Lecture 7

Graph Representation, Traversal, MST

吴蔚琪 2022.11.25

991《数据结构与算法》考纲

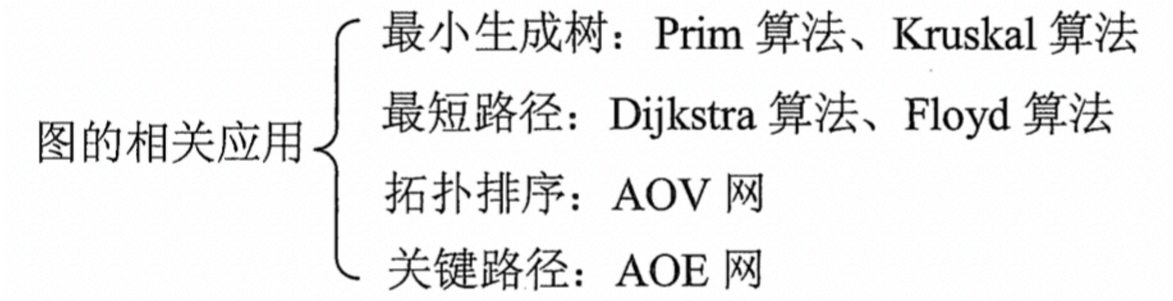
7、图

(1) 图的基本概念。

(2) 图的存储，包括邻接矩阵法、邻接表法。

(3) 图的遍历操作，包括深度优先搜索、广度优先搜索。

(4) 最小生成树，最短路径，关键路径、拓扑排序算法的原理与实现。



Graph Concepts

图的定义

- 图G由顶点集V和边集E组成，记为G=(V, E)
- V(G)表示图G中顶点的有限非空集，E(G)表示图G中顶点之间的关系（边）集合
- |V|表示图G中顶点个数，也称图G的阶
- 线性表可以是空表，树可以是空树，但图不可以是空图。即图的顶点集V一定非空，但边集E可以为空，此时图中只有顶点而没有边。

Checklist of Concepts

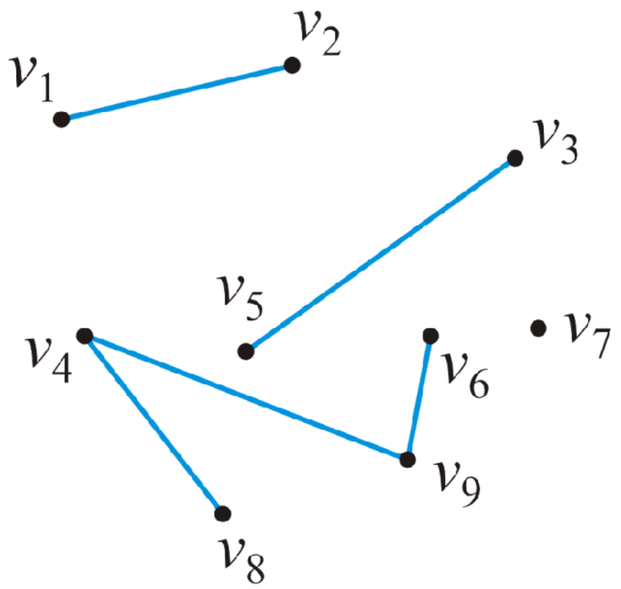
- 有向图/无向图
- 简单图/ 多重图
- 完全图
- 稠密图/稀疏图
- 带权图/无权图
- 子图
- 顶点的度、入度、出度
- 路径、回路
- 简单路径、简单回路
- 连通、连通图、连通分量（极大连通子图）
- 强连通图、强连通分量（极大强连通子图）
- 生成树、生成森林（极小连通子图）

Recap

- Undirected & directed graph

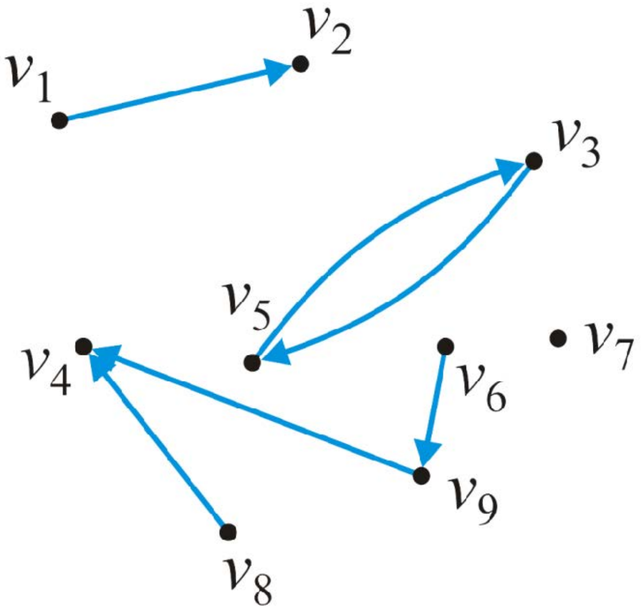
Unordered (v_j, v_k) represents edge connecting v_j and v_k

(v_j, v_k) = (v_k, v_j)



(v_j, v_k) represents connection from v_j to v_k

(v_j, v_k) is different from (v_k, v_j)



Recap

- Edge & vertex

In undirected graph, the maximum number of edges is

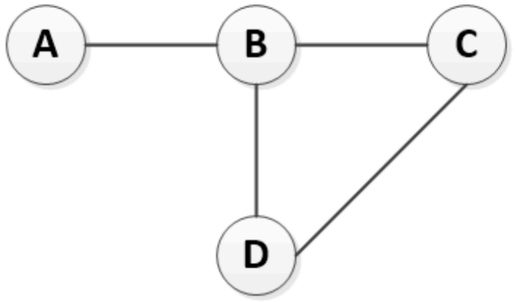
$$|E| \le \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = \mathrm{O}(|V|^2)$$

In directed graph, the maximum number of directed edges is

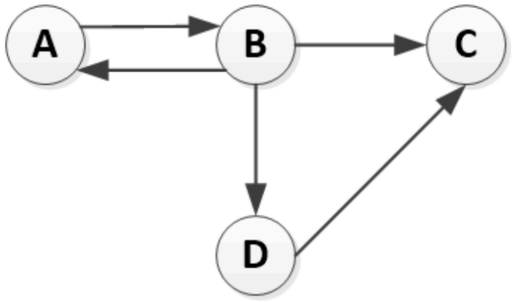
$$|E| \le 2\binom{|V|}{2} = 2\frac{|V|(|V|-1)}{2} = |V|(|V|-1) = \mathrm{O}(|V|^2)$$

Recap

- 简单图:
  - 不存在重复边
  - 不存在顶点到自身的边

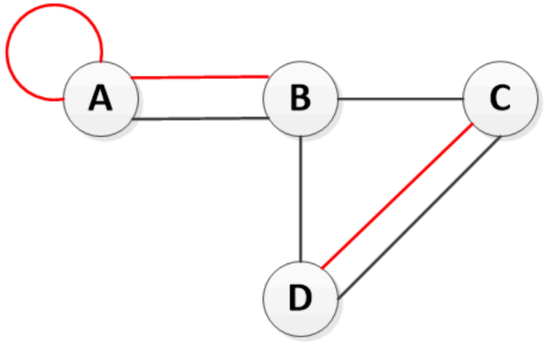


简单无向图

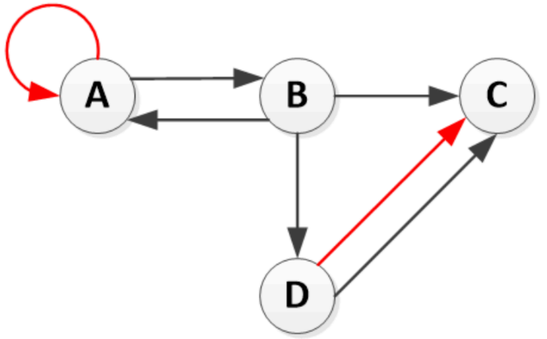


简单有向图

- 多重图
  - 允许图中某两点之间的边数多于一条
  - 允许顶点通过一条边和自己关联



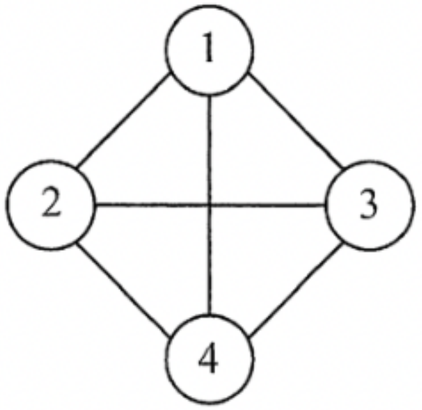
多重无向图



多重有向图

Recap

- 完全图（也称简单完全图）：
  - 无向图：有 |V|(|V|-1)/2 条边，即任意两个顶点之间都存在边



  - 有向图：有 |V|(|V|-1) 条边，即任意两个顶点之间都存在方向相反的两条有向边（弧）

Recap

- 稠密图/稀疏图
  - 有很少条边或弧（边的条数|E|远小于$|V|^2$）的图称为稀疏图（sparse graph）
  - 反之边的条数|E|接近$|V|^2$，称为稠密图（dense graph）
  - 通常以|E|=|V|log|V|作为区别稀疏图与稠密图的标准

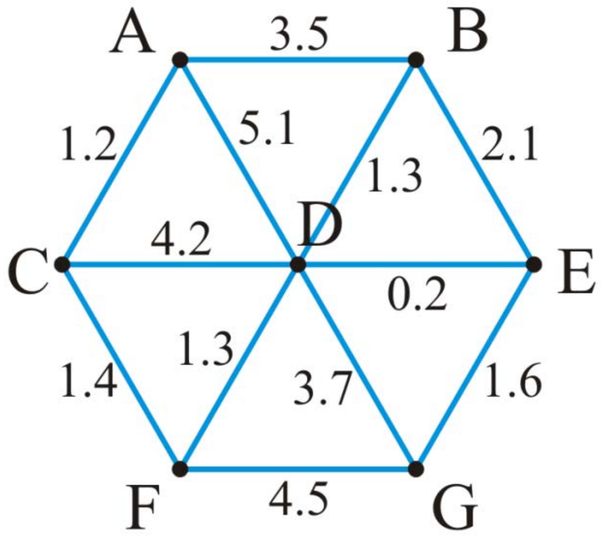
Recap

- Weighted graph

In weighted graph, each weight has a weight.

Each weight could represent distance, energy consumption, cost, etc.

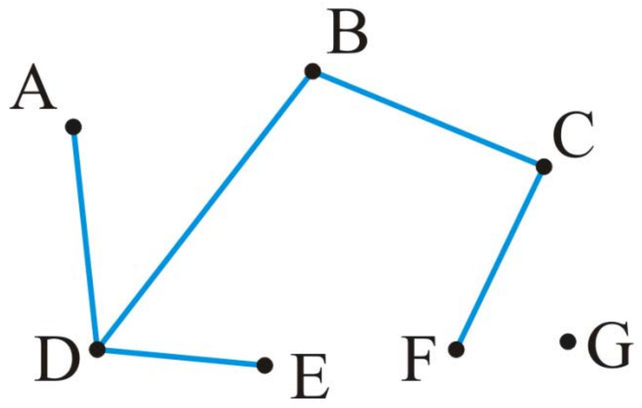
An unweighted graph can be seen as a graph with uniform weight, e.g. 1.

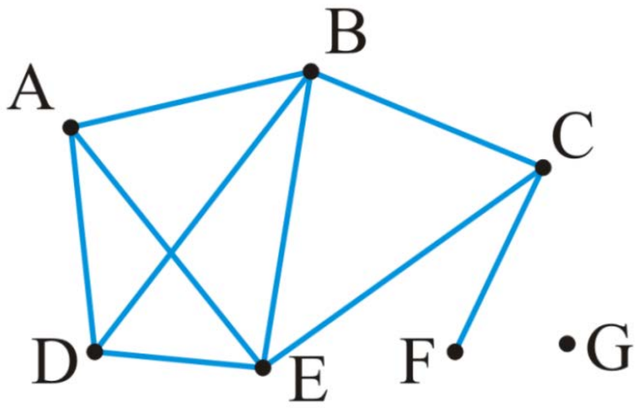


Recap

- Sub-graph

1. V': a subset of original vertices.
2. E': a subset of original edges that connect the subset of the vertices (V') in the original graph





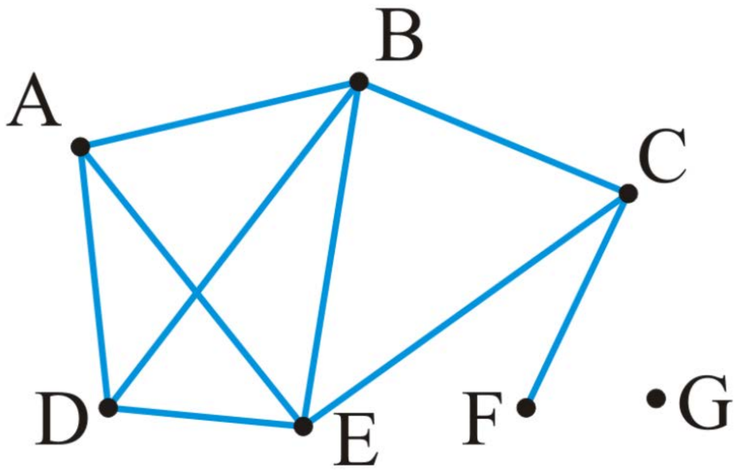
注意：并非 V 和 E 的任何子集都能构成 G 的子图，因为这样的子集可能不是图，即 E 的子集中的某些边关联的顶点可能不在这个 V 的子集中。

Recap

- Degree

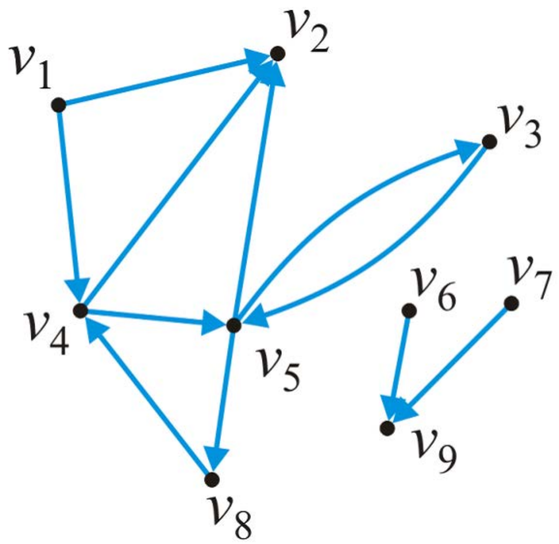
degree of a vertex: number of adjacent vertices (neighbors)

degree(A) = 3
degree(B) = 4
degree(C) = 3
… … …
degree(G) = 0



in-degree: the number of inward edges to the vertex
out-degree: the number of outward edges from the vertex

$in\_deg(v_1) = 0$
$out\_deg(v_1) = 2$
$in\_deg(v_2) = 3$
$out\_deg(v_2) = 0$
… … …

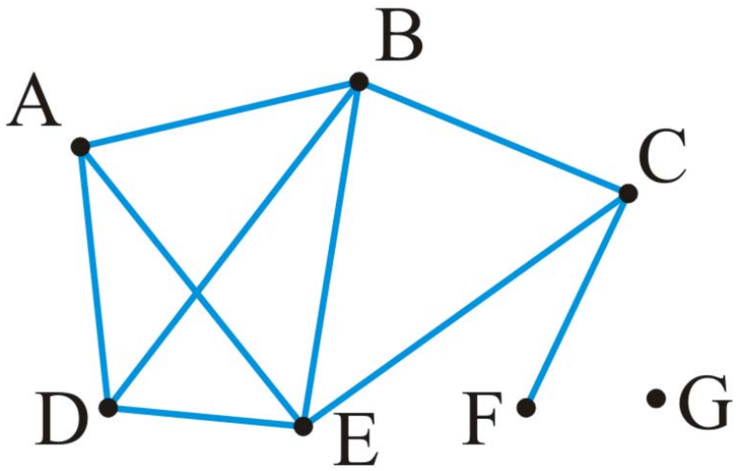


Recap

- Degree

degree of a vertex: number of adjacent vertices (neighbors)

degree(A) = 3
degree(B) = 4
degree(C) = 3
… … …
degree(G) = 0

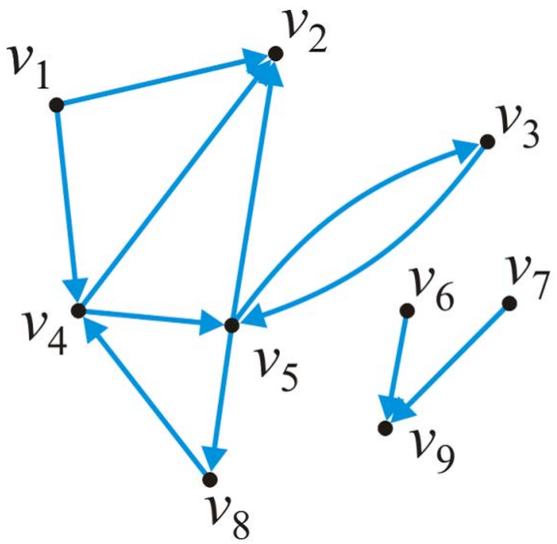


Sources: vertices with an in-degree of zero

Sinks: vertices with an out-degree of zero

Sources: v_1 , v_6 , v_7

Sinks: v_2 , v_9



对于无向图，顶点 v 的度是指依附于该顶点的边的条数，记为 $\mathrm{TD}(v)$。

在具有 n 个顶点、e 条边的无向图中，$\sum_{i=1}^{n}\mathrm{TD}(v_i)=2e$，即无向图的全部顶点的度的和等于边数的 2 倍，因为每条边和两个顶点相关联。

对于有向图，顶点 v 的度分为入度和出度，入度是以顶点 v 为终点的有向边的数目，记为 $\mathrm{ID}(v)$；而出度是以顶点 v 为起点的有向边的数目，记为 $\mathrm{OD}(v)$。顶点 v 的度等于其入度和出度之和，即 $\mathrm{TD}(v)=\mathrm{ID}(v)+\mathrm{OD}(v)$。

在具有 n 个顶点、e 条边的有向图中，$\sum_{i=1}^{n}\mathrm{ID}(v_i)=\sum_{i=1}^{n}\mathrm{OD}(v_i)=e$，即有向图的全部顶点的入度之和与出度之和相等，并且等于边数。这是因为每条有向边都有一个起点和终点。

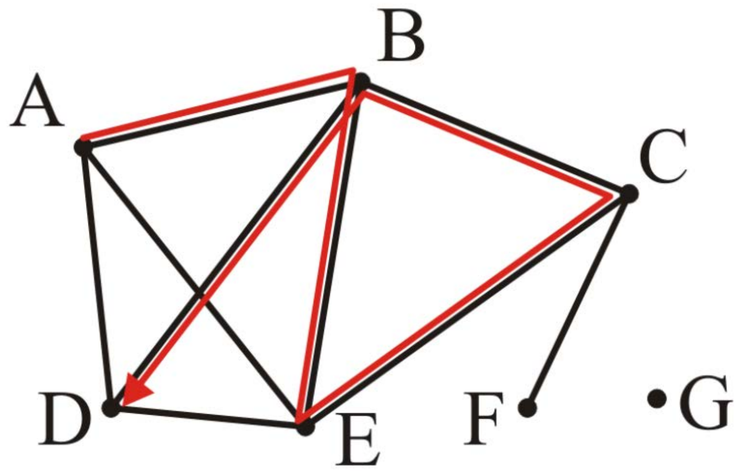
Recap

- Path

A path in an undirected graph is an ordered sequence of vertices (v_0, v_1, v_2, …, v_k) where $\{v_{j-1}, v_j\}$ is an edge for j = 1, …, k.

Represents a path from v_0 to v_k.

(A, B, C, E, B, D)

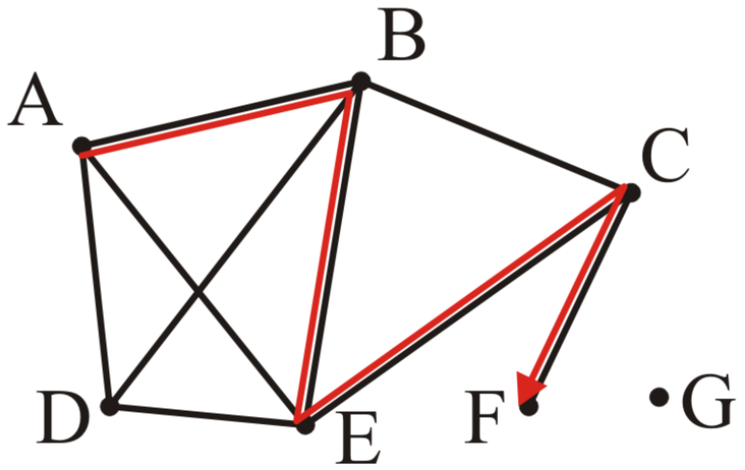


Recap

- Simple path

A simple path has no repetitions of vertices (other than perhaps the first and last vertices)

(A, B, E, C, F)

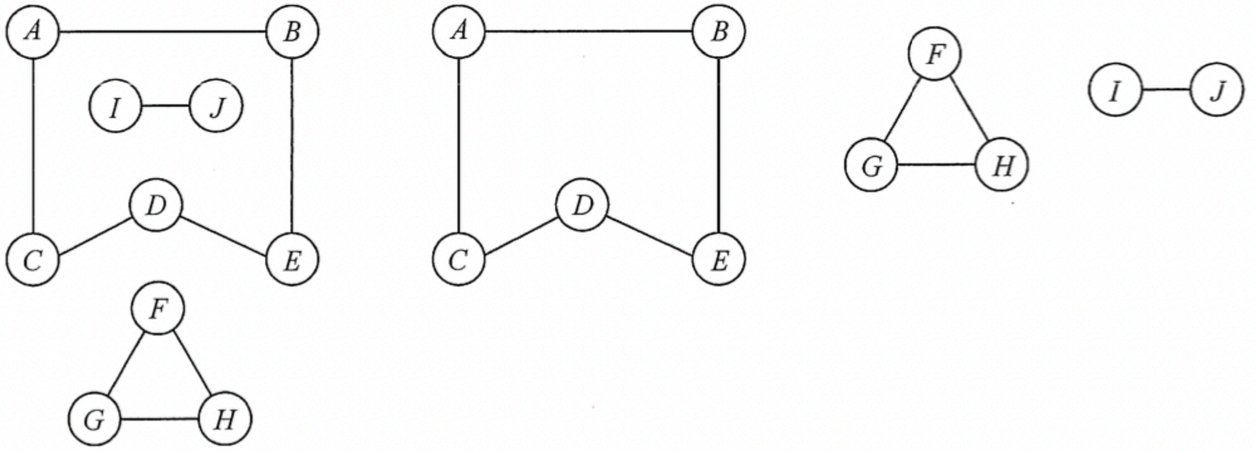


(A, B, E, C, F)

Recap

- 连通、连通图、连通分量

在无向图中，若从顶点 v 到顶点 w 有路径存在，则称 v 和 w 是连通的。若图 G 中任意两个顶点都是连通的，则称图 G 为连通图，否则称为非连通图。无向图中的极大连通子图称为连通分量。若一个图有 n 个顶点，并且边数小于 $n-1$，则此图必是非连通图。如图 6.2(a)所示，图 G_4 有 3 个连通分量，如图 6.2(b)所示。



(a) 无向图G_4　　(b) G_4的三个连通分量

图 6.2　无向图及其连通分量

Recap

- 连通、连通图、连通分量
  - 极大连通子图
    - 连通图只有一个极大连通子图，就是它本身（是唯一的）
    - 非连通图有多个极大连通子图（非连通图的极大连通子图叫做连通分量，每个分量都是一个连通图）
    - 称为极大是因为如果此时加入任何一个不在图的点集中的点都会导致它不再连通

Recap

- 强连通图、强连通分量

在有向图中，若从顶点 v 到顶点 w 和从顶点 w 到顶点 v 之间都有路径，则称这两个顶点是强连通的。若图中任何一对顶点都是强连通的，则称此图为强连通图。有向图中的极大强连通子图称为有向图的强连通分量，图 G_1 的强连通分量如图 6.3 所示。

注意：强连通图、强连通分量只是针对有向图而言的。一般在无向图中讨论连通性，在有向图中考虑强连通性。

图 6.3　图 G_1 的强连通分量

(a) 有向图G_1

Recap

- Graph & tree

A graph is a tree if it is connected and there is a unique path between any two vertices.

i.e. No cycle in the graph

-> convert to a tree:

1. Choosing any vertex to be the root
2. Defining its neighboring vertices as its children

and then recursively defining:

3. All neighboring vertices other than that one designated its parent to be its children

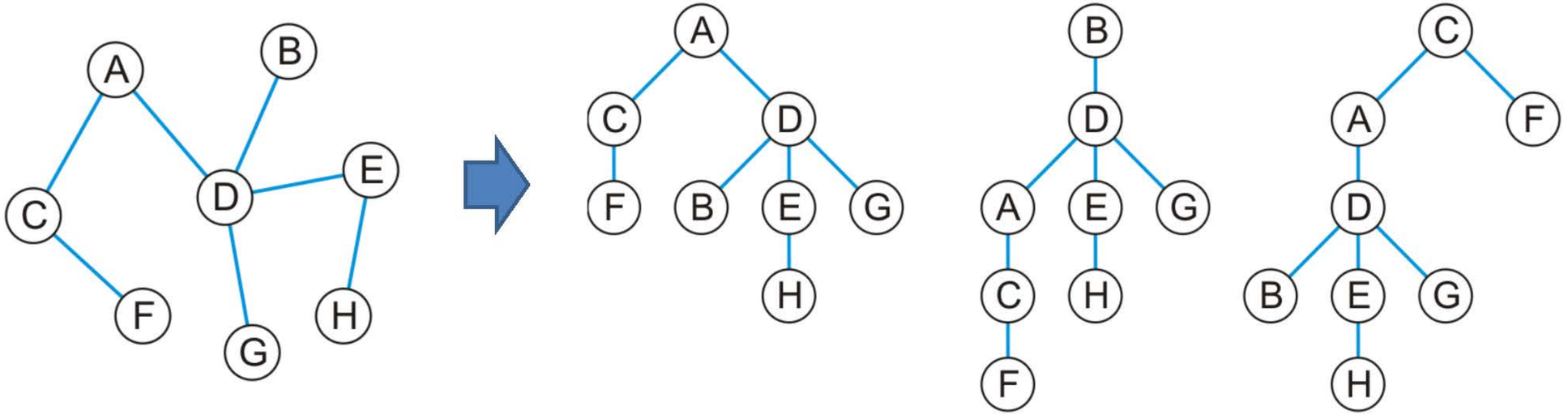
Recap

- Graph & tree

1. Choosing any vertex to be the root
2. Defining its neighboring vertices as its children

and then recursively defining:

3. All neighboring vertices other than that one designated its parent to be its children



Recap

- 生成树
  - 连通图的生成树是包含了图中全部顶点的一个极小连通子图
    - 极小连通子图
      - 即保持图连通又使得边数最少的子图
      - 如果删除一条边就非连通，就无法构成生成树
    - 同一个连通图可以有不同的生成树
    - 若图中顶点为n,则生成树含有n-1条边
- 生成森林
  - 非连通图中，连通分量的生成树构成了非连通图的生成森林

3. Given an undirected graph with n nodes and m edges, select the correct statement(s) below.

（ ）

A. $m = n - 1$ if the graph is a tree

B. $n = m - 1$ if the graph is a tree

C. $m < n$ if the graph is a forest

D. $n < m$ if the graph is a forest

E. None of the above

给定一个具有 n 个节点和 m 条边的无向图，选择下列正确的称述？（ ）

A. 如果该图是一棵树，那么 $m = n - 1$

B. 如果该图是一棵树，那么 $n = m - 1$

C. 如果该图是一个森林，那么 $m < n$

D. 如果该图是一个森林，那么 $n < m$

E. 以上都不正确

Graph Representation

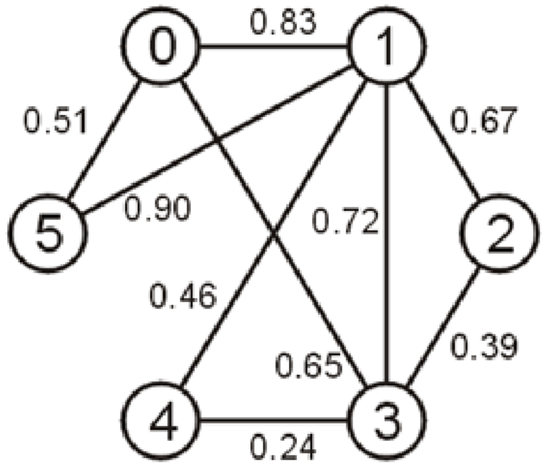
Adjacency Matrix

Undirected graph:

If the vertices v_i and v_j are connected with weight w, then set a_{ij} = w and a_{ji} = w

The matrix is symmetric

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



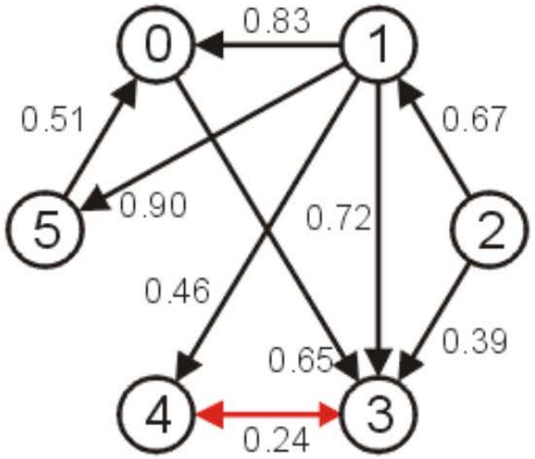
Adjacency Matrix

directed graph:

If the edge (v_i, v_j) has weight w, then set $a_{ij} = w$

If the graph was directed, then the matrix would not necessarily be symmetric

	0	1	2	3	4	5
0				0.65		
1	0.83			0.72	0.46	0.90
2		0.67		0.39		
3					0.24	
4				0.24		
5	0.51					



图的邻接矩阵存储结构定义如下：

```
#define MaxVertexNum 100                                    //顶点数目的最大值
typedef char VertexType;                                    //顶点的数据类型
typedef int EdgeType;                                       //带权图中边上权值的数据类型
typedef struct{
    VertexType Vex[MaxVertexNum];                           //顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum]; //邻接矩阵，边表
    int vexnum,arcnum;                                      //图的当前顶点数和弧数
}MGraph;
```

图的邻接矩阵存储表示法具有以下特点：

① 无向图的邻接矩阵一定是一个对称矩阵（并且唯一）。因此，在实际存储邻接矩阵时只需存储上（或下）三角矩阵的元素。

② 对于无向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的度 $TD(v_i)$。

③ 对于有向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的出度 $OD(v_i)$［或入度 $ID(v_i)$］。

④ 用邻接矩阵法存储图，很容易确定图中任意两个顶点之间是否有边相连。但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。

⑤ 稠密图适合使用邻接矩阵的存储表示。

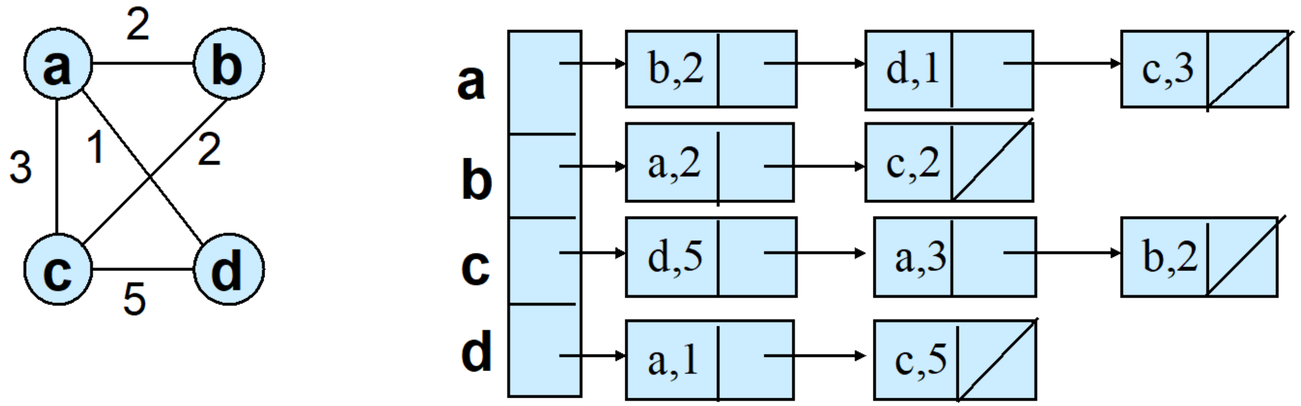
⑥ 设图 G 的邻接矩阵为 A，A^n 的元素 $A^n[i][j]$等于由顶点 i 到顶点 j 的长度为 n 的路径的数目。该结论了解即可，证明方法请参考离散数学教材。

https://www.zhihu.com/question/556030443/answer/2696996661

Adjacency List

Undirected graph: use an array of linked lists to store edges

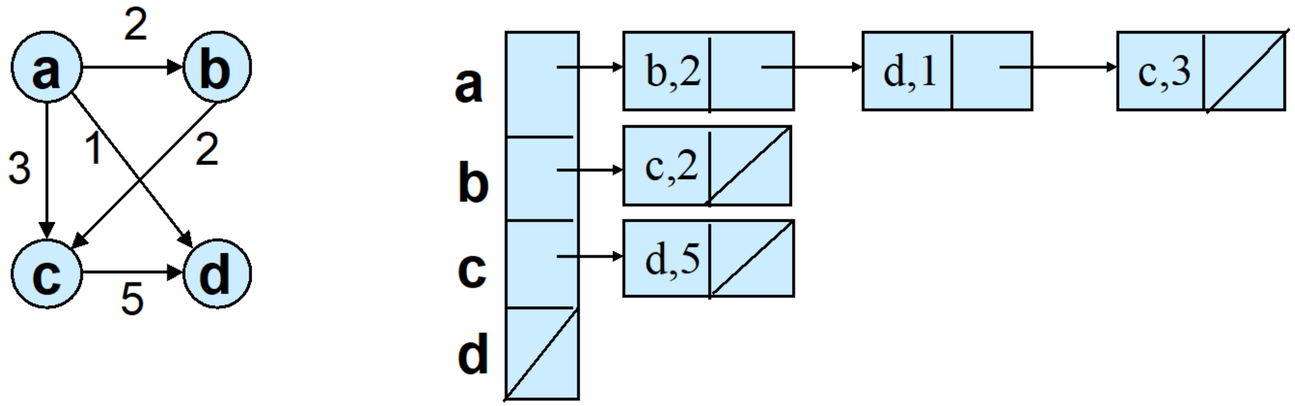
- Each vertex has a linked list that stores all the edges connected to the vertex.
- Each node in a linked list must store two items of information: the connecting vertex and the weight.



Adjacency List

Undirected graph: use an array of linked lists to store edges

- Each vertex has a linked list that stores all the edges originated from the vertex
- Each node in a linked list stores two items of information: the vertex that the edge connects to, the weight



图的邻接表存储方法具有以下特点：

① 若 G 为无向图，则所需的存储空间为 $O(|V|+2|E|)$；若 G 为有向图，则所需的存储空间为 $O(|V|+|E|)$。前者的倍数 2 是由于无向图中，每条边在邻接表中出现了两次。

② 对于稀疏图，采用邻接表表示将极大地节省存储空间。

③ 在邻接表中，给定一顶点，能很容易地找出它的所有邻边，因为只需要读取它的邻接表。在邻接矩阵中，相同的操作则需要扫描一行，花费的时间为 $O(n)$。但是，若要确定给定的两个顶点间是否存在边，则在邻接矩阵中可以立刻查到，而在邻接表中则需要在相应结点对应的边表中查找另一结点，效率较低。

④ 在有向图的邻接表表示中，求一个给定顶点的出度只需计算其邻接表中的结点个数；但求其顶点的入度则需要遍历全部的邻接表。因此，也有人采用逆邻接表的存储方式来加速求解给定顶点的入度。当然，这实际上与邻接表存储方式是类似的。

⑤ 图的邻接表表示并不唯一，因为在每个顶点对应的单链表中，各边结点的链接次序可以是任意的，它取决于建立邻接表的算法及边的输入次序。

6. The space complexity of an adjacency-matrix representation of a graph is independent of the number of edges in the graph. （ ）

图的邻接矩阵表示的空间复杂度与图的边数无关。（ ）

Graph Traversal

Outline

- Objective
  - Understand the algorithm of BFS, DFS in graph.
  - Understand their classical applications:

    Determining Connections

    Determining Distances (in an unweighted graph)

    Bipartite Graphs

Outline

- Graph traversal
  - The only difference with tree traversal using BFS and DFS:
    - To avoid visiting a vertex for multiple times, we have to track which vertices have already been visited.
    - We may have an indicator variable in each vertex or with other approaches.

BFS

- Algorithm:
  - – Choose any vertex, mark it as visited and push it onto queue
    - – While the queue is not empty:
      - Pop the top vertex v from the queue
      - For each vertex adjacent to v that has not been visited:
        - – Mark it visited, and
        - – Push it onto the queue

广度优先搜索算法的伪代码如下：

```
bool visited[MAX_VERTEX_NUM];   //访问标记数组
void BFSTraverse(Graph G){      //对图 G 进行广度优先遍历
    for(i=0;i<G.vexnum;++i)
        visited[i]=FALSE;       //访问标记数组初始化
    InitQueue(Q);               //初始化辅助队列 Q
    for(i=0;i<G.vexnum;++i)     //从 0 号顶点开始遍历
        if(!visited[i])         //对每个连通分量调用一次 BFS
            BFS(G,i);           //vi 未访问过，从 vi 开始 BFS
}
void BFS(Graph G,int v){        //从顶点 v 出发，广度优先遍历图 G
    visit(v);                   //访问初始顶点 v
    visited[v]=TRUE;            //对 v 做已访问标记
    Enqueue(Q,v);               //顶点 v 入队列 Q
    while(!isEmpty(Q)){
        DeQueue(Q,v);           //顶点 v 出队列
        for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
                                //检测 v 所有邻接点
            if(!visited[w]){    //w 为 v 的尚未访问的邻接顶点
                visit(w);       //访问顶点 w
                visited[w]=TRUE;//对 w 做已访问标记
                EnQueue(Q,w);   //顶点 w 入队列
            }//if
    }//while
}
```

1．BFS 算法的性能分析

无论是邻接表还是邻接矩阵的存储方式，BFS 算法都需要借助一个辅助队列 Q，n 个顶点均需入队一次，在最坏的情况下，空间复杂度为 $O(|V|)$。

采用邻接表存储方式时，每个顶点均需搜索一次（或入队一次），故时间复杂度为 $O(|V|)$，在搜索任一顶点的邻接点时，每条边至少访问一次，故时间复杂度为 $O(|E|)$，算法总的时间复杂度为 $O(|V|+|E|)$。采用邻接矩阵存储方式时，查找每个顶点的邻接点所需的时间为 $O(|V|)$，故算法总的时间复杂度为 $O(|V|^2)$。

DFS

- Algorithm:
  - – Choose any vertex, mark it as visited and push it onto stack
    - – From that vertex:
      - If there is another adjacent vertex not yet visited, go to it
      - Otherwise, go back to the previous vertex
    - – Continue until no visited vertices have unvisited adjacent vertices

```
bool visited[MAX_VERTEX_NUM];   //访问标记数组
void DFSTraverse(Graph G){      //对图 G 进行深度优先遍历
    for(v=0;v<G.vexnum;++v)
        visited[v]=FALSE;       //初始化已访问标记数据
    for(v=0;v<G.vexnum;++v)     //本代码中是从 v=0 开始遍历
        if(!visited[v])
            DFS(G,v);
}
void DFS(Graph G,int v){        //从顶点 v 出发，深度优先遍历图 G

    visit(v);                   //访问顶点 v
    visited[v]=TRUE;            //设已访问标记
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighor(G,v,w))
        if(!visited[w]){        //w 为 u 的尚未访问的邻接顶点
            DFS(G,w);
        }  //if
}
```

1．DFS 算法的性能分析

DFS 算法是一个递归算法，需要借助一个递归工作栈，故其空间复杂度为 $O(|V|)$。

遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于所用的存储结构。以邻接矩阵表示时，查找每个顶点的邻接点所需的时间为 $O(|V|)$，故总的时间复杂度为 $O(|V|^2)$。以邻接表表示时，查找所有顶点的邻接点所需的时间为 $O(|E|)$，访问顶点所需的时间为 $O(|V|)$，此时，总的时间复杂度为 $O(|V|+|E|)$。

Connected

First, let us determine whether one vertex is connected to another

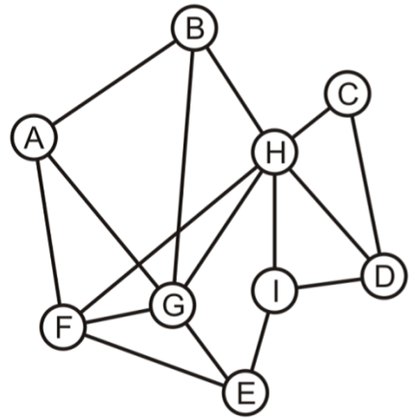
- v_j is connected to v_k if there is a path from the first to the second

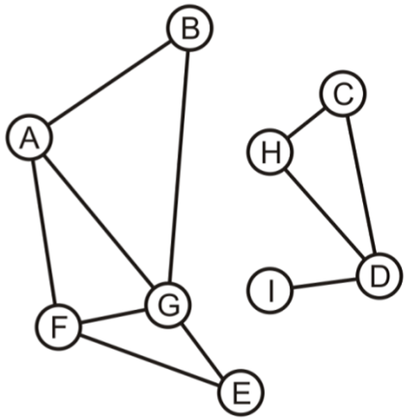
Strategy:

- Perform a breadth-first traversal starting at v_j
- If the vertex v_k is ever found during the traversal, return true
- Otherwise, return false

Determining Connections

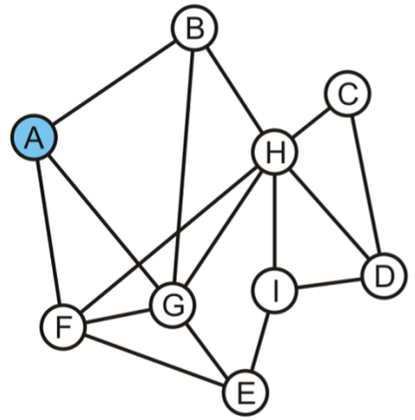
Is A connected to D?

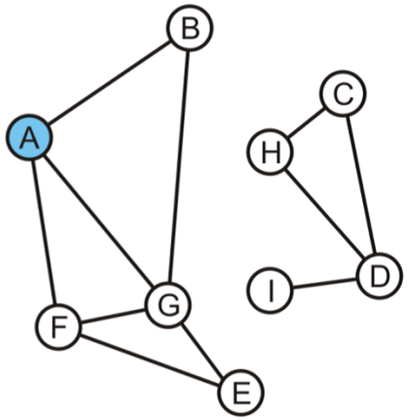




Determining Connections

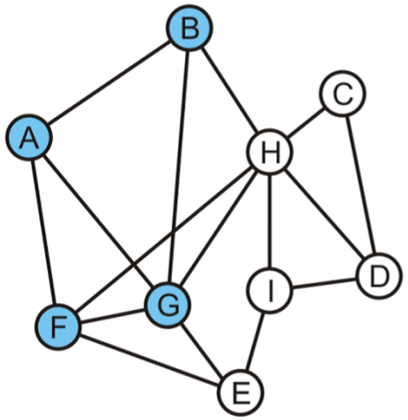
Vertex A is marked as visited and pushed onto the queue



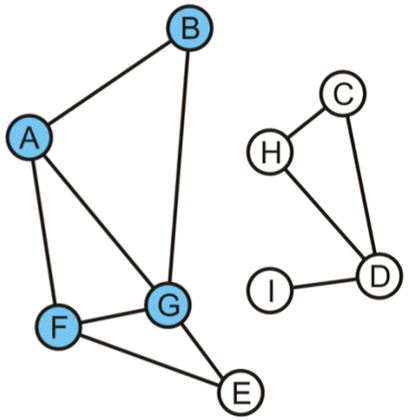


Determining Connections

Pop the head, A, and mark and push B, F and G



B	F	G			

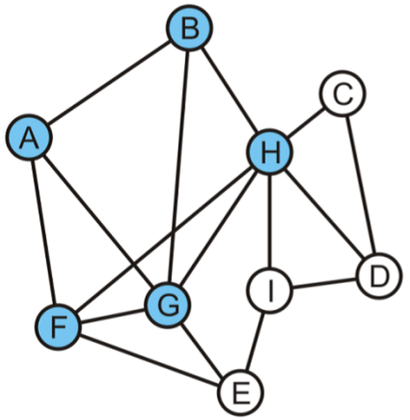


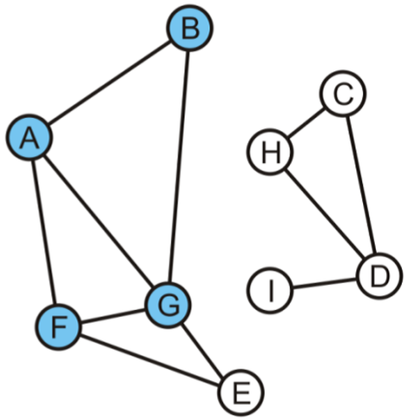
B	F	G			

Determining Connections

Pop B and mark and, in the left graph, mark and push H

- On the right graph, B has no unvisited adjacent vertices



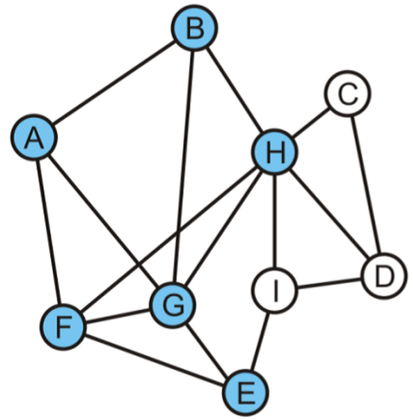


F	G	H			

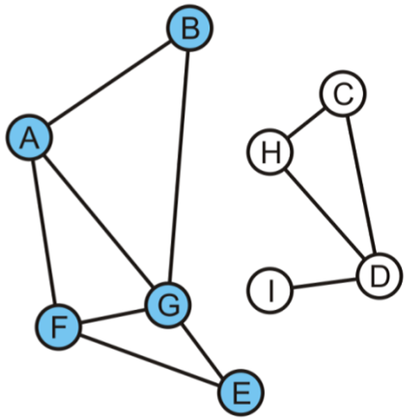
F	G				

Determining Connections

Popping F results in the pushing of E



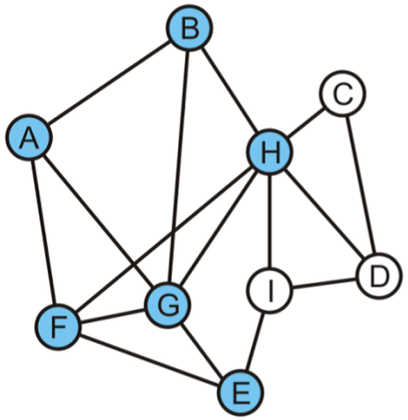
G	H	E			



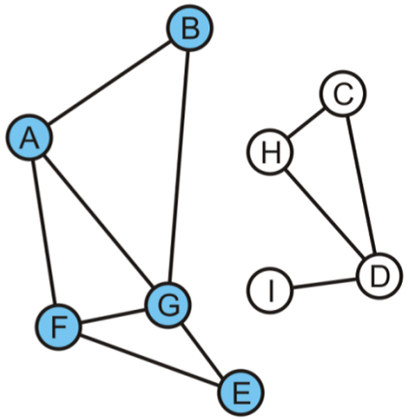
G	E				

Determining Connections

In either graph, G has no adjacent vertices that are unvisited



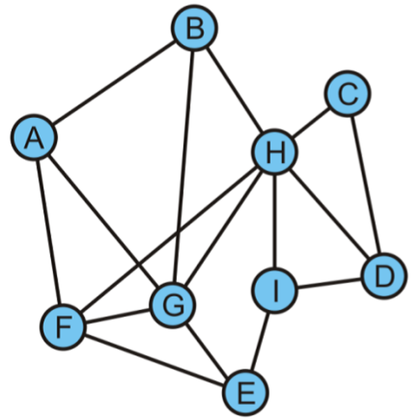
H	E				

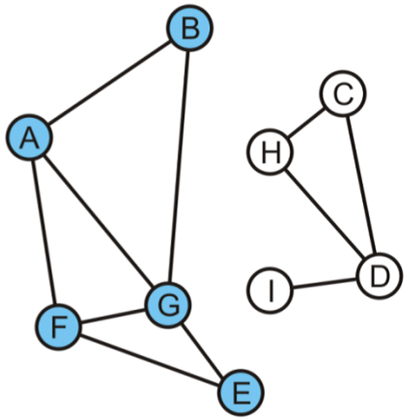


E					

Determining Connections

Popping H on the left graph results in C, I, D being pushed



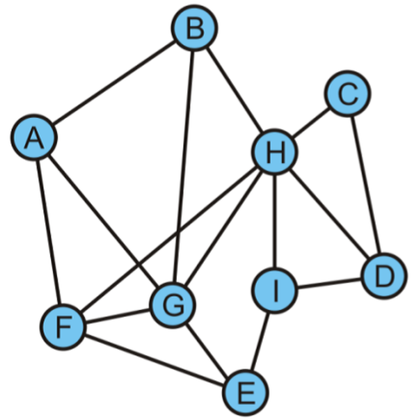


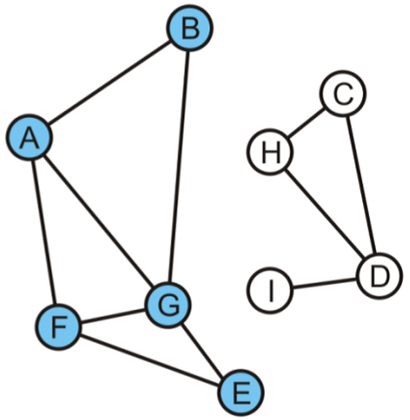
E	C	I	D		

Determining Connections

On the left, D is now visited

- We determine A is connected to D



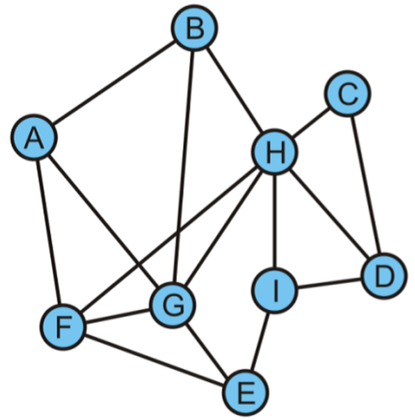


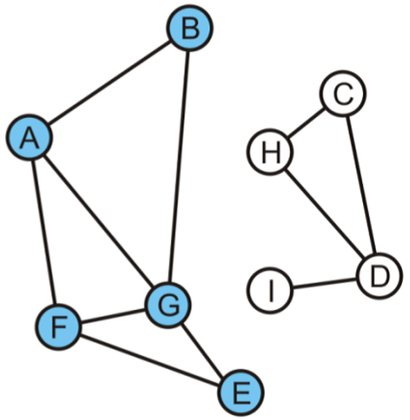
E	C	I	D		

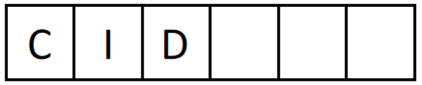
Determining Connections

On the right, the queue is empty and D is not visited

- We determine A is not connected to D







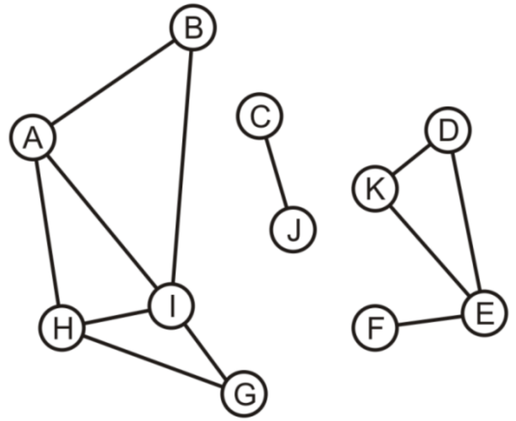
Connected Components

- Suppose we want to partition the vertices into connected sub-graphs
  - While there are unvisited vertices in the tree:
    - Select an unvisited vertex and perform a traversal on that vertex
    - Each vertex that is visited in that traversal is added to the set initially containing the initial unvisited vertex
  - Continue until all vertices are visited

- We would use a disjoint set data structure for maximum efficiency

Connected Components

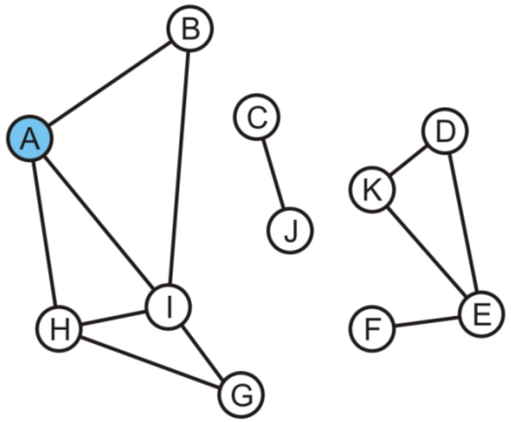
Here we start with a set of singletons



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

Connected Components

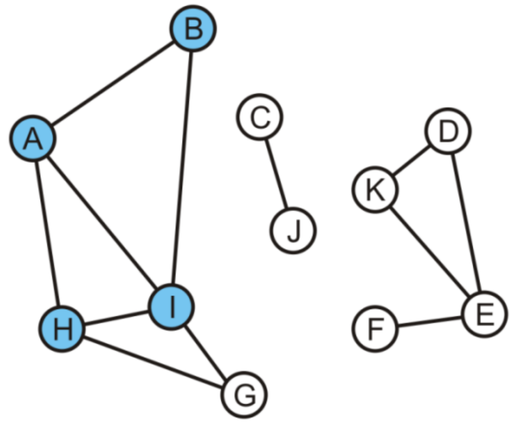
The vertex A is unvisited, so we start with it



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

Connected Components

Take the union of with its adjacent vertices: {A, B, H, I}

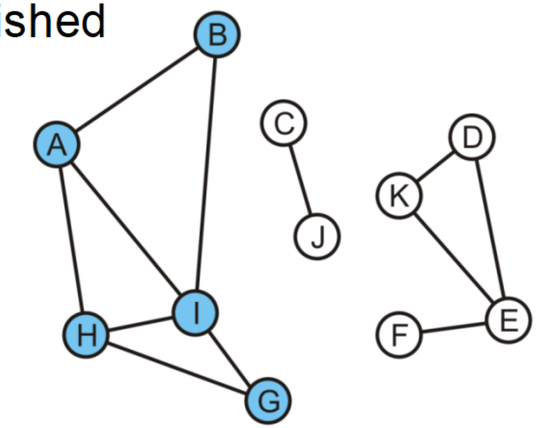


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**G**	**A**	**A**	**J**	**K**

Connected Components

As the traversal continues, we take the union of the set {G} with the set containing H: {A, B, G, H, I}

– The traversal is finished

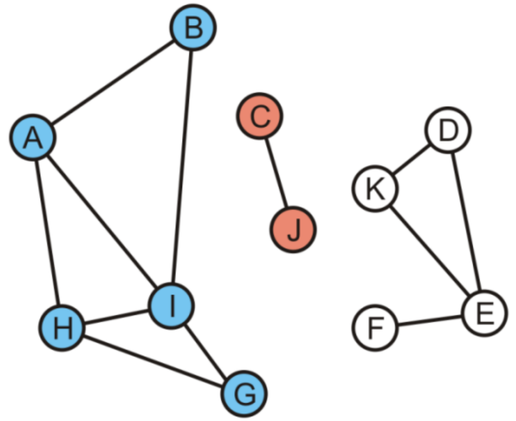


A	B	C	D	E	F	G	H	I	J	K
A	**A**	**C**	**D**	**E**	**F**	**A**	**A**	**A**	**J**	**K**

Connected Components

We take the union of {C} and its adjacent vertex J: {C, J}

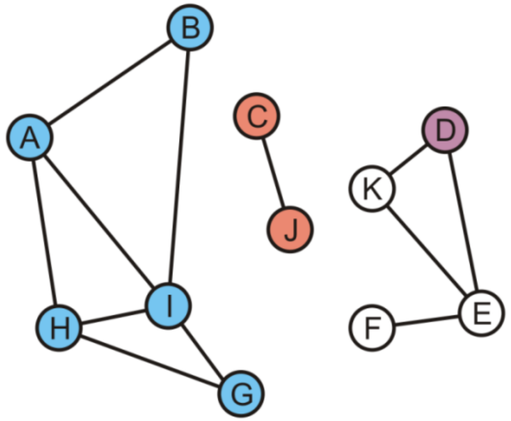
– This traversal is finished



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

Connected Components

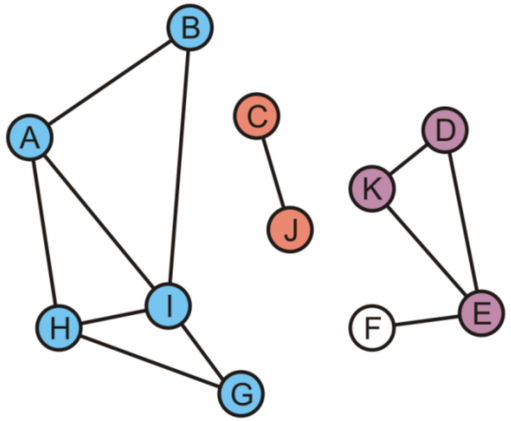
We start again with the set {D}



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

Connected Components

K and E are adjacent to D, so take the unions creating {D, E, K}

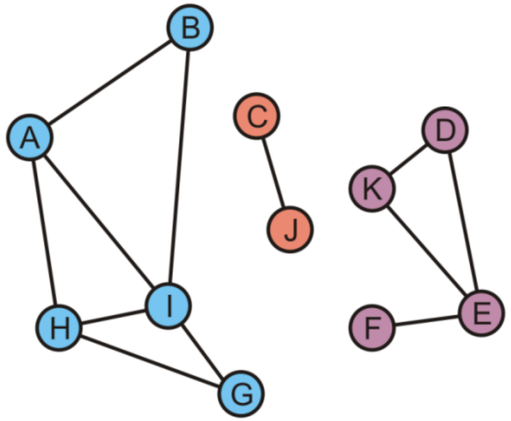


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	F	A	A	A	C	D

Connected Components

Finally, during this last traversal we find that F is adjacent to E

– Take the union of {F} with the set containing E: {D, E, F, K}

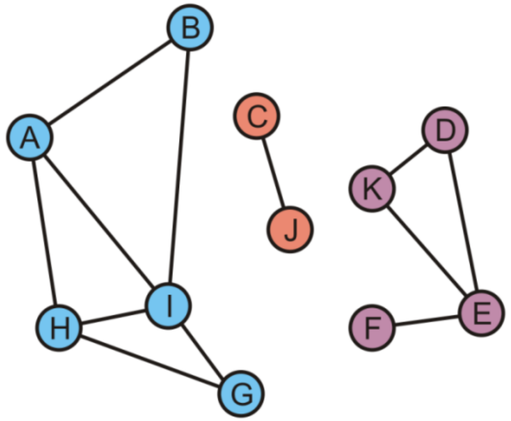


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	D	A	A	A	C	D

Connected Components

All vertices are visited, so we are done

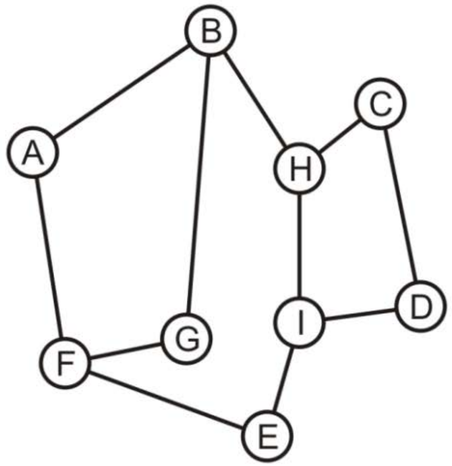
- There are three connected sub-graphs {A, B, G, H, I}, {C, J}, {D, E, F, K}

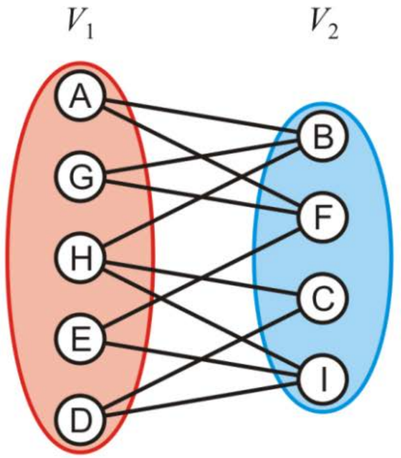


A	B	C	D	E	F	G	H	I	J	K
A	A	C	D	D	D	A	A	A	C	D

Bipartite graph

- Definition
  - A bipartite graph is a graph where the vertices V can be divided into two disjoint sets V_1 and V_2 such that every edge has one vertex in V_1 and the other in V_2





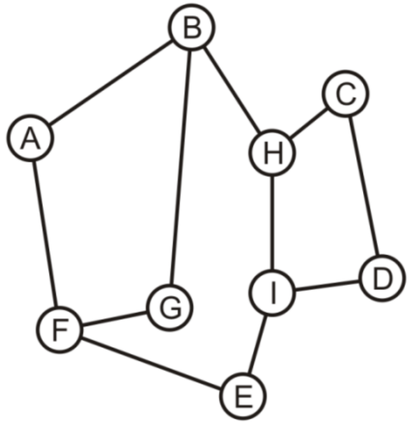
Bipartite Graphs

- Use a breadth-first traversal for a connected graph:
  - Choose a vertex, mark it belonging to V_1 and push it onto a queue
  - While the queue is not empty, pop the front vertex v and
    - Any adjacent vertices that are already marked must belong to the set not containing v, otherwise, the graph is not bipartite (we are done);
    - Any unmarked adjacent vertices are marked as belonging to the other set and they are pushed onto the queue
  - If the queue is empty, the graph is bipartite

Bipartite Graphs

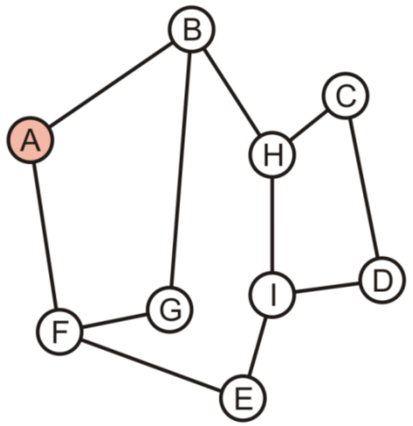
With the first graph, we can start with any vertex

- We will use colours to distinguish the two sets



Bipartite Graphs

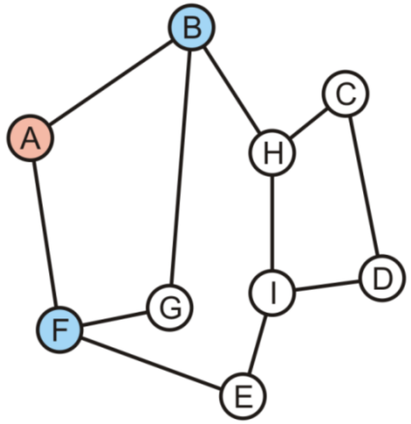
Push A onto the queue and colour it red



Bipartite Graphs

Pop A and its two neighbours are not marked:

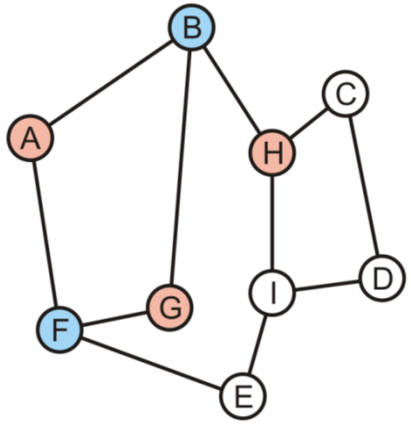
- Mark them as blue and push them onto the queue



Bipartite Graphs

Pop B—it is blue:

- Its one marked neighbour, A, is red
- Its other neighbours G and H are not marked: mark them red and push them onto the queue

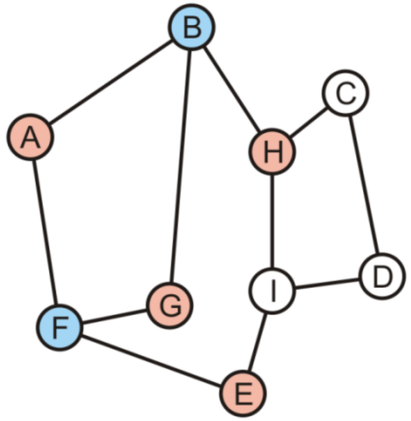


F	G	H			

Bipartite Graphs

Pop F—it is blue:

- Its two marked neighbours, A and G, are red
- Its neighbour E is not marked: mark it red and pus it onto the queue

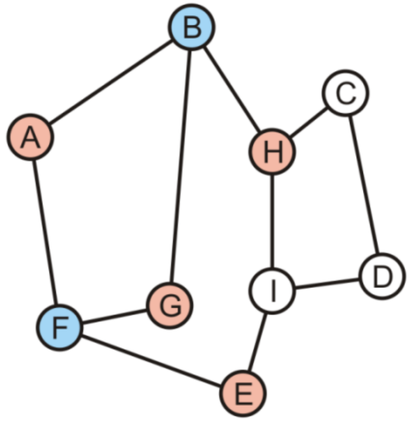


G	H	E			

Bipartite Graphs

Pop G—it is red:

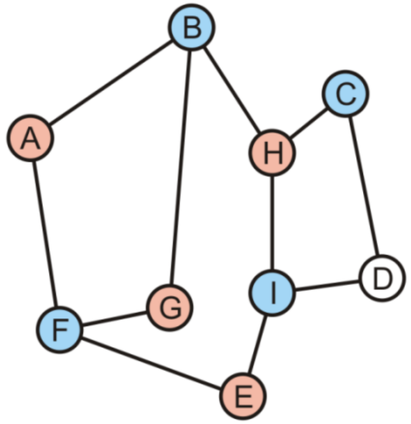
- Its two marked neighbours, B and F, are blue



Bipartite Graphs

Pop H—it is red:

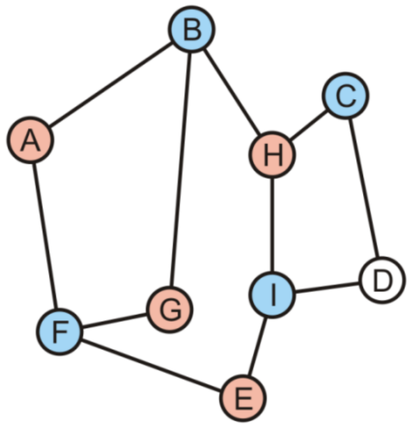
- Its marked neighbours, B, is blue
- It has two unmarked neighbours, C and I; mark them blue and push them onto the queue



Bipartite Graphs

Pop E—it is red:

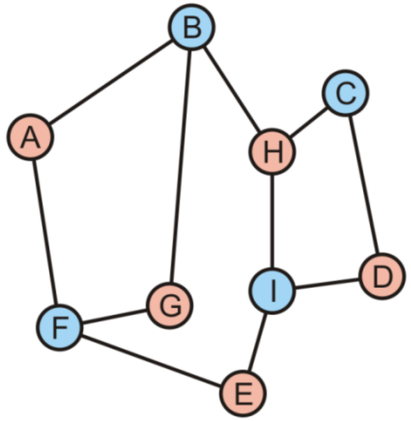
- Its marked neighbours, F and I, are blue



Bipartite Graphs

Pop C—it is blue:

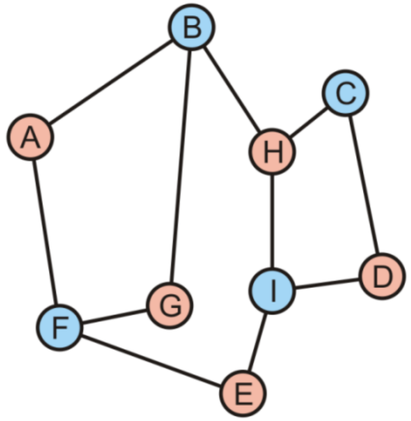
- Its marked neighbour, H, is red
- Mark D as red and push it onto the queue



Bipartite Graphs

Pop I—it is blue:

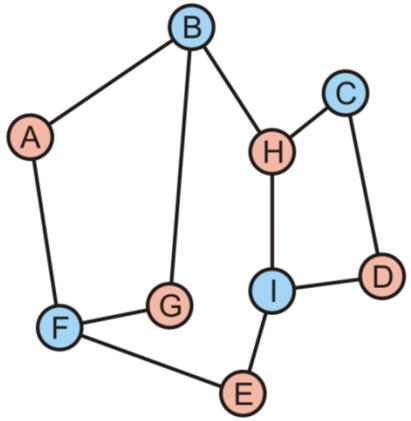
- Its marked neighbours, H, D and E, are all red



Bipartite Graphs

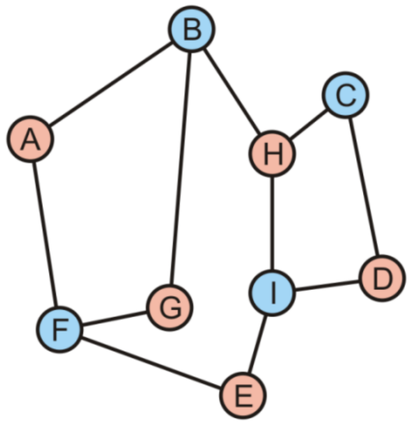
Pop D—it is red:

- Its marked neighbours, C and I, are both blue



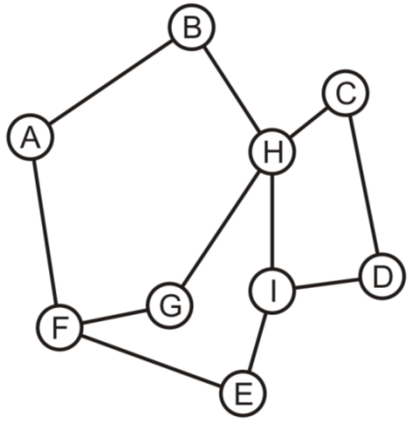
Bipartite Graphs

The queue is empty, the graph is bipartite



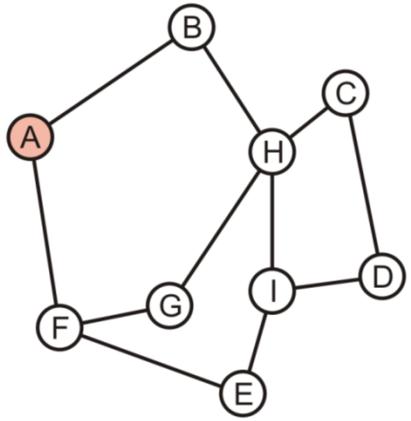
Bipartite Graphs

Consider the other graph which was claimed to be not bipartite



Bipartite Graphs

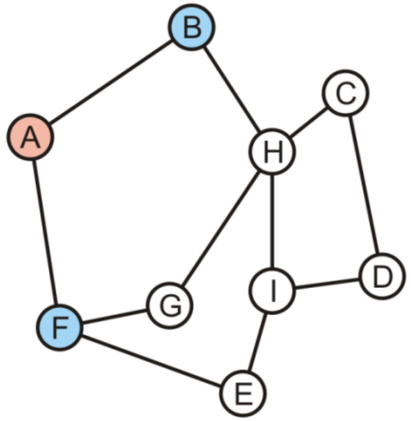
Push A onto the queue and colour it red



Bipartite Graphs

Pop A off the queue:

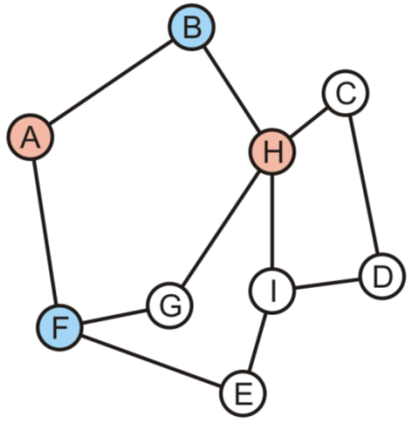
- Its neighbours are unmarked: colour them blue and push them onto the queue



Bipartite Graphs

Pop B off the queue:

- Its one neighbour, A, is red
- The other neighbour, H, is unmarked: colour it red and push it onto the queue

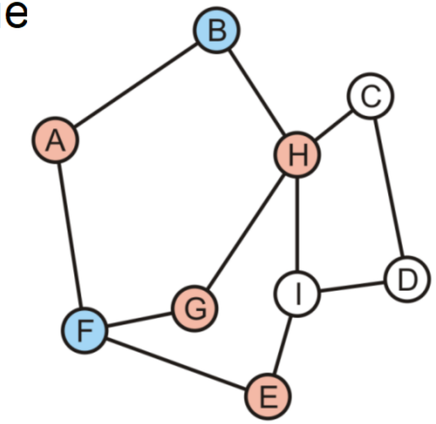


F	H				

Bipartite Graphs

Pop F off the queue:

- Its one neighbour, A, is red
- The other neighbours, E and G, are unmarked: colour them red and push it onto the queue

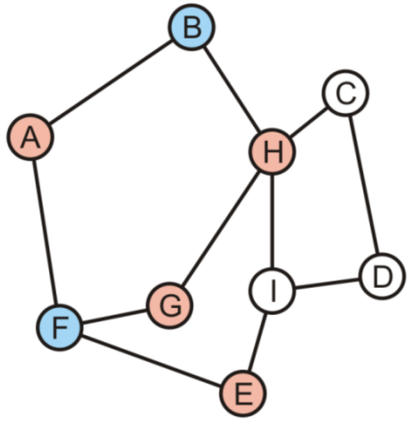


H	E	G			

Bipartite Graphs

Pop H off the queue—it is red:

- Its one neighbour, G, is already red
- The graph is not bipartite



E	G				

7. In the breadth-first search procedure of graphs, each vertex must be enqueued exactly once.
（ ）

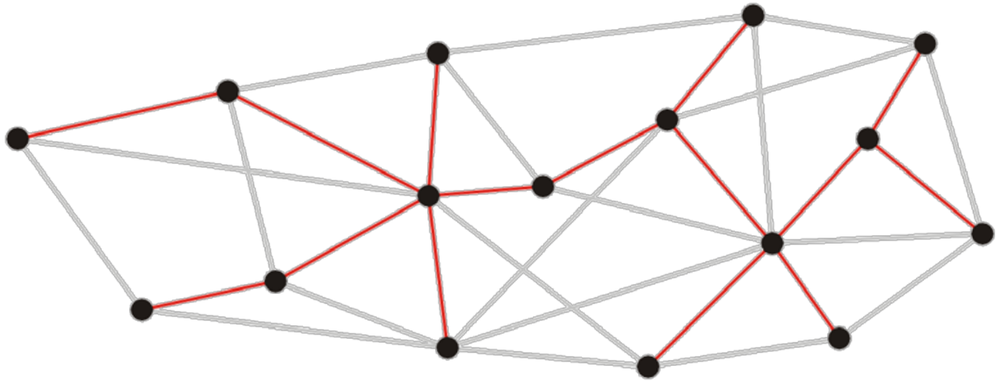
在图的广度遍历算法中，每个节点恰好仅入队一次。（ ）

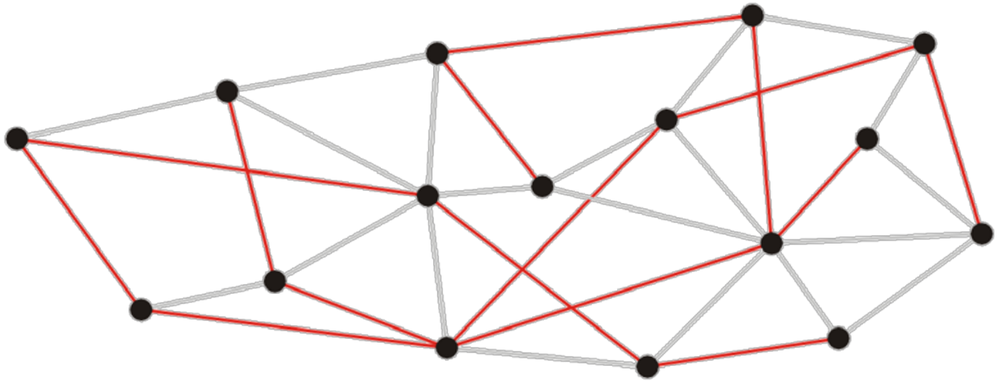
Minimum Spanning Tree

Spanning trees

Given a connected graph with n vertices, a spanning tree is defined as a subgraph that is a tree and includes all the n vertices

- It has $n - 1$ edges

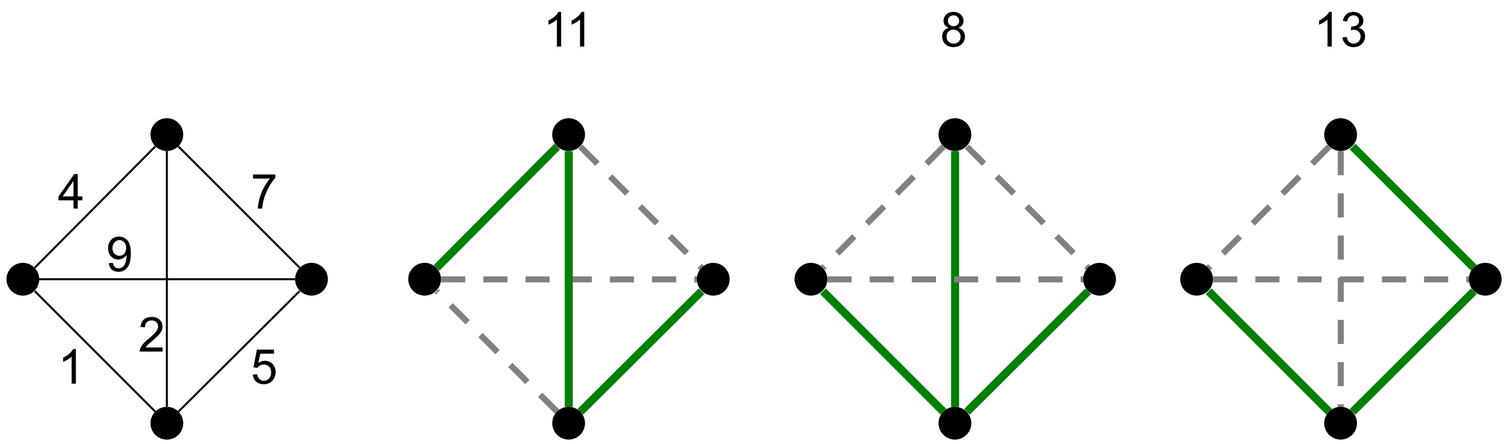




A spanning tree is not necessarily unique

Spanning trees on weighted graphs

The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree



Minimum Spanning Trees

Simplifying assumption:

- All edge weights are distinct

This guarantees that given a graph, there is a unique minimum spanning tree.

Prim's Algorithm

Prim's algorithm for finding the minimum spanning tree states:

- Start with an arbitrary vertex to form a minimum spanning tree on one vertex
- At each step, add the edge with least weight that connects the current minimum spanning tree to a new vertex
- Continue until we have $n - 1$ edges and n vertices

Prim's Algorithm

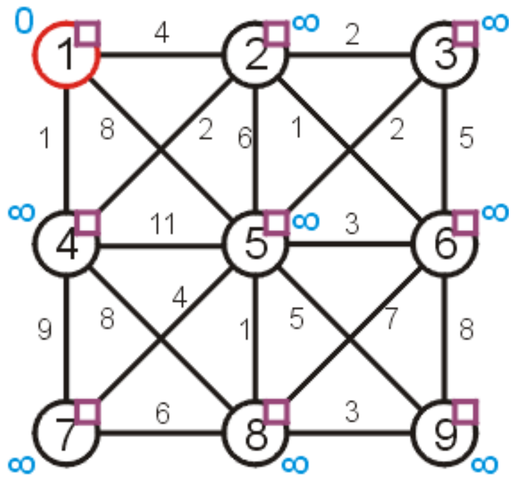
Associate with each vertex three items of data:

- A Boolean flag indicating if the vertex has been visited,
- The minimum distance (weight of a connecting edge) to the partially constructed tree, and
- A pointer to a vertex which will form the parent node in the resulting tree

Implementation:

- Add three member variables to the vertex class
- Or track three tables

Prim's Algorithm

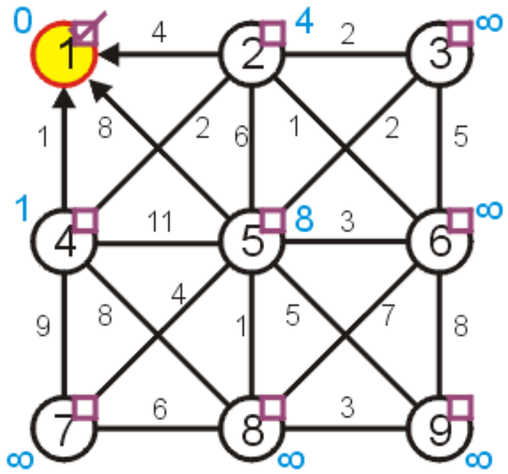


Visited or not

		Distance	Parent
1			
2			
3			
4			
5			
6			
7			
8			
9			

Prim's Algorithm

Visiting vertex 1, we update vertices 2, 4, and 5

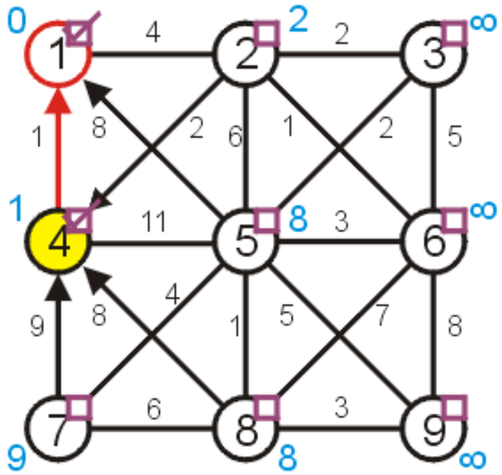


		Distance	Parent
1	T	0	0
2	F	4	1
3	F	∞	0
4	F	1	1
5	F	8	1
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim's Algorithm

The next unvisited vertex with minimum distance is vertex 4

- Update vertices 2, 7, 8
- Don't update vertex 5

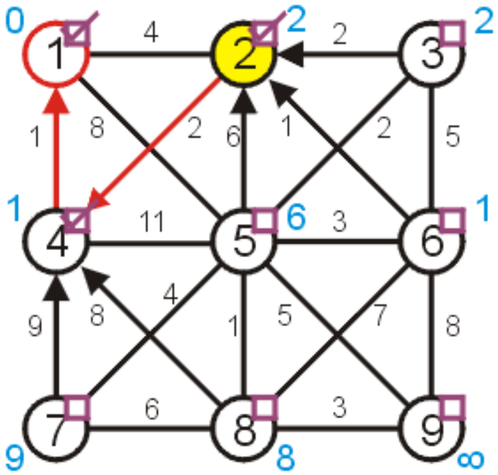


		Distance	Parent
1	T	0	0
2	F	2	4
3	F	∞	0
4	T	1	1
5	F	8	1
6	F	∞	0
7	F	9	4
8	F	8	4
9	F	∞	0

Prim's Algorithm

Next visit vertex 2

- Update 3, 5, and 6

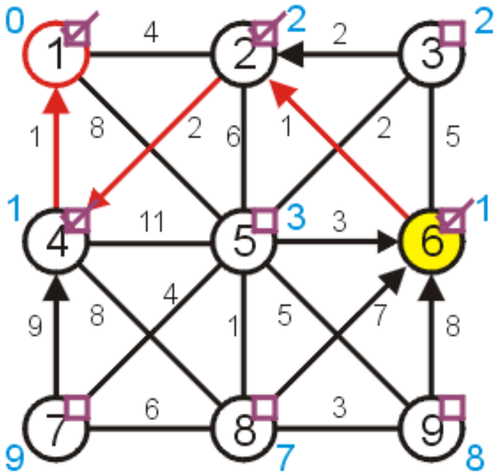


		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	6	2
6	F	1	2
7	F	9	4
8	F	8	4
9	F	∞	0

Prim's Algorithm

Next, we visit vertex 6:

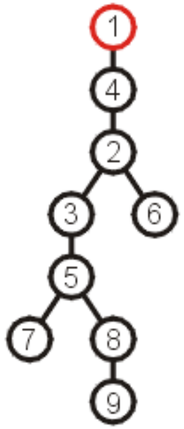
- update vertices 5, 8, and 9

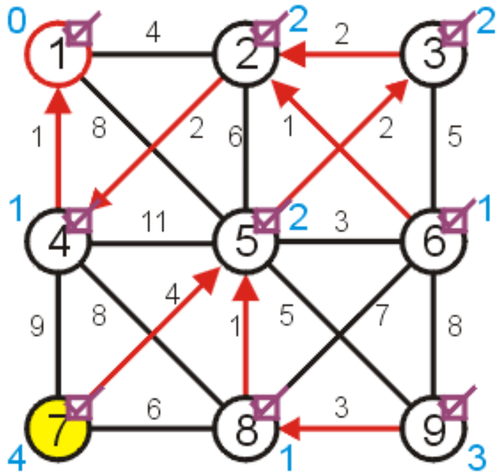


		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	3	6
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim's Algorithm

And neither are there any vertices to update when visiting vertex 7

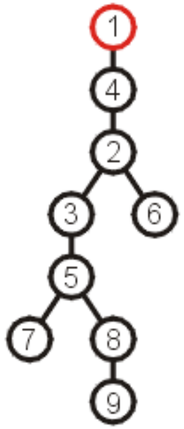


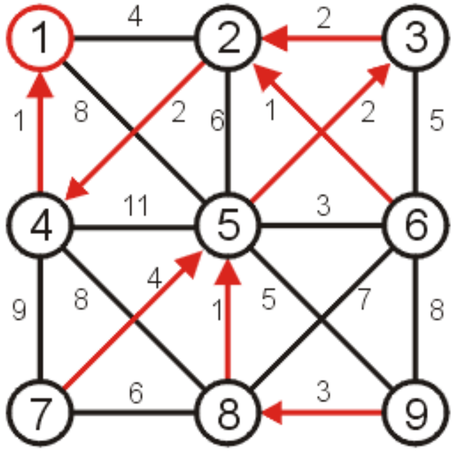


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Prim's Algorithm

Using the parent pointers, we can now construct the minimum spanning tree





		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Implementation: Prim's Algorithm

Use a priority queue.

- Maintain set of explored nodes S.
- For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge from v to a node in S.
- O(n2) with an array; O(m log n) with a binary heap.

```
Prim(G, c) {
    foreach (v ∈ V) a[v] ← ∞
    Initialize an empty priority queue Q
    foreach (v ∈ V) insert v onto Q
    Initialize set of explored nodes S ← ϕ

    while (Q is not empty) {
        u ← delete min element from Q
        S ← S ∪ { u }
        foreach (edge e = (u, v) incident to u)
            if ((v ∉ S) and (c_e < a[v]))
                decrease priority a[v] to c_e
}
```

Kruskal's Algorithm

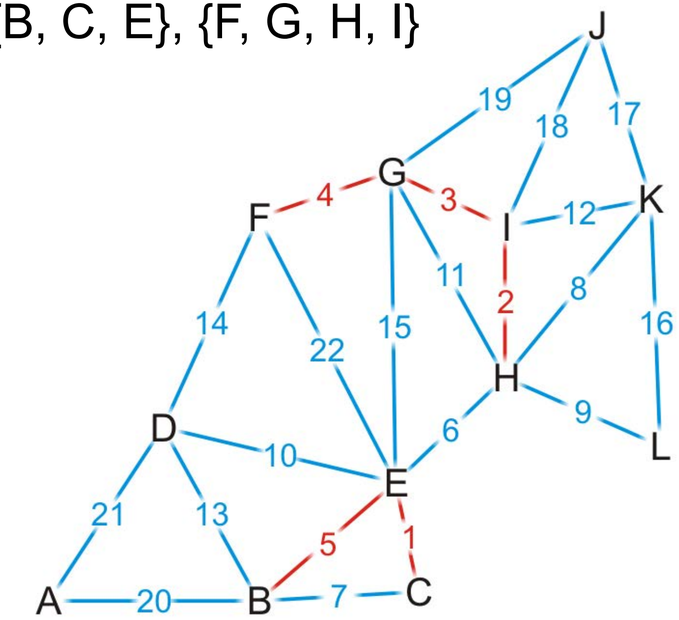
- Sort the edges by weight
- Go through the edges from least weight to greatest weight
  - add the edges to the spanning tree so long as the addition does not create a cycle
  - Does this edge belong to the minimum spanning tree?
    - Yes! The cut property (consider the subtree connected to one end of the edge as the set *S*).
- Repeatedly add more edges until:
  - $|V| - 1$ edges have been added, then we have a minimum spanning tree
  - Otherwise, if we have gone through all the edges, then we have a forest of minimum spanning trees on all connected sub-graphs

Analysis

We could use disjoint sets

- Consider edges in the same connected sub-graph as forming a set

{B, C, E}, {F, G, H, I}



Analysis

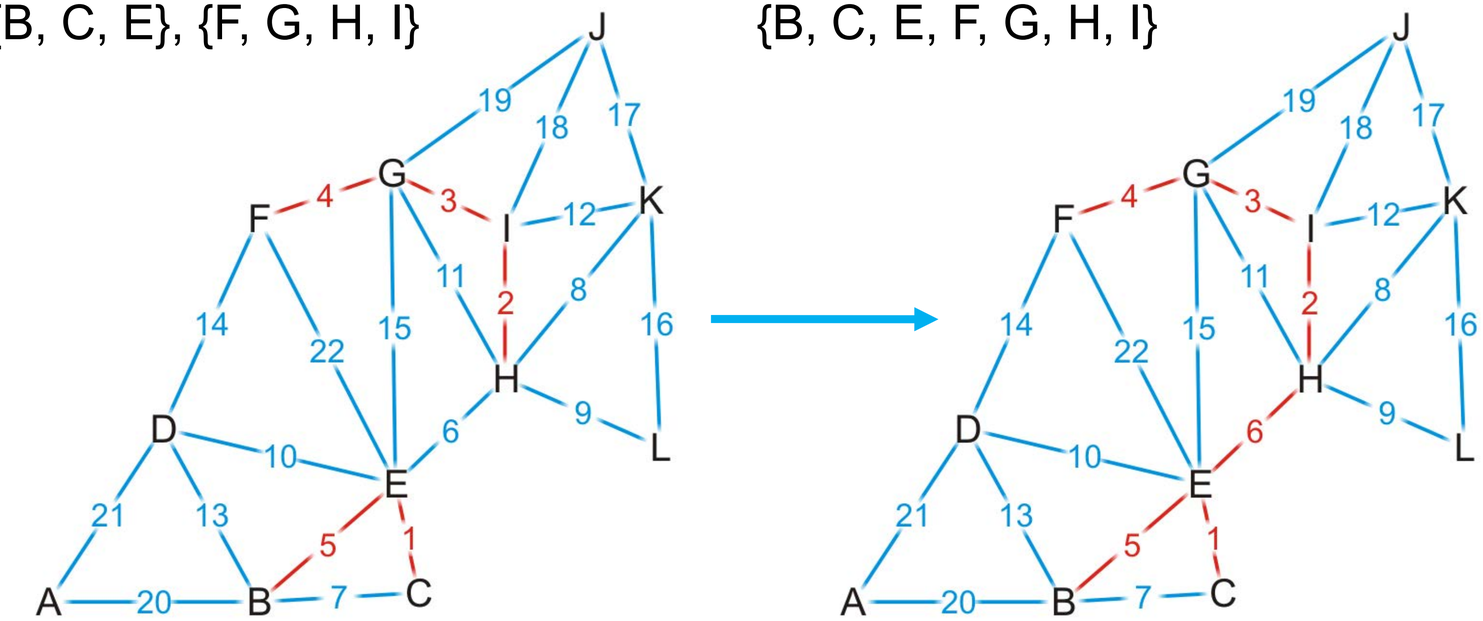
Instead, we could use disjoint sets

- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets, take the union of the two sets

{B, C, E}, {F, G, H, I}

Add edge (E, H)?

{B, C, E, F, G, H, I}



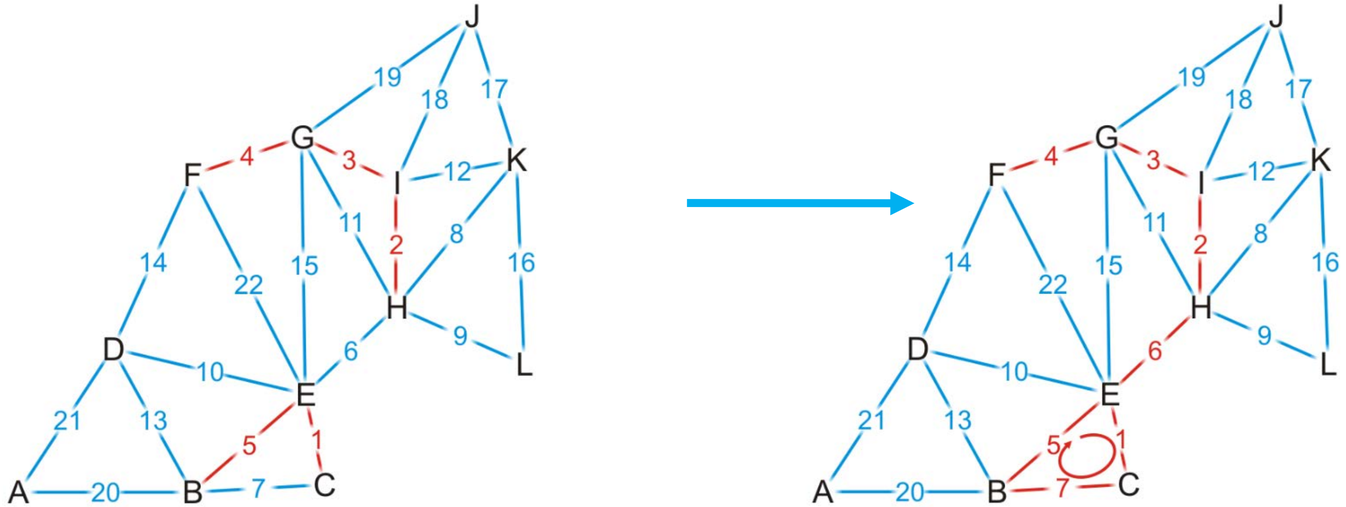
Analysis

Instead, we could use disjoint sets

- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets, take the union of the two sets
- Do not add an edge if both vertices are in the same set

Add edge (B, C)?

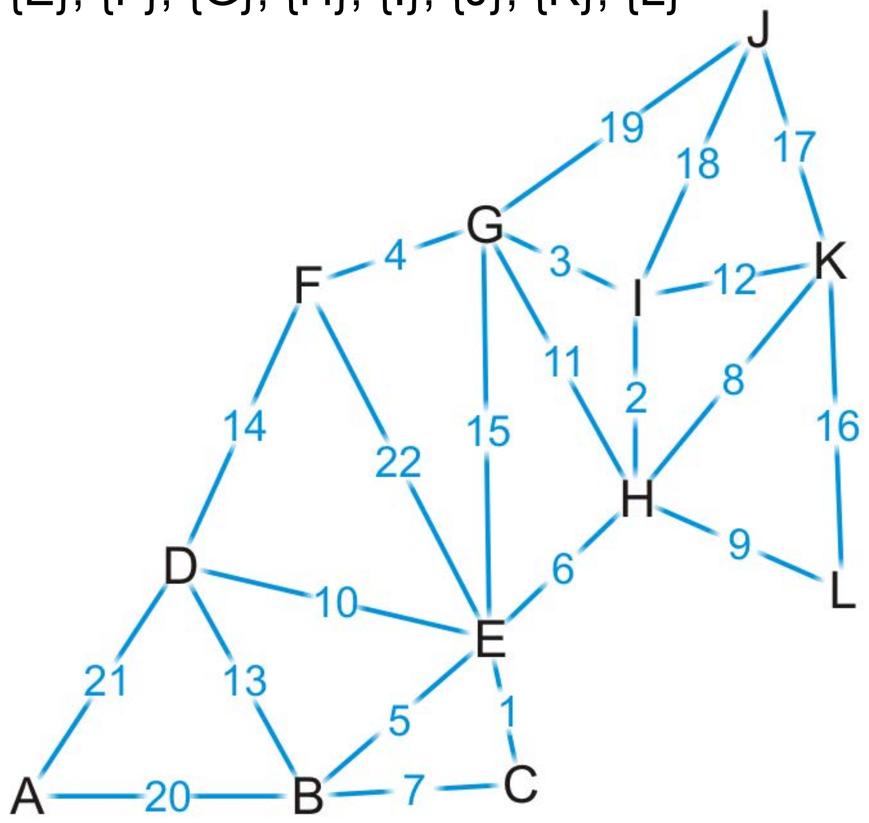
{B, C, E}, {F, G, H, I}



Example

We start with twelve singletons

{A}, {B}, {C}, {D}, {E}, {F}, {G}, {H}, {I}, {J}, {K}, {L}



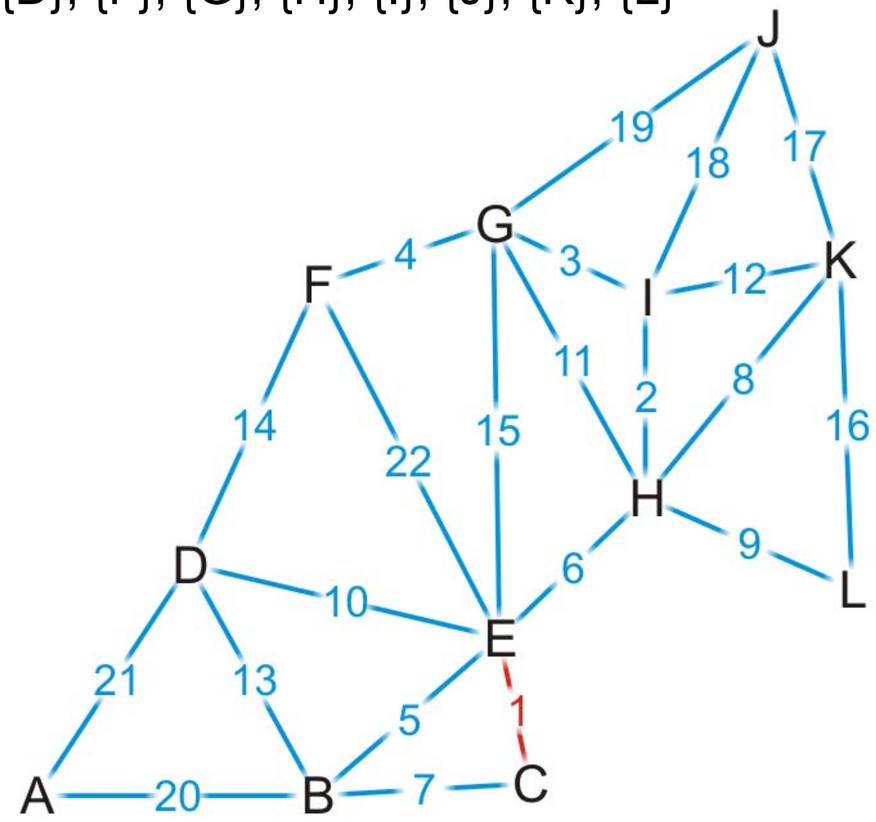
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

We start by adding edge {C, E}

{A}, {B}, {C, E}, {D}, {F}, {G}, {H}, {I}, {J}, {K}, {L}

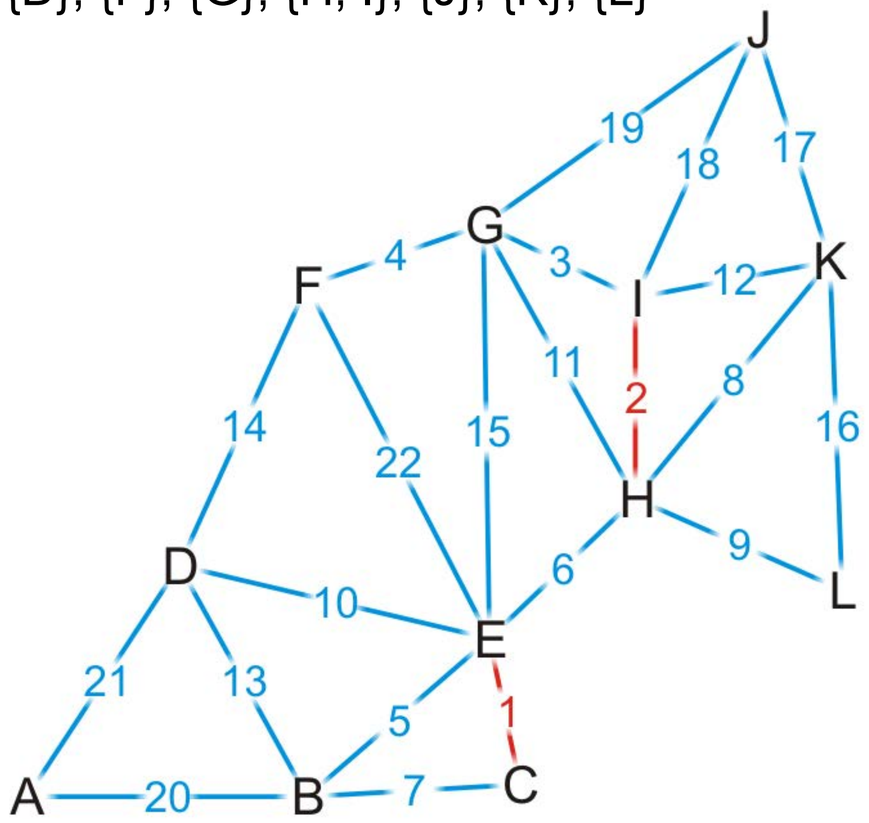


Example

{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

We add edge {H, I}

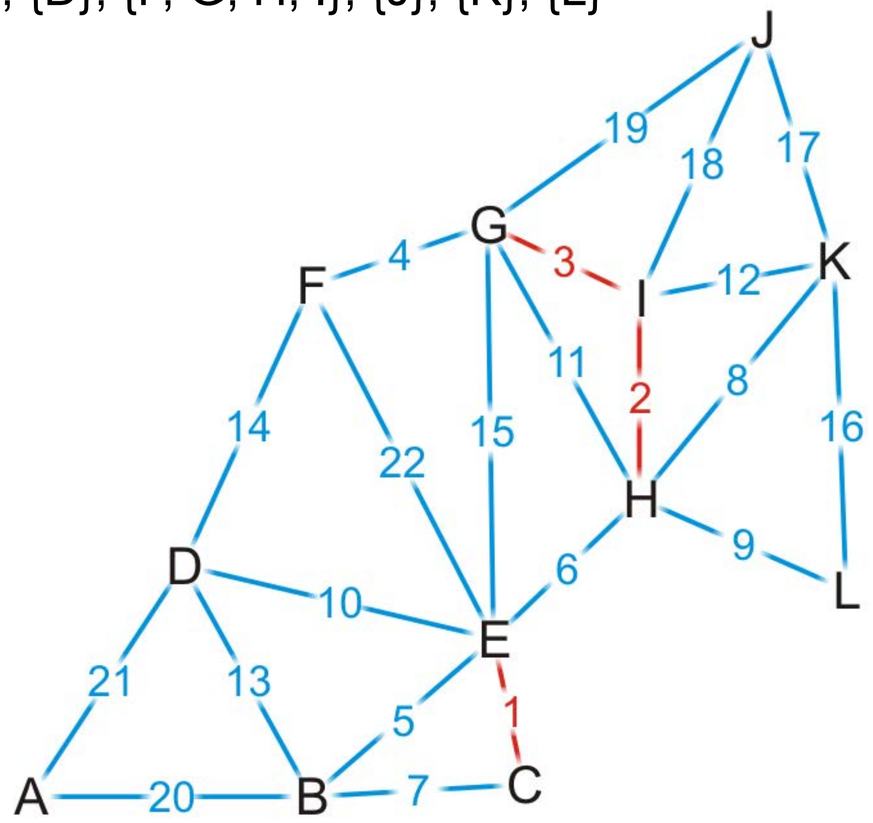
{A}, {B}, {C, E}, {D}, {F}, {G}, {H, I}, {J}, {K}, {L}



Example

Similarly, we add {G, I}, {F, G}, {B, E}

{A}, {B, C, E}, {D}, {F, G, H, I}, {J}, {K}, {L}

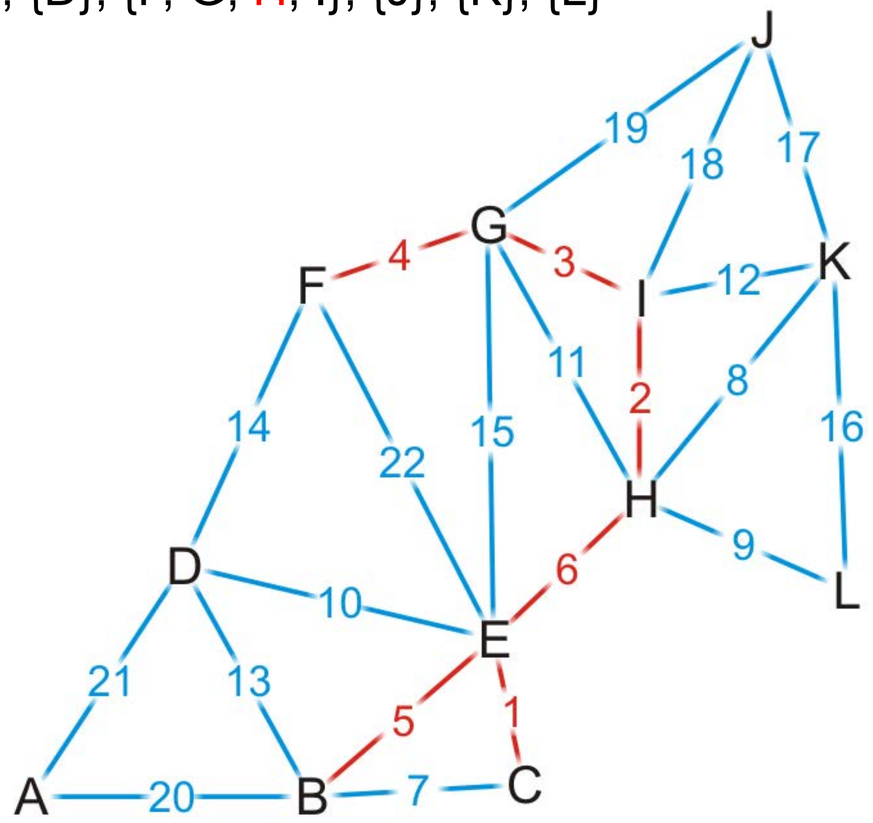


{C, E}
{H, I}
→ {G, I}
→ {F, G}
→ {B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

The vertices of {E, H} are in different sets

{A}, {B, C, E}, {D}, {F, G, H, I}, {J}, {K}, {L}

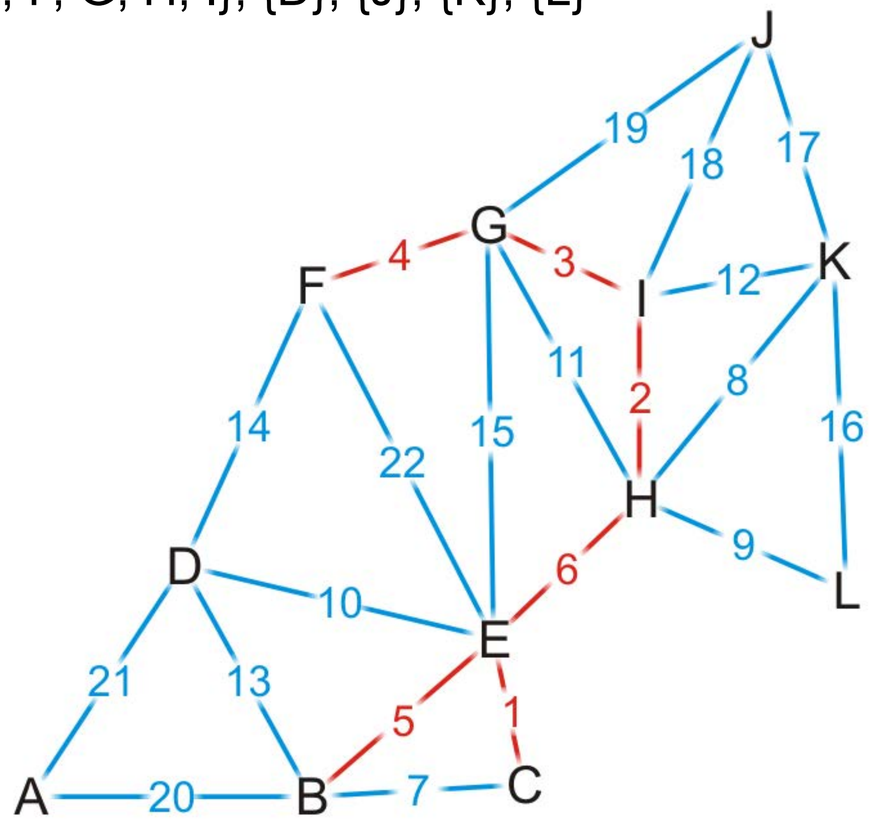


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
→ {E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Adding edge {E, H} creates a larger union

{A}, {B, C, E, F, G, H, I}, {D}, {J}, {K}, {L}

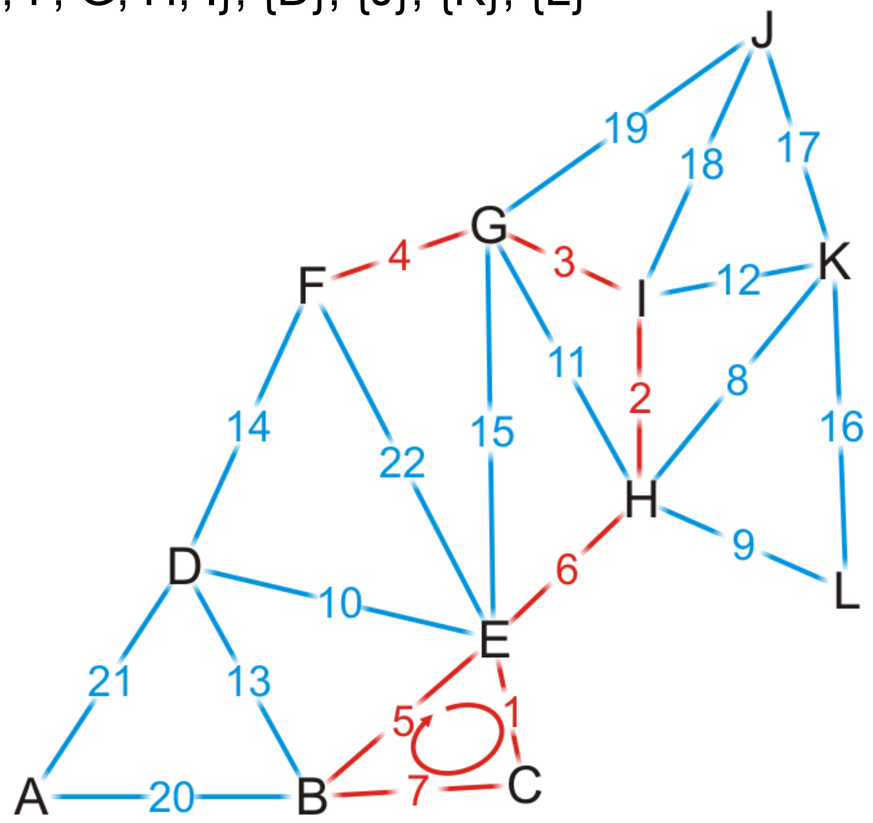


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
→ {E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We try adding {B, C}, but it creates a cycle

{A}, {B, C, E, F, G, H, I}, {D}, {J}, {K}, {L}

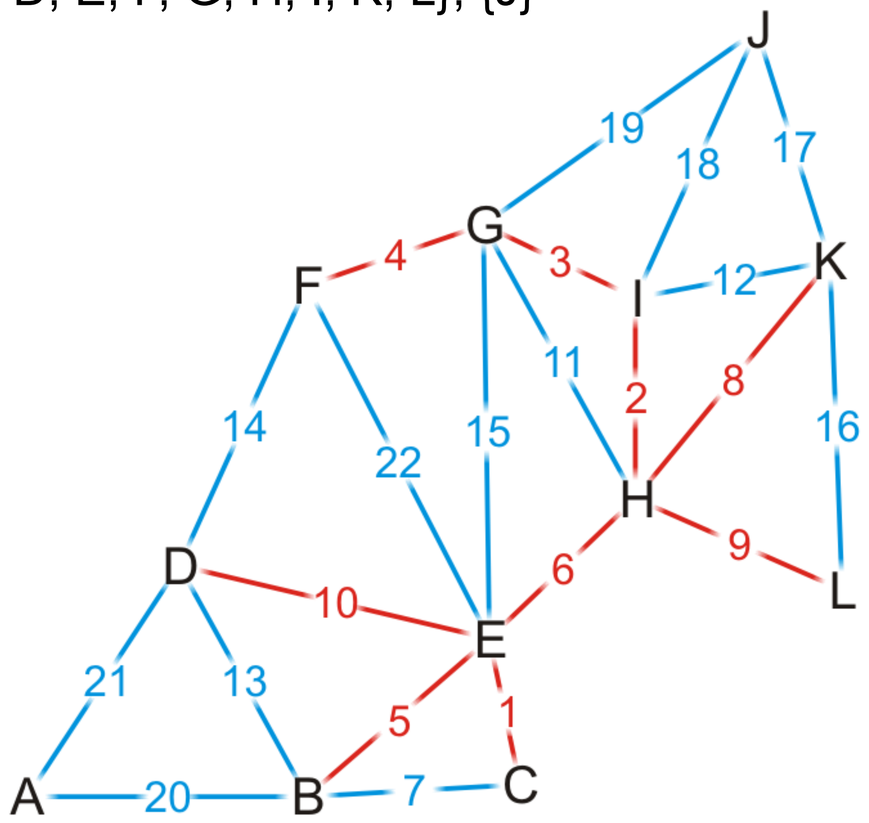


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
→ {B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We add edge {H, K}, {H, L} and {D, E}

{A}, {B, C, D, E, F, G, H, I, K, L}, {J}

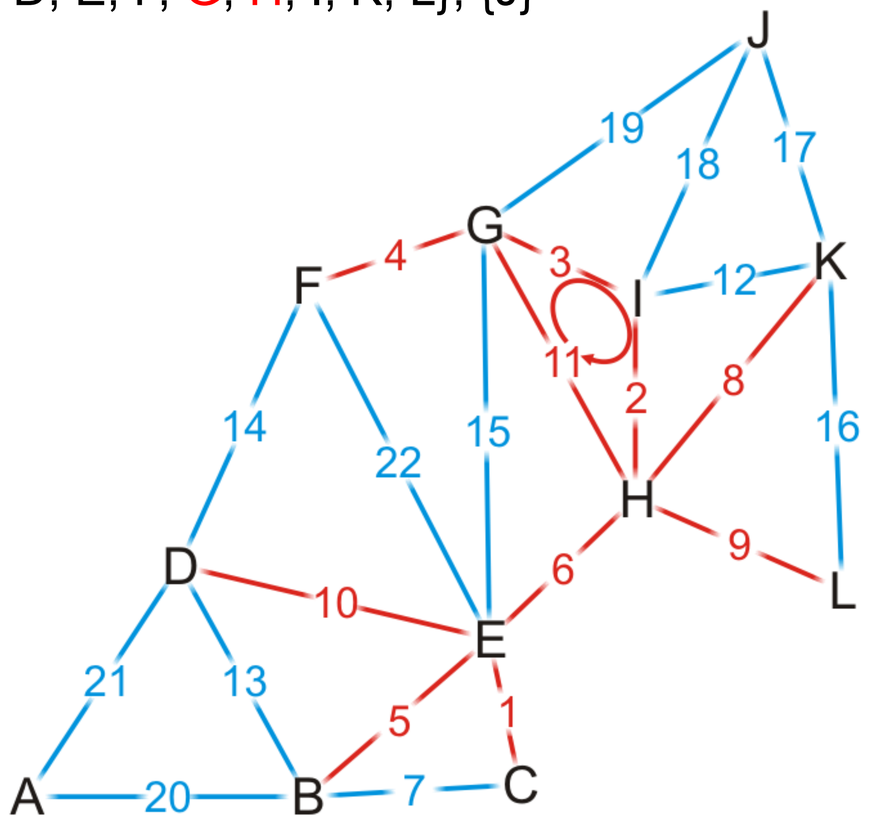


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
→ {H, K}
→ {H, L}
→ {D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both G and H are in the same set

{A}, {B, C, D, E, F, G, H, I, K, L}, {J}

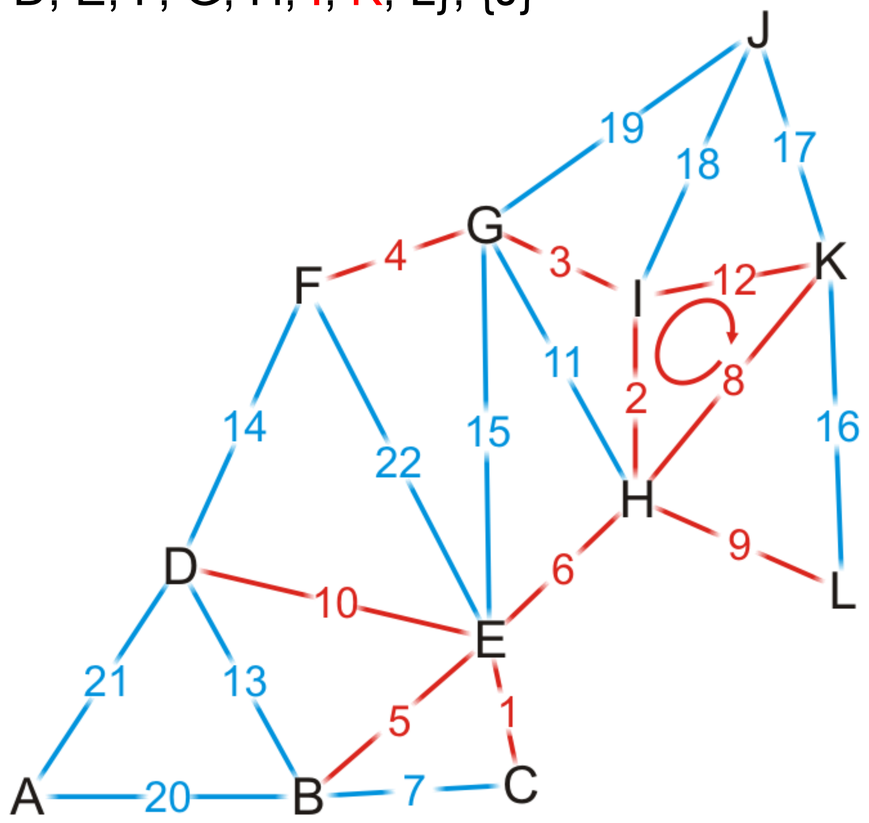


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
→ {G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both {I, K} are in the same set

{A}, {B, C, D, E, F, G, H, I, K, L}, {J}

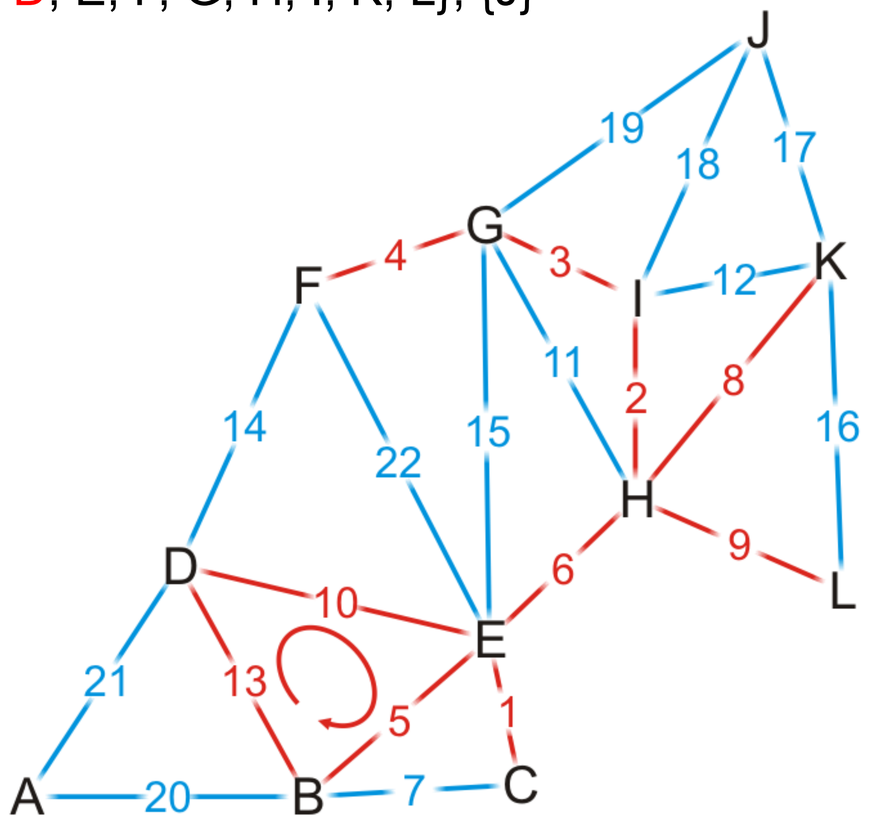


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
→ {I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both {B, D} are in the same set

{A}, {B, C, D, E, F, G, H, I, K, L}, {J}

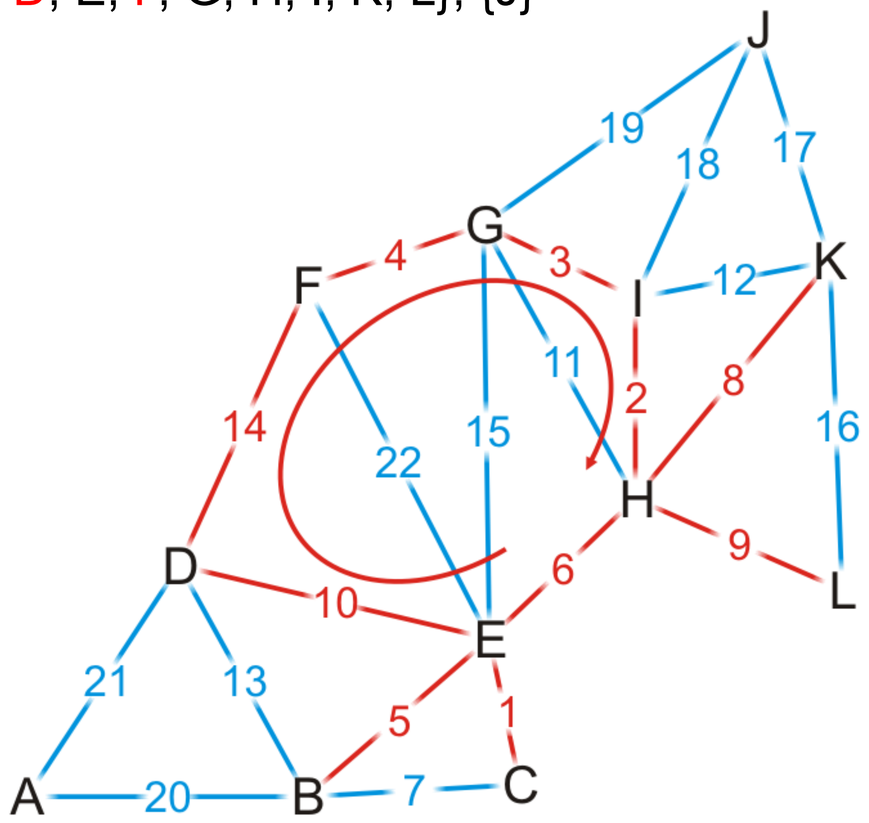


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
→ {B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both {D, F} are in the same set

{A}, {B, C, D, E, F, G, H, I, K, L}, {J}

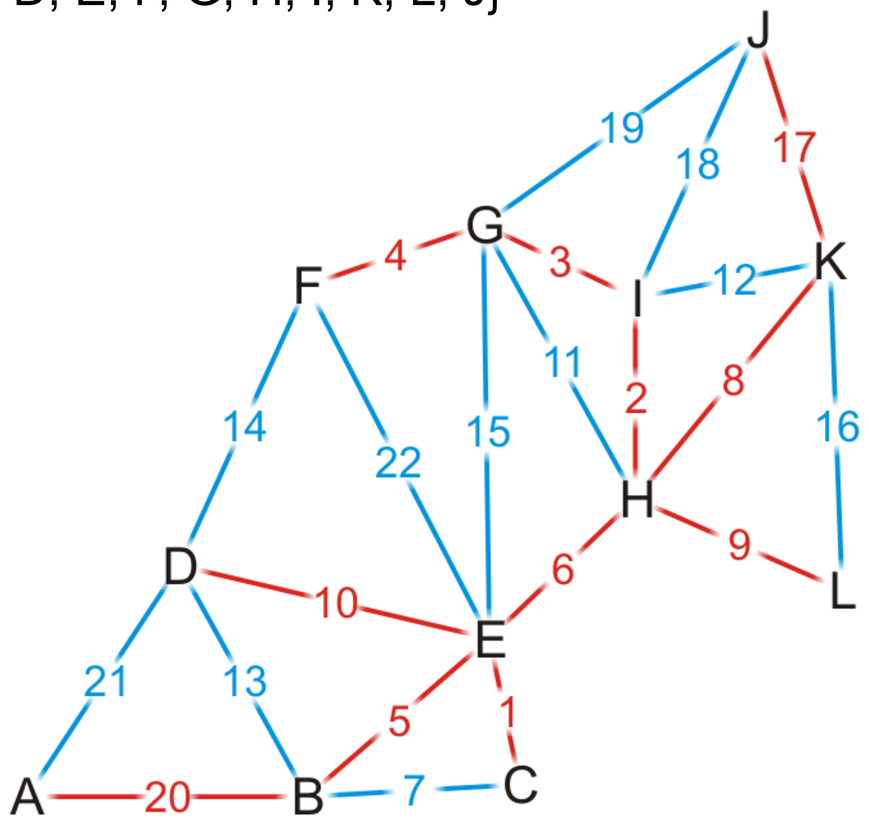


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
→ {D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We end when there is only one set, having added (A, B)

{A, B, C, D, E, F, G, H, I, K, L, J}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
→ {A, B}
{A, D}
{E, F}

Implementation: Kruskal's Algorithm

Use the union-find data structure.

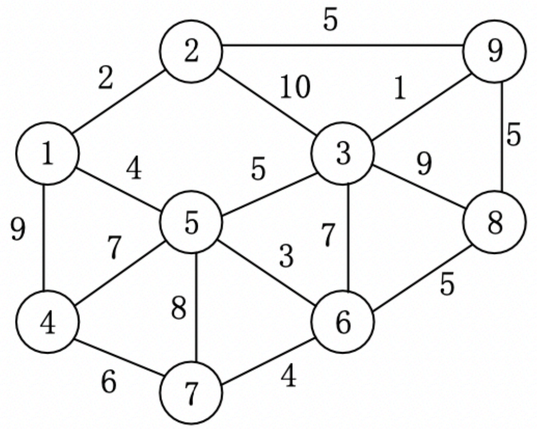
- Build set T of edges in the MST.
- Maintain set for each connected component.
- O(m log m) for sorting.

```
Kruskal(G, c) {
   Sort edges weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m                  are u and v in different connected components?
      (u,v) = e_i                          ↙
      if (u and v are in different sets) {
         T ← T ∪ {e_i}
         merge the sets containing u and v
      }                                ↖
   return T                      merge two components
}
```

4. Consider the following weighed, undirected graph. Which of the following must be true? （ ）



A. The above graph has four minimum spanning trees.

B. There is at least one minimum spanning tree containing the edge (6,8)

C. There is at least one minimum spanning tree containing the edge (5,6)

D. All the minimum spanning trees contain the edge (2,9)

E. All the minimum spanning trees contain the edge (4,7)

15) 下面关于无向图的描述哪个是正确的（　　）

A．树是一个无环图

B．无环图一定是树

C．图的最小生成树上的一条路径是连接这条路径任意两个顶点的最短路径

D．以上均不对