent vertices: two vertices are adjacent if there is an edge between them. In the graph above, v_2 and v_3 are adjacent.

Adjacent edges: two edges are adjacent if they have an endpoint in common. In the graph above, e_5 and e_6 are adjacent.

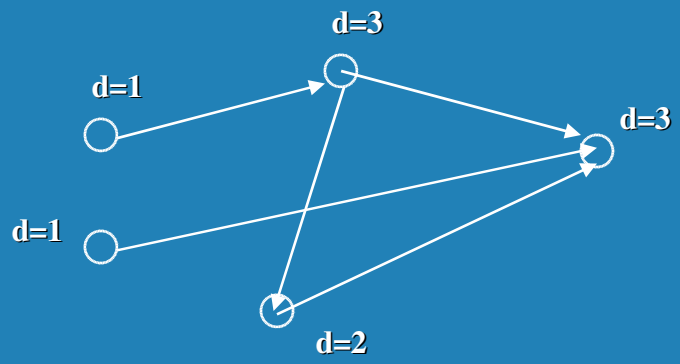
Degree of a vertex: number of edges incident to it. The degree of v_2 is 3, or $d(v_2) = 3$.



4.1 Introduction to Graphs

The size of the set V is denoted by m, and size of the set E denoted by m.

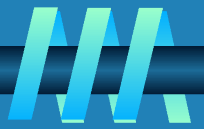
For a given graph, is there any relation between the number of edges and number of vertices?



Sum of all the degrees = 10, m = 5

Theorem:
The sum of the all the degrees in the graph is equal to twice the number of edges in the graph.

4.1 Introduction to Graphs



Paths: sequence of vertices such that every pair of adjacent vertices in the sequence are also adjacent vertices in the graph. The beginning vertex of the path is called the source and the last vertex is called the destination of the path.

Example: P = (v_2 , v_3 , v_4 , v_2 , v_1 ,v_5) is a path in G above

Circuit: is a path in which the beginning and ending vertex are same.

Example: C = (v_2 , v_3 , v_4 , v_2 , v_1 ,v_5 ,v_4 ,v_2) is a circuit in G above

Simple Path: is a path in which no vertex or edge is repeated.

Example: P = (v_2 , v_3 , v_4) is a simple path in G above

Cycle : is a simple path except beginning and ending vertex are same.

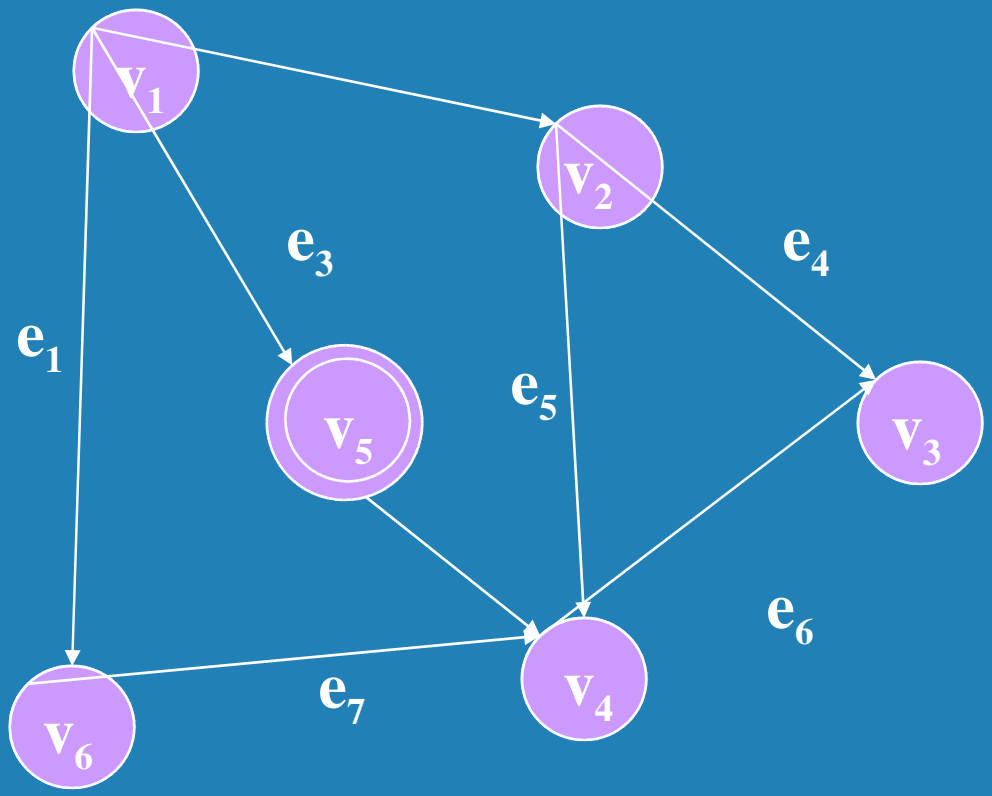
Example: C = (v_2 , v_3 , v_4 , v_2) is a cycle in G above.

Note: Whenever there is a path between two vertices, there is also a simple path between the two vertices.

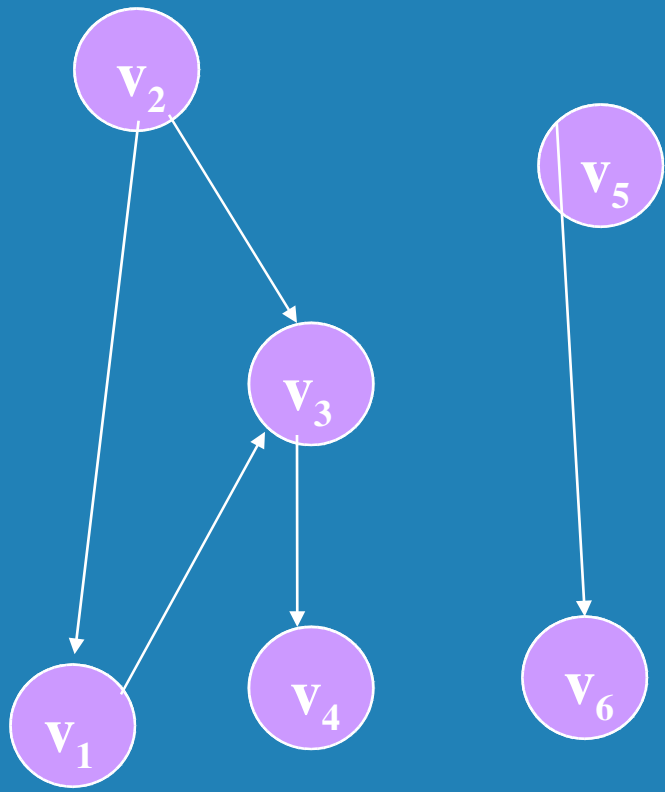
4.1 Introduction to Graphs

Connected Graph: there is a simple path between any two vertices.

Disconnected Graph: A graph that is not connected.



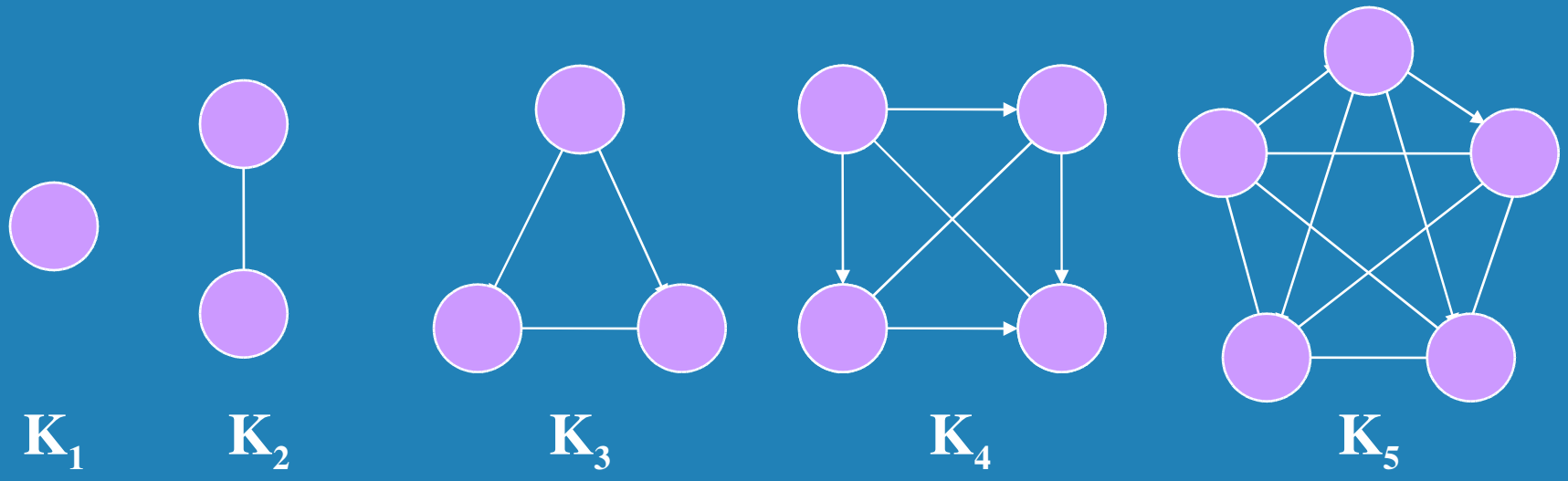
Connected Graph with n=6



Disconnected Graph with n=6

4.1 Introduction to Graphs

A complete graph on n vertices is a graph that contains every possible edge among n vertices. A complete graph on n vertices is denoted by K_n



Number of Edges in K_5 = 4 + 4 + 4 + 4 + 4 = 20/2 = 10 edges

4.1 Introduction to Graphs

How many edges does a complete graph on n vertices K_n, for any n have?

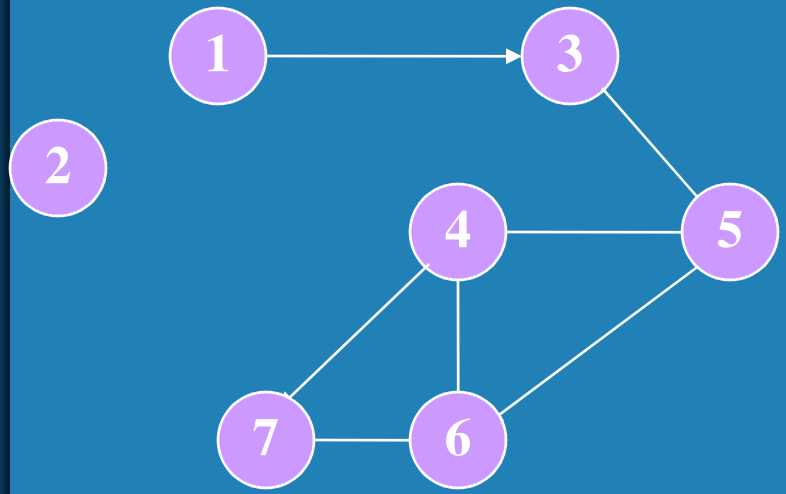
Since the degree of each vertex is n-1 and there are n vertices, the sum of all the degrees is n*(n-1).

Using a theorem above, the number of edges is half of the sum of the degrees. That is,

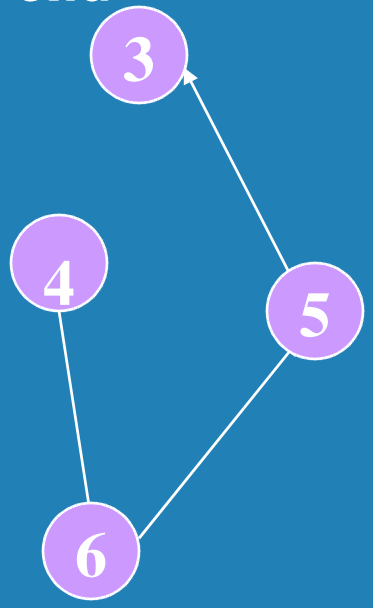
$$m = \frac{n(n-1)}{2}$$

4.1 Introduction to Graphs

Subgraph: of a graph G =(V,E) is a graph G′ = (V′,E′) such that V′ ⊆ V and E′ ⊆ E and E′ is chosen such that every edge in E′ has its end points in V′



A graph G

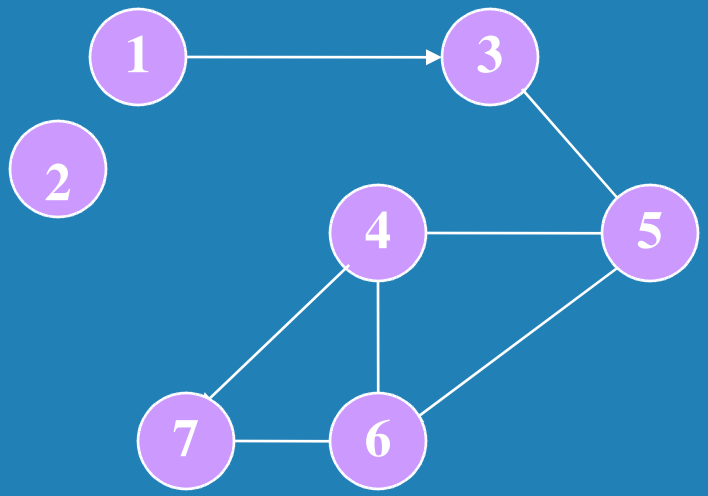


A subgraph of G

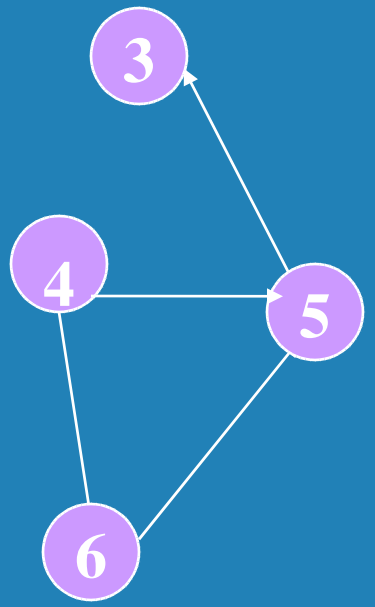
To construct a subgraph, we first select a subset of vertices V′ and then we select some edges from G whose end points are in V′. Note that we have not selected the edge (4,5) to be part of the subgraph.

4.1 Introduction to Graphs

Induced Subgraph: of a graph G =(V,E) is a graph G′ = (V′,E′) such that V′ ⊆ V and E′ is the collection of all edges in E whose end points are in V′.The subgraph is said to be induced by V′



A graph G



Subgraph induced by {3,4,5,6}

Question: How many induced subgraphs are there for a graph G and a given vertex set V′ ⊆ V?

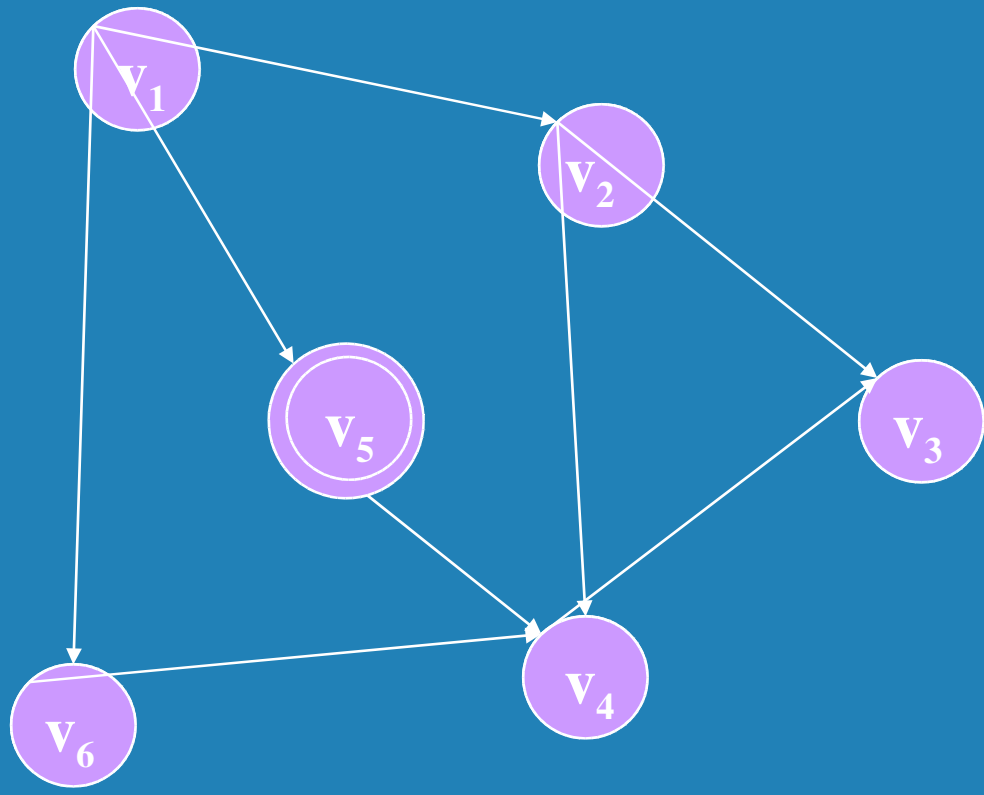
Question: What do the induced subgraphs of K_n look like?

4.1 Representation of Graphs

Adjacency matrix representation is one in which we save a matrix of size n×n in which the $(i,j)^{th}$ entry is 1 if there an edge between v_i and v_j, otherwise $(i,j)^{th}$ entry is 0.

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	0	0	1	1
v_2	1	0	1	1	0	0
v_3	0	1	0	1	0	0
v_4	0	1	1	0	1	1
v_5	1	0	0	1	0	0
v_6	1	0	0	1	0	0

Adjacency matrix for G



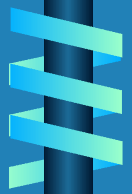
A graph G

4.1 Representation of Graphs

The storage space used is n^2 irrespective of how many edges are there in the graph.

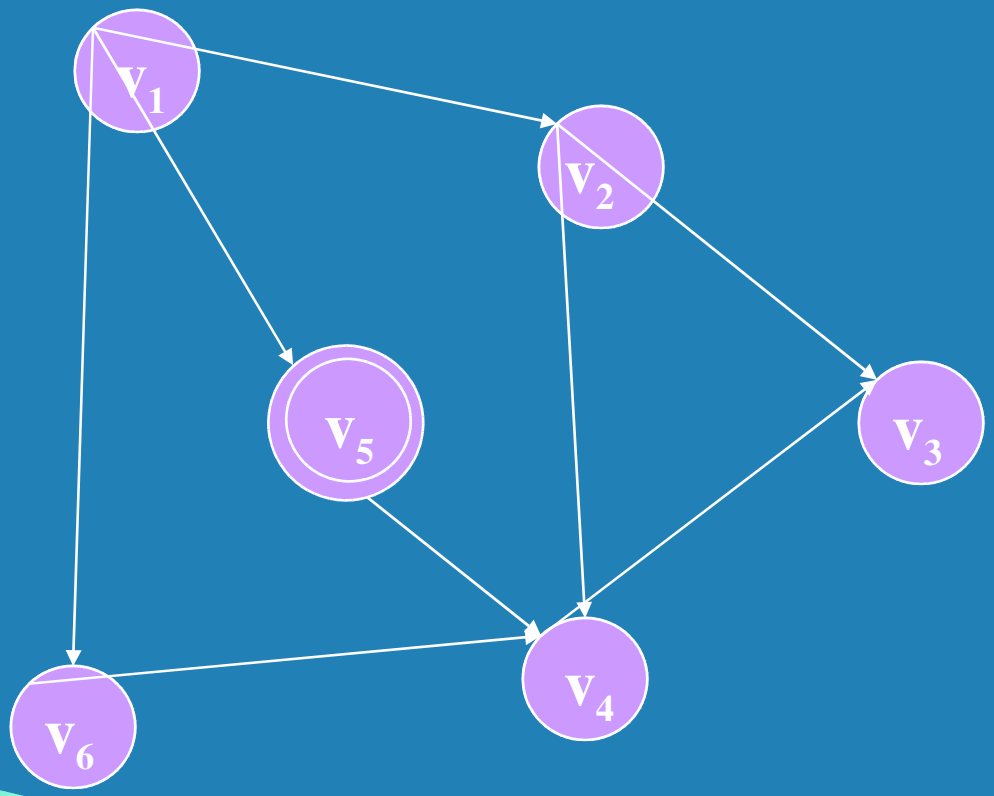
It is easy get the degree of each vertex but finding paths in graph etc will require algorithms.

For dense graphs (graphs with many edges) it is useful, but for sparse graphs, (graphs with few edges) the representation uses a unnecessary storage.

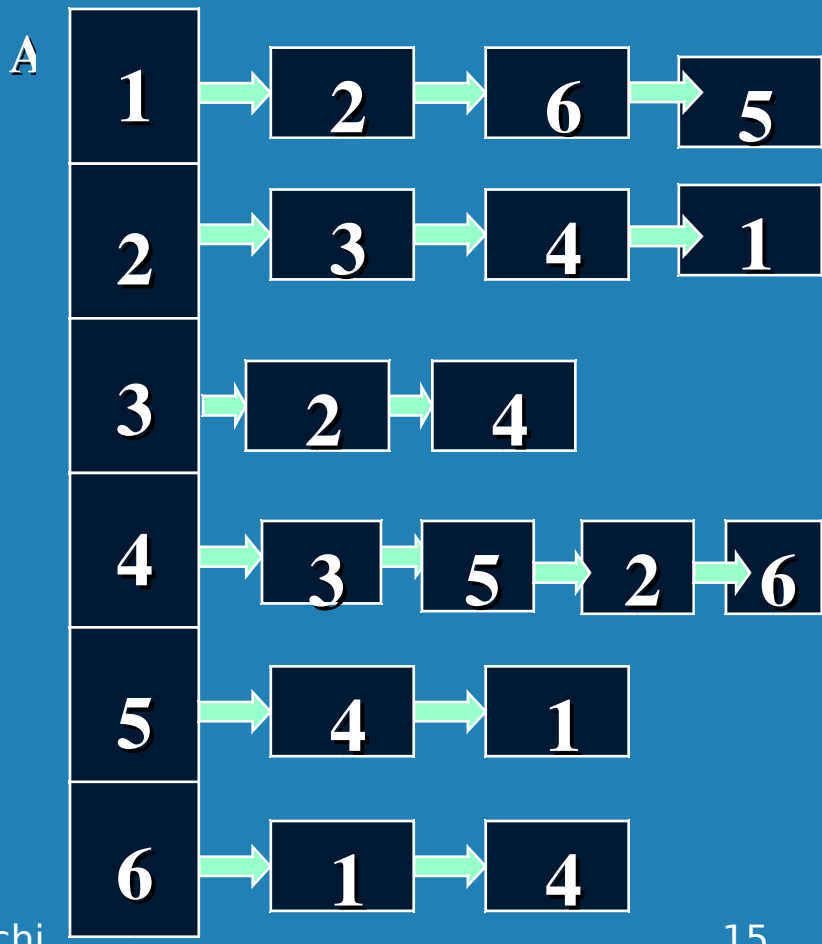


4.1 Representation of Graphs

Adjacency list representation is one in which we have n size array A of linked list, each containing the list of vertices that A[i] is adjacent to.

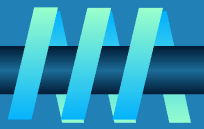


A graph G



Adjacency List for G

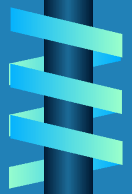
4.1 Representation of Graphs



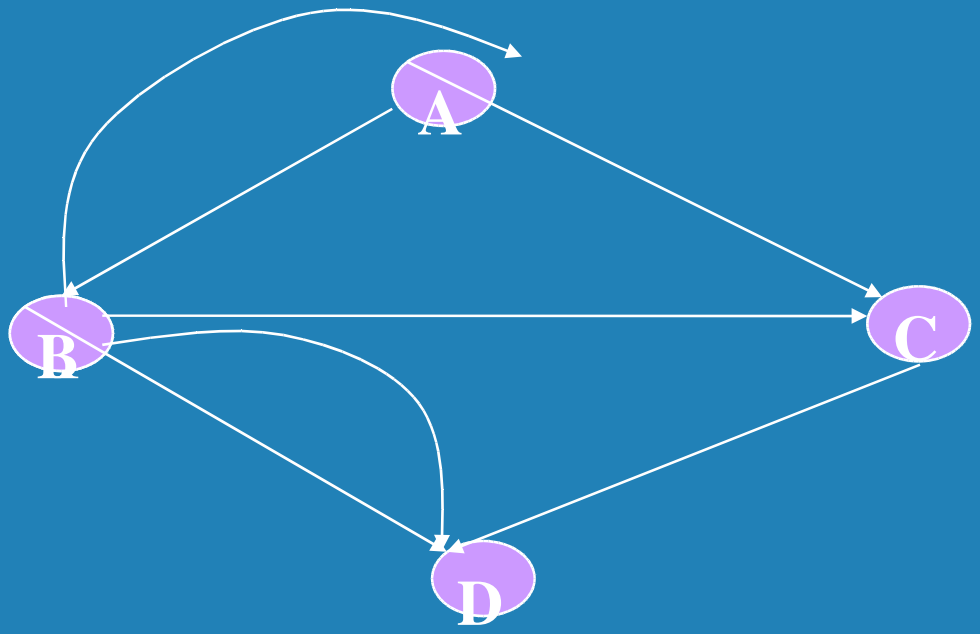
The storage space used is n + 2m

It is easy get the degree of each vertex but finding paths in graph etc will require algorithms.

This representation is considered very efficient since the storage does represent the size of the graph.



4.2 Eulerean tour



Is there a way to start at any one node, travel each edge once, and get back to the same node? This is called EULEREAN TOUR Problem

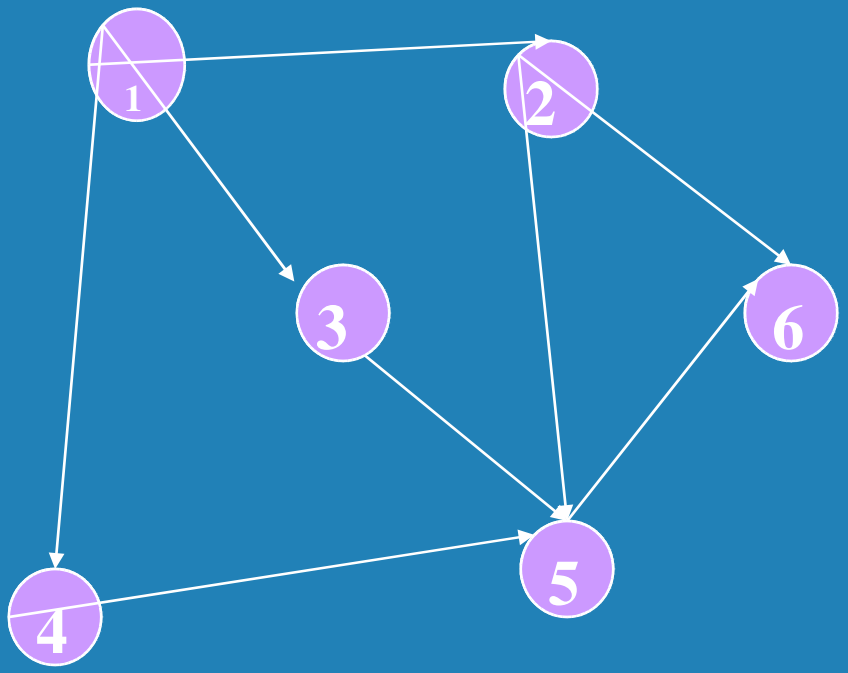
Such a tour exists if and only if the number of vertices with odd degree is at most two.

We travel enter each vertex using an edge (starting at odd vertex, if any) and then leave the vertex using a different edge. (ending at the other odd vertex if there is one).

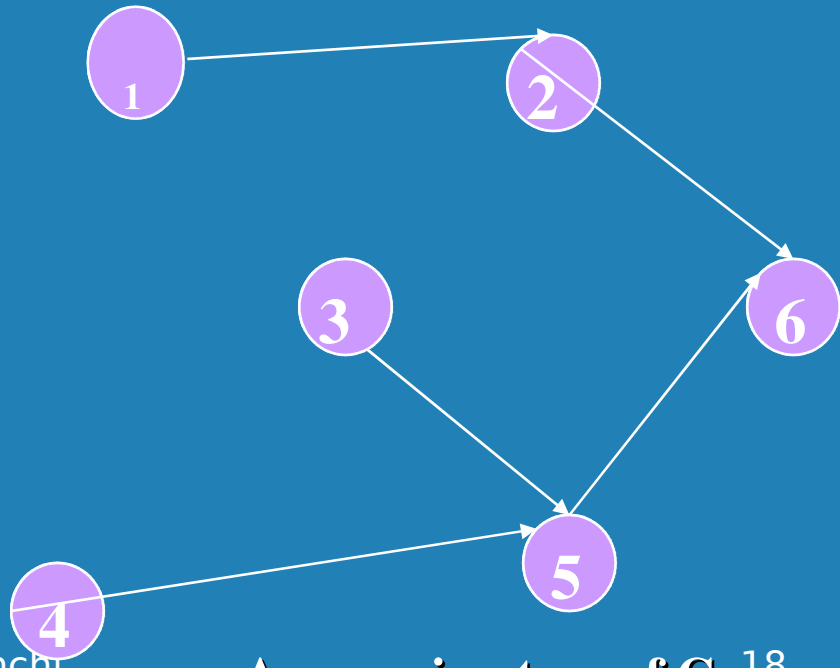
4.2 Spanning Trees

In order to travel the graph, to find path between two vertices, or check if a graph is connected or to check if a graph is a complete graph etc, we need some structures in the graph.

Spanning tree of a connected graph G=(V,E) is a subgraph of G such that it is a tree which contains each vertex of V.



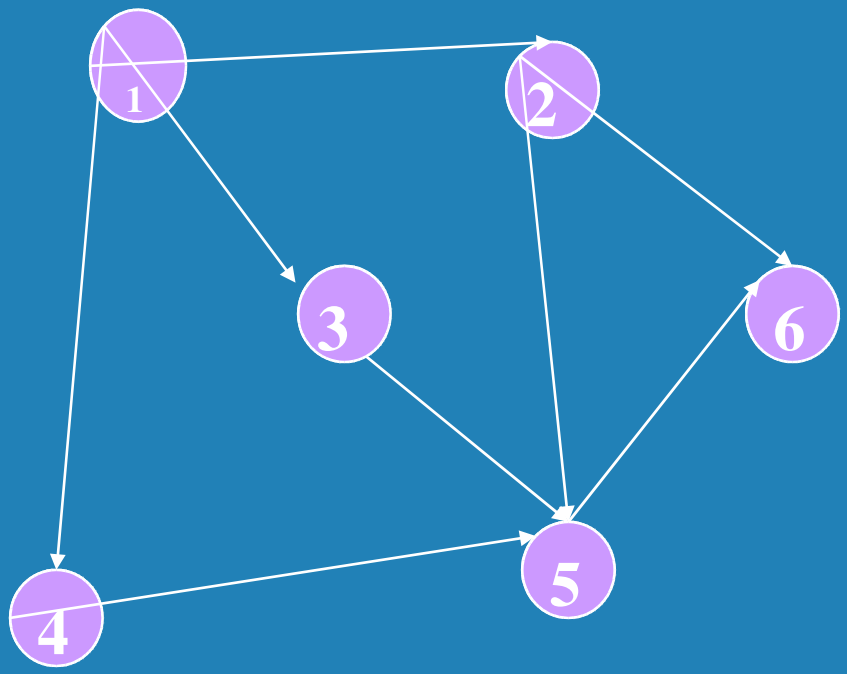
A graph G



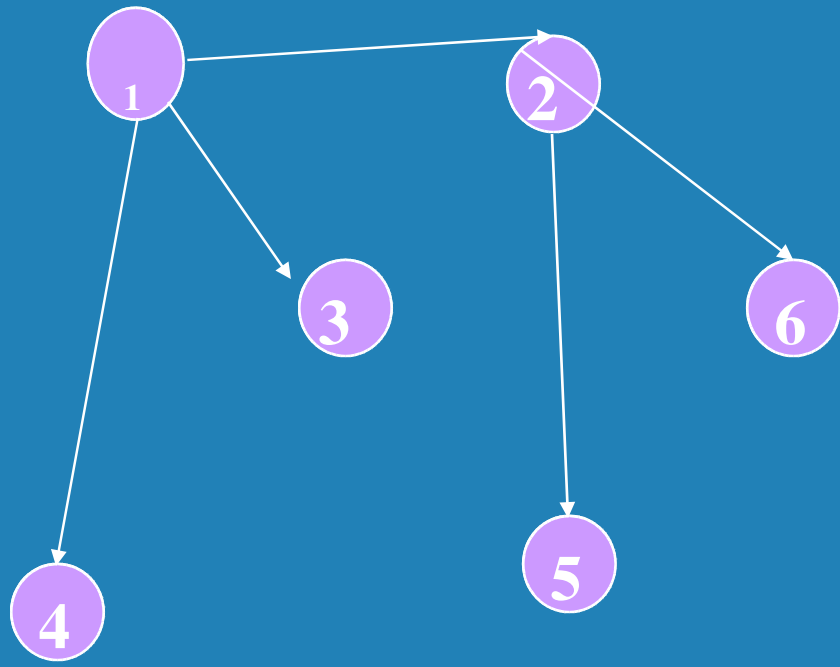
A spanning tree of G

4.2 Spanning Trees

A graph has many spanning trees:



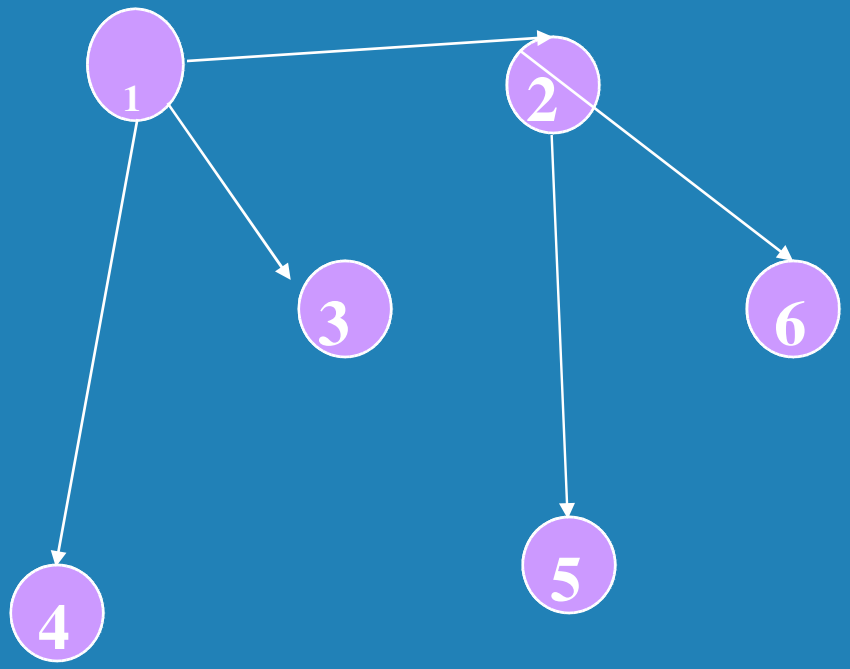
A graph G



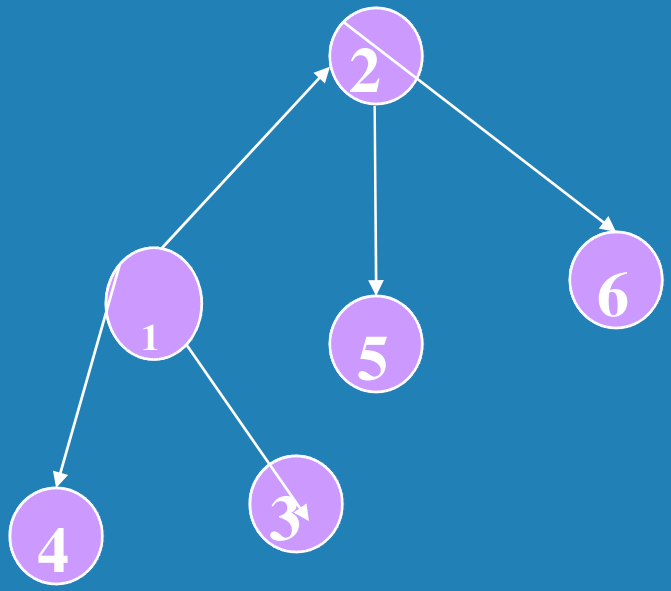
Another spanning tree of G

4.2 Spanning Trees

Spanning trees can be rooted at some vertex, and then such trees can be used to find a path between any two vertices and other related algorithms.



Spanning tree of a graph



Rooted at vertex 2

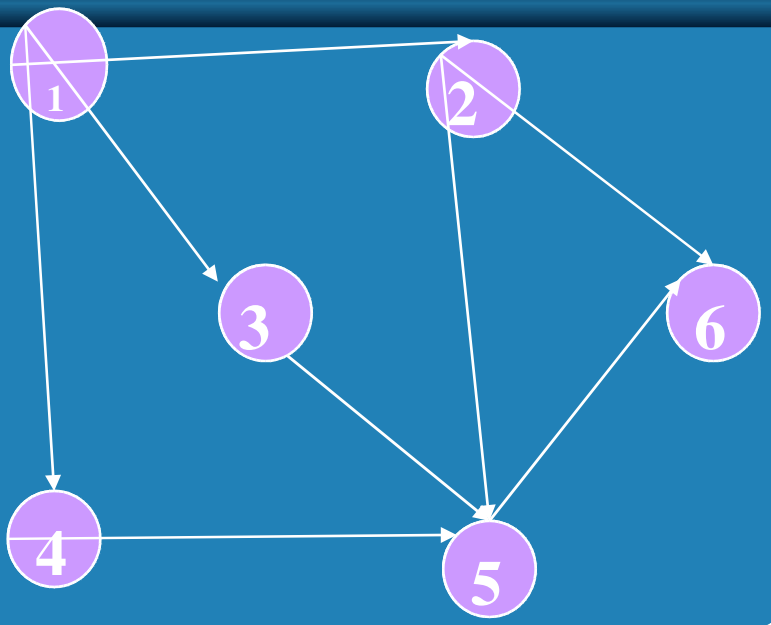
4.2 Breadth First Search Trees

Breadth First Search (BFS) trees are special spanning trees of a graph that is obtained by breadth first search traversal technique.

In breadth first search traversal technique, we simply start at a vertex and make that node the root. Now we add all the vertices adjacent to the root in the graph to the tree as the children of the root.

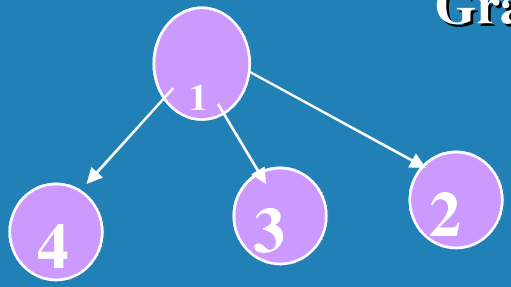
Now we start each of the children and add the nodes adjacent to each as long it is not yet added. We construct the tree level by level, by expanding all possible nodes at each level.

4.2 Breadth-First Search Tree

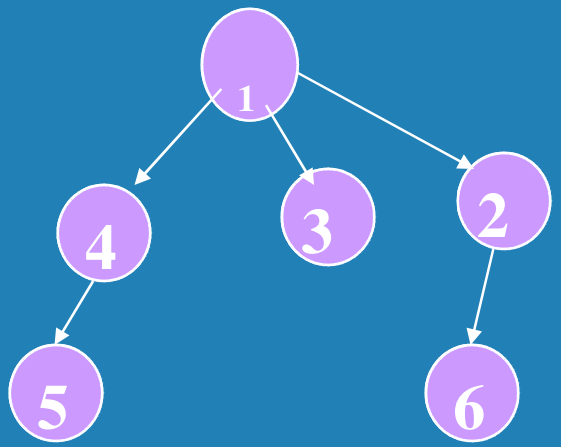


Graph G

Add 1 as root



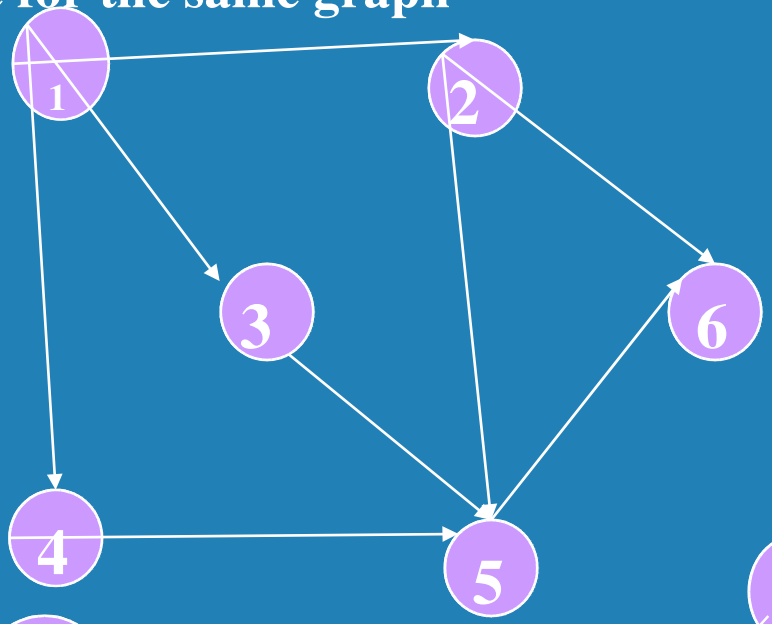
Add nodes adjacent to 1



Now explore 4, 3 and 2 in that order and add unvisited nodes

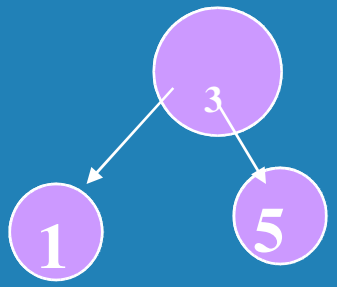
4.2 Breadth-First Search Tree

Another BFS tree for the same graph

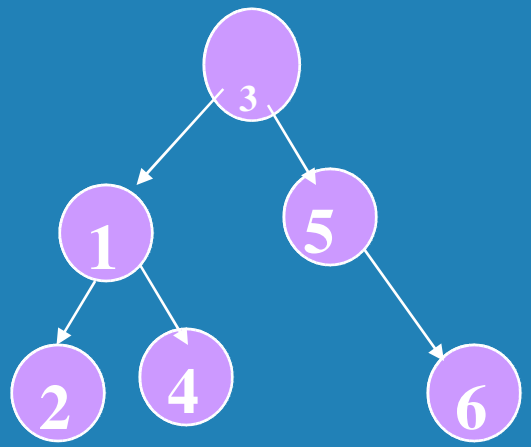


Graph G

Add 3 as root

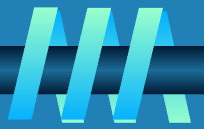


Add nodes adjacent to 3



Now explore 1 and 5 and add unvisited nodes

4.2 Breadth-First Search Tree



Questions:

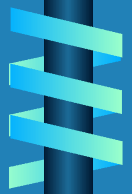
When should we stop adding vertices?

How many BFS trees are there?

Could we obtain different trees even when we start at the same vertex?

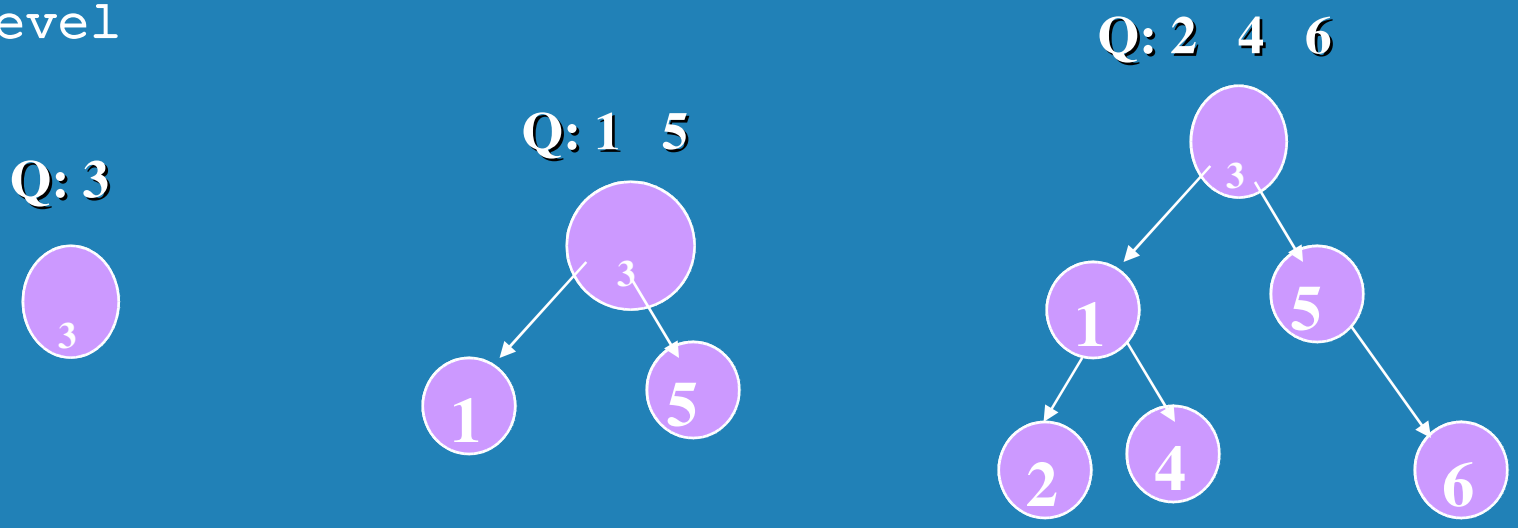
Could we observe anything about the edges that do not belong to the BFS tree? (co-tree edges?)

How do we construct the tree given the adjacency list representation of the graph?



4.2 Breadth First Search Trees

We will use a Queue to store the vertices that we added in one level so that we can expand each node in a level



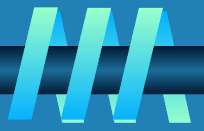
First add 3 to the Queue

Remove 3 and process 3 by adding nodes adjacent to 3 in the graph that are not yet in the tree. Add 1 and 5 to the Queue

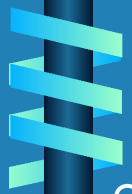
Remove 1 and and add 2 4 to the queue

Remove 5 and add 6 to the queue.

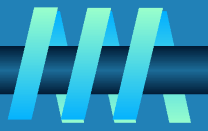
4.2 Breadth First Search Tree



```
procedure BreadthFirstTraversal (var G: Graph);
var      Q:Queue;
         p: node;
         visited:array[1..n] of boolean;
begin
     // Initialize the visited array to false.
     // Pick the first vertex, 1 as the root.
     Enque(Q,1);
     visited[1] := true;
     while not isEmpty(Q) do
        p:= deque(Q);
        for each vertex v in the adjcency list of G[p]
            if (visited[v] = false)
                visited[v] := true;
                enque(Q,v);
            endif
        endfor
    endwhile
end;
```



4.2 Breadth First Search Tree

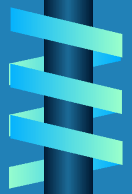


The time complexity of the inner for loop is two times the number of edges in the graph. Every edge is visited once when you travel the adjacency list of each vertex once.

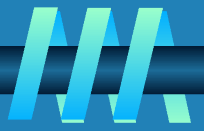
Note that the algorithm does not actually output the BFS tree, but this can be added to the code.

Time complexity is O(m)
Space Complexity is O(n) [why?]

Question: How do we use BFS algorithm to check if a graph is connected?

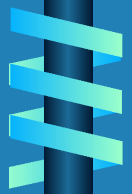


4.2 Depth First Search Tree

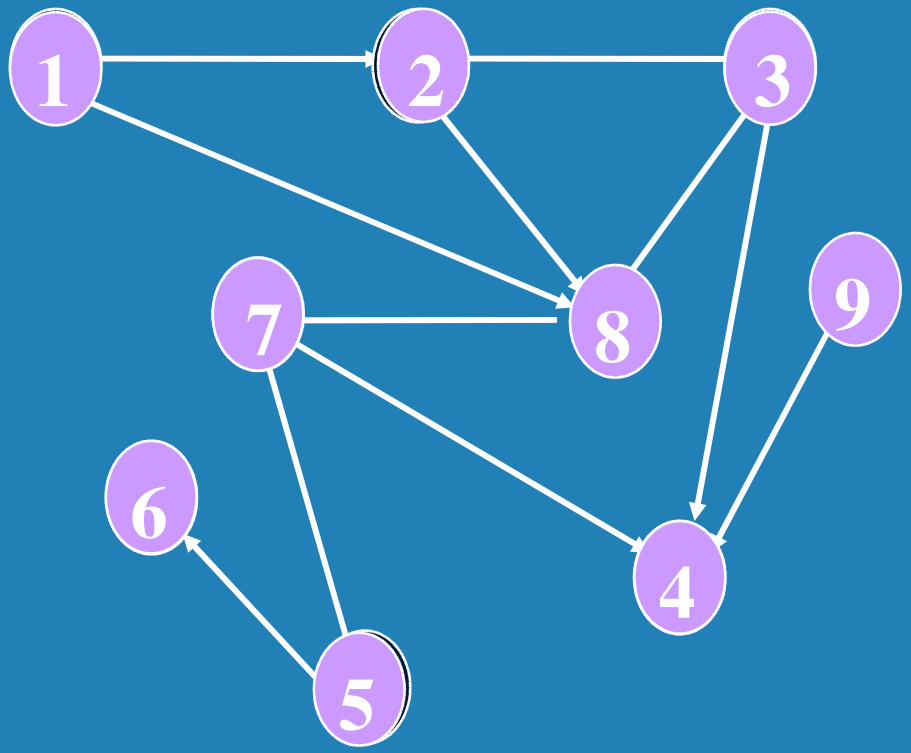


Breadth first search tree travels the graph in a breadth first manner, but depth first search would travel the graph as “deep” as possible.

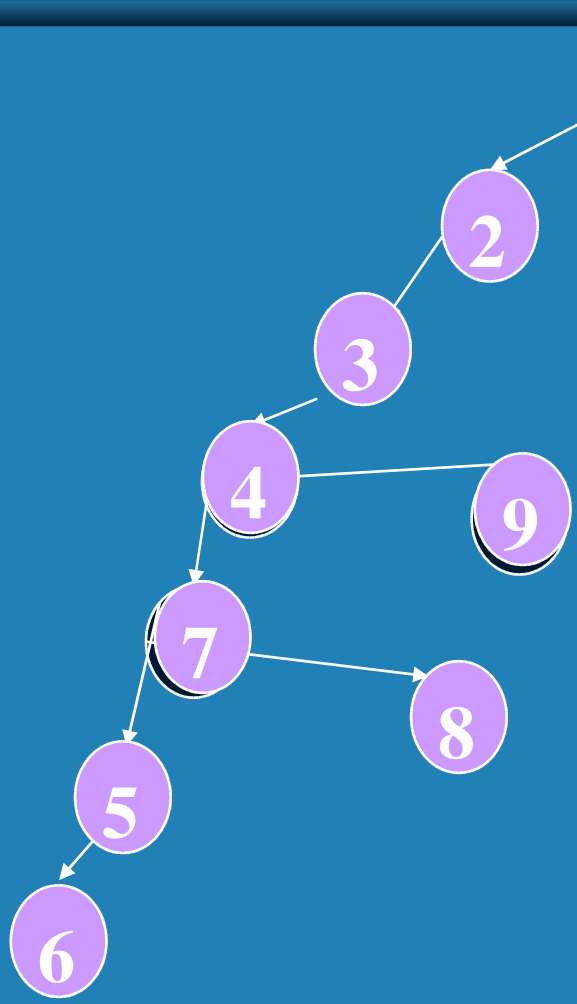
The idea is to start at some random vertex and mark it root. Now travel one node that is adjacent to root and make it the child of root. Then go from this child to find its child. When we come a vertex for which all its adjacent nodes are already in the tree, then we backtrack to the parent of this node.



4.2 Depth First Search Tree

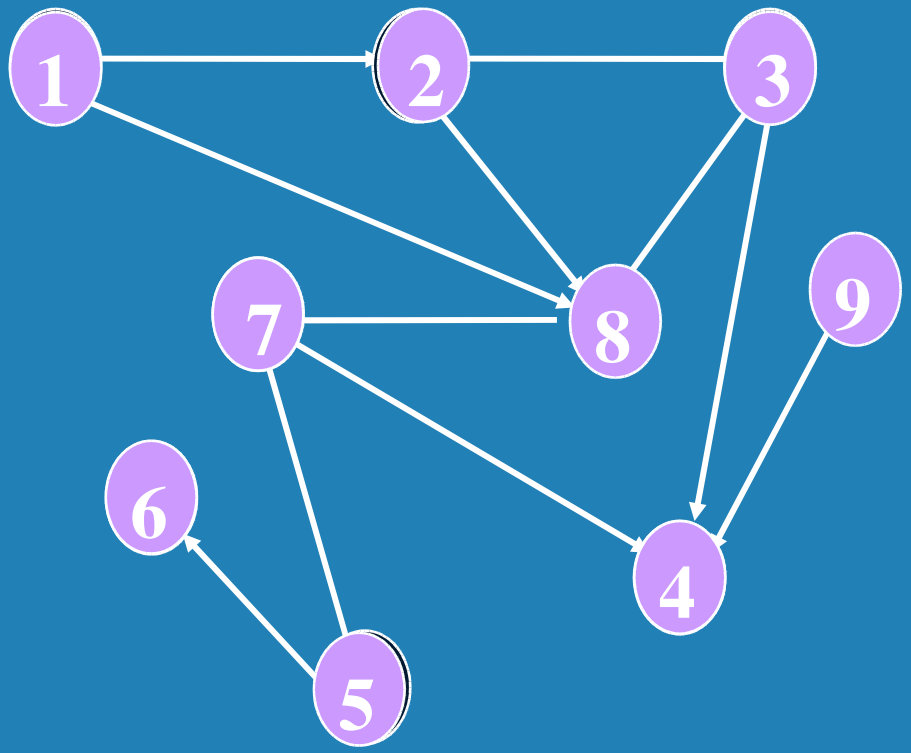


Graph G

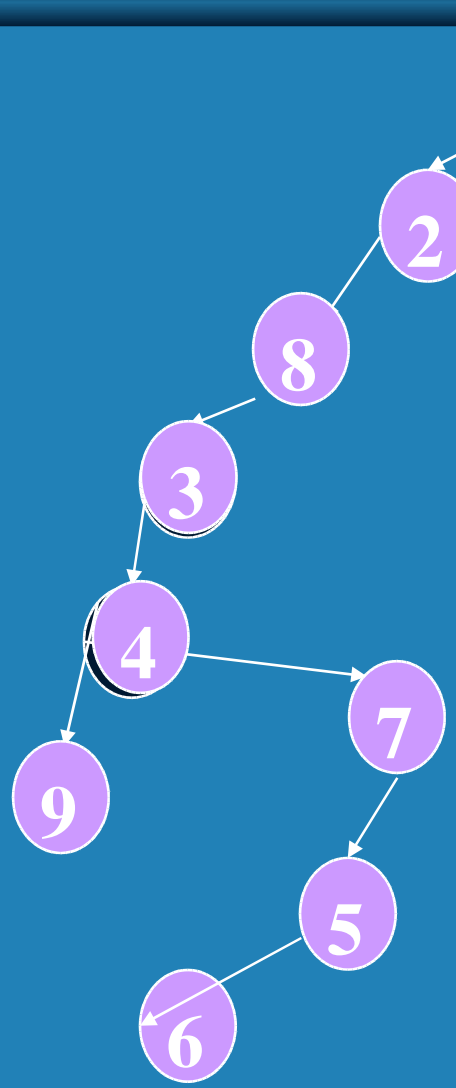


A DFS tree T

4.2 Depth First Search Tree

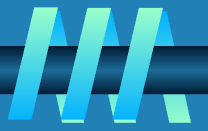


Graph G



Another DFS tree

4.2 Depth First Search Tree



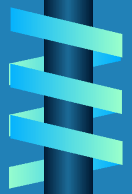
Questions:

How many DFS trees are there?

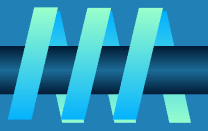
Could we obtain different trees even when we start at the same vertex?

Could we observe anything about the edges that do not belong to the BFS tree? (co-tree edges?)

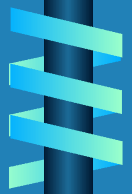
How do we construct the DFS tree given the adjacency list representation of the graph?



4.2 Depth-First Search Tree

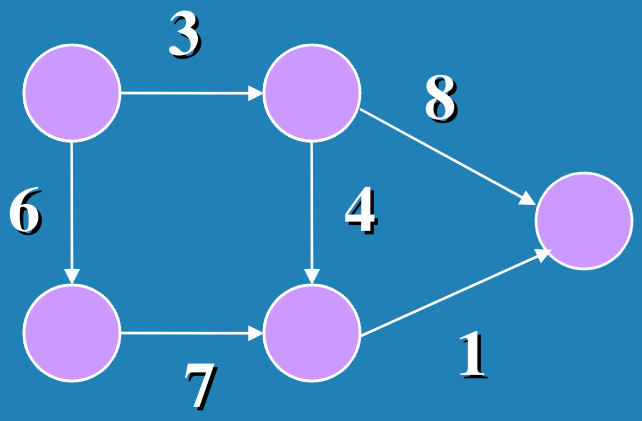


```
// Graph G is global
// Visited is an array of size n of boolean initialized
to false
// call the recursive procedure dfs(1);

procedure dfs(var node:GraphNode)
begin
      visited[node]:=true;
      for each node v in the adjacency list G[node]
        if (visited[v] is false)
              dfs(v);
        endif
     endfor;
end;
```



4.3 Weighted Graphs

Weighted Graphs are graphs in which edges have a positive numerical weight.

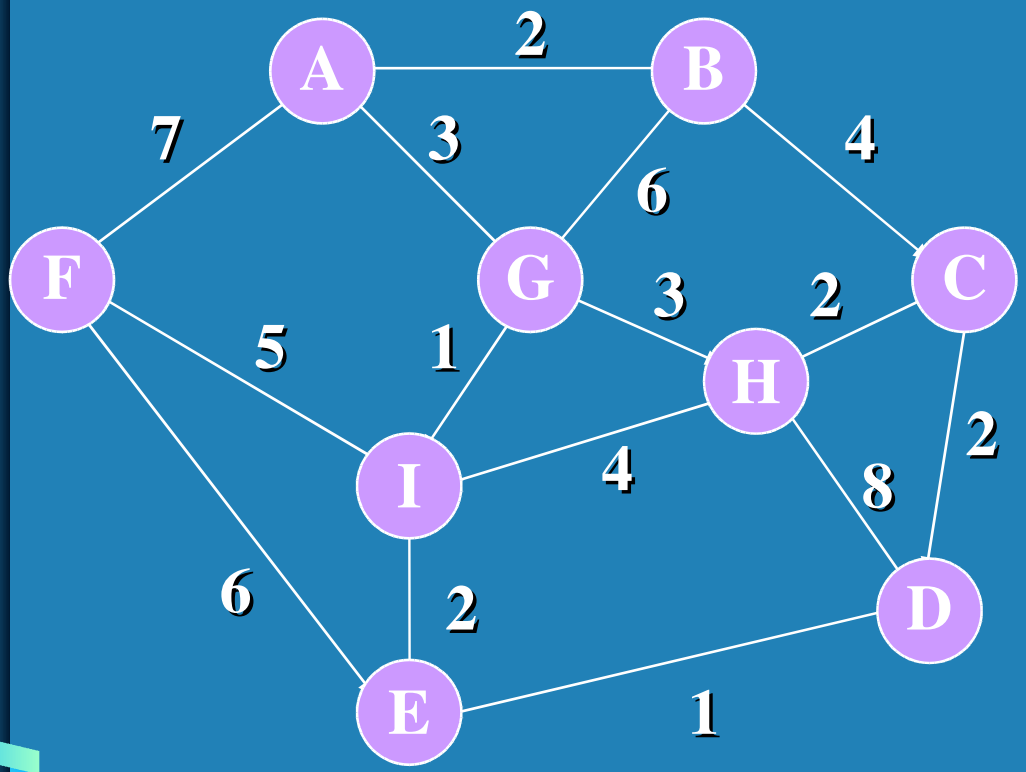


Application of Weighted Graphs:

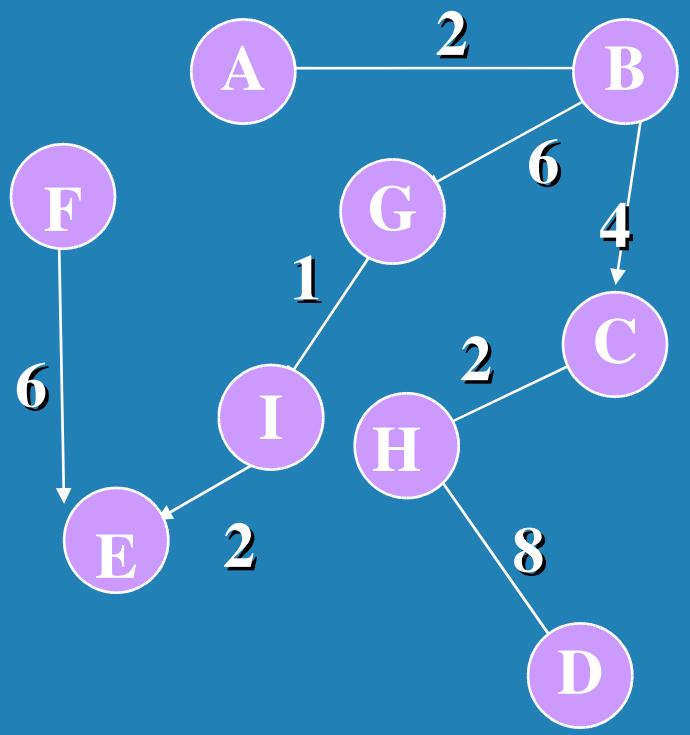
Traveling Salesman
Distance maps
Network loads
Neural networks

4.3 Weighted Graphs

Just like simple graphs, weighted graphs also have spanning trees. Now we can associate a number called weight of a tree with the spanning tree. The weight of a tree is the sum of the weights of all the edges in the tree.

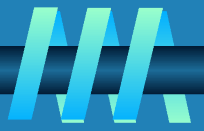


The graph G



A spanning tree of weight 31

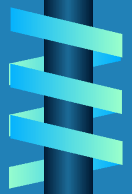
4.3 Minimum Spanning Trees



Given a weighted graph, it is common question to find the shortest distance between two given nodes, or distance of a path between two vertices that goes through a specified vertex C, etc.

To answer questions like that, we first find a spanning tree with minimum weight. Such trees are called minimum spanning trees.

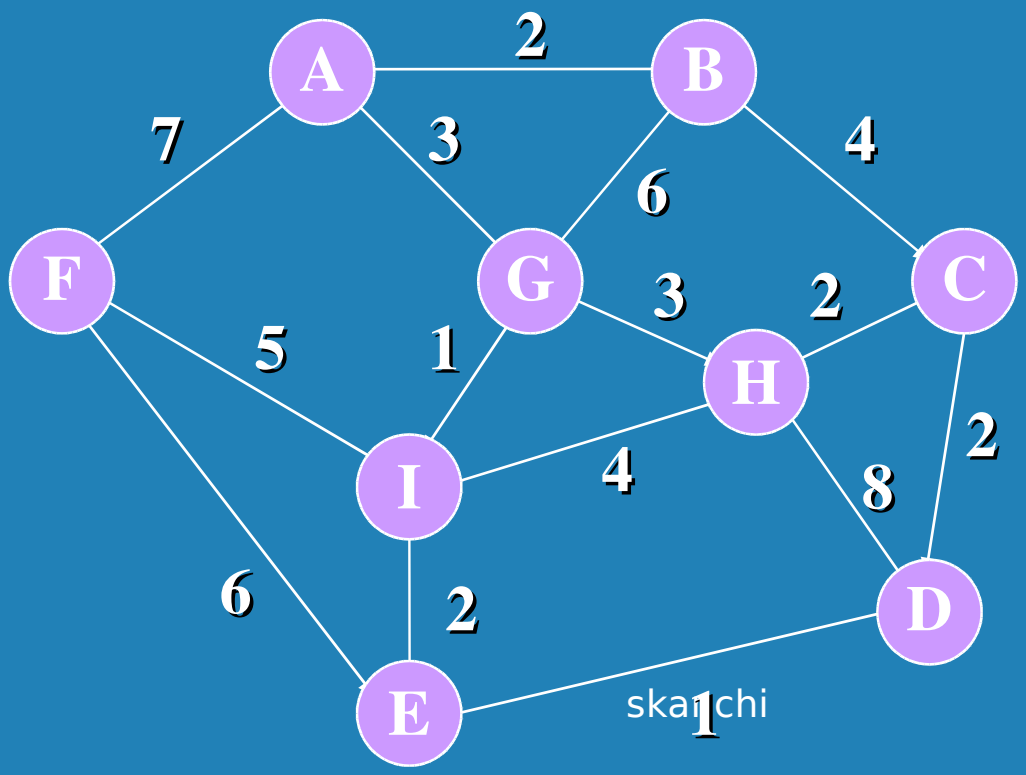
There are two different algorithms to find the minimum spanning trees(MST).



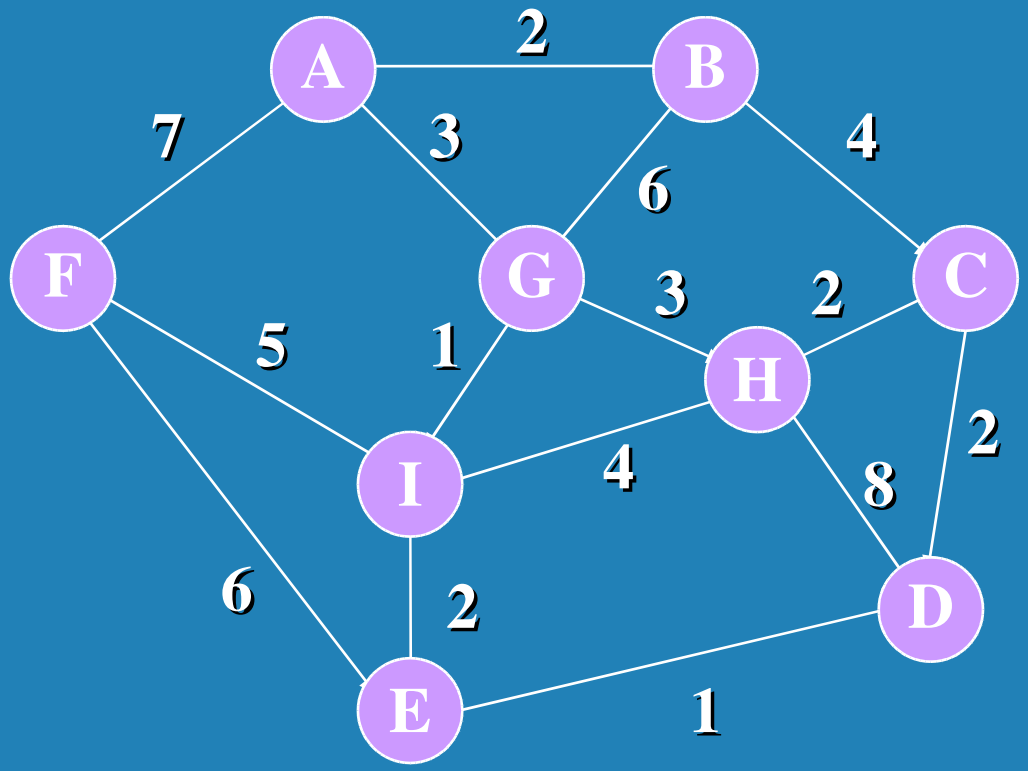
4.3 Kruskal's Algorithm for MST

In this algorithm, we simply sort the edges on the basis of their weight. We keep adding edges from the sorted list, to the tree, as long as the edge that we add does not form a cycle when added to the tree. When we have added n-1 edges, we stop.

Let us find the MST for the following graph.



4.3 Kruskal's MST-Demonstration-0



The graph G

Sorted List of edges

(G,I) =1 (F,E)=6

(E,D)=1 (B,G)=6

(C,H)=2 (A,F)=7

(C,D)=2 (H,D)=8

(A,B)=2

(I,E)=2

(G,H) =3

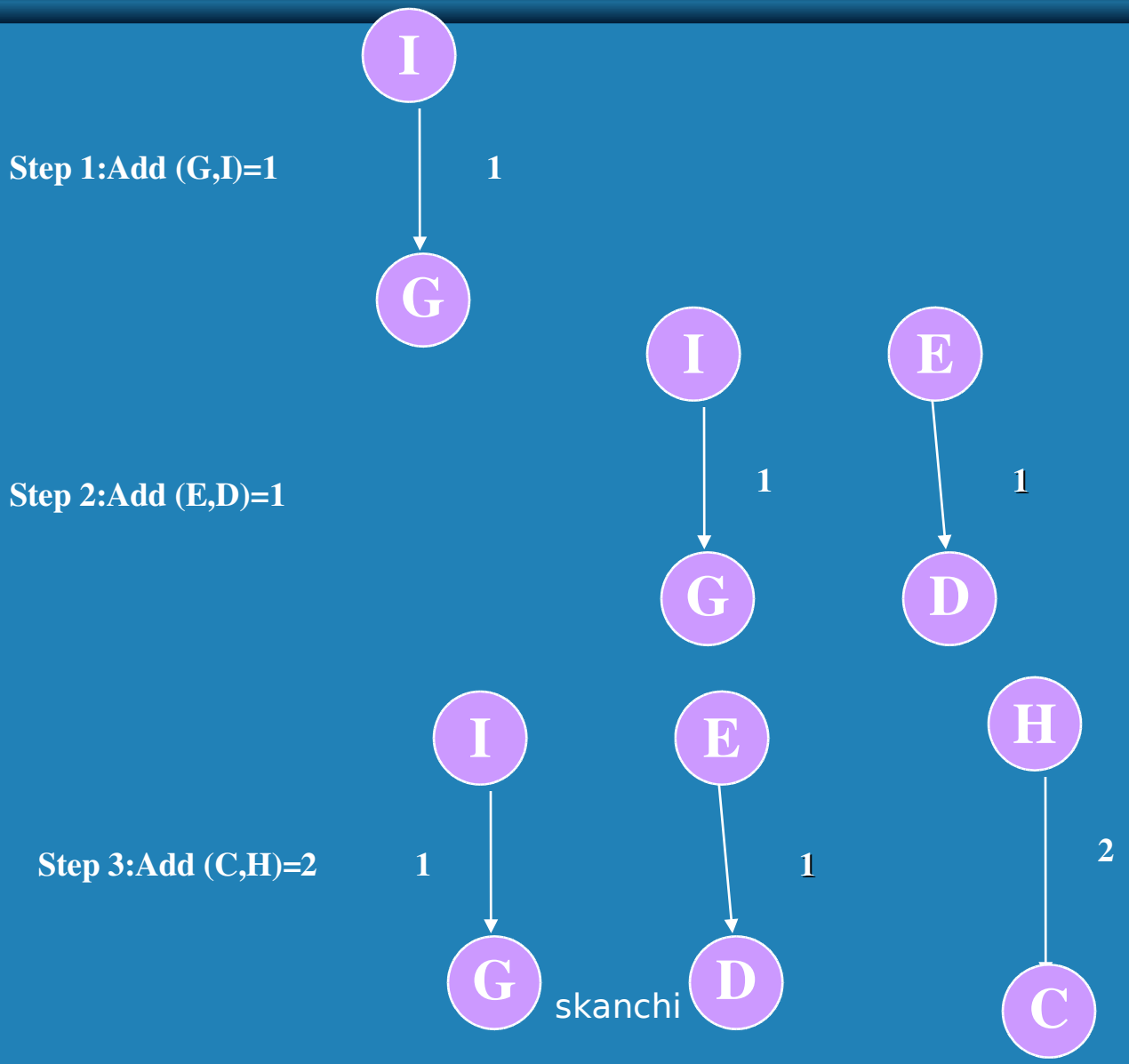
(A,G)=3

(B,C)=4

(I,H)=4

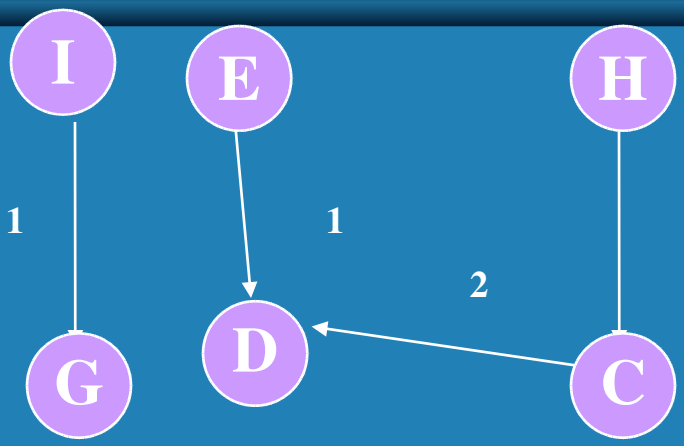
(F,I)=5

4.3 Kruskal's MST- Demonstration-1

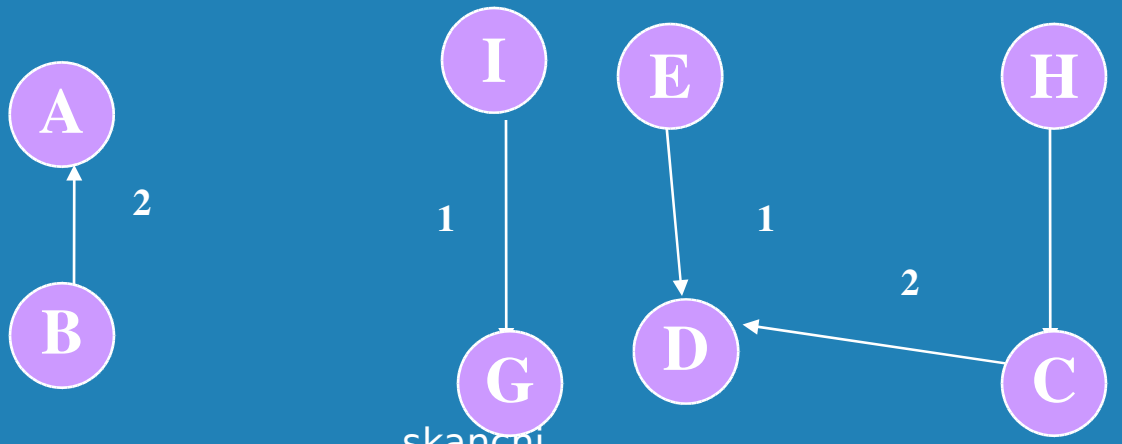


4.3 Kruskal's MST- Demonstration-2

Step 4:Add (C,D)=2

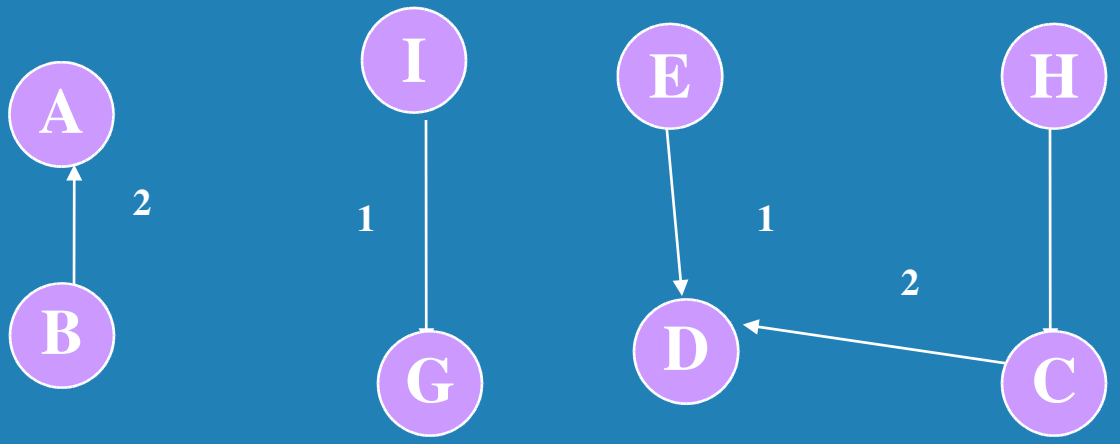


Step 5:Add (A,B)=2

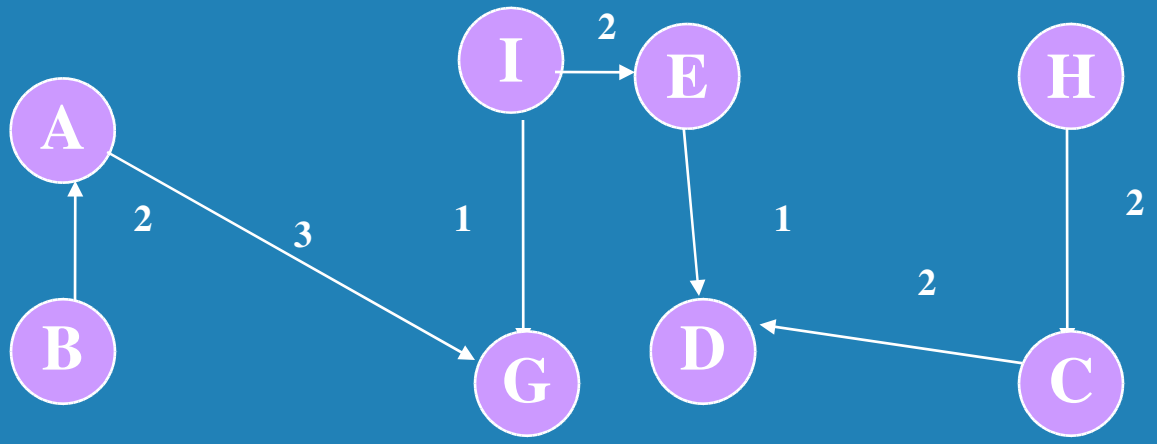


4.3 Kruskal's MST- Demonstration-4

Step 6:Add (I,E)=2

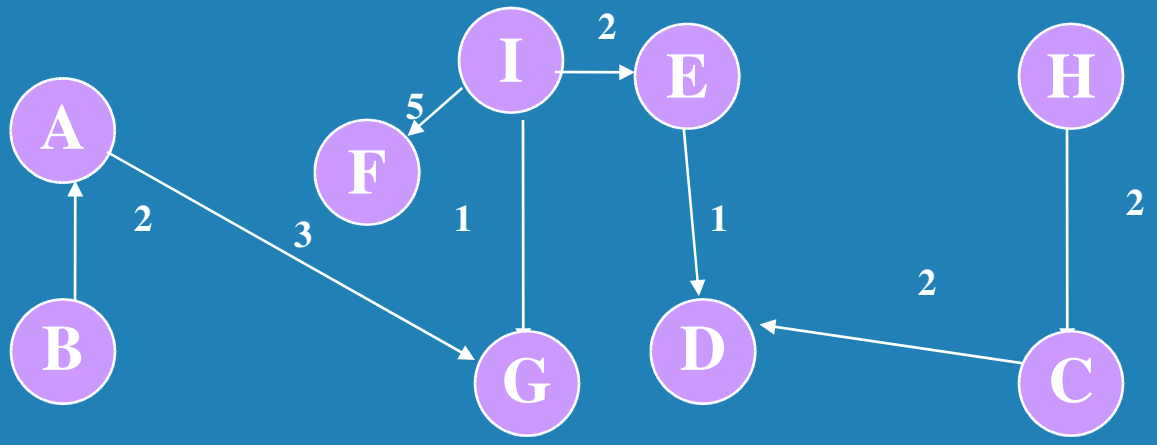


Step 7:Add (A,G)=3



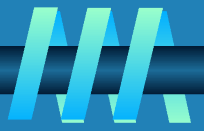
4.3 Kruskal's MST- Demonstration-5

Step 8:Add (F,I)=5

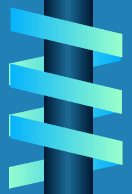


Since n-1 vertices are added we stop. The weight of the MST is 18.

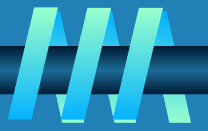
4.3 Kruskal's Algorithm:Code



```
procedure KruskalMST(G:Graph)
begin
   sort all the edges of E into a sorted
List L.
   add the first edge from L to tree
(forest)T;

   while (n-1 edges are not yet added)
      find the next edge e in L that does
      not  form a cycle when added to the
forest T;

      add e to T;

   endwhile
end;
```



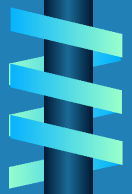
4.3 Kruskal's Algorithm:Code



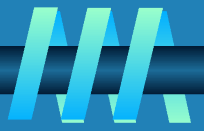
The time complexity is the time to sort which is O(mlogm).

The while loop executes m times, once for each edge. However, to check if an edge forms a cycle it takes long time since we may have to run a BFS or DFS algorithm on the current tree we are forming.

To check if an edge forms a cycle, we need to do a BFS or DFS algorithm, which takes time m. Since we do this for each edge, the time would be m^2



Set Union-Find Algorithms

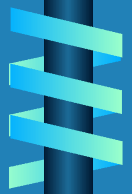


Keep an array of size n of pointers, each array[i] pointing to the node i in tree structure.

A

Initially there are n trees each containing the value of the node. The root of the tree is the node itself.

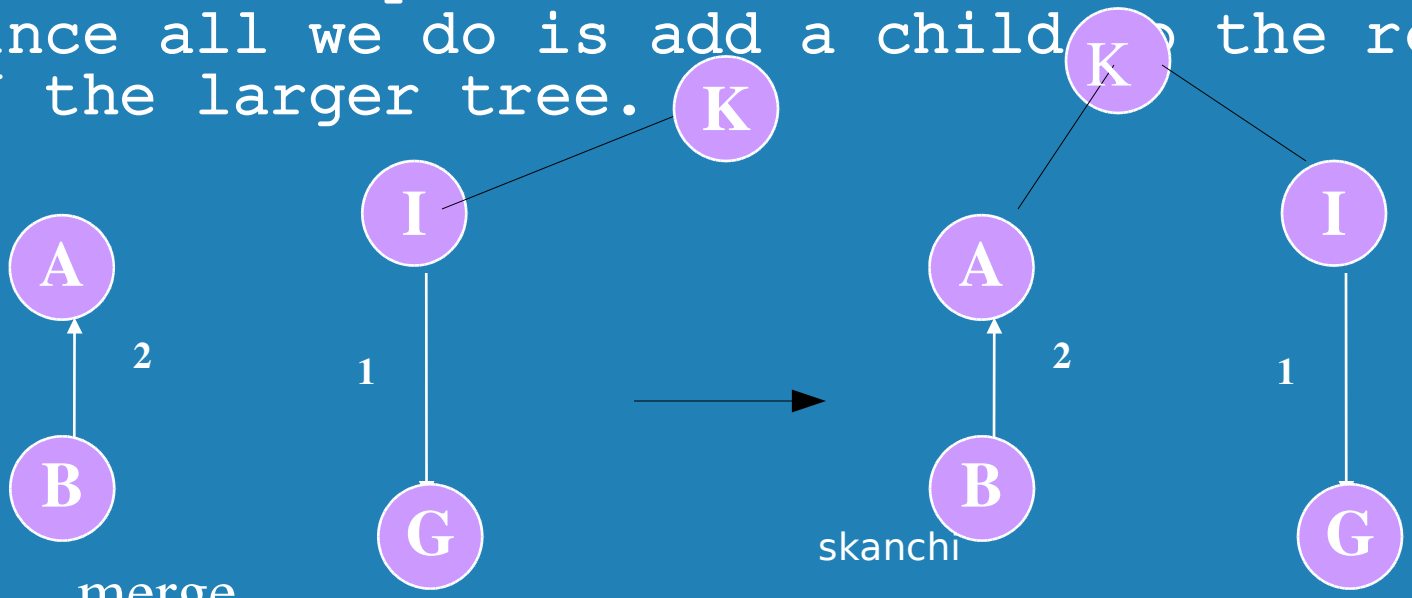
When an edge (K,L) is added, we first find the roots of trees containing vertices K and L respectively, where K and L are the endpoints of the edge to add.



Set Union-Find Algorithms

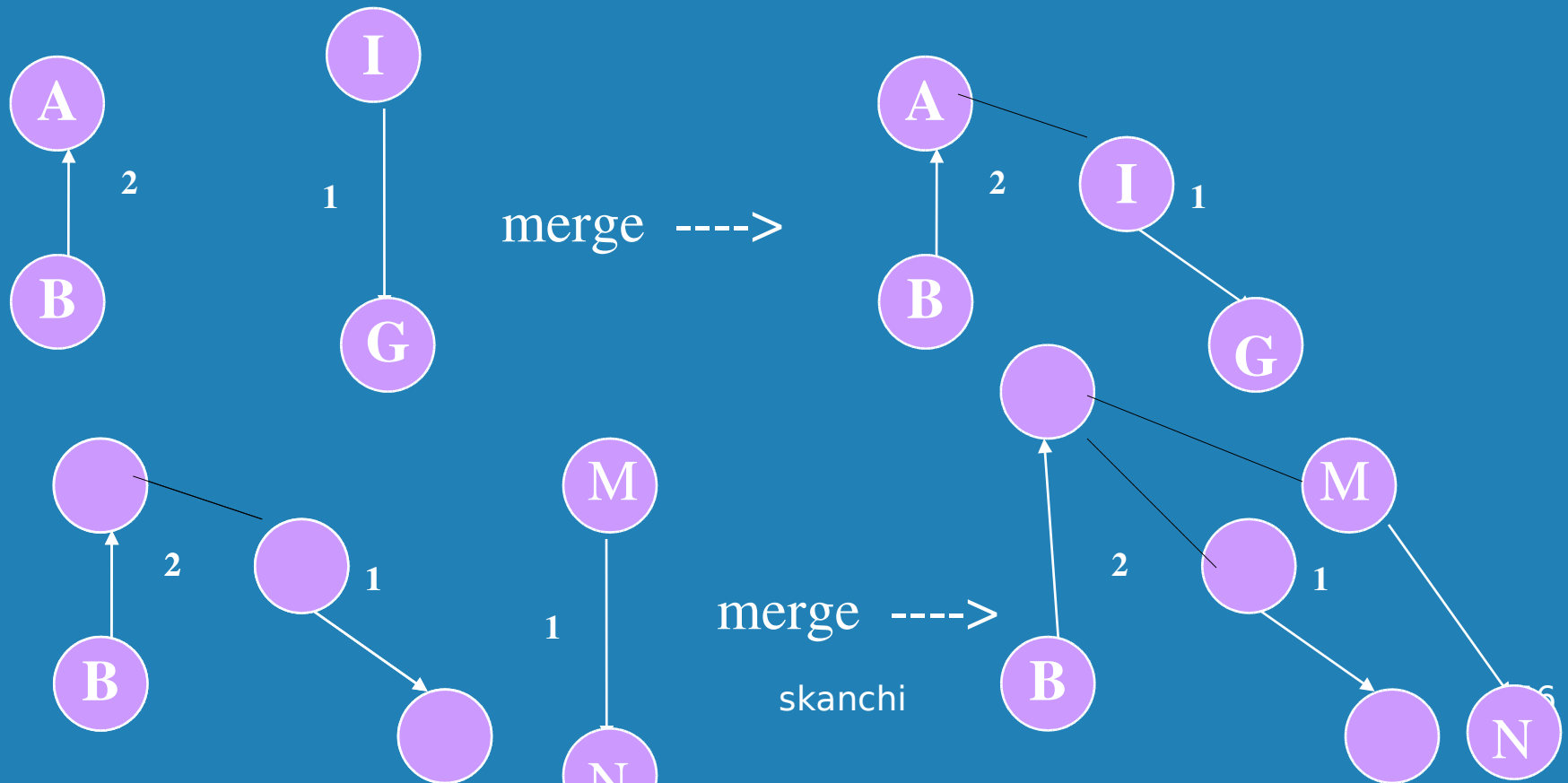
If the roots of K and L are different, then we add the edge (K,L) and merge, (union) the two trees.

To perform the union, the smaller tree is made the child of the larger tree and the size of the total number of nodes is stored at the root.
The time to perform the union is time 1, since all we do is add a child to the root of the larger tree.

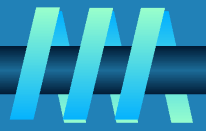


Set Union-Find Algorithms

If we always merge the smaller with the larger tree, while doing union, tree height can never be more than logn!



Time Complexity of Kruskal's Algorithm



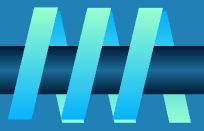
Therefore Kruskal's algorithm, which does 2m finds, finding the two endpoints for each edge that it tries to add, takes 2mlogn time. Once it finds it will do an union if needed. Since n-1 edges are added to make a tree, there are at most n-1 unions. Each union takes only time 1.

n-1 + 2mlogn

for the adding edges part of the algorithm and

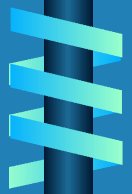
mlogm for sorting the list of edges by weight.

4.3 Dijkstra-Prim Algorithm for MST

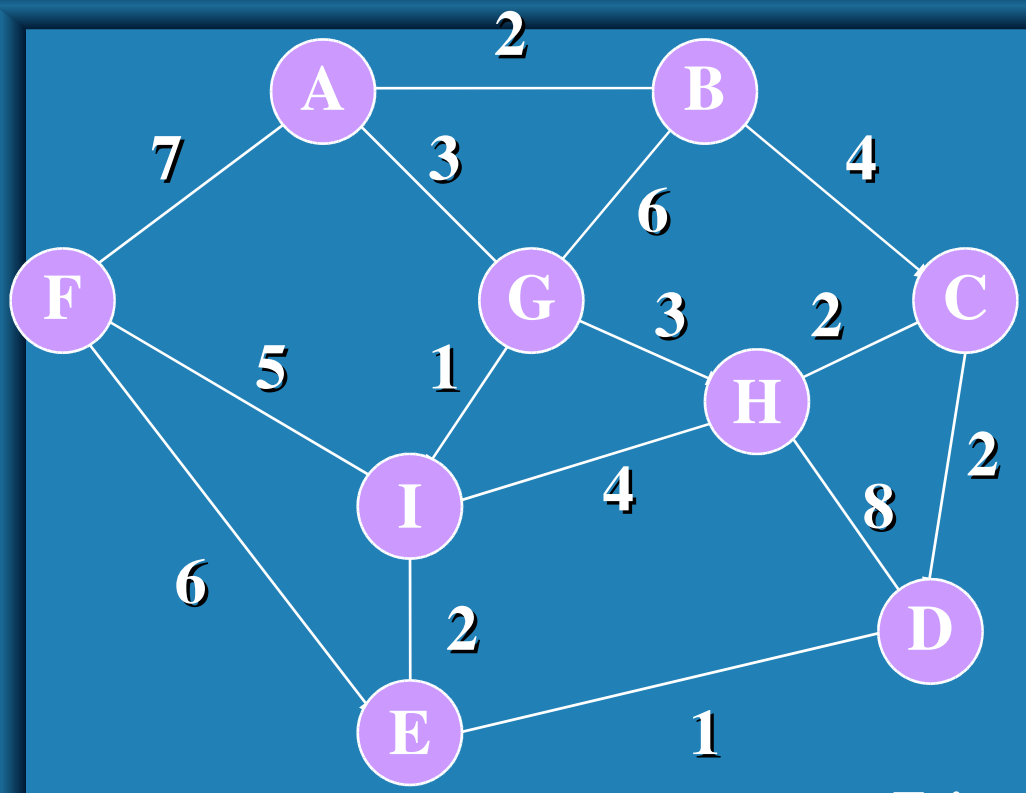


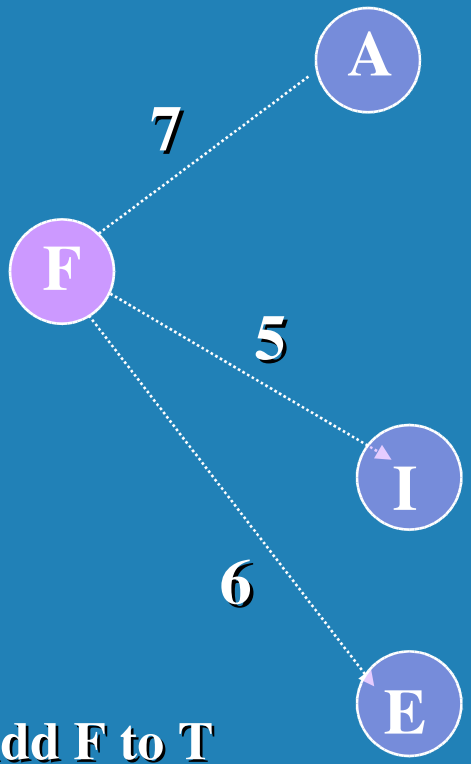
Unlike the Kruskal's algorithm which has a forest along the way of finding the MST, Dijksta's algorithm always keeps the tree connected.

The idea is have a list of fringe vertices, that are candidates to be added in the next iterations. The distance to the tree from each vertex is maintained. The fringe node with the smallest distance to the tree is added to the tree. Whenever a node is added to the tree, its adjacencies are examined to see additional fringe nodes exist and distance to the existing fringe nodes are updated.



4.3 Dijkstra-Prim MST-Demo-0

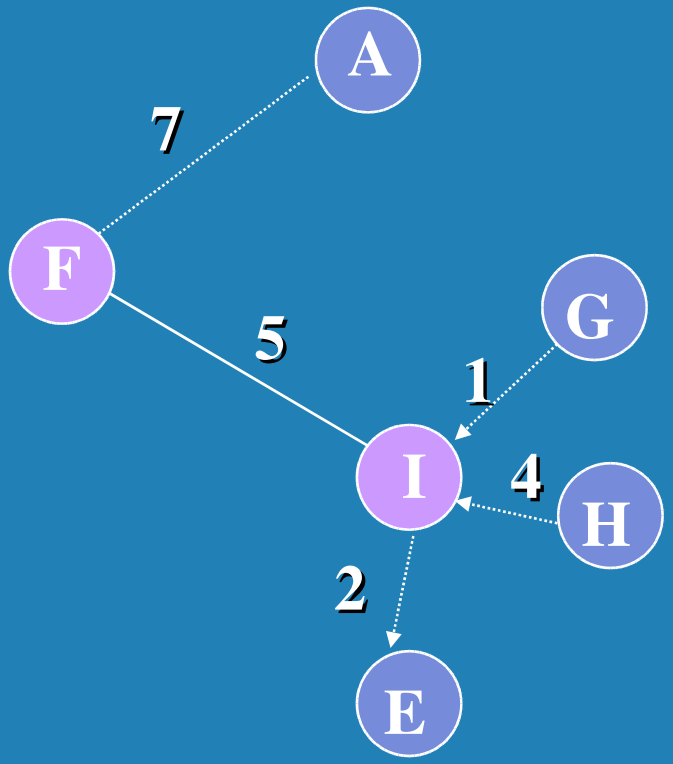


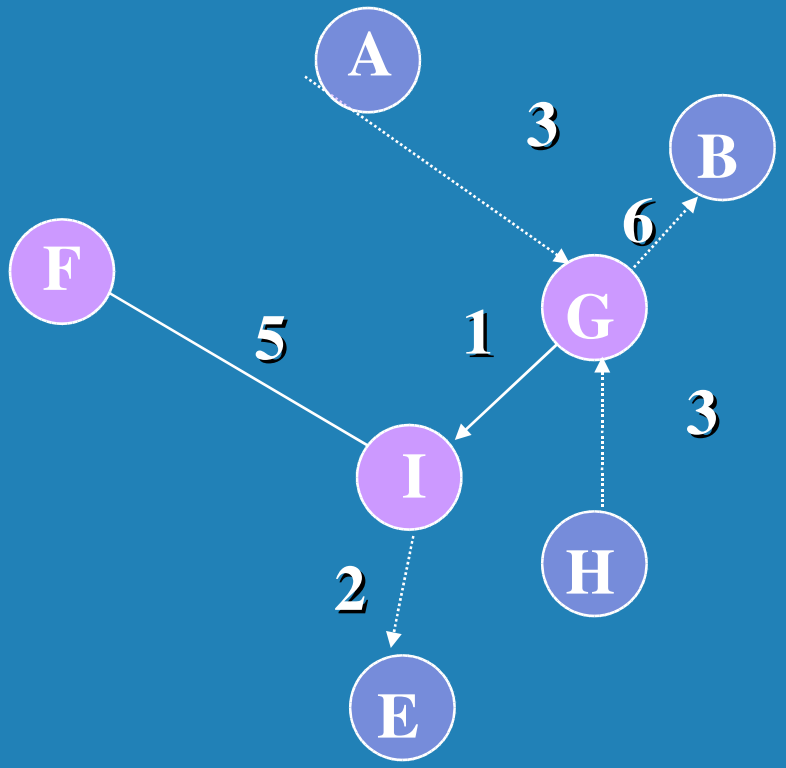


Fringe List: Add F to T

A	B	C	D	E	F	G	H	I
7				6	*			5

4.3 Dijkstra-Prim MST-Demo-1





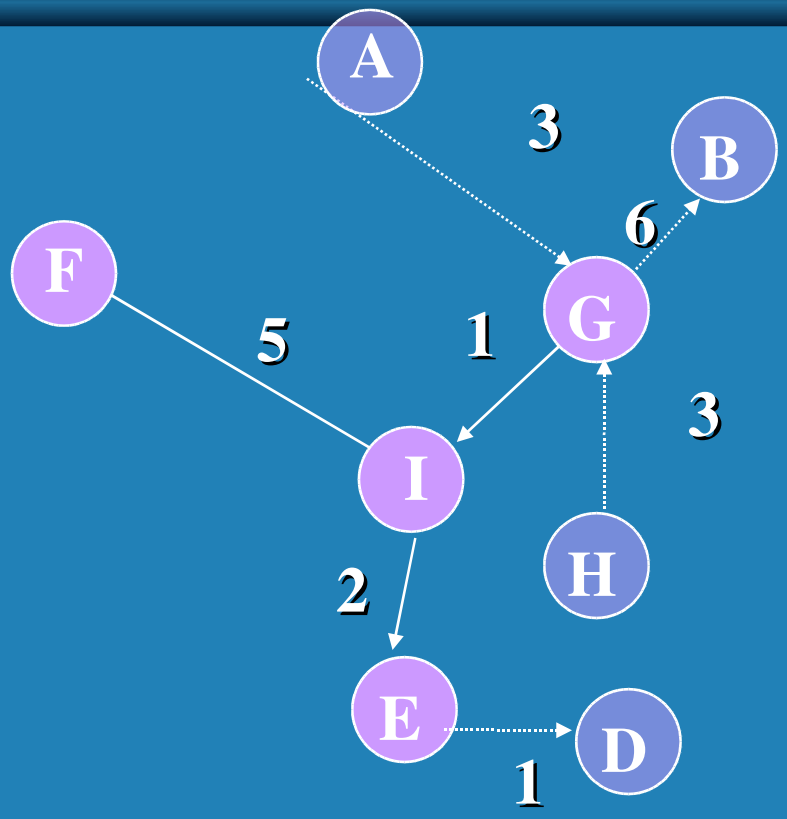
Fringe List: Add I to T

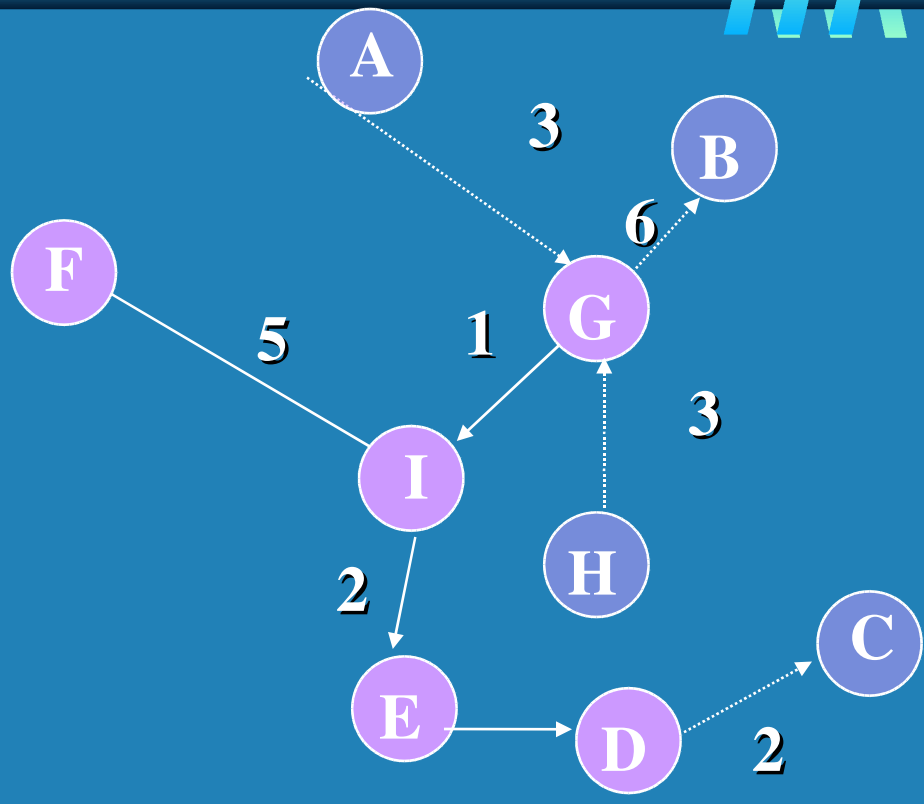
A	B	C	D	E	F	G	H	I
7				2	*	1	4	*

Fringe List: Add G to T

A	B	C	D	E	F	G	H	I
3	6			2	*	*	3	*

4.3 Dijkstra-Prim MST-2





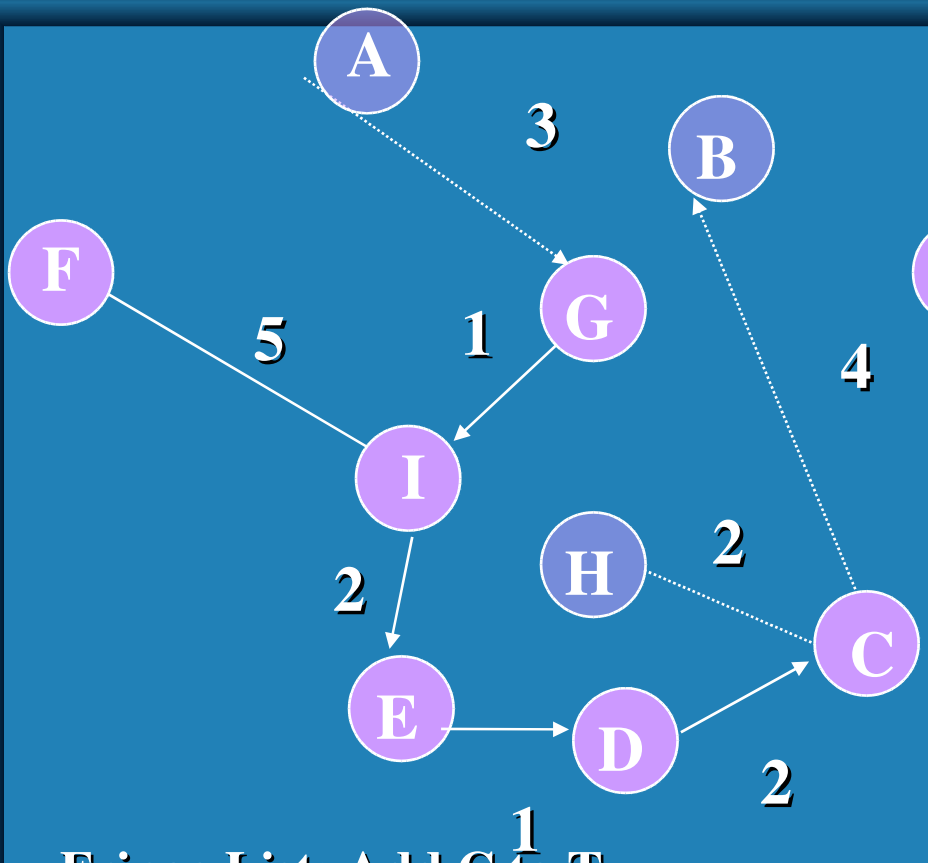
Fringe List: Add E to T

A	B	C	D	E	F	G	H	I
3	6		1	*	*	*	3	*

Fringe List: Add D to T

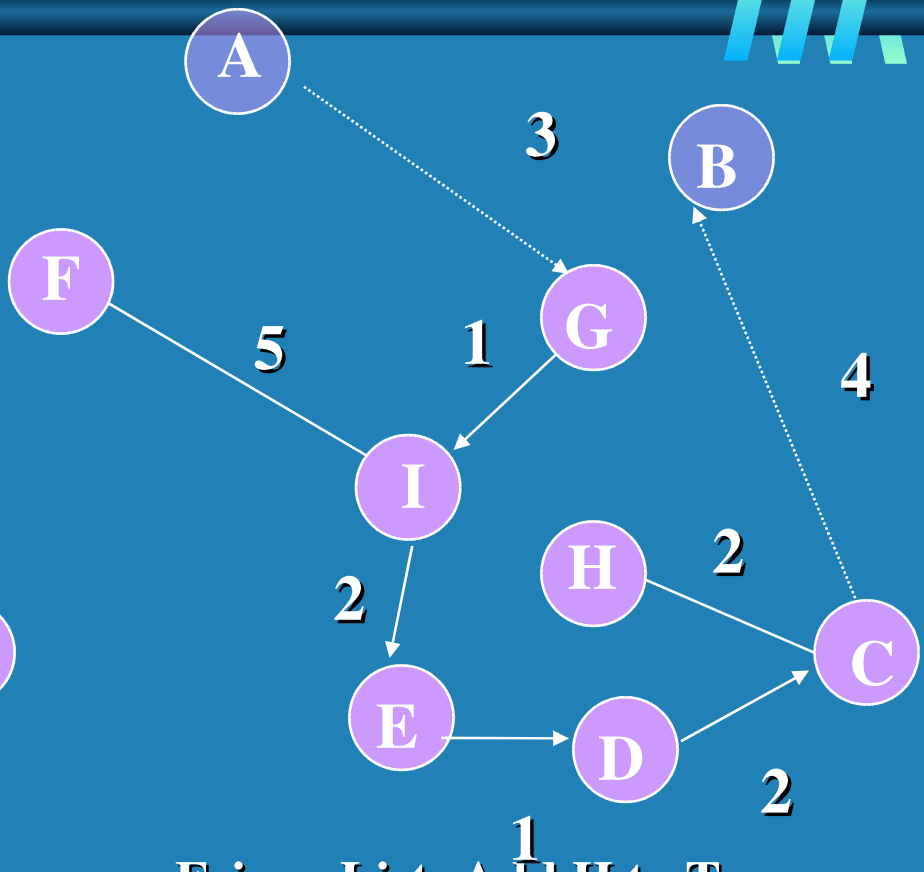
A	B	C	D	E	F	G	H	I
3	6	2	*	*	*	*	3	*

4.3 Dijkstra-Prim MST-Demo-3



Fringe List: Add C to T

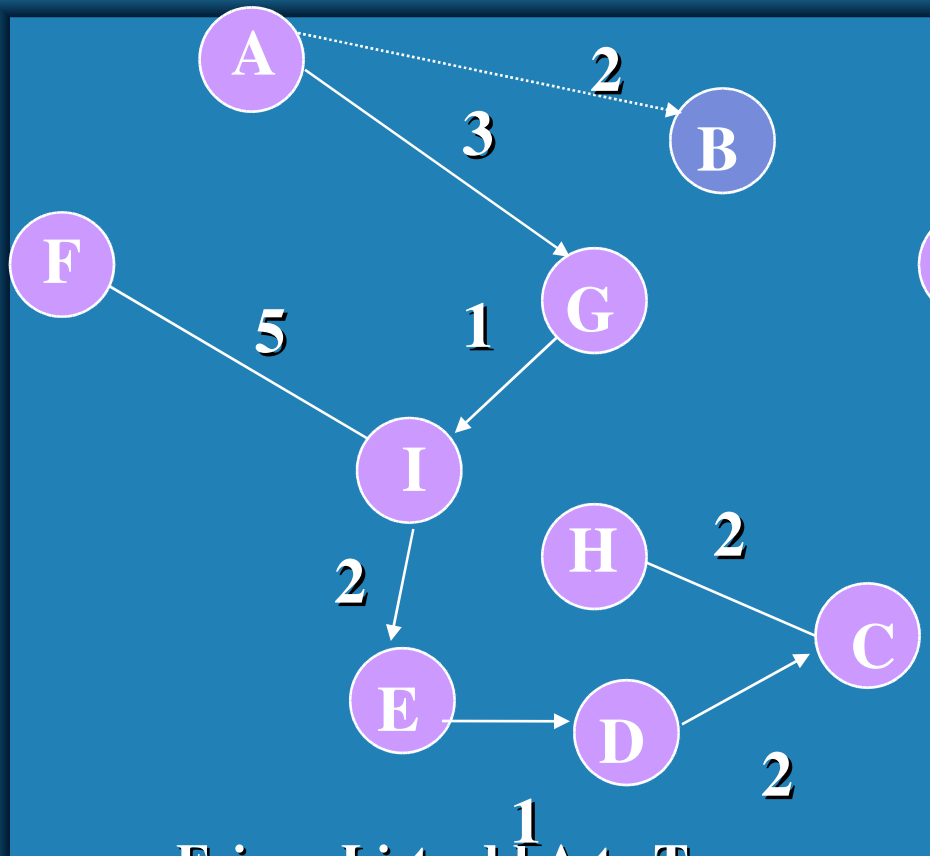
A	B	C	D	E	F	G	H	I
3	4	*	*	*	*	*	2	*



Fringe List: Add H to T

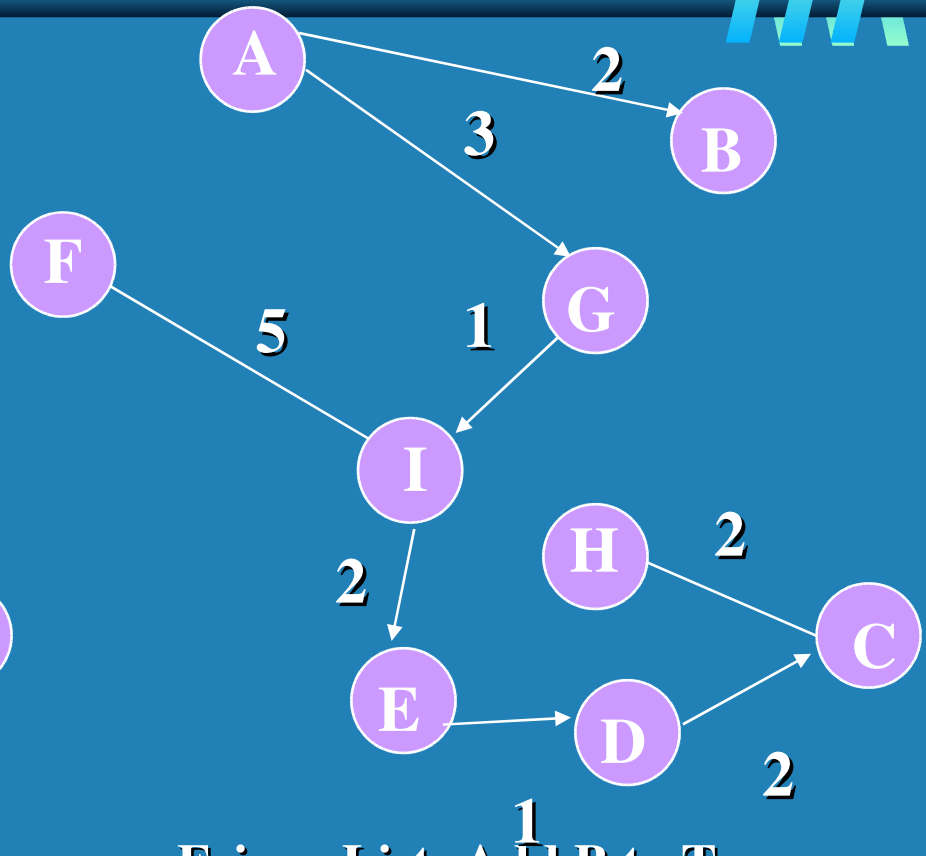
A	B	C	D	E	F	G	H	I
3	4	*	*	*	*	*	*	*

4.3 Dijkstra-Prim MST- Demo-4



Fringe List add A to T

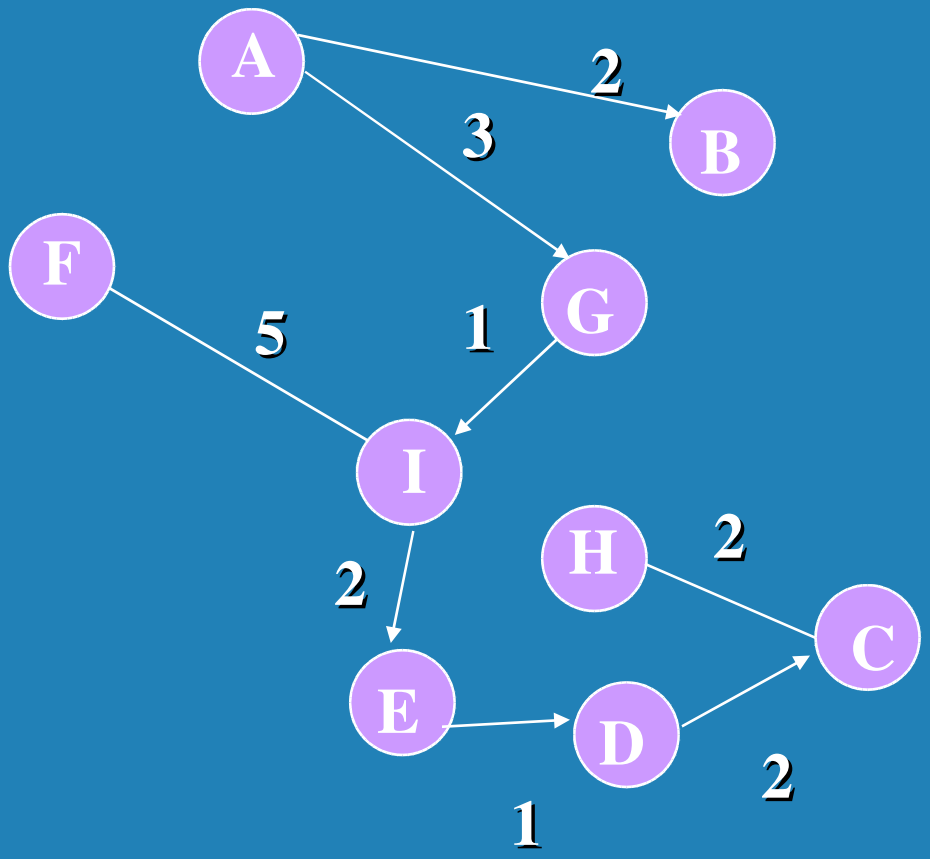
A	B	C	D	E	F	G	H	I
*	2	*	*	*	*	*	*	*



Fringe List: Add B to T

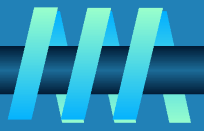
A	B	C	D	E	F	G	H	I
*	*	*	*	*	*	*	*	*

4.3 Dijkstra-Prim MST- Demo-5



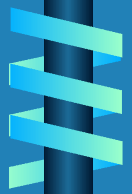
MST of weight 18

4.3 Dijkstra-Prim MST-Code



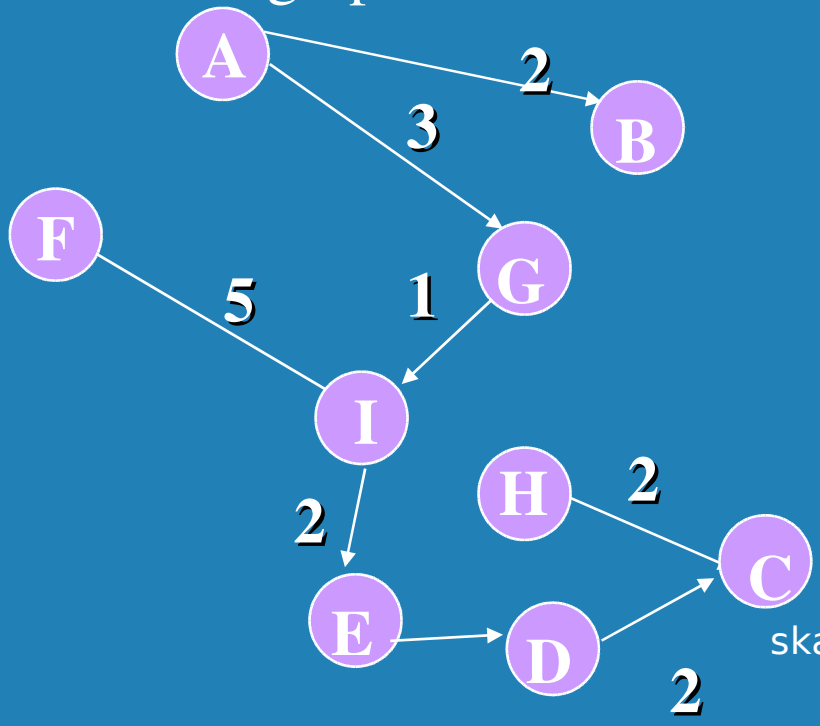
```
procedure DijkstraPrimMST(G:Graph)
Begin
      Intialize fringe[v] := inf for each vertex v.
      Add a random vertex v to the tree T;

      while (the tree has less than n vertices)
       for each vertex v' adj. to v and v'not in T,
            if fringe[v']> weight(v,v')
                 fringe[v'] := weight(v,v');
            endif
       Find vertex v that has minimum fringe[].
       Add v to the tree;
      endwhile;
end;
```

The time complexity is the time $O(m+n^2)$.[why?]



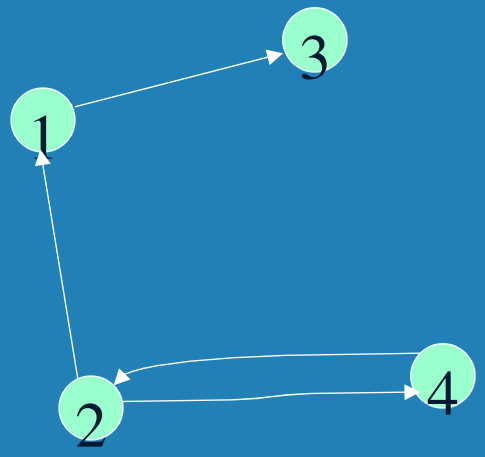
4.4 Directed Graphs

Directed Graphs are those graphs in which every edge has direction. Therefore the edge (x,y) is different from the edge (y,x). We allow at most two edges between a pair of vertices. Directed graphs may be weighted or unweighted just like undirected graphs.

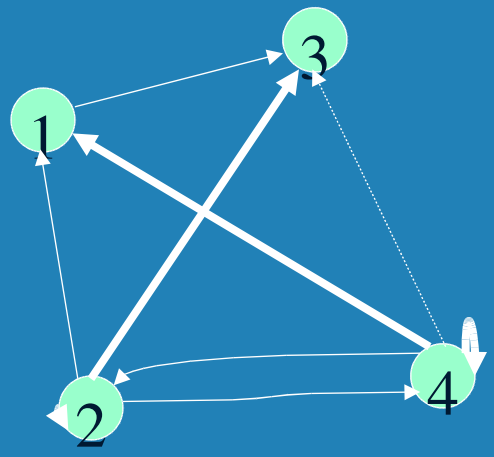


4.4 Transitive Closure

- Given a directed graphs, the transitive closure is defined as follows: If there is an edge between (x,y) and (y,z) then there is an edge (x,z).
- **Example of transitive closure:**



0 0 1 0
1 0 0 1
0 0 0 0
0 1 0 0



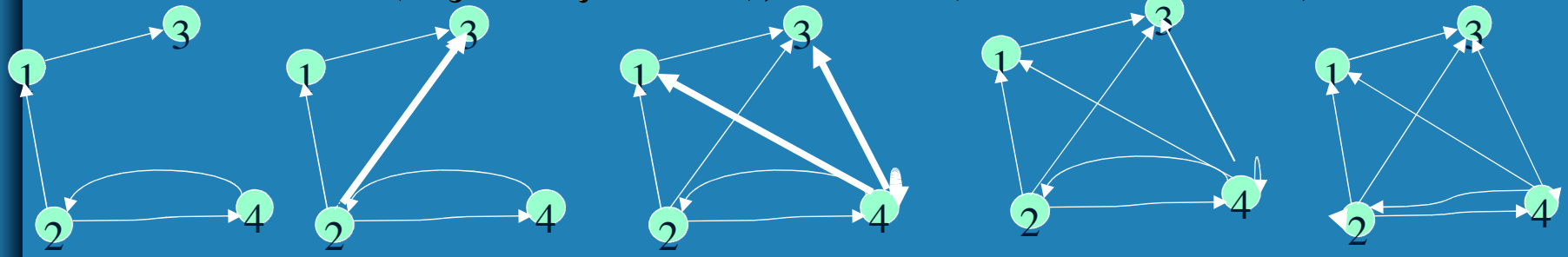
0 0 1 0
1 **1** **1** 1
0 0 0 0
1 1 **1** **1**

4.4 Warshall's Algorithm for Transitive Closure

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ where

$R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate

Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

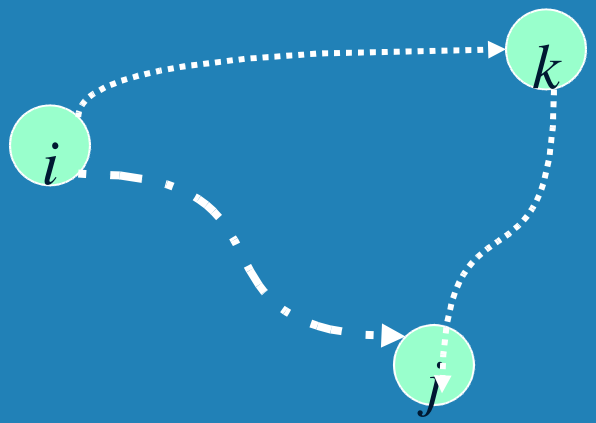


$R^{(0)}$	$R^{(1)}$	$R^{(2)}$	$R^{(3)}$	$R^{(4)}$
0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0
1 0 0 1	1 0 **1** 1	1 0 1 1	1 0 1 1	1 **1** 1 1
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
0 1 0 0	0 1 0 0	**1** 1 **1** **1**	1 1 1 1	1 1 1 1

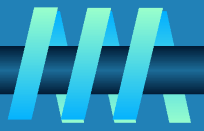
4.4 Warshall's Algorithm for Transitive Closure

On the *k*-th iteration, the algorithm determines for every pair of vertices *i, j* if a path exists from *i* and *j* with just vertices 1,…,*k* allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just 1 ,…,}k\text{-1)} \\ \text{or} & \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from i to k and from } k \text{ to } i \text{ using just 1 ,…,}k\text{-1)} \end{cases}$$



4.4 Warshall's Algorithm for Transitive Closure



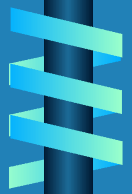
Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

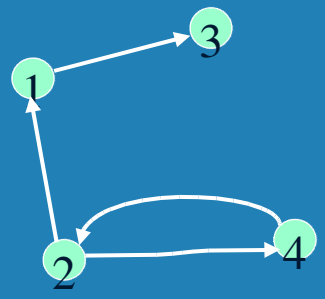
It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$

Rule 2 If an element in row i and column j 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 and the element in its column j and row k are both 1's in $R^{(k-1)}$



4.4 Warshall's Algorithm (example)



$R^{(0)} =$

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0

$R^{(1)} =$

0	0	1	0
1	0	**1**	1
0	0	0	0
0	1	0	0

$R^{(2)} =$

0	0	1	0
1	0	1	1
0	0	0	0
1	1	**1**	**1**

$R^{(3)} =$

0	0	1	0
1	0	1	1
0	0	0	0
1	1	1	1

$R^{(4)} =$

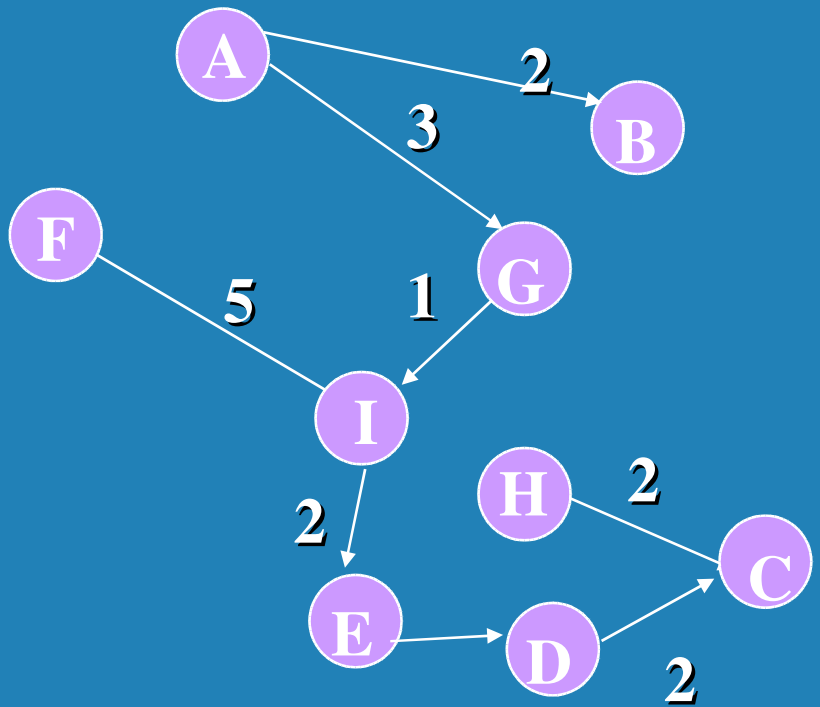
0	0	1	0
1	**1**	1	1
0	0	0	0
1	1	1	1

4.4 Warshall's Algorithm (pseudocode)

```
procedure Warshall(A[1..n][1..n]) : Nodes[1..n][1..n]
   var R:Nodes[1..n][1..n];
   var i,j, k:Index;
  begin
            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])
                    endfor
                  endfor
             endfor
                 Warshall = R;

end;
```

Time efficiency: $0(n^3)$ Space Efficiency = ?

4.5 Weighted Directed Graphs

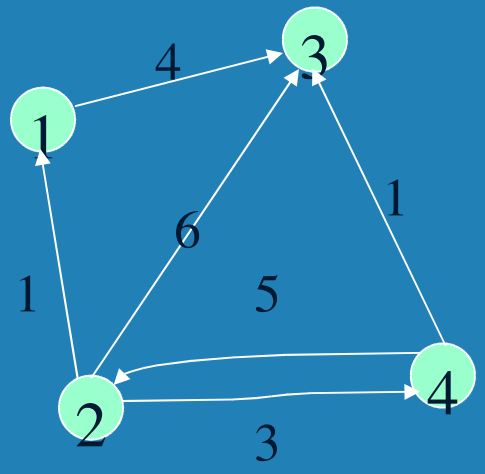


4.5 All pairs shortest paths

Problem: In a weighted directed graph, find shortest paths between every pair of vertices

Dynamic Programming construct solution through series of matrices $D^{(0)}, \ldots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

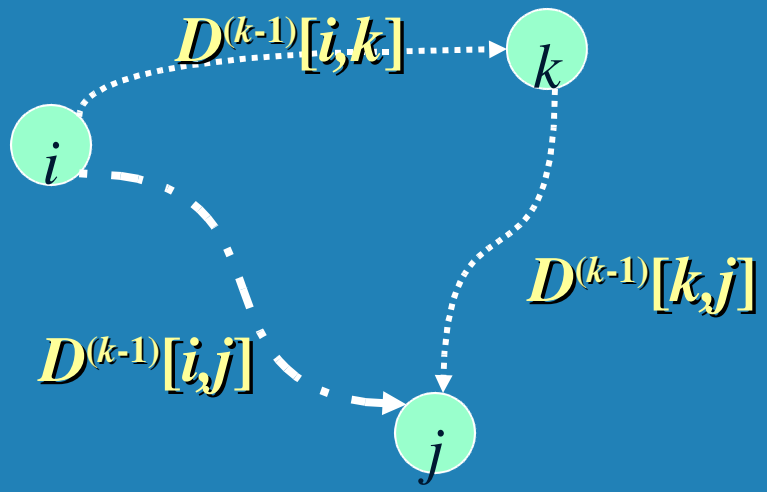
Example:



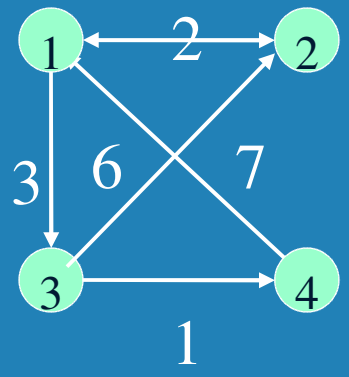
4. 5 Floyd's Algorithm

On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1,…,*k* as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



4.5 Floyd's Algorithm (example)



$D^{(0)} =$

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

$D^{(1)} =$

0	∞	3	∞
2	0	**5**	∞
∞	7	0	1
6	∞	**9**	0

$D^{(2)} =$

0	∞	3	∞
2	0	5	∞
9	7	0	1
6	∞	9	0

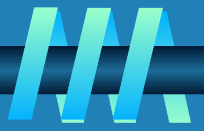
$D^{(3)} =$

0	**10**	3	**4**
2	0	5	**6**
9	7	0	1
6	**16**	9	0

$D^{(4)} =$

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

4.5 Floyd's Algorithm (pseudocode)



```
procedure Floyd(A[1..n][1..n]) : Nodes[1..n][1..n]
    var D:Nodes[1..n][1..n];
    var i,j, k:Index;
  begin
        D = A;
        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])
              endfor
           endfor
         endfor
      Floyd = D;

end;
```

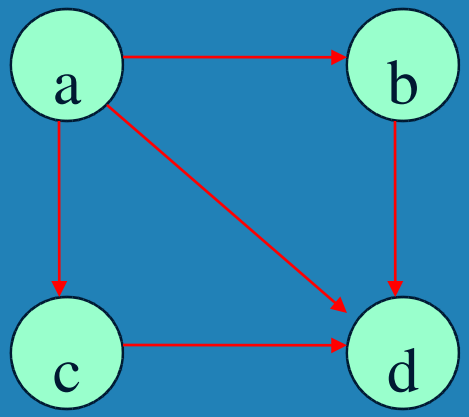
Time efficiency: $\Theta(n^3)$ Space efficiency: ?

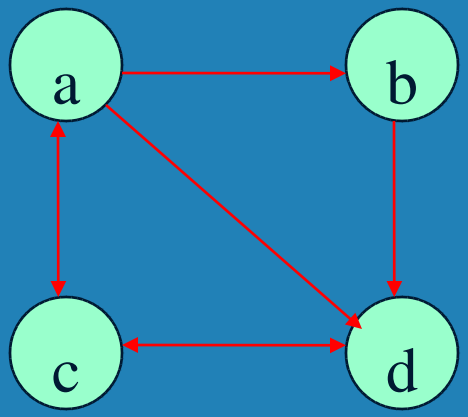
Note: Shortest paths themselves can be found, too

4.6 Directed Acyclic Graphs

A *dag*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles

a dag



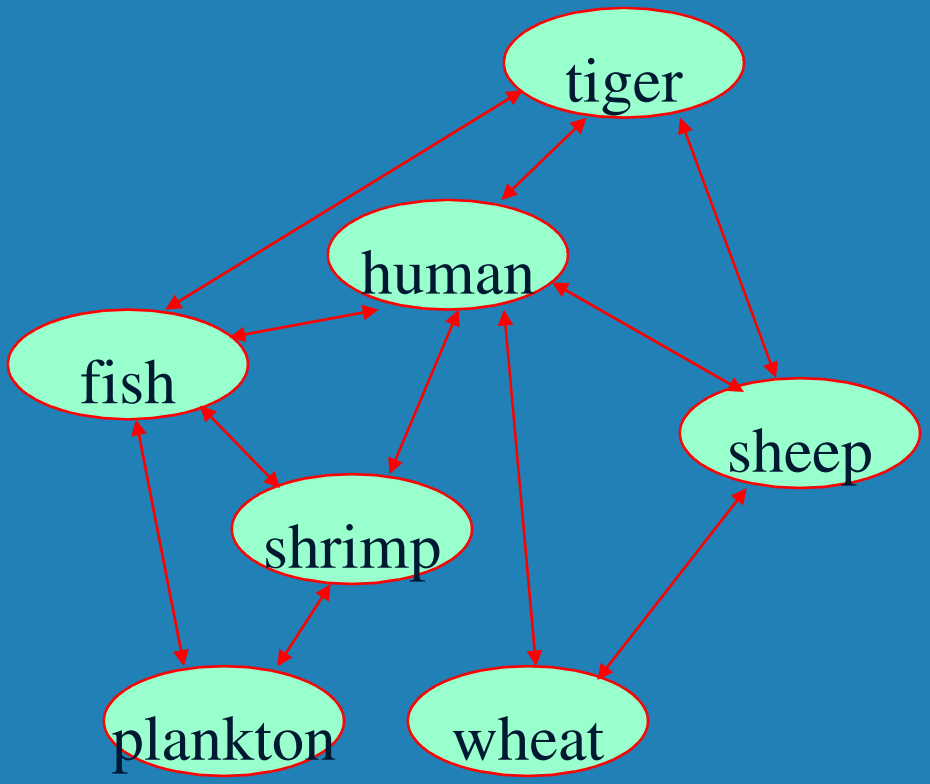


not a dag

Arise in modelling many problems that involve prerequisite constraints (construction projects, document version control)

4. 6 Topological Sorting

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting be possible. Order the following items in a food chain

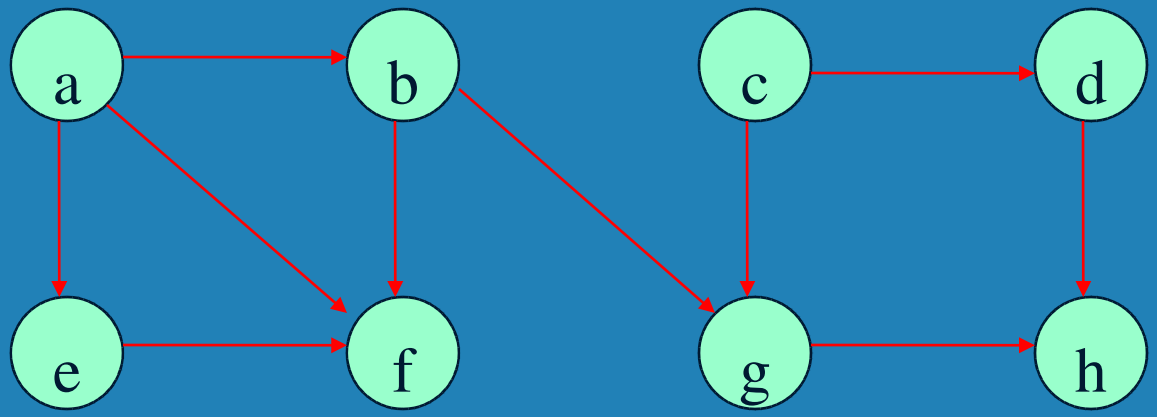


4.6 DFS-based Algorithm

DFS-based algorithm for topological sorting

- **Perform DFS traversal, noting the order vertices are popped off the traversal stack**
- **Reverse order solves topological sorting problem**
- **Back edges encountered?→ NOT a dag!**

Example:



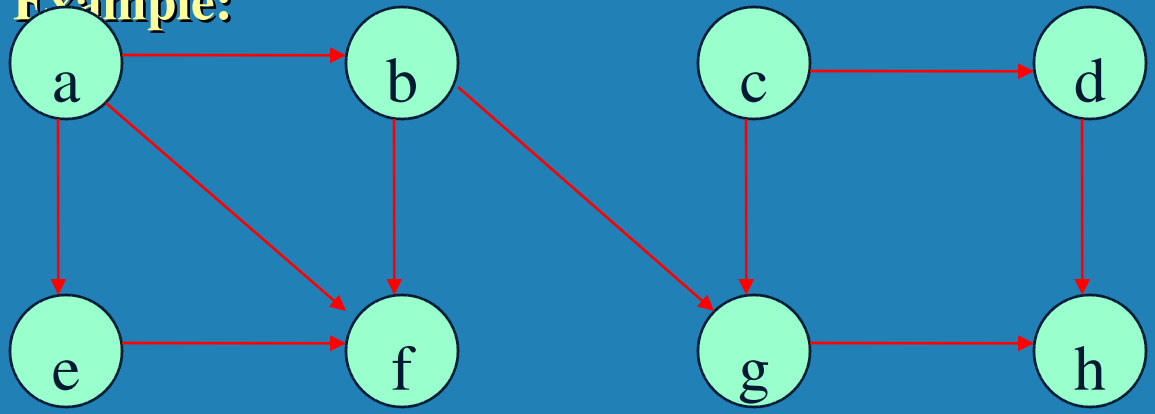
Efficiency:

4. 6 Source Removal Algorithm

Source removal algorithm

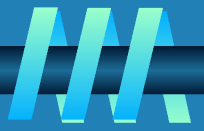
Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm

Exercises



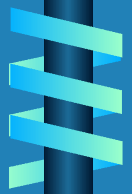
1. Write an algorithm to find a simple path between x and y, given a path between x and y in a graph G. Develop the time complexity analysis for your algorithm.

2. a)Let A be an adjacency matrix of an undirected graph. Explain what property of the matrix indicates that

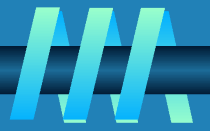
i) The graph is complete

ii) the graph has a self-loop, i.e an edge connecting a vertex to itself.

iii) The graph has an isolated vertex, i.e a vertex that has no edges incident to it.



Exercises



3. Answer the above questions if the A was a adjacency list representation.

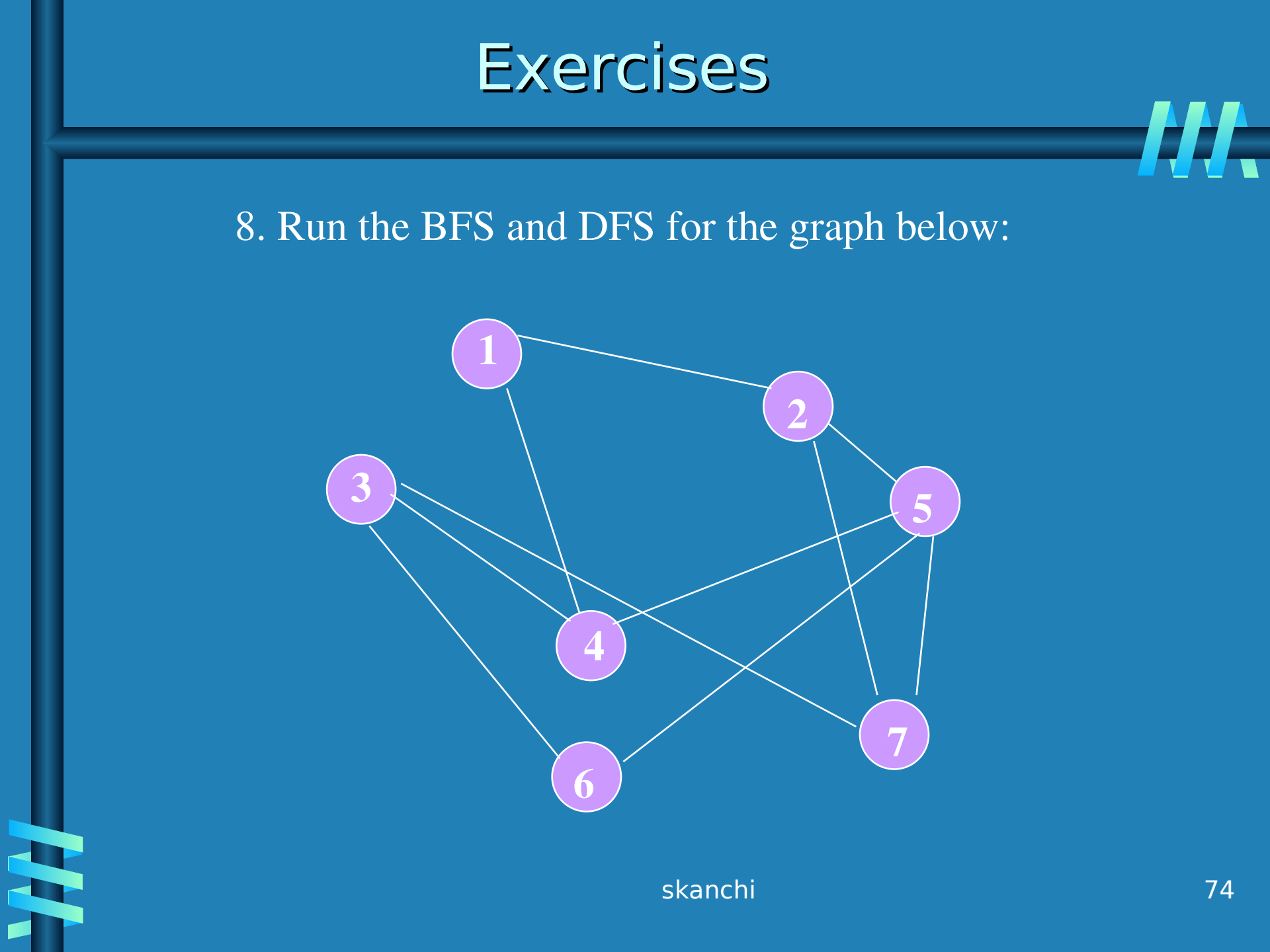
4. Write an algorithm to find a simple path between two given vertices in a rooted tree in its adjacency list representation.

5. Write an algorithm to check if a given graph is connected?

6. Write the detailed algorithm for BFS using adjacency list as the input data structure. Make sure the resulting is tree is created as a tree data structure.

7. Write the detailed algorithm for DFS using adjacency list as the input data structure. Make sure the resulting is tree is created as a tree data structure.

Exercises

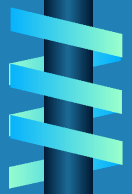


8. Run the BFS and DFS for the graph below:

Exercises

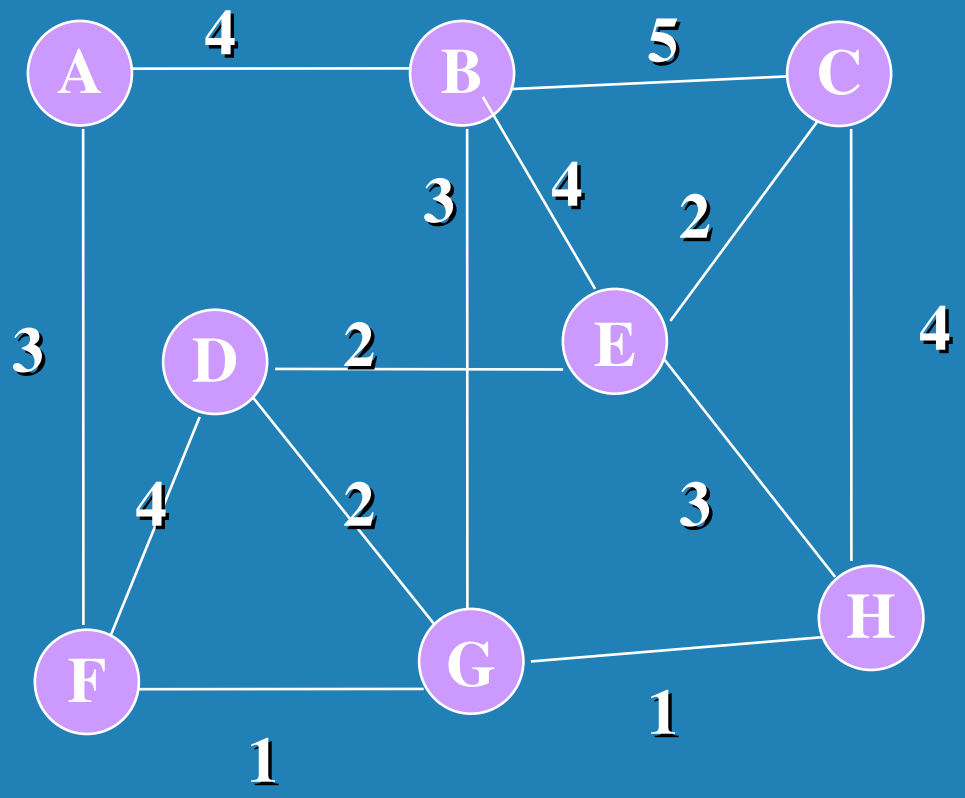
9. Write the detailed algorithm for Kruskal's MST using adjacency list the input data structure. Ensure the time complexity of the algorithm is described in the notes.

10. Write the detailed algorithm for Dijkstra-Prim's MST using adjacen list as the input data structure. Ensure the time complexity of the algorithm is as described in the notes.

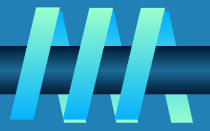


Exercise

11. Run the Kruskal's MST and Dijkstra-Prim's MST algorithm on the graph below: Show the details of each step.



Exercises



12. Write an algorithm to find Euler circuit and Euler path whenever it exists. Develop the time complexity analysis for your algorithm.

13. Consider the clique problem - given a graph G and a positive integer k, determine whether the graph contains a clique of size k, i.e a complete subgraph of k vertices. Design an exhaustive-search algorithm for this problem.

14. What is the time efficiency of the DFS-based algorithm for topological sorting.

15. How can one modify the DFS-based algorithm to avoid reversing the vertex ordering generated by the DFS?

Exercises

16. Apply the DFS based topological sorting algorithm to solve the topological sorting problem for the following digraph.

17. Apply Warshall's Dynamic Programming algorithm for finding the transitive closure of the digraph given by the following adjacency matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

18. Prove that the time efficiency of Warshall's algorithm is cubic.

Exercises

19. Solve the all-pairs shortest-path problem for digraph with the weight matrix using Floyd's Dynamic Programming technique

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 18 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

20. Give an example of a graph or digraph with negative weights for which Floyd's algorithm does not yield correct results.