Lecture 10
Graph Algorithms

Euiseong Seo
(euiseong@skku.edu)

Graph Theory

- Study of the properties of graph structures
  - It provides us with a language with which to talk about graphs
- Keys to solving problems
  - Identifying the fundamental graph theoretic notion underlying the situation
  - Using classical algorithms to solve the resulting problem

Degrees

- Number of edges connected to a vertex
- For undirected graphs
  - Sum of all degrees = 2 * edges
- For directed graph
  - Sum of in-degree = sum of out-degree

Connectivity

- A graph is connected if there is an undirected path between every pair of vertices
- Existence of a spanning tree -> connectivity
  - BFS or DFS connected component algorithms to find connected components
- Articulation vertex
  - A single vertex whose deletion disconnects the graph
- Biconnected graphs
  - Any graphs without an articulation vertex
- Bridge
  - A single edge whose deletion disconnects the graph

Connectivity

- Testing for articulation vertices or bridges
  - Brute force
  - Be sure to add that vertex/edge back before doing the next deletion!
- Strongly connected components
  - For directed graphs only
  - Partitions the graph into chunks such that there are directed paths between all pairs of vertices within a given chunk
  - Road networks should be strongly connected

Cycles in Graphs

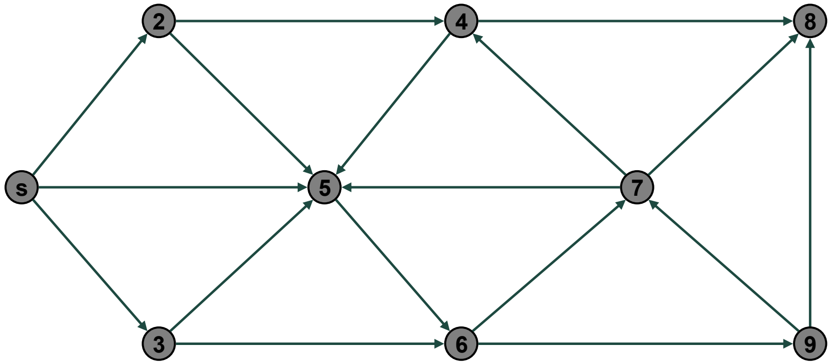
- All non-tree connected graphs contain cycles
- Eulerian cycle
  - A tour which visits every edge of the graph exactly once
  - Condition for a undirected graph to have an Eulerian cycle?
  - Find an Eulerian cycle by building it once cycle at a time
    - Finding a back edge using DFS
    - Deleting the edges on this cycle leaves each vertex with even degree

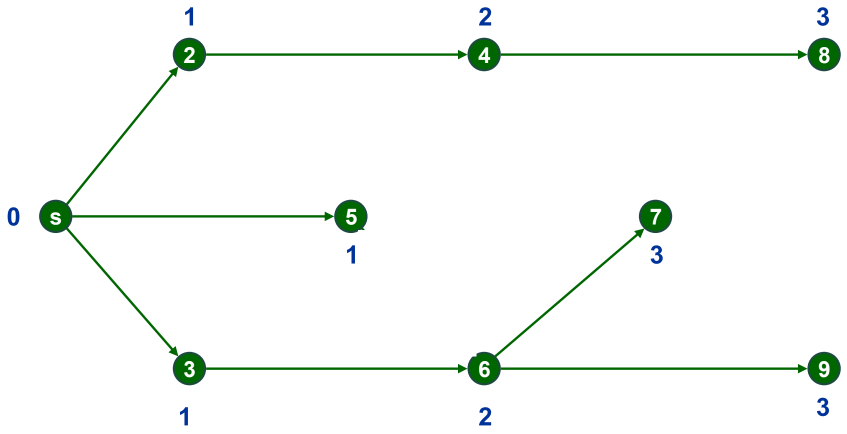
Cycles in Graphs

- Hamiltonian cycles
  - A tour which visits every vertex of the graph exactly once
  - Traveling salesman problem = the shortest Hamiltonian cycle on a weighted graph
  - No efficient algorithm to find one
    - Backtracking!

Spanning Tree

- Given a graph $G = (V, E)$ and tree $T = (V, E')$
    - $E' \subset E$
    - for all (u, v) in E' $u, v \in V$
    - for all connected graph, there exists a spanning tree



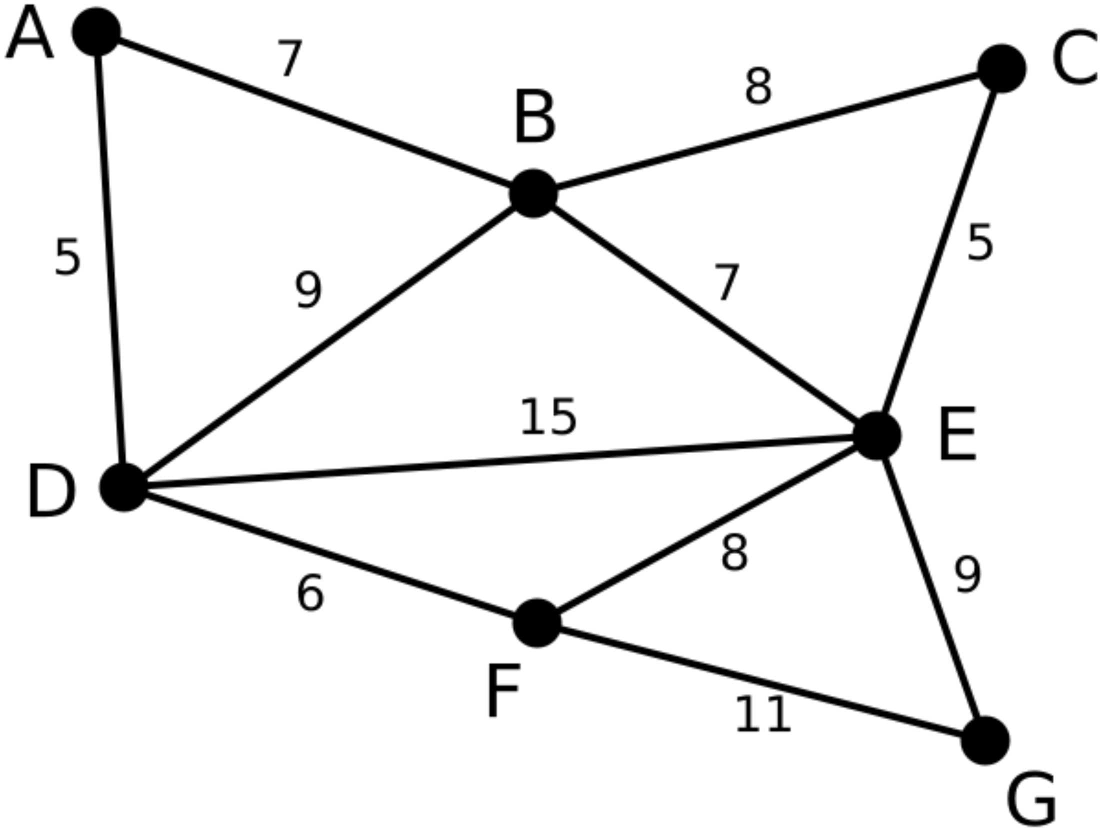


- A spanning tree can be constructed using DFS or BFS

Minimum Spanning Tree

- A spanning tree whose sum of edge weights is minimal
- Importance of MSTs
  - When to need connect a set of points by the smallest amount of roadway, wire or pipe
- Prim’s and Kruskal’s algorithms

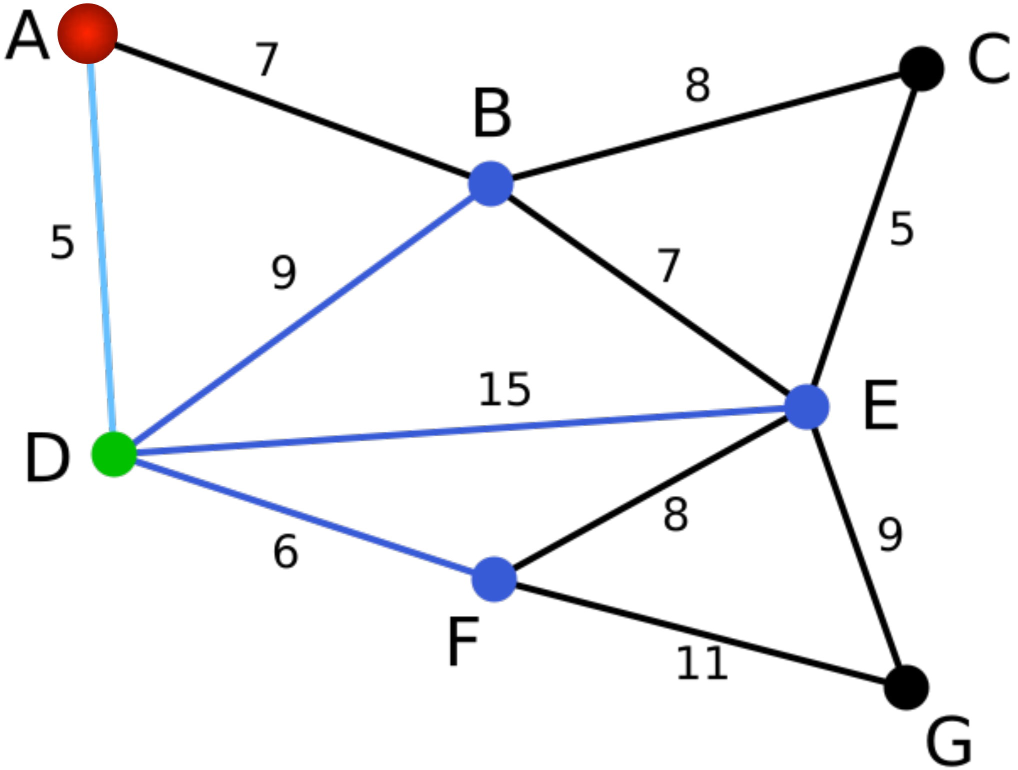
Example of MST: Prim's Algorithm



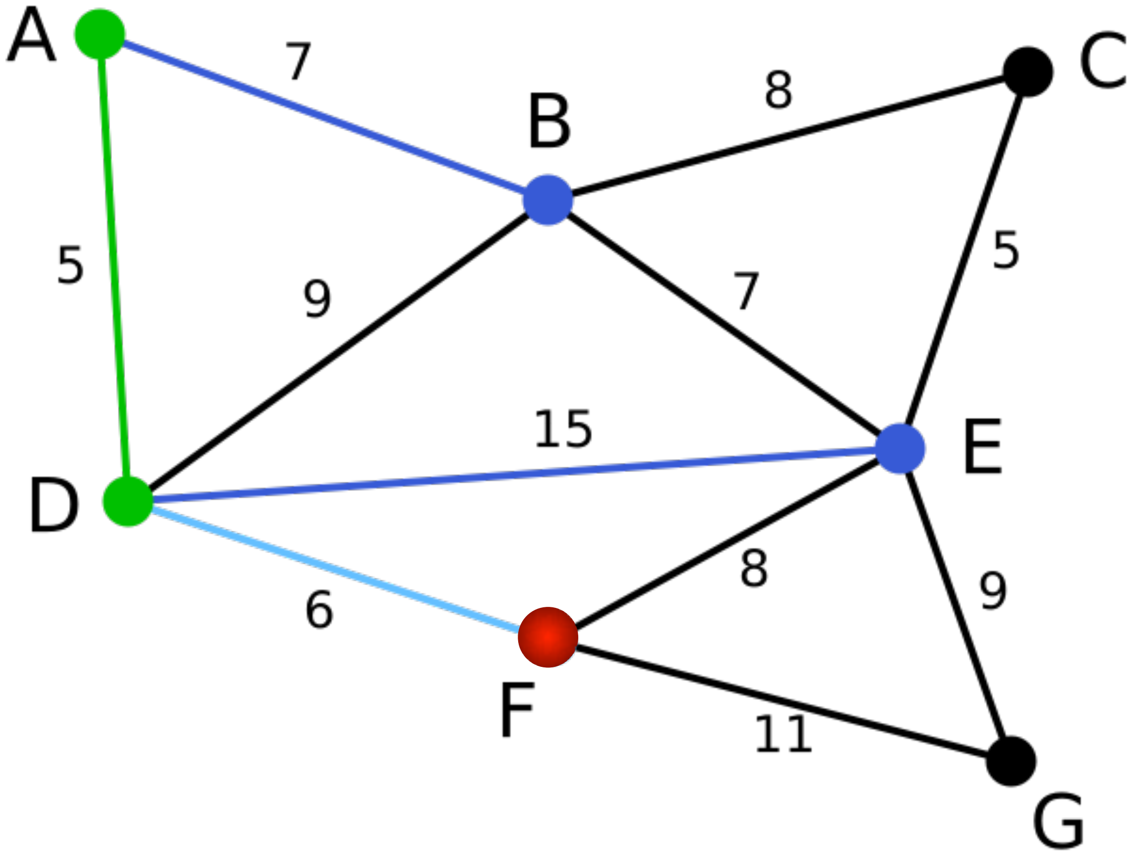
1. Vertex D has been chosen as a starting point

① Vertices A, B, E, F are connected to D through a single edge.

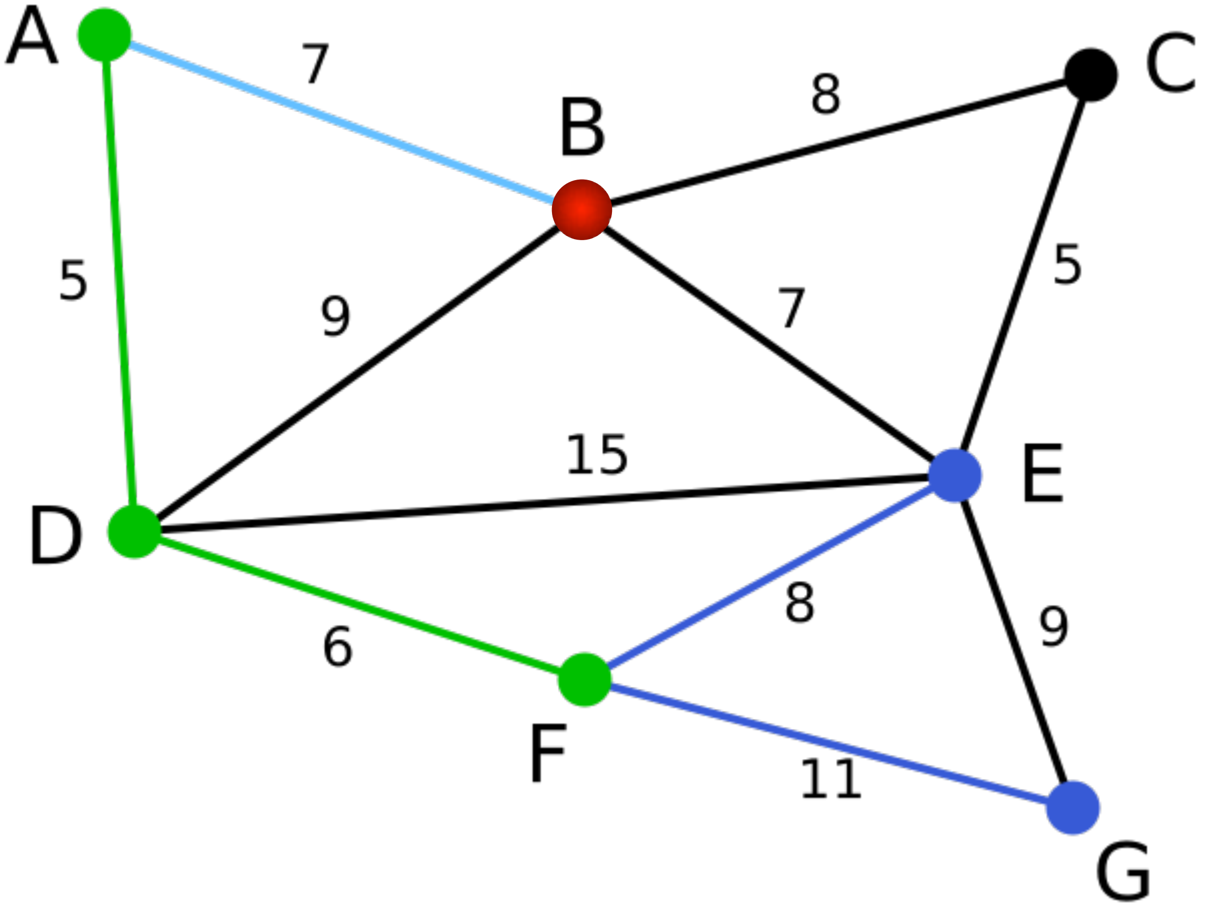
② A is the nearest to D and thus chosen as the 2nd vertex along with the edge AD



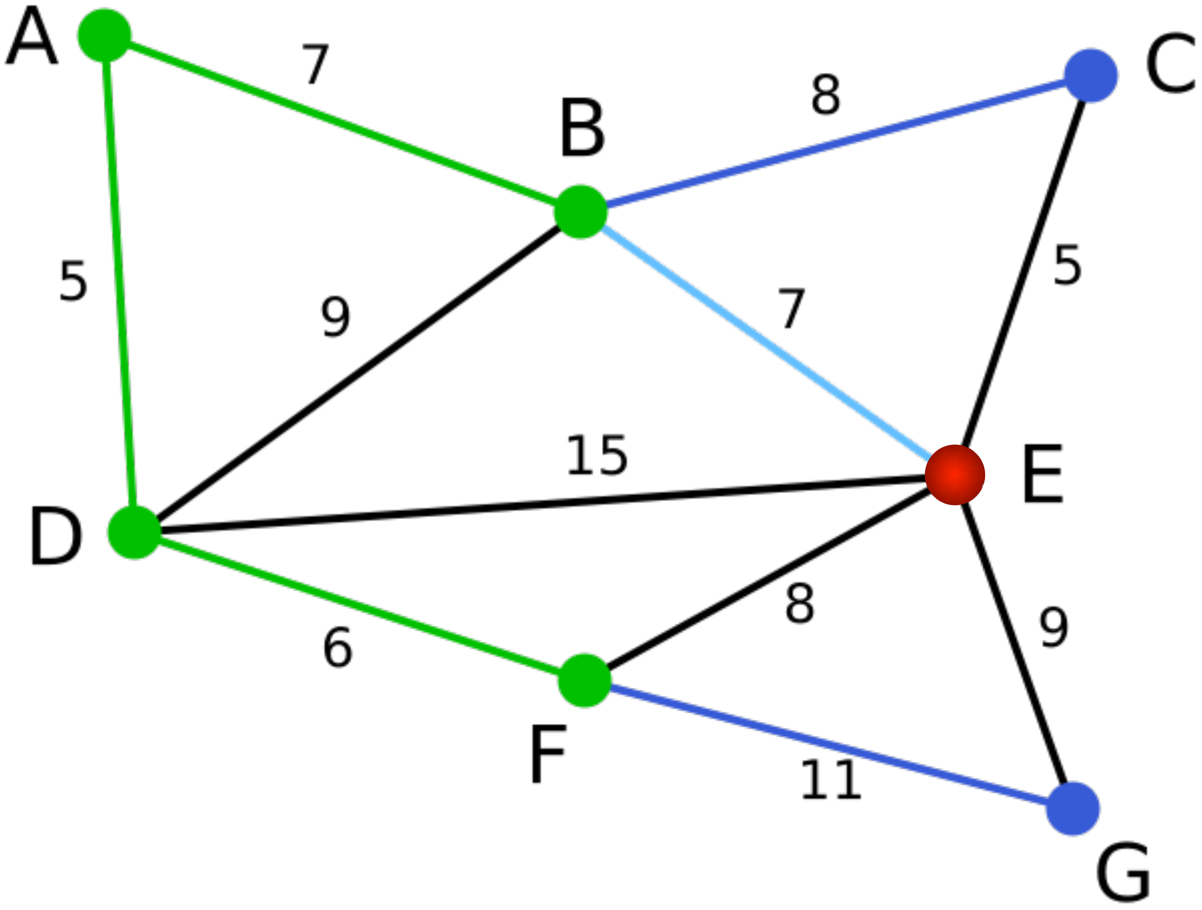
2. The next vertex chosen is the vertex nearest to either D or A. So the verte x F is chosen along with the edge DF



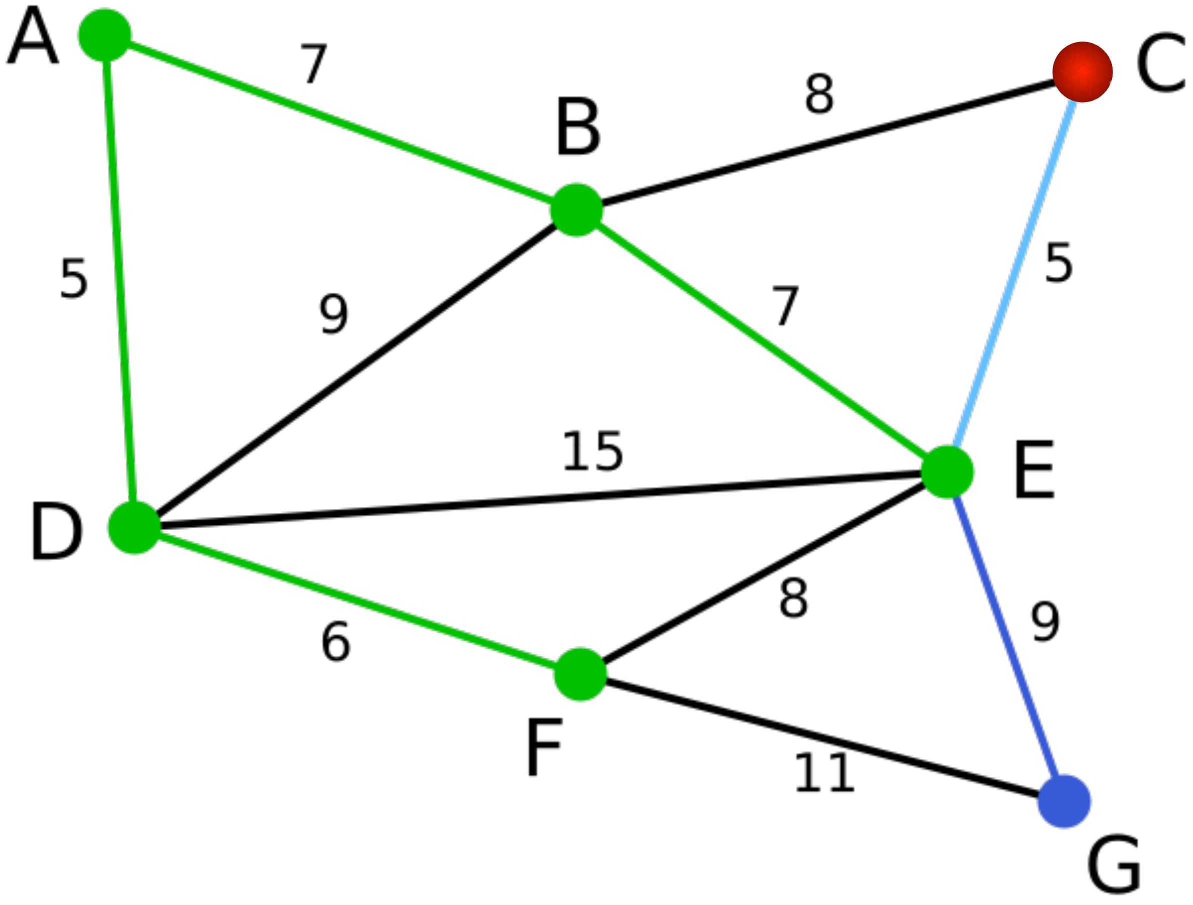
3. same as 2, Vertex B is chosen.



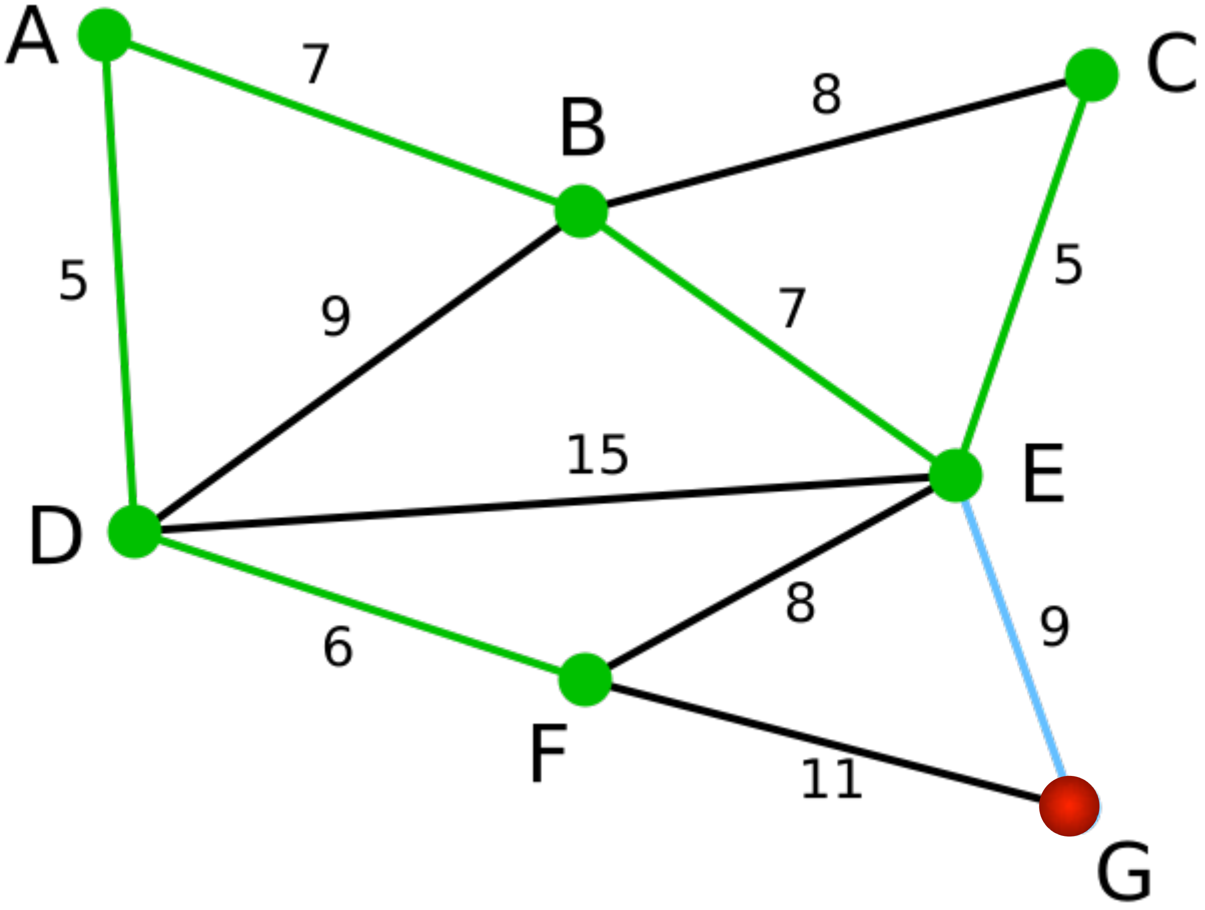
4. among C, E, G, E is chosen.



5. among C, G, C is chosen.

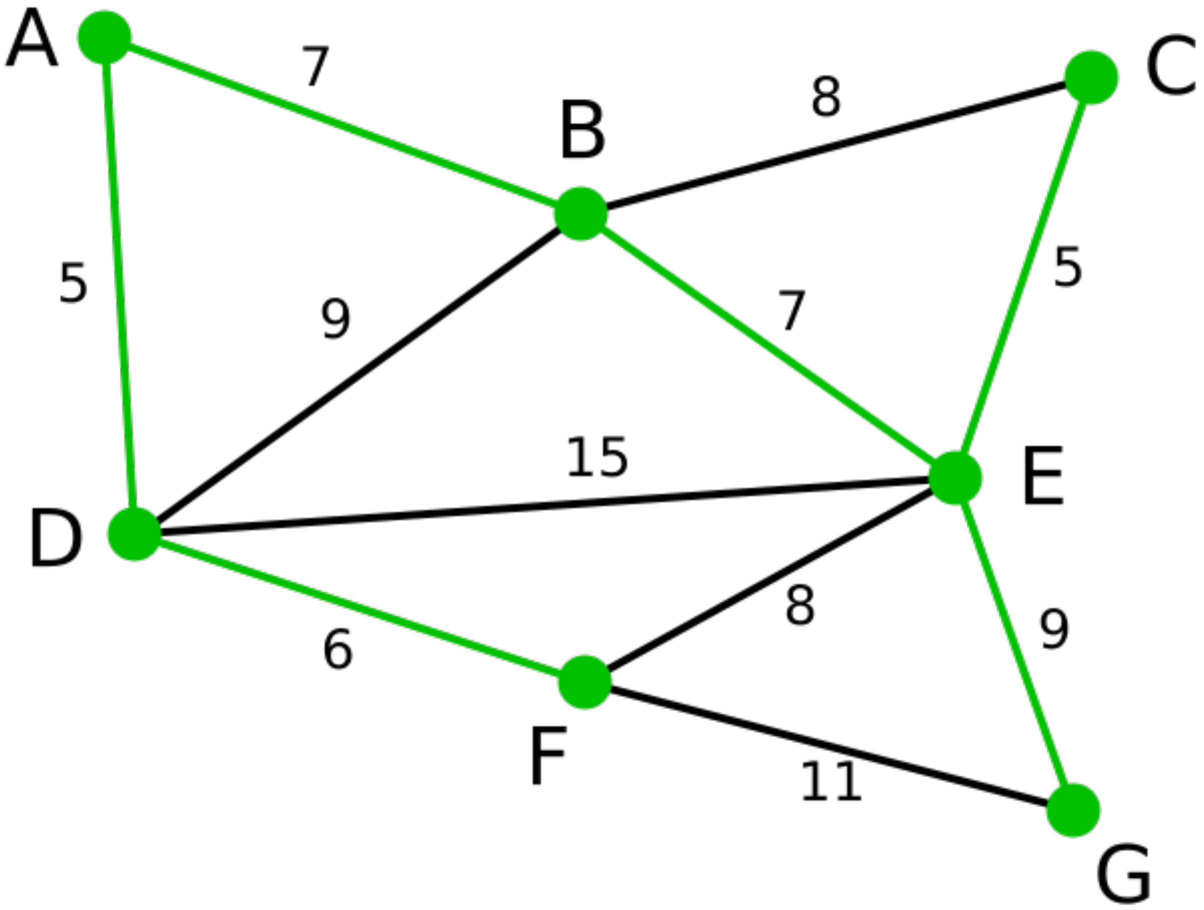


6. G is the only remaining vertex. E is chosen.



7. The finally obtained minimum spanning tree

⇒ the total weight is 39



Applications of Prim's

- Maximum spanning trees
- Minimum product spanning trees
- Minimum bottleneck spanning tree

Weighted Graph Data Type

```
typedef struct {
        int v;                          /* neighboring vertex */
        int weight;                     /* edge weight */
} edge;

typedef struct {
        edge edges[MAXV+1][MAXDEGREE];  /* adjacency info */
        int degree[MAXV+1];             /* outdegree of vertex */
        int nvertices;                  /* number of vertices */
        int nedges;                     /* number of edges in graph */
} graph;
```

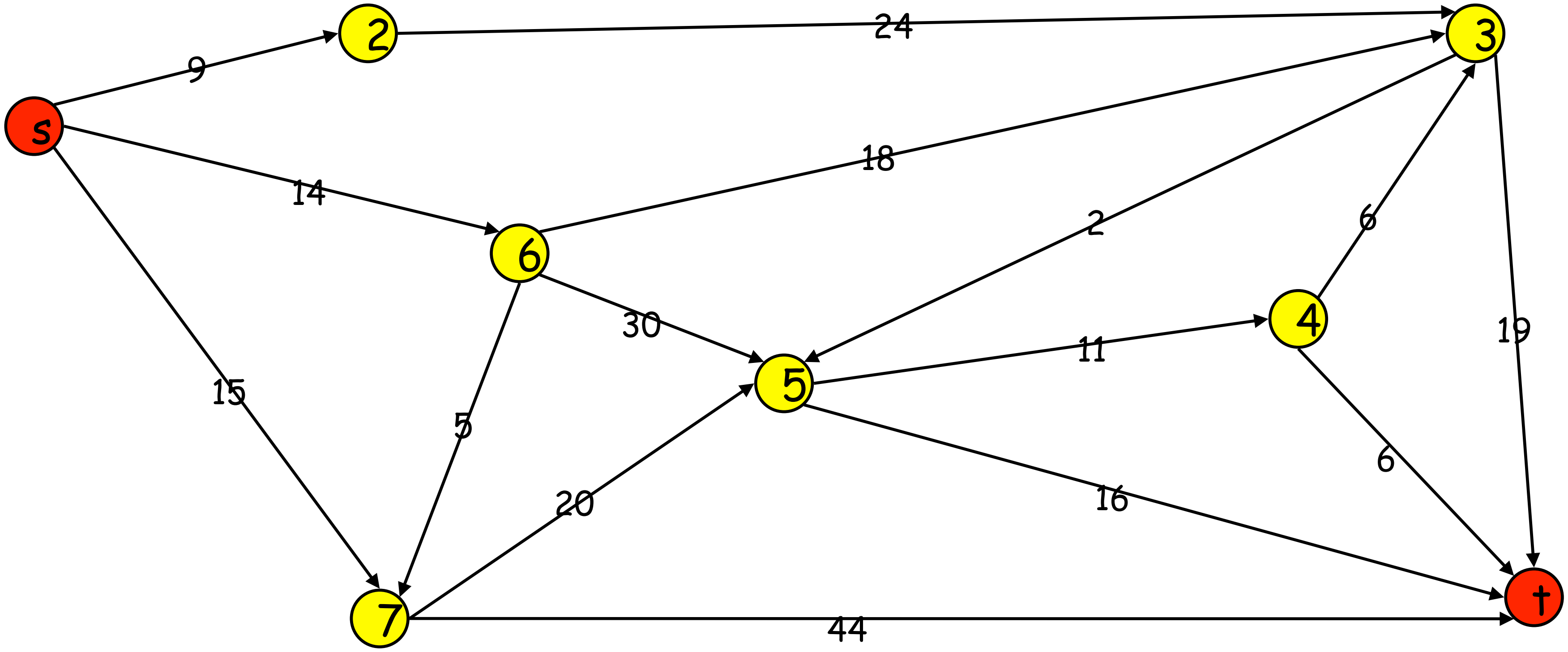
Prim’s Algorithm

```
prim(graph *g, int start) {
   int i,j;                        /* counters */
   bool intree[MAXV];              /* is vertex in the tree yet? */
   int distance[MAXV];             /* vertex distance from start */
   int v;                          /* current vertex to process */
   int w;                          /* candidate next vertex */
   int weight;                     /* edge weight */
   int dist;                       /* shortest current distance */

   for (i=1; i<=g->nvertices; i++) {
           intree[i] = FALSE;
           distance[i] = MAXINT;
           parent[i] = -1;
   }
   distance[start] = 0;
   v = start;

   while (intree[v] == FALSE) {
       intree[v] = TRUE;
       for (i=0; i<g->degree[v]; i++) {
            w = g->edges[v][i].v;
            weight = g->edges[v][i].weight;
            if ((distance[w] > weight) && (intree[w]==FALSE)) {
                    distance[w] = weight;
                    parent[w] = v;
            }
       }

       v = 1;
       dist = MAXINT;
       for (i=2; i<=g->nvertices; i++)
            if ((intree[i]==FALSE) && (dist > distance[i])) {
                    dist = distance[i];
                         v = i;
                 }
        }
}
```

Shortest Path

- For unweighted graphs
  - BFS suffices
- For weighted graphs
  - Dijkstra's algorithm
- Dijkstra's algorithm is very similar to Prim's

Dijkstra's Algorithm



Dijkstra’s Algorithm

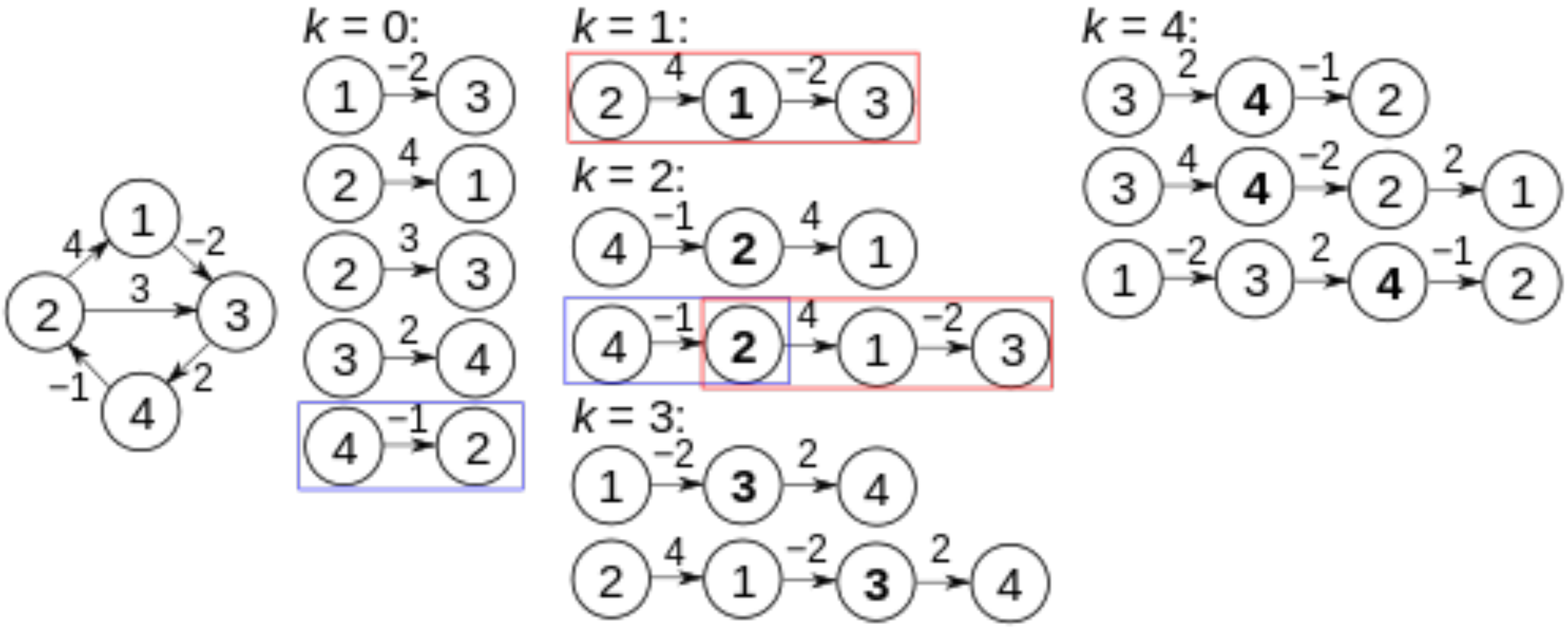
```
dijkstra(graph *g, int start)           /* WAS prim(g,start) */
{
   int i,j;                        /* counters */
   bool intree[MAXV];              /* is vertex in the tree yet? */
   int distance[MAXV];             /* vertex distance from start */
   int v;                          /* current vertex to process */
   int w;                          /* candidate next vertex */
   int weight;                     /* edge weight */
   int dist;                       /* shortest current distance */

   for (i=1; i<=g->nvertices; i++) {
           intree[i] = FALSE;
           distance[i] = MAXINT;
           parent[i] = -1;
   }
   distance[start] = 0;
   v = start;

   while (intree[v] == FALSE) {
       intree[v] = TRUE;
       for (i=0; i<g->degree[v]; i++) {
            w = g->edges[v][i].v;
            weight = g->edges[v][i].weight;
/* CHANGED */    if (distance[w] > (distance[v]+weight)) {
/* CHANGED */            distance[w] = distance[v]+weight;
                         parent[w] = v;
                 }
       }
       v = 1;
       dist = MAXINT;
       for (i=2; i<=g->nvertices; i++)
            if ((intree[i]==FALSE) && (dist > distance[i])) {
                    dist = distance[i];
                    v = i;
            }
   }
}
```

All Pairs Shortest Path

- When you want to know the center vertex for a new business
  - Dijksta’s?
- We need another one
  - Floyd’s all pairs shortest path algorithm

All Pairs Shortest Path

- Floyd’s algorithm
  - W(i,j,k) = min(W(i,j,k-1), W(i,k,k-1)+W(k,j,k-1))



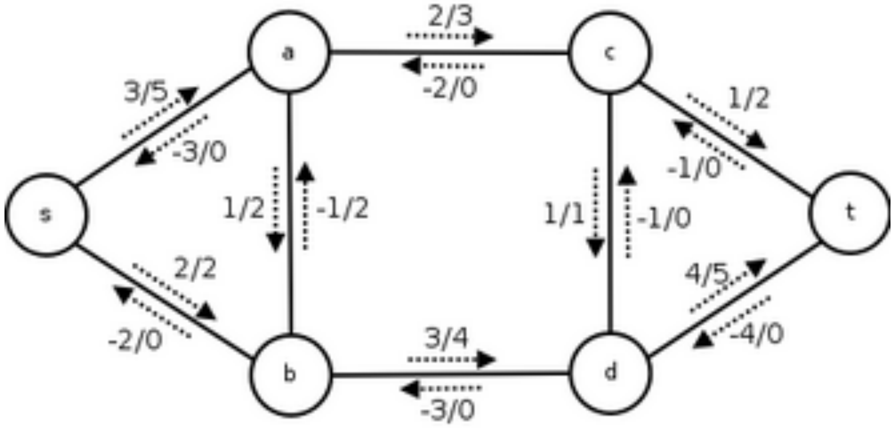
Transitive Closure

- Sometimes we are interested in which vertices are reachable from a given node
- The Blackmail problem

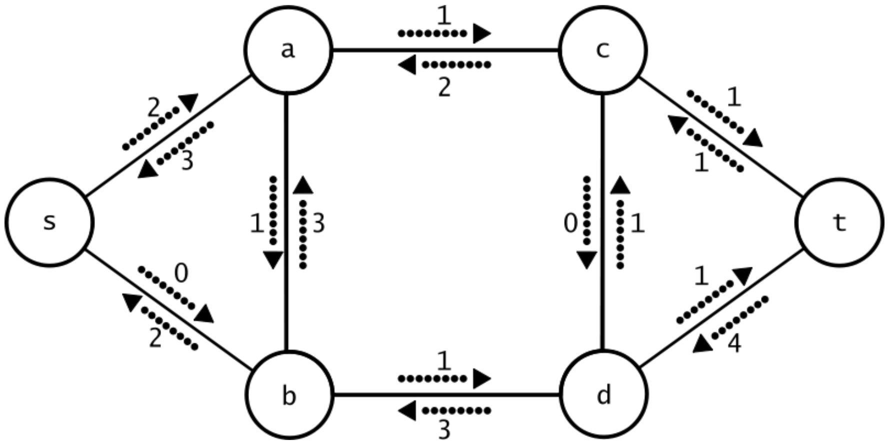
Network Flow

- Any edge-weighted graph can be thought of as a network of pipes
  - Weight of edge (i, j) measures the capacity of the pipe
- Network flow problem
  - Given source and sink vertices of a graph
  - The maximum amount of flow which can be sent from source to sink while respecting the maximum capacities of each pipe

Network Flow

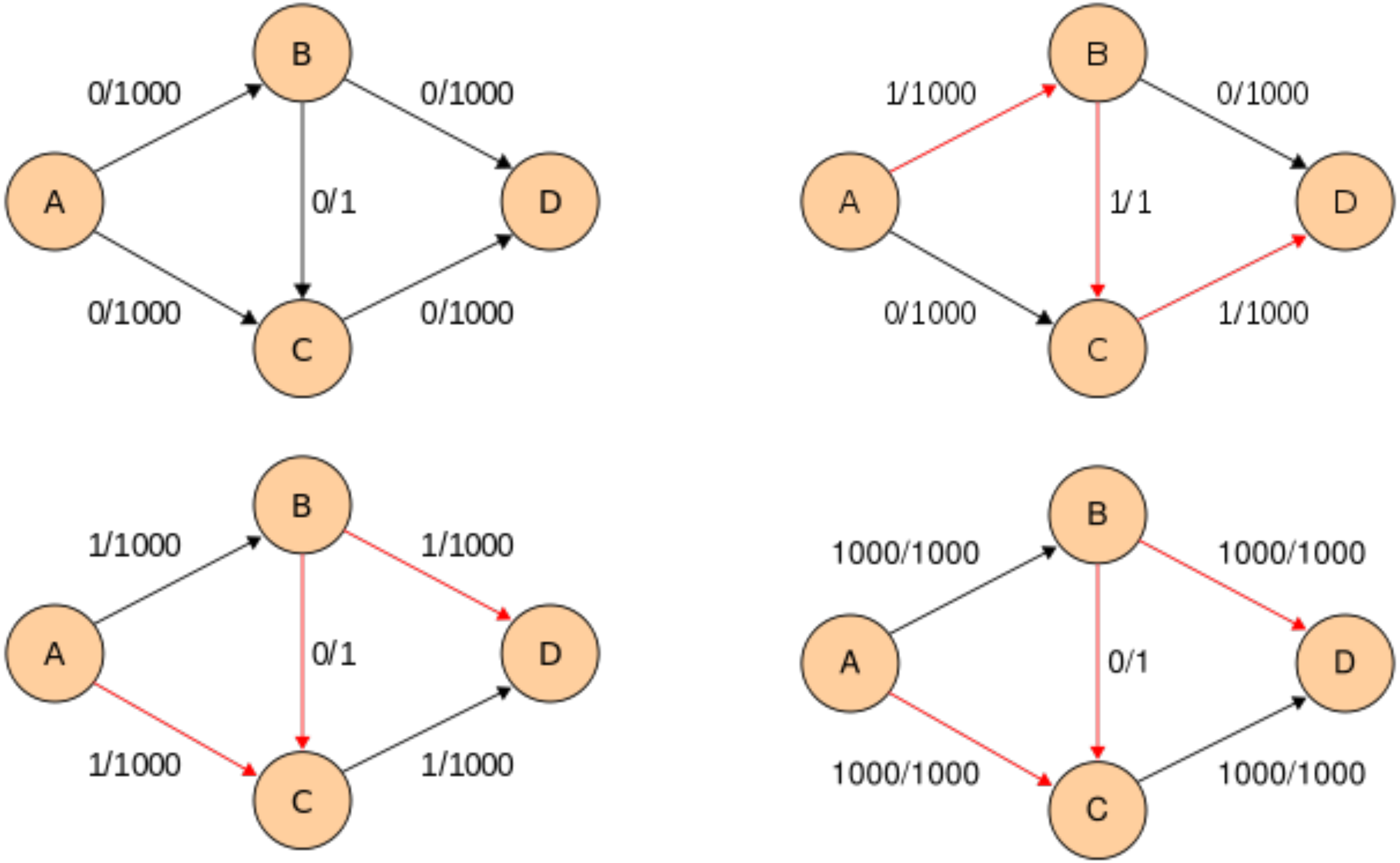


Flow network (flow/capacity)



Residual network for the above flow network

Ford-Fulkerson Algorithm



Bipartite Graphs

- A graph G is bipartite or two-colorable if the vertices can be divided into two sets
- A matching in a graph G = (V, E) is a subset of edges E’ ⊂ E such that no two edges in E’ share a vertex
  - Job-worker pairs
  - Marriage graphs
- The largest possible bipartite matching can be found using network flow

Problem: Fire Station

A city is served by a number of fire stations. Residents have complained that the distance between certain houses and the nearest station is too far, so a new station is to be built. You are to choose the location of the new station so as to reduce the distance to the nearest station from the houses of the poorest-served residents.

The city has up to 500 intersections, connected by road segments of various lengths. No more than 20 road segments intersect at a given intersection. The locations of houses and fire stations alike are considered to be at intersections. Furthermore, we assume that there is at least one house associated with every intersection. There may be more than one fire station per intersection.

Problem: Fire Station

Input

The input begins with a single line indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive inputs.

The first line of input contains two positive integers: the number of existing fire stations f ($f \leq 100$) and the number of intersections i ($i \leq 500$). Intersections are numbered from 1 to i consecutively. Then f lines follow, each containing the intersection number at which an existing fire station is found. A number of lines follow, each containing three positive integers: the number of an intersection, the number of a different intersection, and the length of the road segment connecting the intersections. All road segments are two-way (at least as far as fire engines are concerned), and there will exist a route between any pair of intersections.

Output

For each test case, output the lowest intersection number at which a new fire station can be built so as to minimize the maximum distance from any intersection to its nearest fire station. Separate the output of each two consecutive cases by a blank line.

Problem: Fire Station

Sample Input

```
1

1 6
2
1 2 10
2 3 10
3 4 10
4 5 10
5 6 10
6 1 10
```

Sample Output

```
5
```