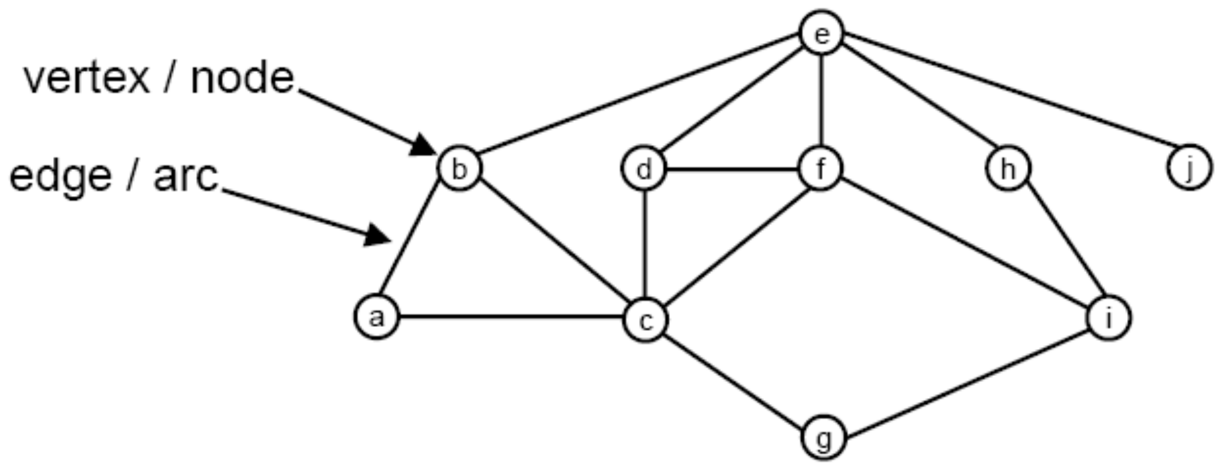
Graphs

EECS 233

Many Data Structures Learned

- Lists (arrays, linked lists)
- Stacks and queues
- Trees
  - With various structural and key ordering constraints
  - In-memory and external
- Hash tables
  - In-memory and external (extendible)
- Next up: graphs

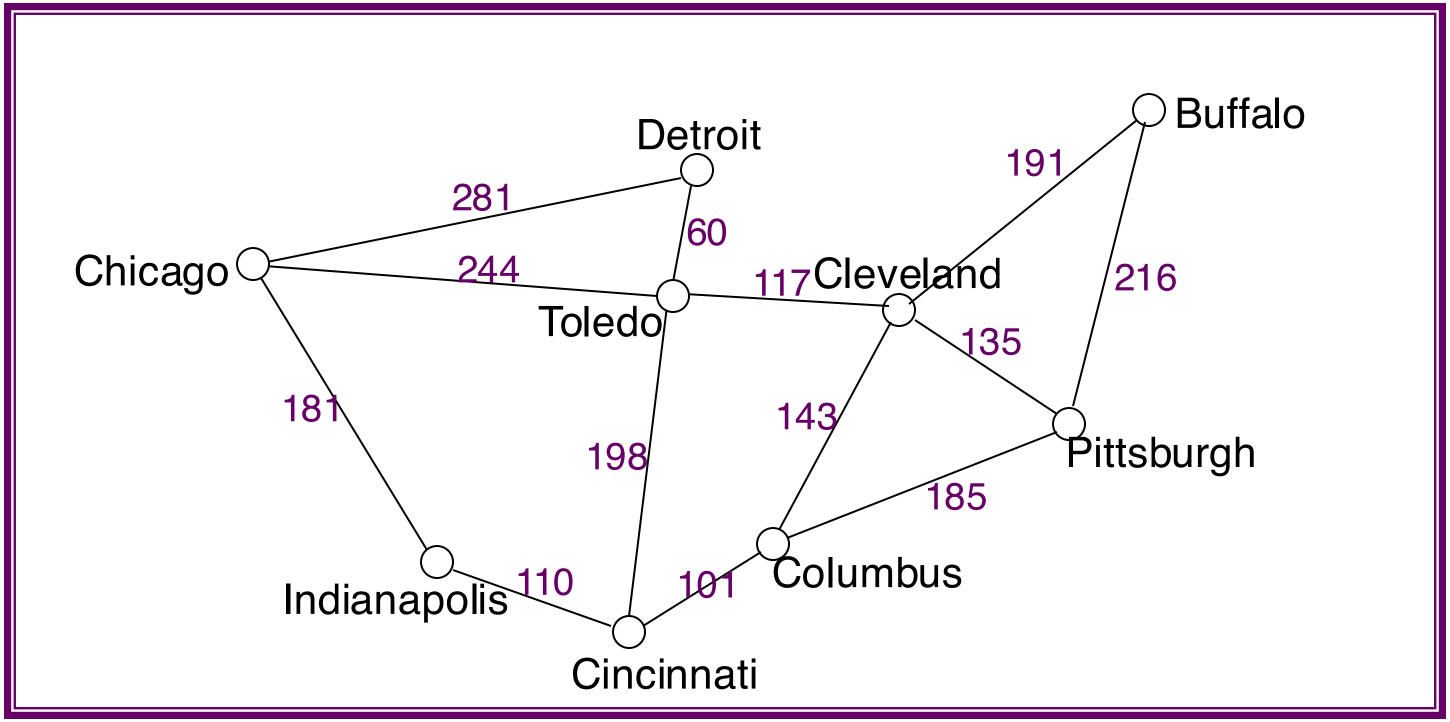
What Is A Graph?



- A graph consists of:
  - a set of *vertices* (also known as *nodes*)
  - a set of *edges* (also known as *arcs*), each of which connects a pair of vertices

- Q: Can any two vertices reach each other?

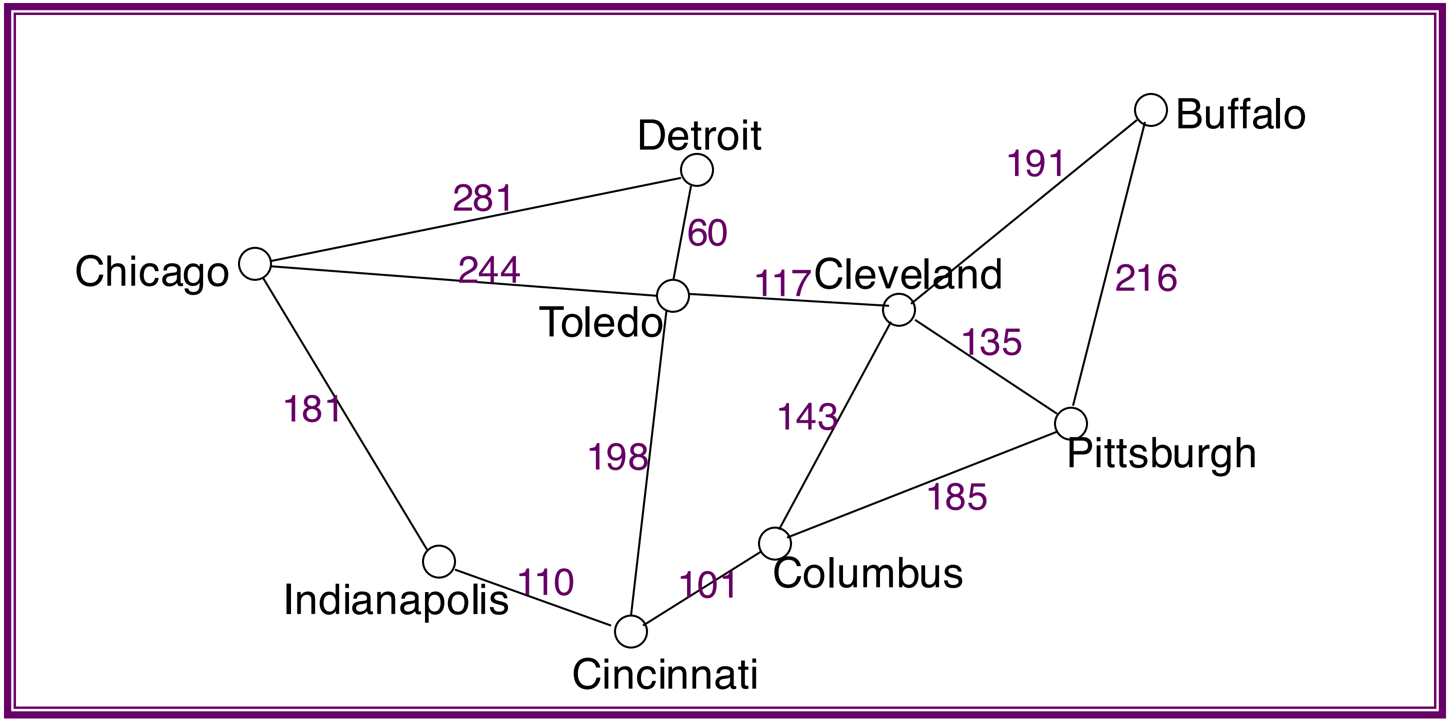
Example: A Highway Map



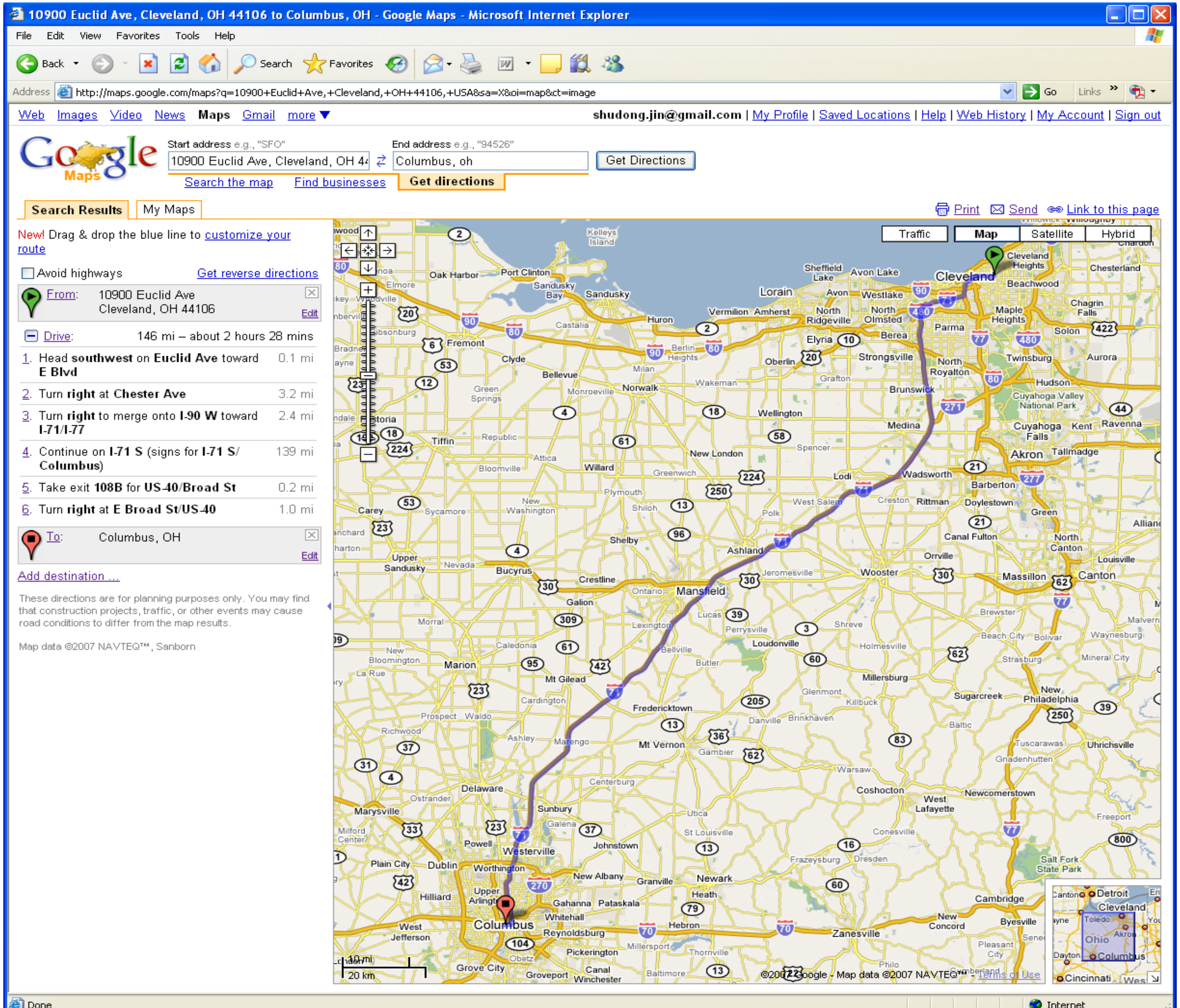
- Vertices represent cities, and edges represent highways.
- This is a *weighted* graph: it associates a *cost* with each edge (in this example, the cost represents mileage)

Why Graphs

- Many applications of graphs (and hence graph data structures and algorithms)
- Example 1: What is the shortest path between Pittsburgh and Chicago in the example graph?



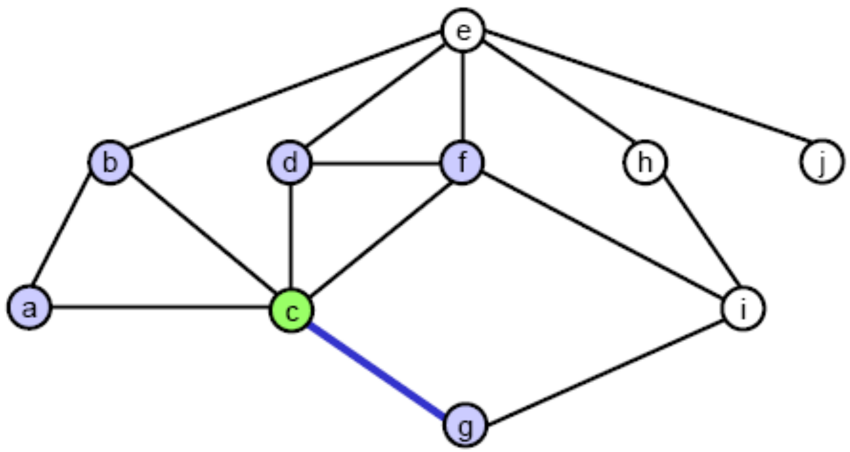
Google Maps uses graph(s)



Social network is a graph

- People = nodes
- Friend relationships = undirected edges
- “Following” relationship = directed edges
- Degree of separation = shortest paths

Terminology

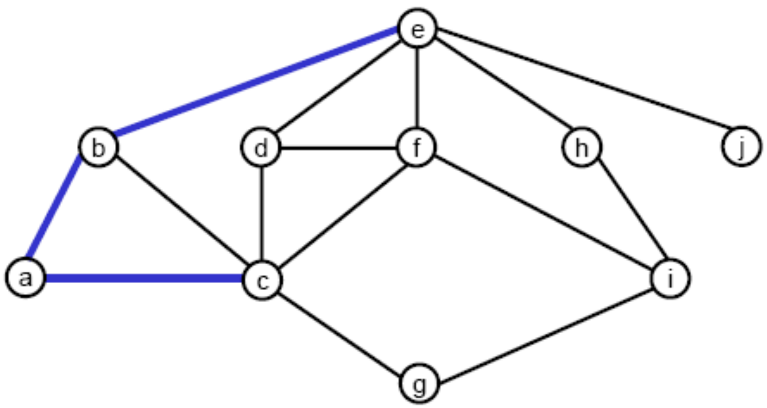


- Two vertices are ***adjacent*** if they are connected by a single edge.
  - *example:* c and g are adjacent, but c and i are not

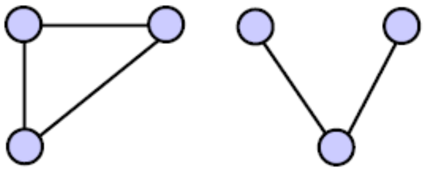
- The collection of vertices that are adjacent to a vertex v are referred to as v's ***neighbors***.
  - *example:* c's neighbors are a, b, d, f, and g

Terminology

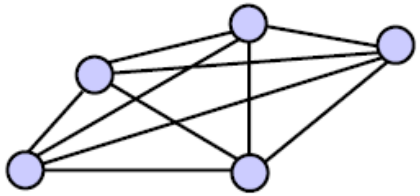
- A ***path*** is a sequence of edges that connects two vertices.
  - *example:* the path connects c and e



- A graph is ***connected*** if there is a path between any two vertices.
  - *example:* the graph at right is *not* connected

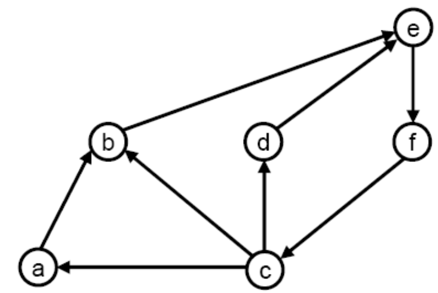


- A graph is ***complete*** if there is an edge between every pair of vertices.
  - *example:* the graph at right is complete



Directed versus Undirected Graphs

- A ***directed* graph** has a direction associated with each edge, which is depicted using an arrow:

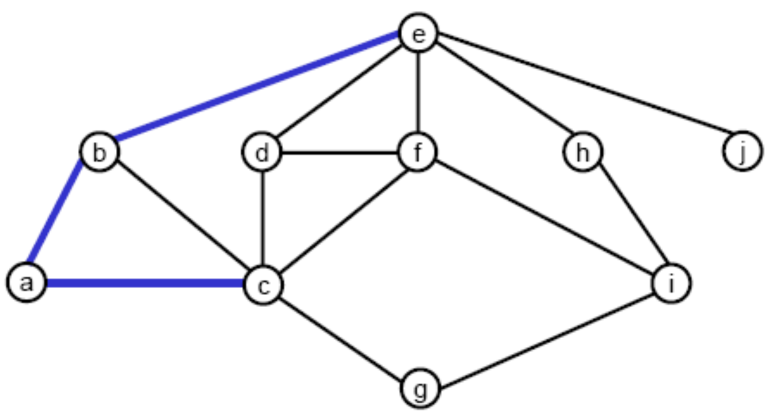


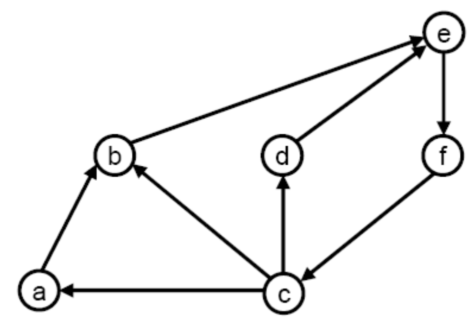
- Edges in a directed graph are often represented as ordered pairs of the form (start vertex, end vertex).
  - *example:* (a, b) is an edge in the graph above, but (b, a) is not.

- **A path in a directed graph** is a sequence of edges in which the end vertex of edge *i* must be the same as the start vertex of edge *i* + 1.
  - *ex:* { (a, **b**), (**b**, **e**), (**e**, f) } is a valid path.
  - { (a, **b**), (**c**, **b**), (**c**, a) } is not.

More Terminology

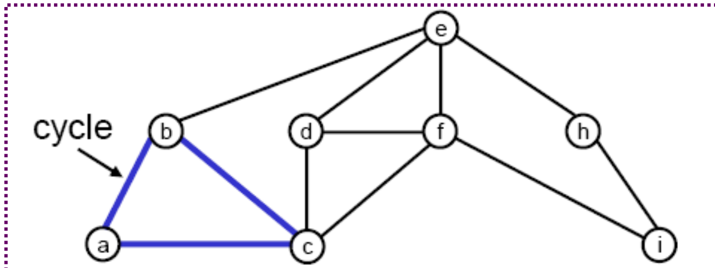
- Degree
  - In-degree and out-degree
- Distance, hop-distance or path length
- DAG – directed acyclic graph



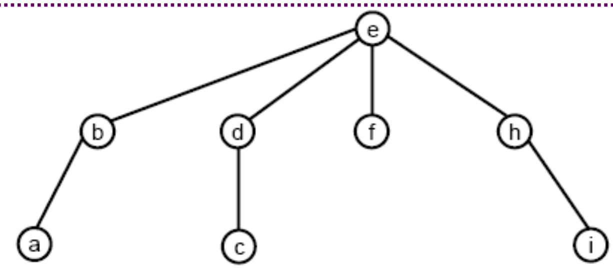


Trees versus Graphs

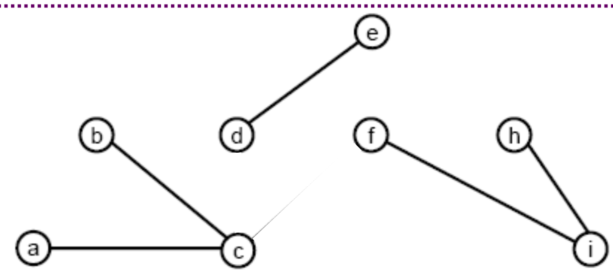
- A tree is a special type of graph.
  - it is connected and undirected
  - it is *acyclic:* there is no path containing distinct edges that starts and ends at the same vertex
  - we usually single out one of the vertices to be the root of the tree, although graph theory doesn't require this



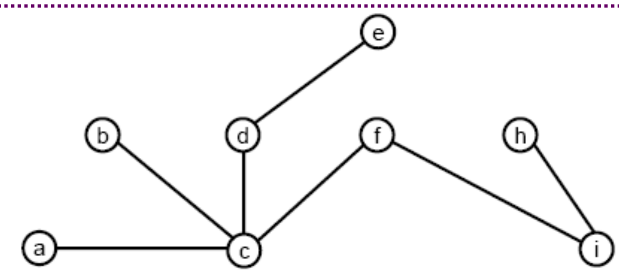
a graph that is *not* a tree



a tree using the same nodes



A graph that is not a tree

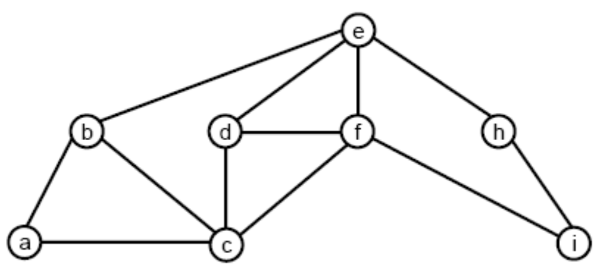


a tree using the same nodes

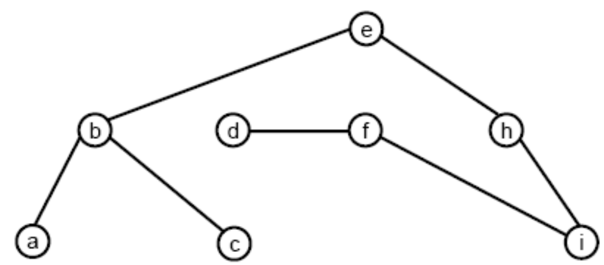
Spanning Trees

- A spanning tree is a subset of a connected graph that contains:
  - all of the *N* vertices
  - a subset of the edges that form a tree (the subset contains *N*-1 edges)

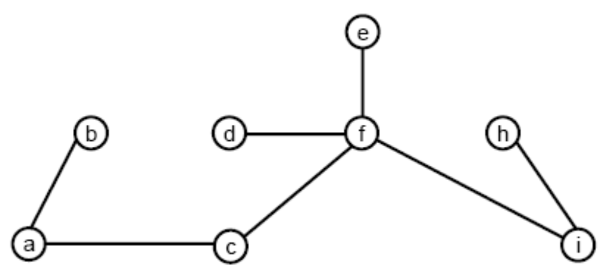
- Examples:



the original graph



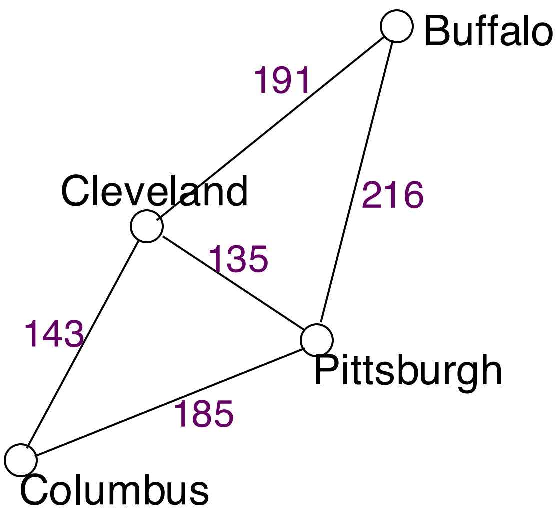
another spanning tree for this graph



another spanning tree for this graph

Graph Representation - Matrix

- Adjacency matrix = a two-dimensional array that is used to represent the edges and any associated costs
  - edge[r][c] = the cost of the link from vertex r to vertex c
  - Use a special value to there is no direct link from r to c.

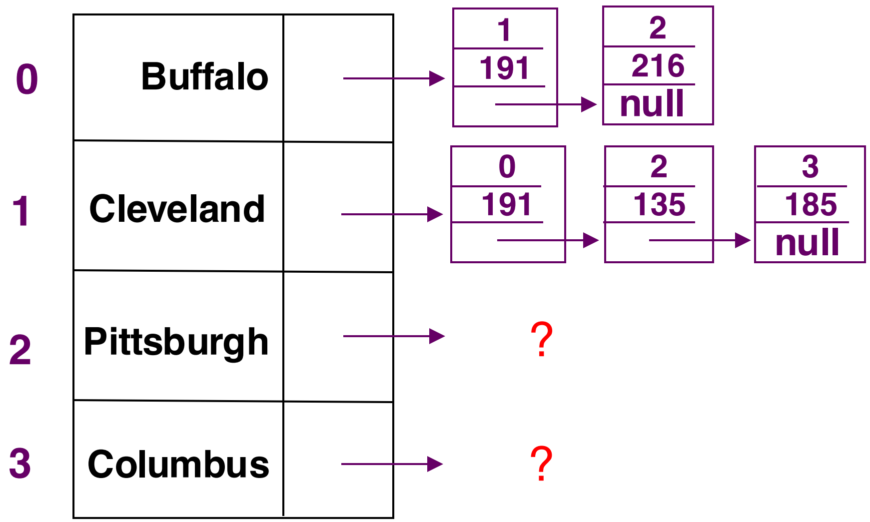


	Buf	Cle	Pitt	Col
Buffalo	**X**	**191**	**216**	**?**
Cleveland	**191**	**X**	**135**	**143**
Pittsburgh	**216**	**135**	**X**	**185**
Columbus	**?**	**143**	**185**	**X**

- This representation is good if the graph is *dense* – if most nodes are connected – but wastes memory if the graph is *sparse* – if it has few edges per vertex.

Graph Representation - Lists

- Adjacency lists = a set (either an array or linked list) of linked lists that is used to represent the edges and any associated costs



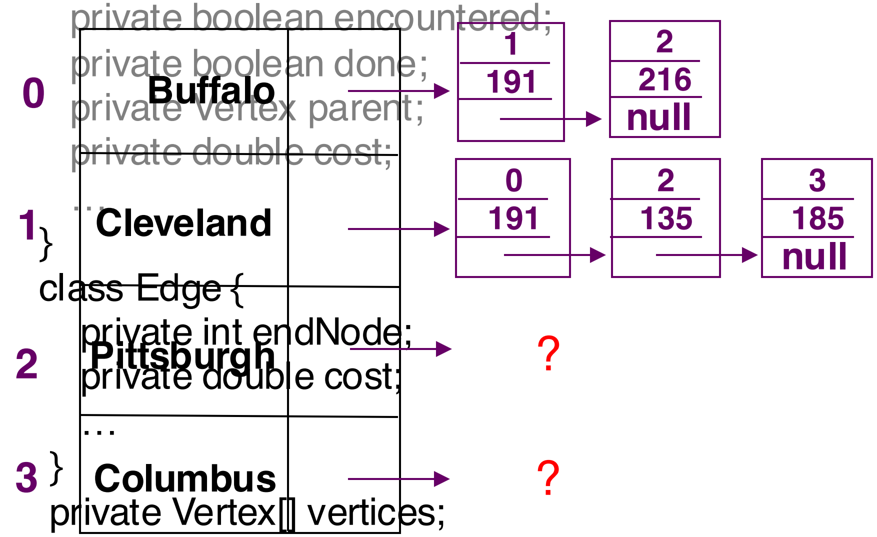
	Buf	Cle	Pitt	Col
Buffalo	**X**	**191**	**216**	**?**
Cleveland	**191**	**X**	**135**	**143**
Pittsburgh	**216**	**135**	**X**	**185**
Columbus	**?**	**143**	**185**	**X**

- This representation is good if a graph is sparse, but inefficient if the graph is dense.
  - no memory is allocated for non-existent edges
  - the references in the linked lists use extra memory
  - List traversal to find a link

Graph Representation with Lists

```
public class Graph {
        class Vertex {
                private String id;
                private Edge edgeHead; // adjacency list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                private Edge next;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

```
public class Graph {
    class Vertex {
        private String id;
        private LinkedList <Edge> edges; // adjacency list
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }
    class Edge {
        private int endNode;
        private double cost;
        …
    }
    private Vertex[] vertices;
    private int numVertices;
    private int maxNum;
        …
}
```



Adding Nodes

```
public class Graph {
        class Vertex {
                private String id;
                private LinkedList <Edge> edges; // adjacency
                    list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

```
public Graph(int maximum) {
        vertices = new Vertex[maximum];
        numVertices = 0;
        maxNum = maximum;
}

public int addNode(String id)
{
        // code to grow vertices if array “vertices”
        // is too small to have another node
        …

        vertices[numVertices] = new Vertex(id);
        // any further initialization
        …
        numVertices++;
        return numVertices-1; // return the index
}
```

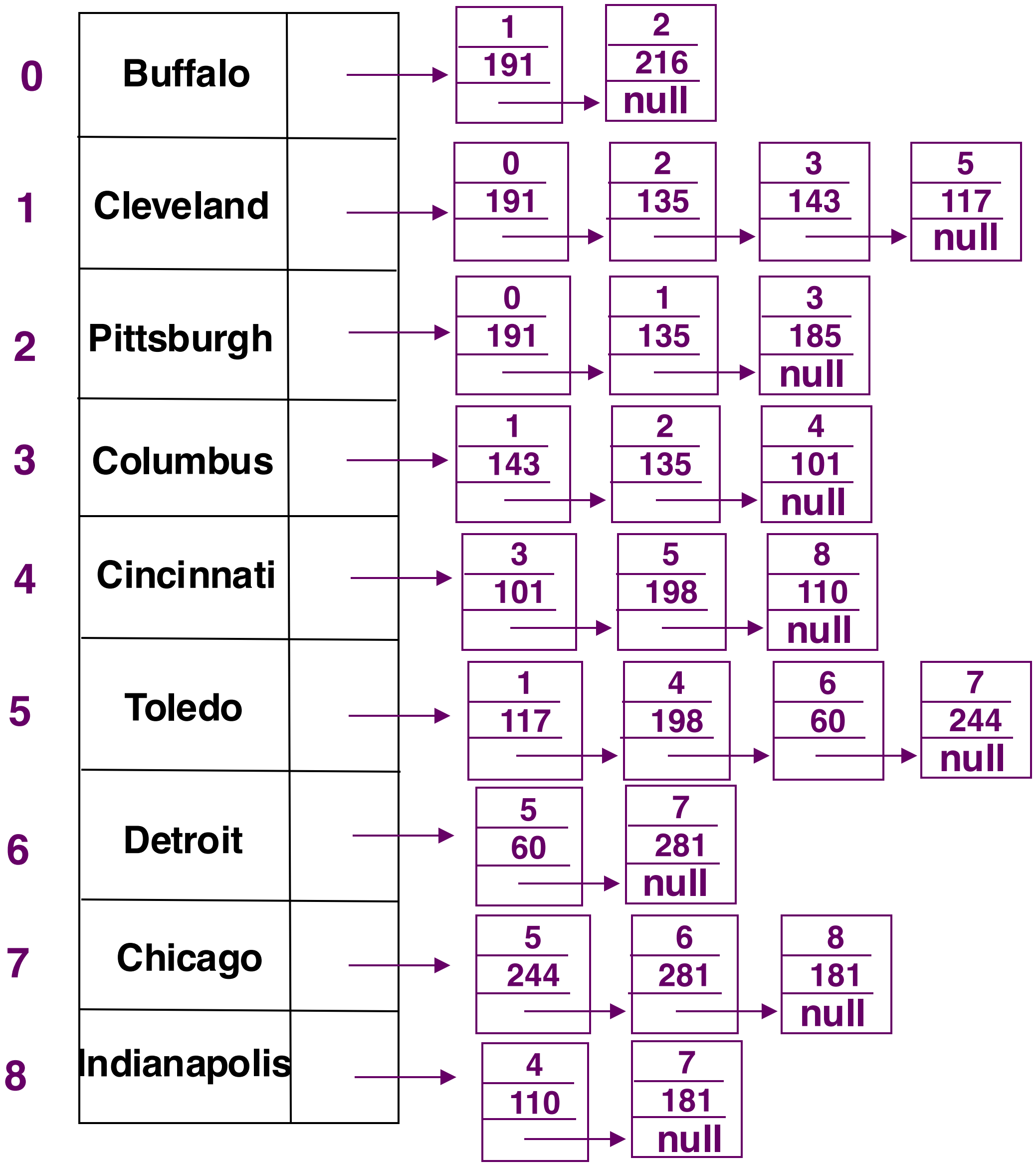
Adding Edges

```
public class Graph {
        class Vertex {
                private String id;
                private LinkedList <Edge> edges; // adjacency
                    list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

```
public void addEdge(int i, int j, double cost)
{
        // add an edge from i to j
        Edge newEdge = new Edge();
        newEdge.endNode = j;
        …

        vertices[i].edges.add(newEdge);


        // add an edge to j for node i
        newEdge = new Edge();
        newEdge.endNode = i;
        …
        vertices[j].edges.add(newEdge);
}
```


0 Buffalo
1 Cleveland
2 Pittsburgh
3 Columbus
4 Cincinnati
5 Toledo
6 Detroit
7 Chicago
8 Indianapolis
1 191
2 216 null
0 191
2 135
3 143
5 117 null
0 191
1 135
3 185 null
1 143
2 135
4 101 null
3 101
5 198
8 110 null
1 117
4 198
6 60
7 244 null
5 60
7 281 null
5 244
6 281
8 181 null
4 110
7 181 null