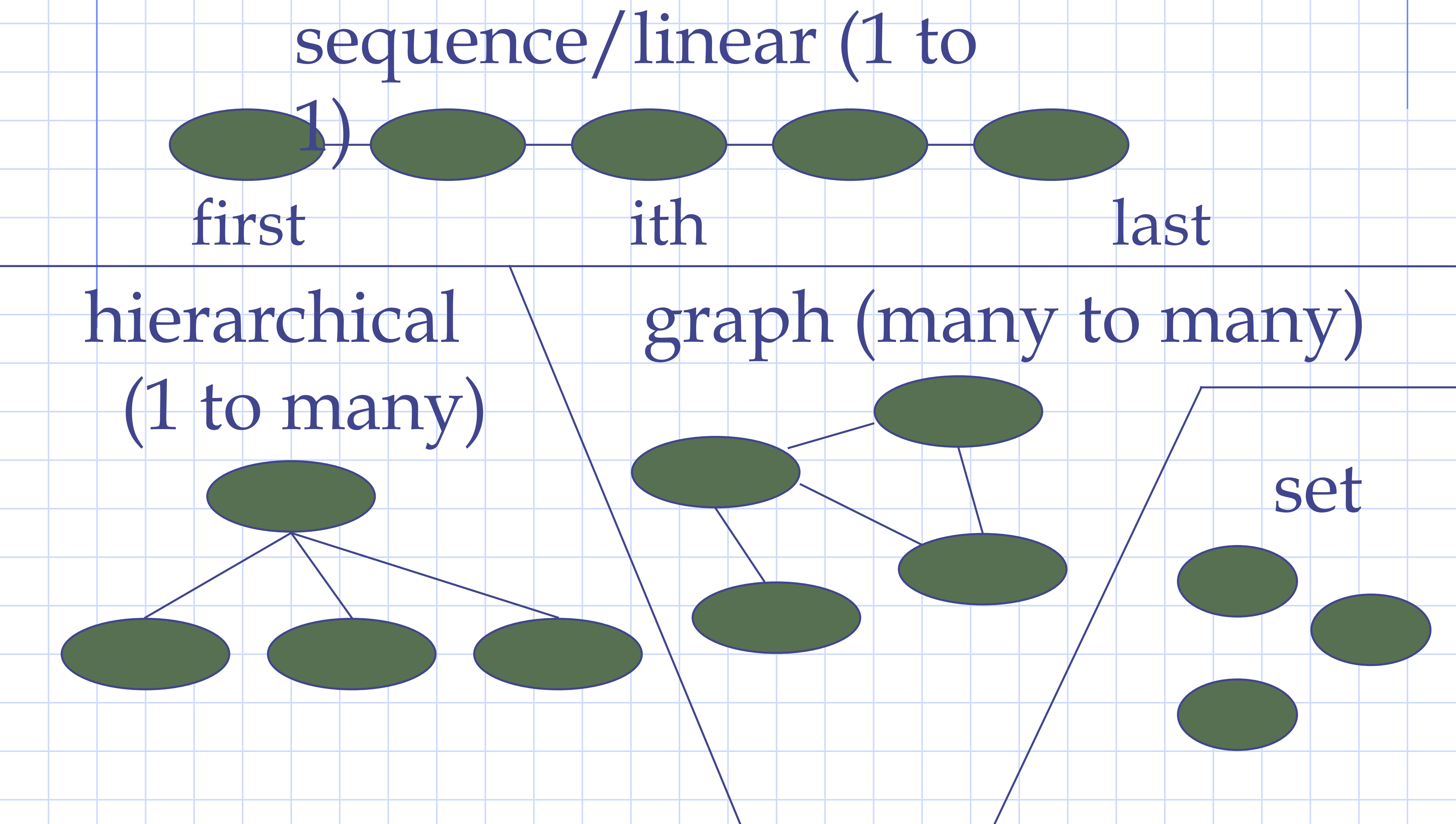
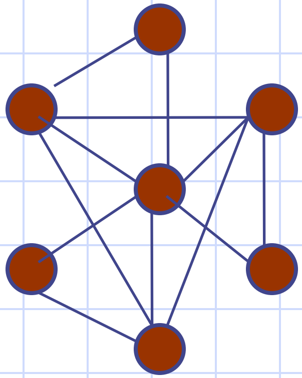
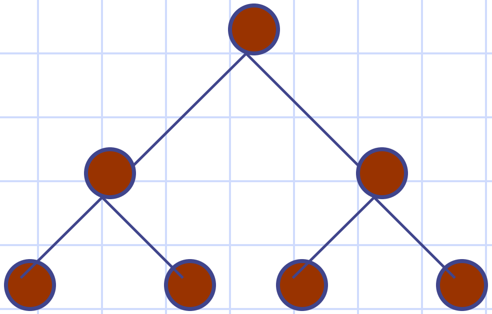
Graph

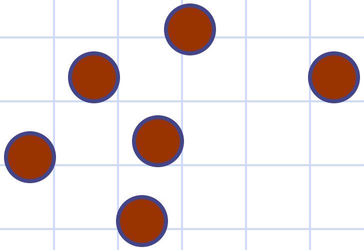
Abstract containers



Graph

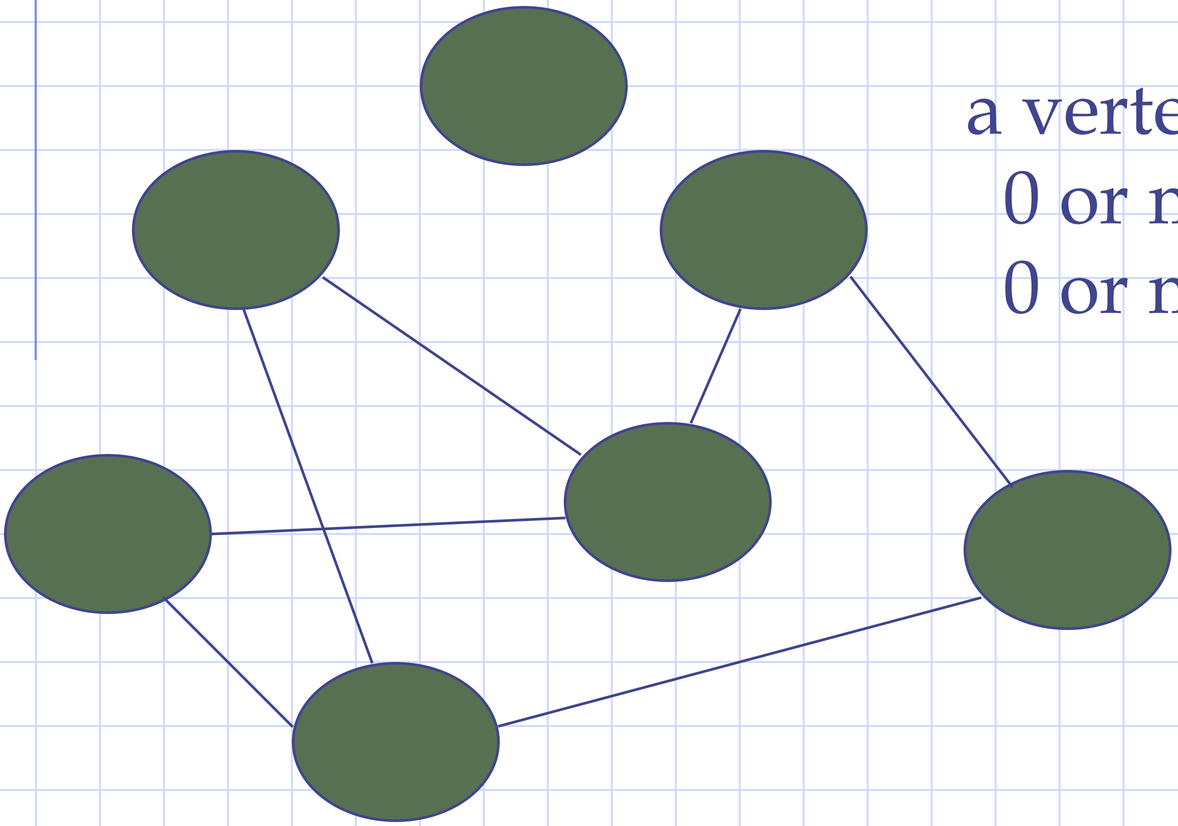






THESE ARE ALL GRAPHS.

G = (V, E)

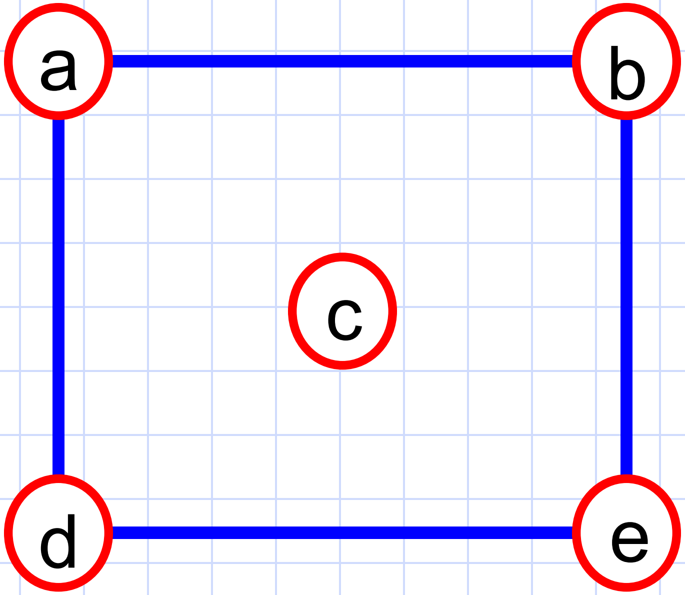


a vertex may have:

- 0 or more predecessors
- 0 or more successors

What is a Graph?

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



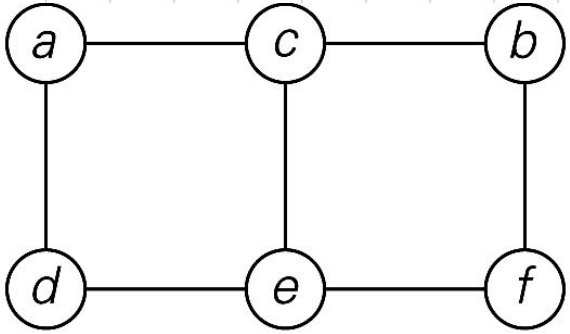
V= {a,b,c,d,e}

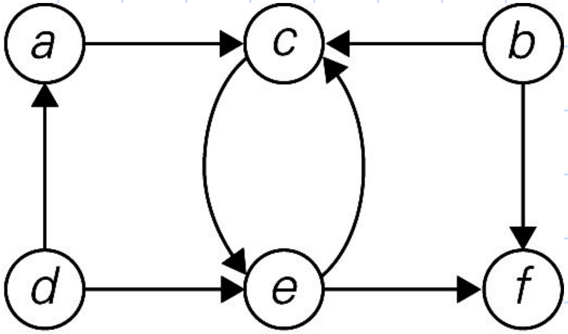
E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Graphs

- Formal definition
  - A graph *G = <V, E>* is defined by a pair of two sets: a finite set V of items called vertices and a set E of vertex pairs called edges.
- Undirected and directed graphs (digraph).
- What's the maximum number of edges in an undirected graph with |V| vertices?
- Complete, dense, and sparse graph
  - A graph with every pair of its vertices connected by an edge is called complete. $K_{|V|}$

(a)





(b)

Terminology:Adjacent and Incident

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

Terminology:Degree of a Vertex

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

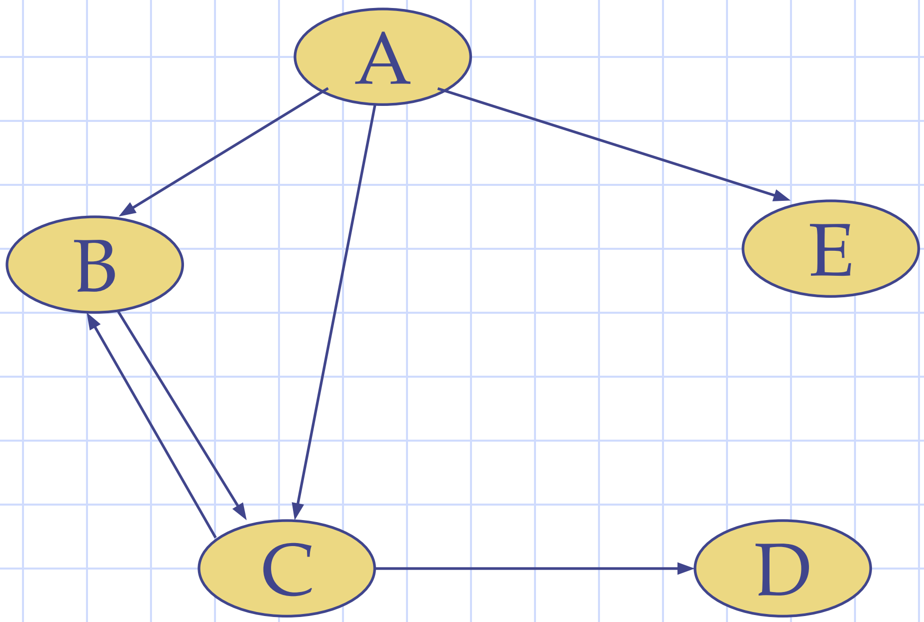
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

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

Graph variations

- undirected graph (graph)
  - edges do not have a direction
    - (V1, V2) and (V2, V1) are the same edge
- directed graph (digraph)
  - edges have a direction
    - <V1, V2> and <V2, V1> are different edges
- for either type, edges may be weighted or unweighted

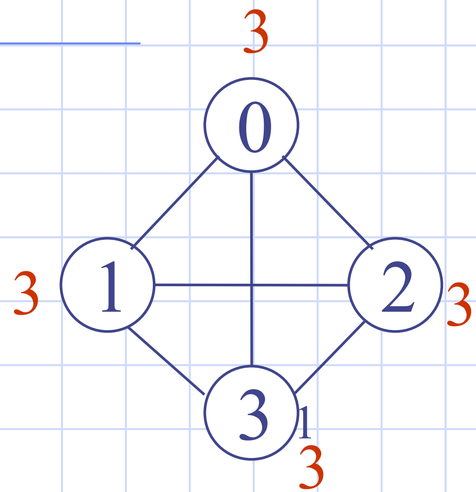
A digraph

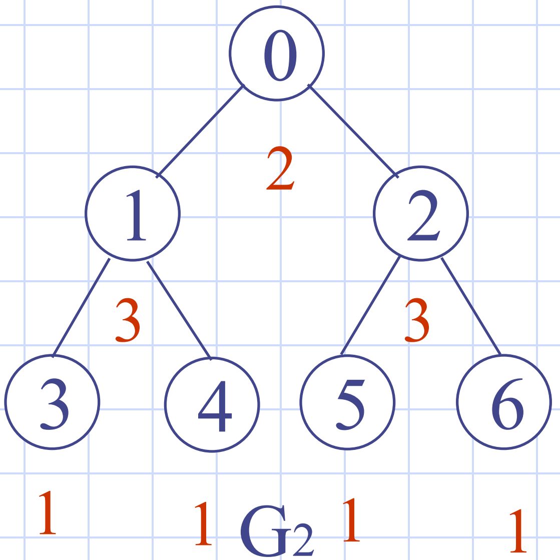


V = [A, B, C, D, E]

E = [<A,B>, <B,C>, <C,B>, <A,C>, <A,E>, <C,D>]

Examples

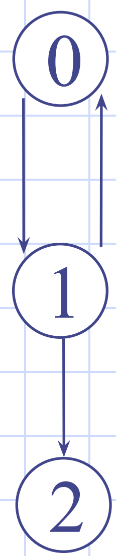




directed graph

in-degree

out-degree



in:1, out: 1

in: 1, out: 2

in: 1, out: 0

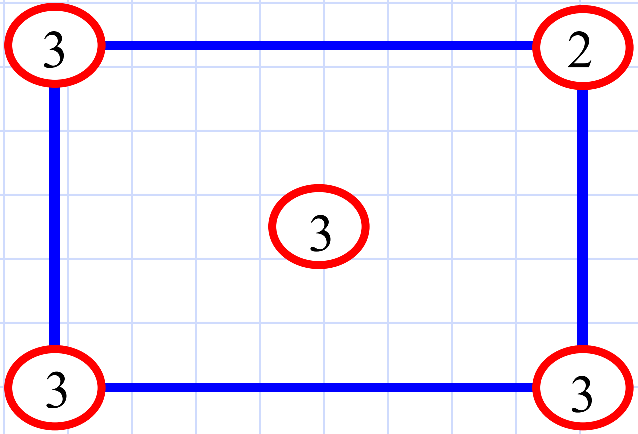
G3

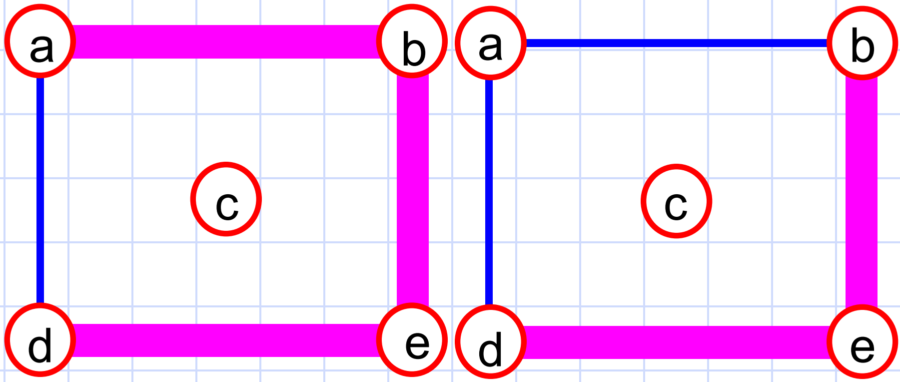
In-degree : the number of edges for which v is the head.

Out-degree : the number of edges for which v is the tail.

Terminology:Path

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





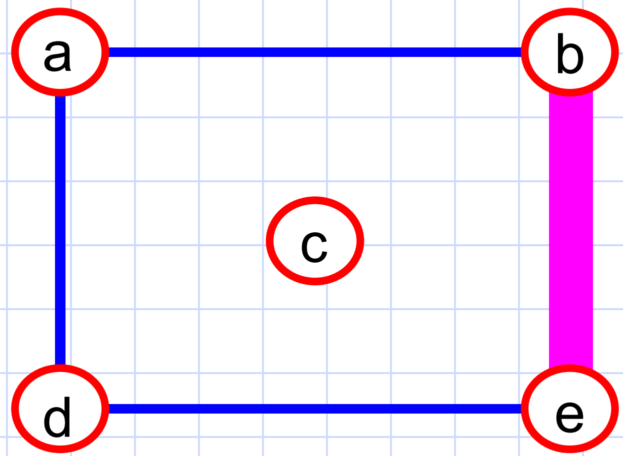
a b e d c

b e d c

More Terminology

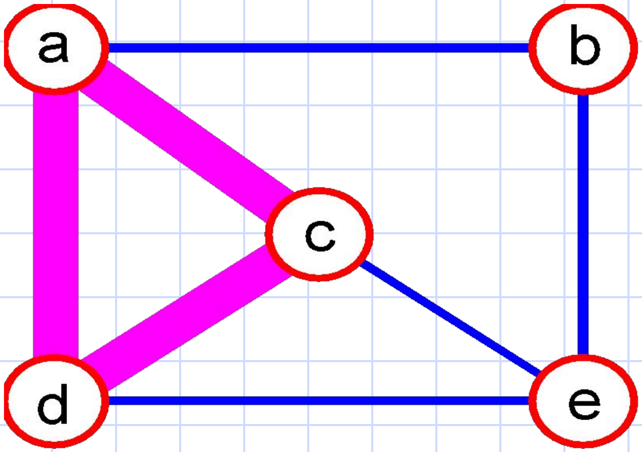
- simple path:

no repeated vertices

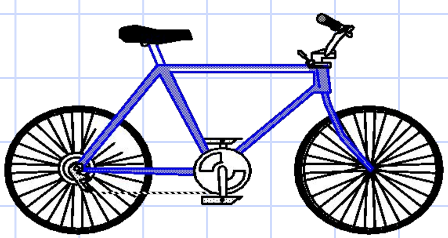


b e c

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

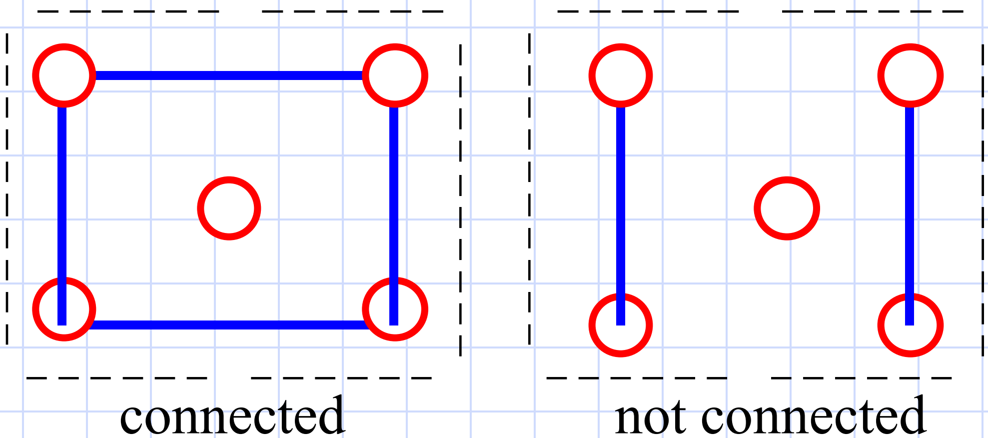


a c d a

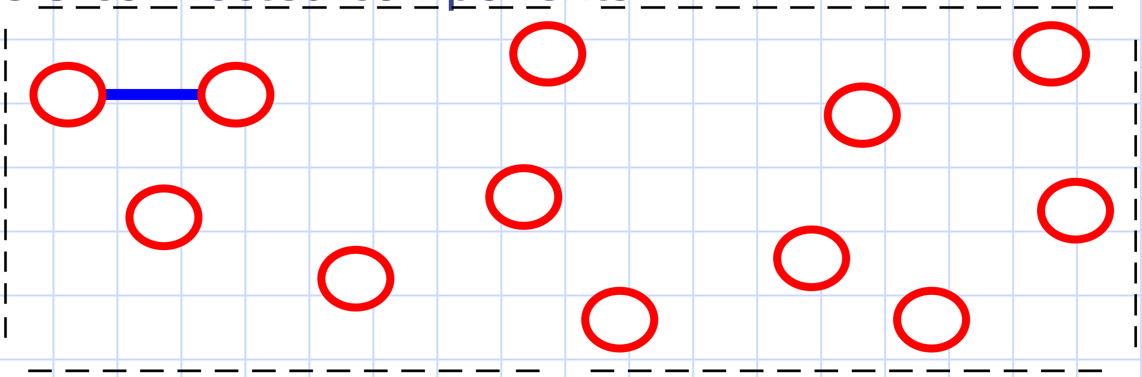


Even More Terminology

- connected graph: any two vertices are connected by some path

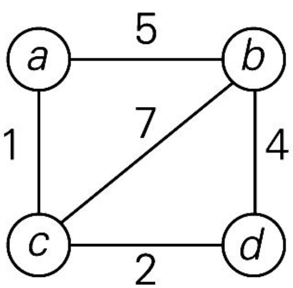


- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Weighted Graphs

- Weighted graphs
  - Graphs or digraphs with numbers assigned to the edges.



(a)

$$
\begin{array}{c|cccc}
 & a & b & c & d \\
\hline
a & \infty & 5 & 1 & \infty \\
b & 5 & \infty & 7 & 4 \\
c & 1 & 7 & \infty & 2 \\
d & \infty & 4 & 2 & \infty
\end{array}
$$

(b)

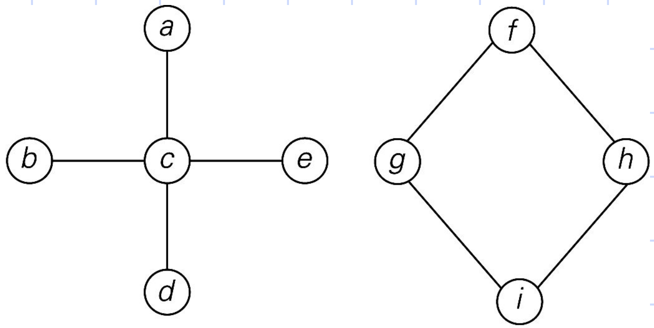
a	$\to b, 5 \to c, 1$
b	$\to a, 5 \to c, 7 \to d, 4$
c	$\to a, 1 \to b, 7 \to d, 2$
d	$\to b, 4 \to c, 2$

(c)

(a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Graph Properties -- Paths and Connectivity

- Paths
  - A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.
  - Simple paths: All edges of a path are distinct.
  - Path lengths: the number of edges, or the number of vertices – 1.
- Connected graphs
  - A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.
- Connected component
  - The maximum connected subgraph of a given graph.

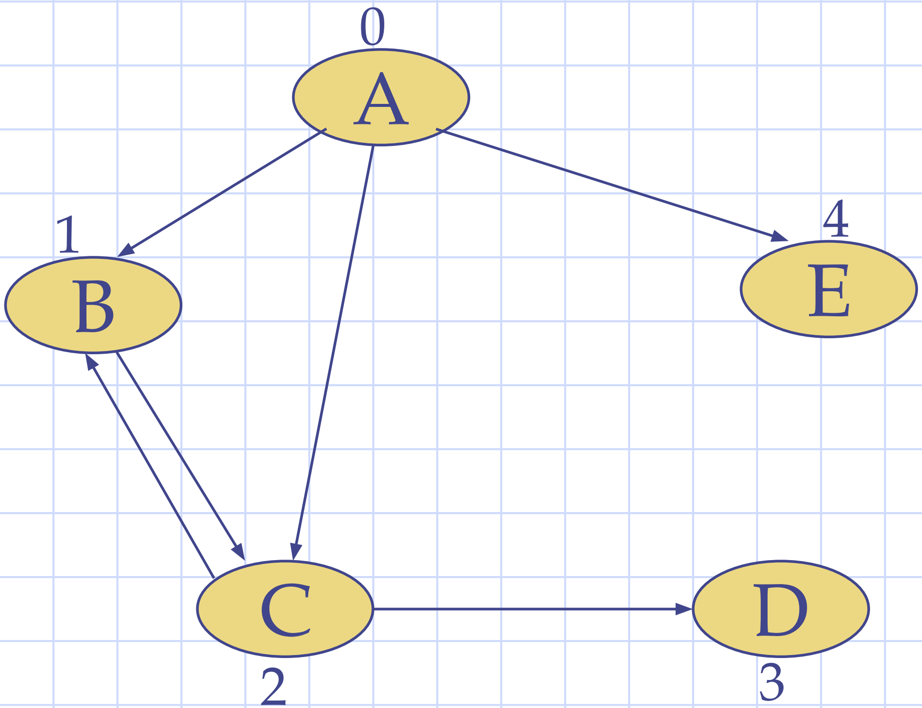


Graph that is not connected

Graph Properties -- Acyclicity

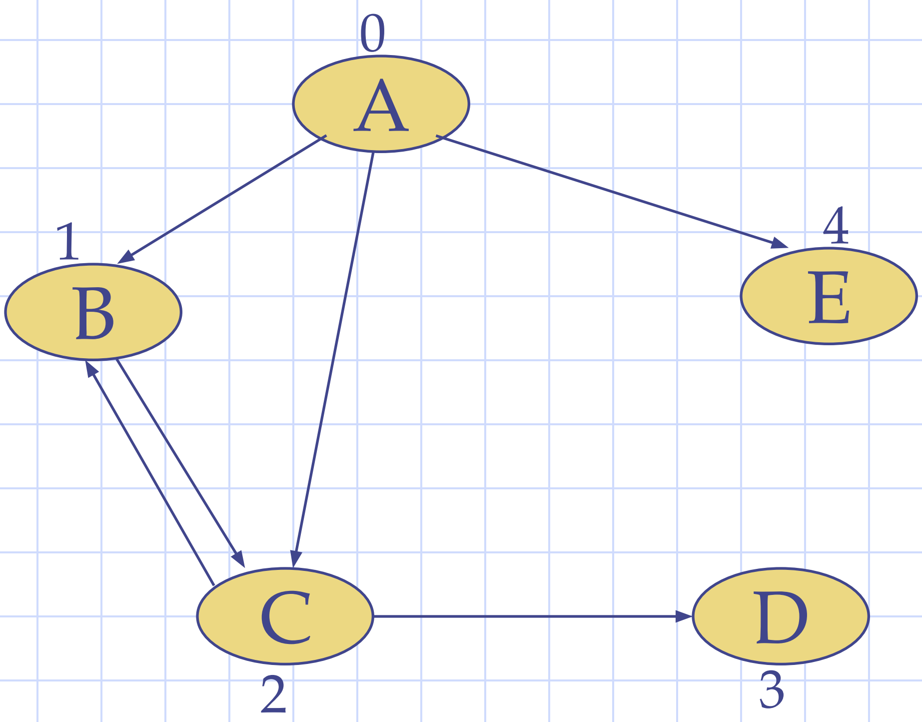
- Cycle
  - A simple path of a positive length that starts and ends at the same vertex.
- Acyclic graph
  - A graph without cycles
  - DAG (Directed Acyclic Graph)

The vertex vector



0	A
1	B
2	C
3	D
4	E

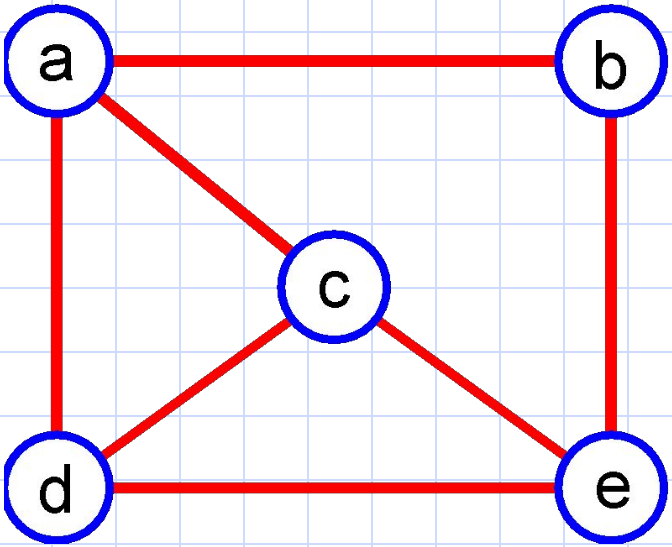
Adjacency matrix



0	A
1	B
2	C
3	D
4	E

	0	1	2	3	4
0	0	1	1	0	1
1	0	0	1	0	0
2	0	1	0	1	0
3	0	0	0	0	0
4	0	0	0	0	0

Adjacency Matrix (traditional)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- matrix M with entries for all pairs of vertices
- M[i,j] = true means that there is an edge (i,j) in the graph.
- M[i,j] = false means that there is no edge (i,j) in the graph.
- There is an entry for every possible edge, therefore:

  Space = $\Theta(N^2)$

Graph Representation

- Adjacency matrix
  - n x n boolean matrix if |V| is n.
  - The element on the ith row and jth column is 1 if there's an edge from ith vertex to the jth vertex; otherwise 0.
  - The adjacency matrix of an undirected graph is symmetric.
- Adjacency linked lists
  - A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.

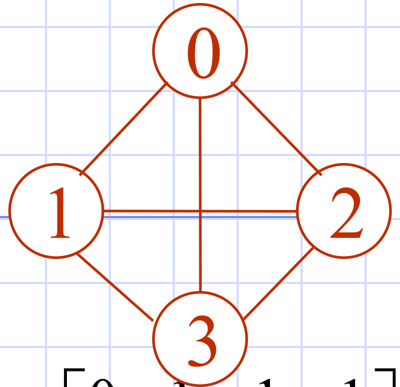
$$\begin{array}{cc} & \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} & \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{|c|l} \hline a & \rightarrow c \rightarrow d \\ \hline b & \rightarrow c \rightarrow f \\ \hline c & \rightarrow a \rightarrow b \rightarrow e \\ \hline d & \rightarrow a \rightarrow e \\ \hline e & \rightarrow c \rightarrow d \rightarrow f \\ \hline f & \rightarrow b \rightarrow e \\ \hline \end{array}$$

Adjacency Matrix

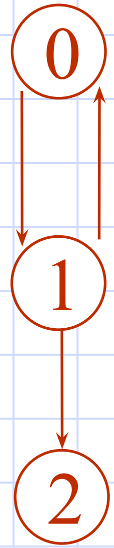
- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix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

Examples for Adjacency Matrix



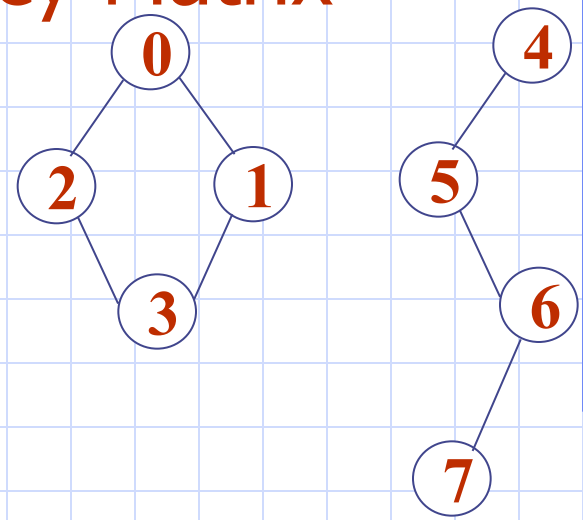
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



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

G_2



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

G_4

symmetric

undirected: $n^2/2$
directed: n^2

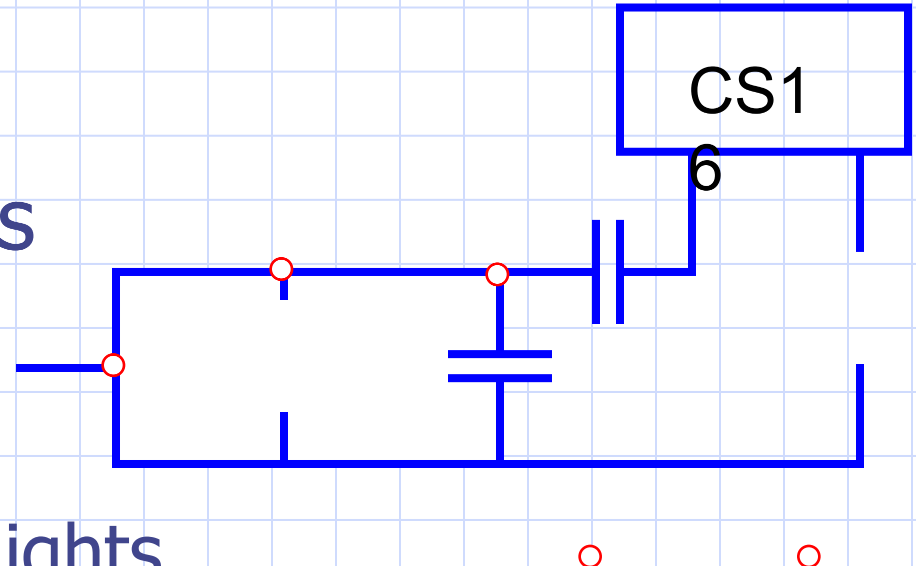
Merits of Adjacency Matrix

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

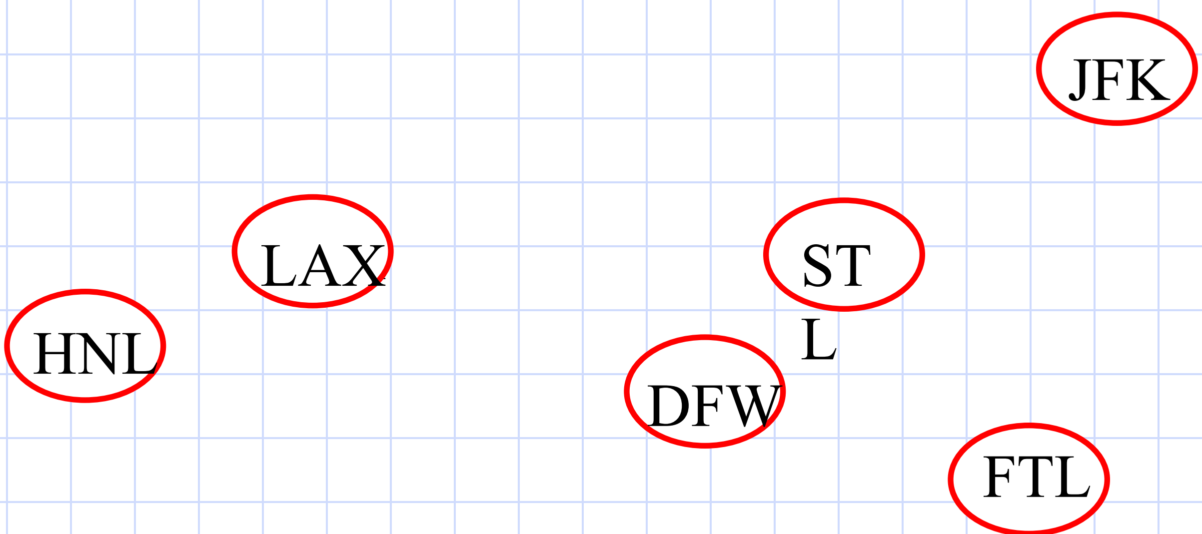
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

Applications

- electronic circuits



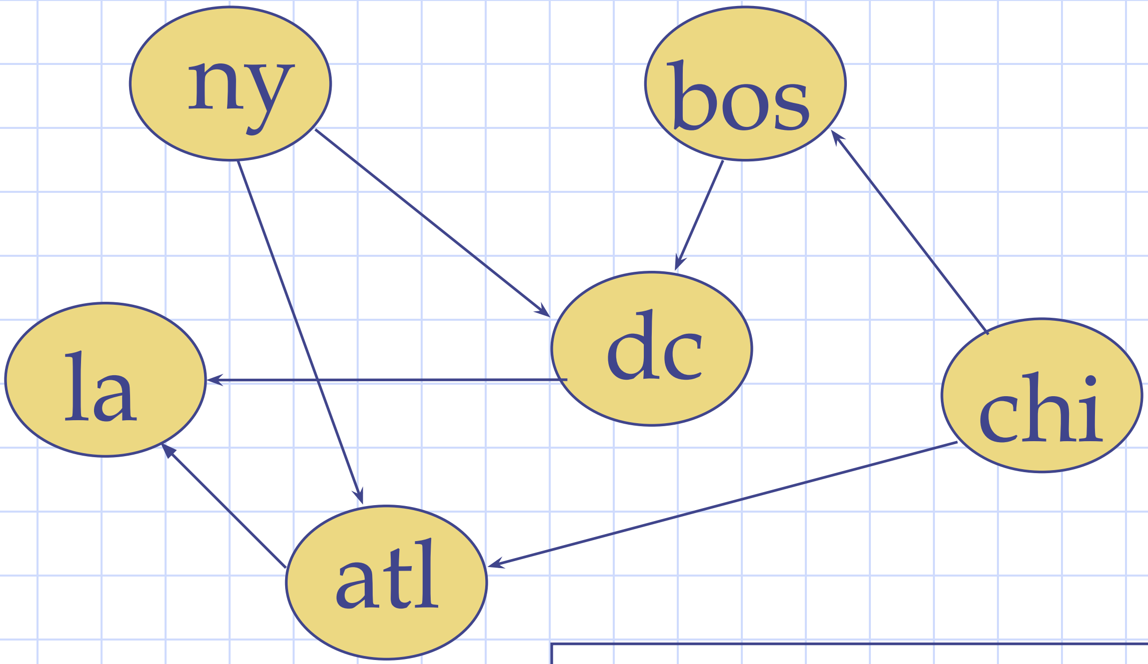
- networks (roads, flights, communications)



Applications

- computer networks
- airline flights
- road map
- course prerequisite structure
- tasks for completing a job
- flow of control through a program
- many more

Traversing a graph



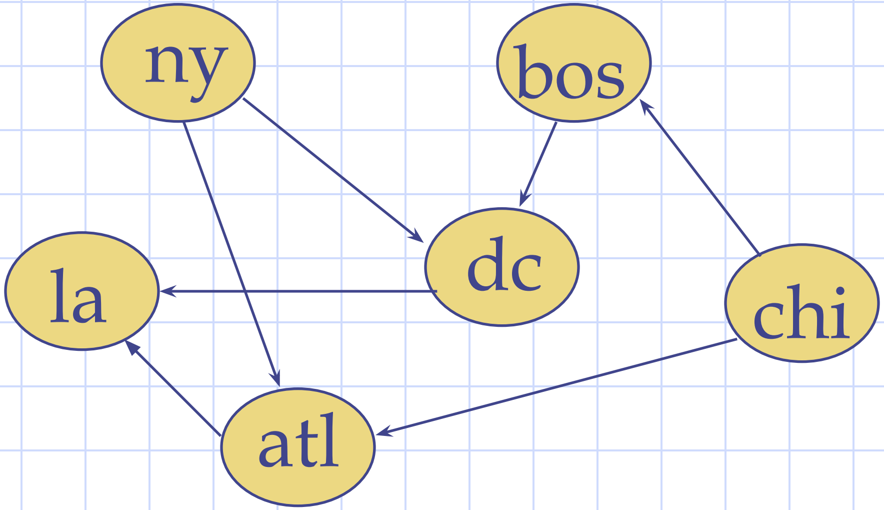
Where to start?
Will all vertices be visited?
How to prevent multiple visits?

Graph Traversal

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

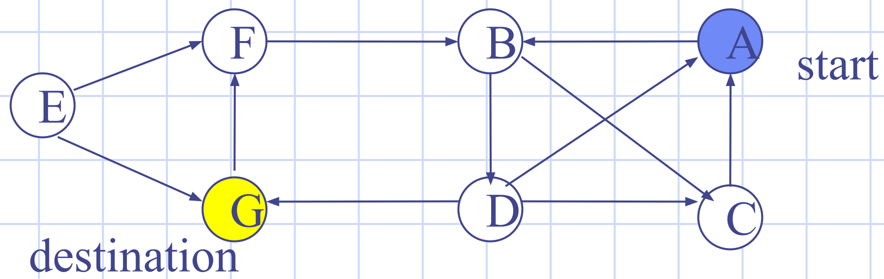
Breadth first traversal

```
breadthFirstTraversal (v)
    put v in Q
    while Q not empty
        remove w from Q
        visit w
        mark w as visited
        for each neighbor (u) of w
            if not yet visited
                put u in Q
```



BFS

BFS Process



rear front
A
Initial call to BFS on A
Add A to queue

rear front
B
Dequeue A
Add B

rear front
D C
Dequeue B
Add C, D

rear front
D
Dequeue C
Nothing to add

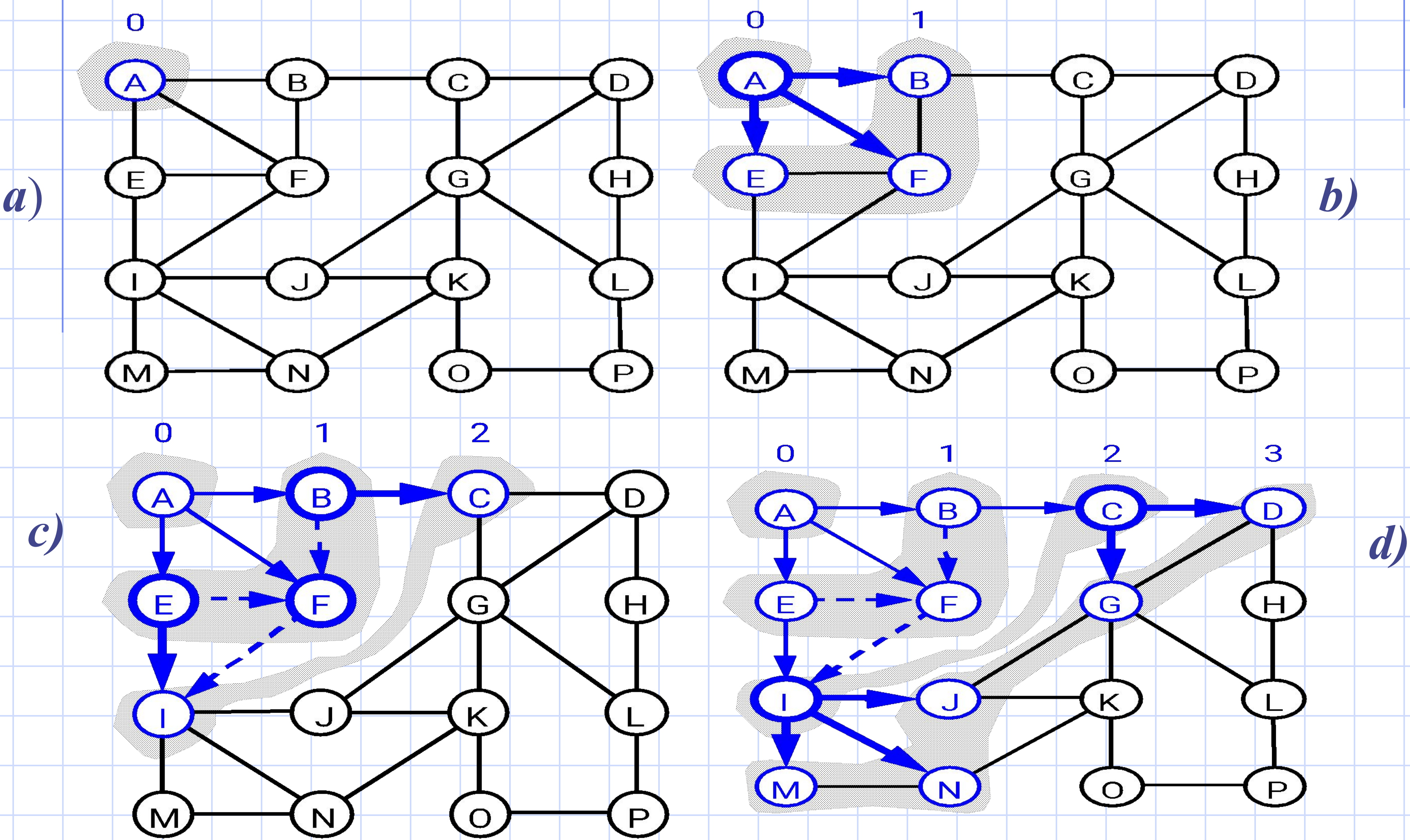
rear front
G
Dequeue D
Add G

found destination - done!
Path must be stored separately

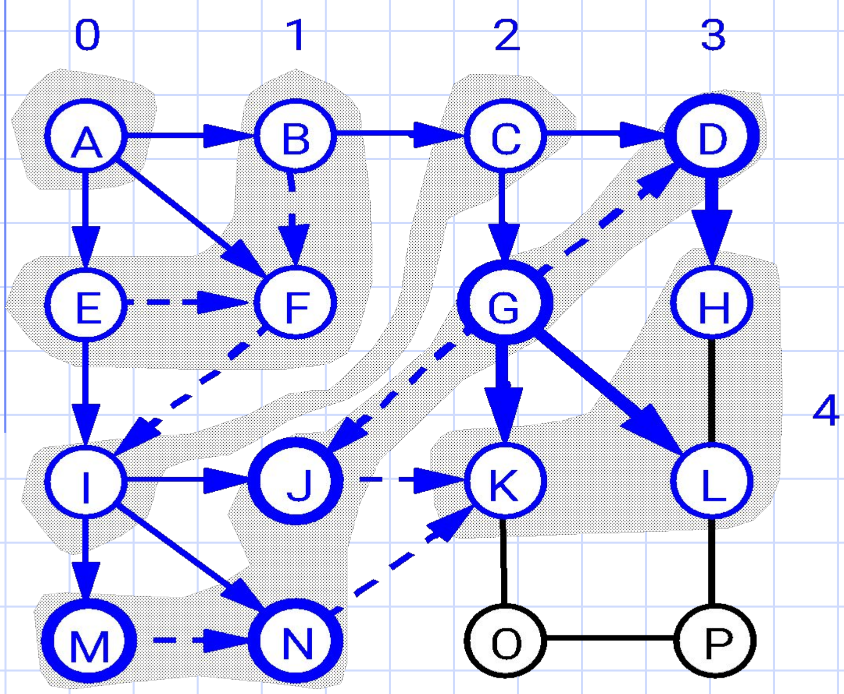
Breadth-First Search

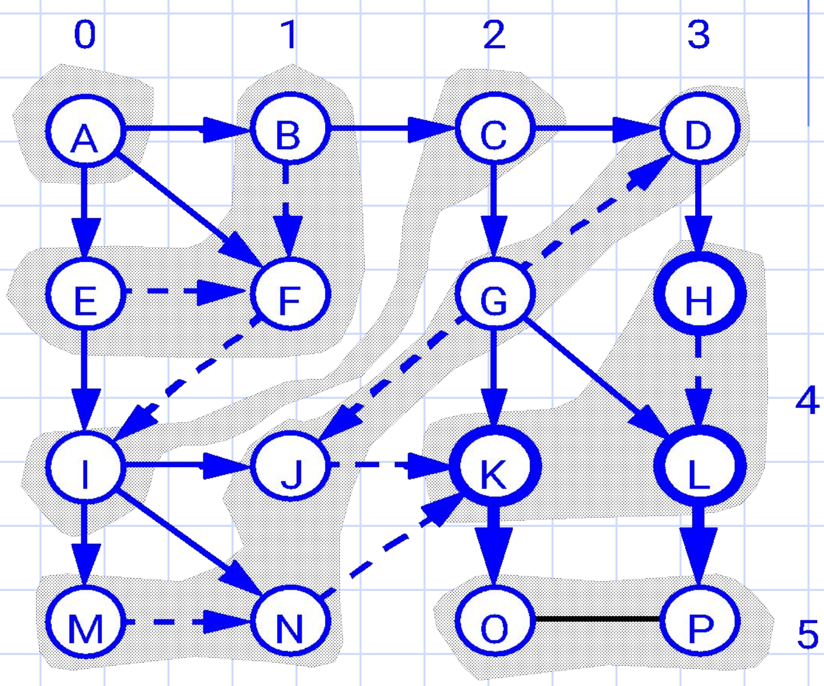
- A Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree with several useful properties.
- The starting vertex *s* has level 0, and, as in DFS, defines that point as an "anchor."
- In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited.
- These edges are placed into level 1
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex ***v*** corresponds to the length of the shortest path from ***s*** to ***v***.

BFS - A Graphical Representation



More BFS





BFS Pseudo-Code

Algorithm BFS(s): Input: A vertex s in a graph
Output: A labeling of the edges as "discovery" edges and "cross edges"

```
initialize container L_0 to contain vertex s
i ← 0
while L_i is not empty do
    create container L_{i+1} to initially be empty
    for each vertex v in L_i do
        if edge e incident on v do
            let w be the other endpoint of e
            if vertex w is unexplored then
                label e as a discovery edge
                insert w into L_{i+1}
            else label e as a cross edge
    i ← i + 1
```

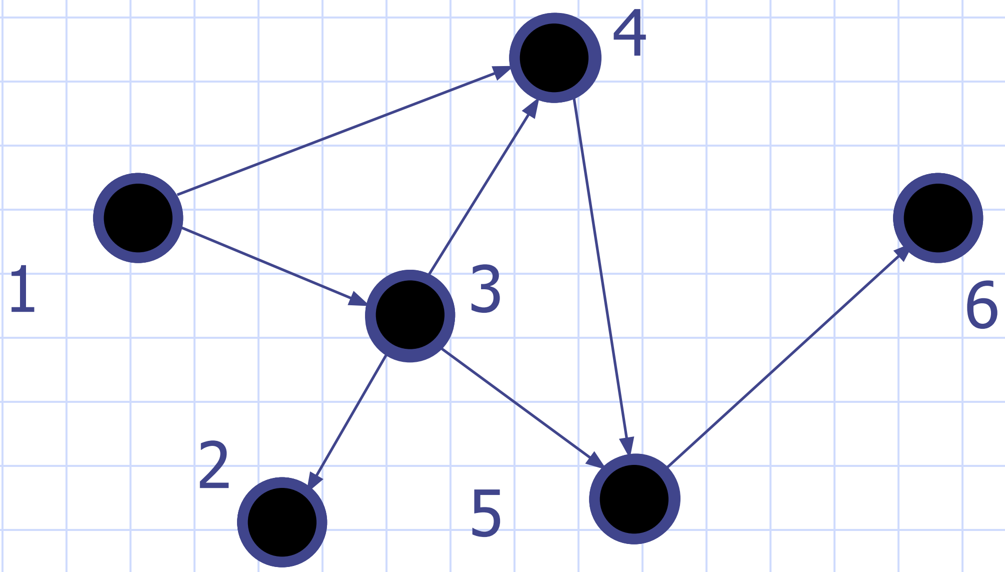
Breadth-First Search (BFS)

- Instead of going as far as possible, BFS tries to search all paths.
- BFS makes use of a queue to store visited (but not dead) vertices, expanding the path from the earliest visited vertices.

Simulation of BFS

- Queue:

1	4	3	5	2	6



Implementation

```
while queue Q not empty
    dequeue the first vertex u from Q
    for each vertex v directly reachable from u
     if v is unvisited
          enqueue v to Q
          mark v as visited
```

- Initially all vertices except the start vertex are marked as *unvisited* and the queue contains the start vertex only

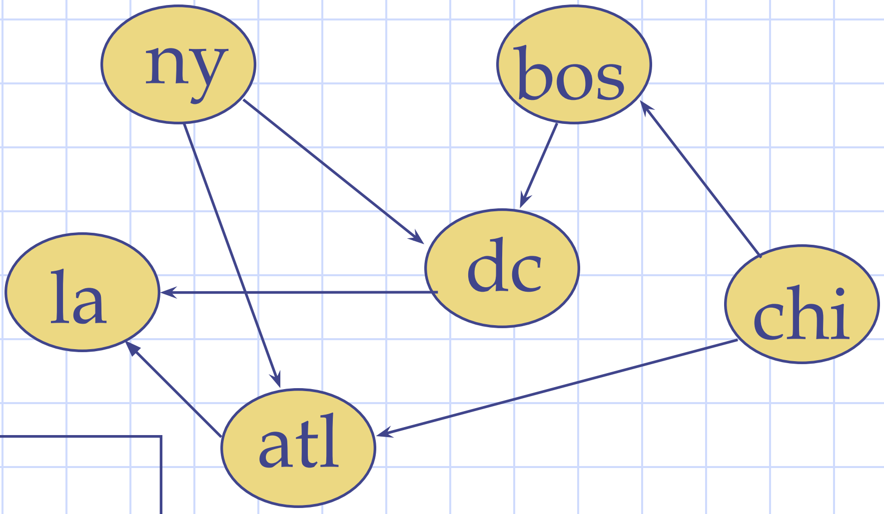
Algorithm BFS

```
BFS(g)
Step 1:  h=g;
Step 2:  visited[g]=1;
Step 3:  repeat
        {
Step 4:  for all vertices w adjacent from h do
         {if (visited [w]=0) then
          {
             add w to q;    //w is unexplored
             visited[w]=1;
          }   //end of the if
         }     // end of for loop
Step 5:  if q is empty then return;
          Delete h from q;
     } until (false);
     // end of the repeat
Step 6:  end   BREADTH FIRST SEARCH
```

Advantages

- Guarantee shortest paths for unweighted graphs
- Use queue instead of recursive functions – Avoiding stack overflow

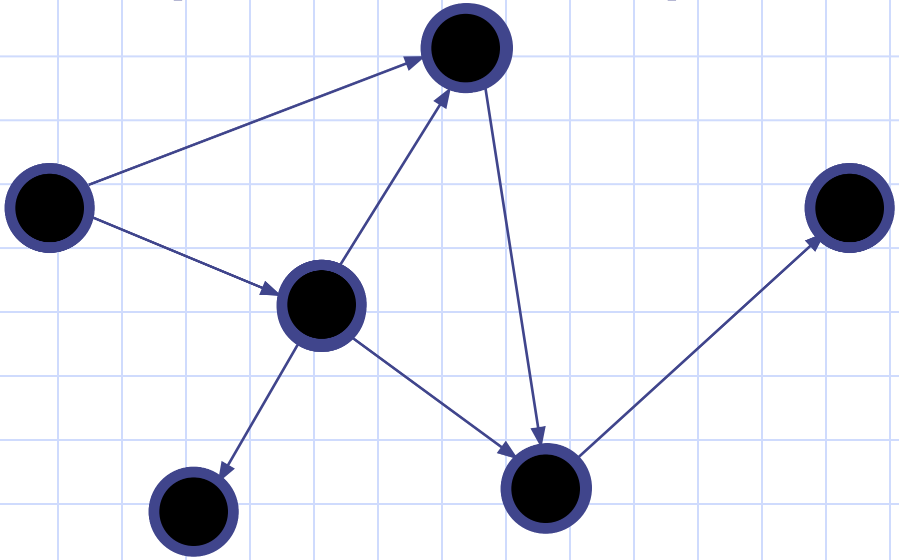
Depth first traversal




```
depthFirstTraversal (v)
  visit v
  mark v as visited
  for each neighbor (u) of v
      if not yet visited
          depthFirstTraversal (u)
```

Depth-First Search (DFS)

- Strategy: Go as far as you can (if you have not visit there), otherwise, go back and try another way



Implementation

```
DFS (vertex u) {
    mark u as visited
    for each vertex v directly reachable from u
     if v is unvisited
          DFS (v)
}
```

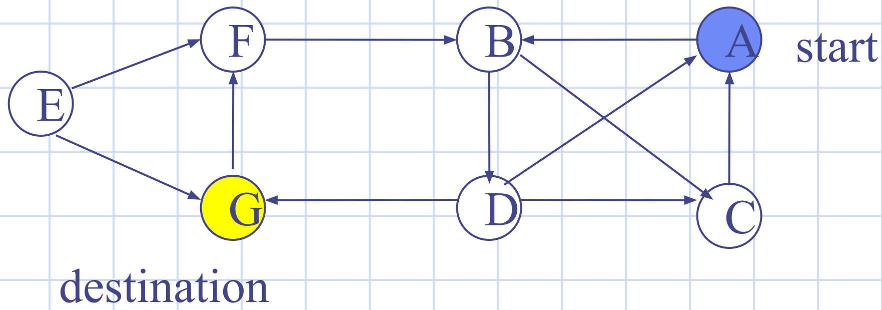
- Initially all vertices are marked as *unvisited*

Algorithm DFS

```
DFS(g)
Step 1:  h=g;
Step 2:  visited[g]=1;
Step 3:  repeat
        {
Step 4:  for all vertices w adjacent from h do
          {if (visited [w]=0) then
            {
               push w to s;    //w is unexplored
               visited[w]=1;
            }   //end of the if
           }     // end of for loop
Step 5:  if s is empty then return;
          Pop h from s;
     } until (false);
     // end of the repeat
Step 6:  end   DEPTH FIRST SEARCH
```

DFS

DFS Process



A — DFS on A

B
A — DFS on B

C — DFS on C
B
A

D — Call DFS on D
B — Return to call on B
A

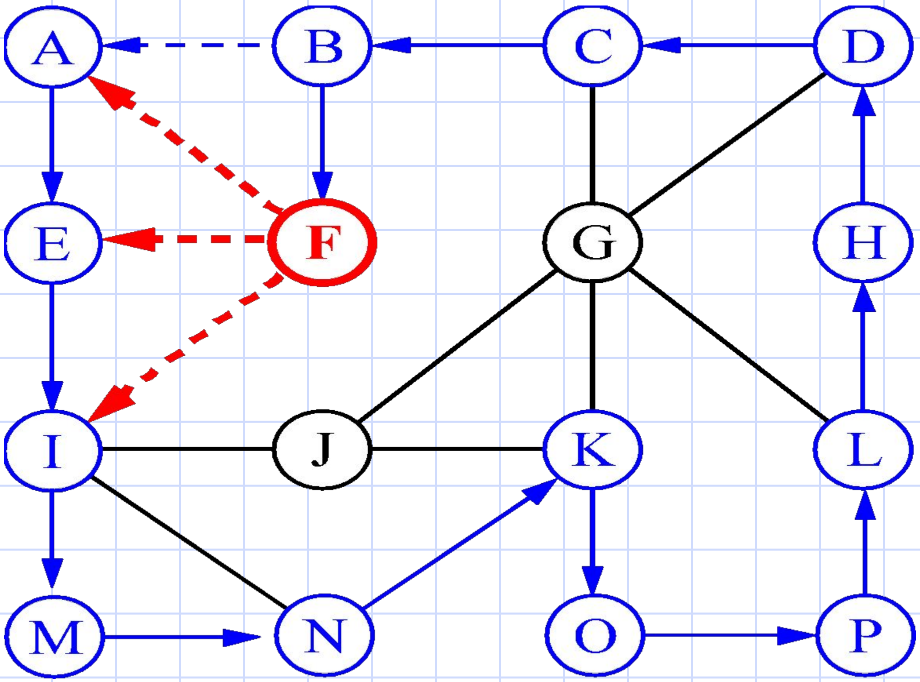
G — Call DFS on G
D
B
A

found destination - done!

Path is implicitly stored in DFS recursion

Path is: A, B, D, G

Depth-First Search



Depth-First Search

Algorithm DFS(v); **Input**: A vertex v in a graph

Output: A labeling of the edges as "discovery" edges and "backedges"

for each edge *e* incident on *v* **do**

if edge *e* is unexplored **then** let *w* be the other endpoint of *e*

if vertex w is unexplored **then** label *e* as a discovery edge

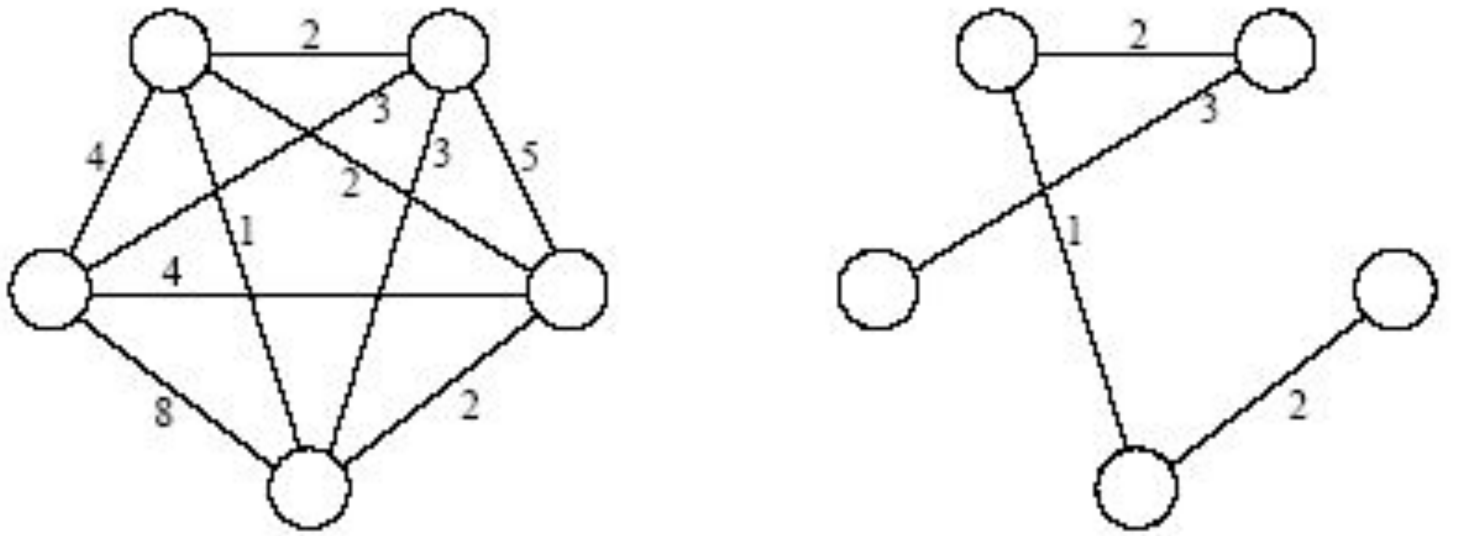
recursively call **DFS(*w*)**

else label *e* as a backedge

Minimum Spanning Tree

- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*.
- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.
- A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.
- there may be a number of minimal spanning trees for a particular undirected graph with the same total weight.

Minimum Spanning Tree



An undirected graph and its minimum spanning tree.

Algorithms for Determining the Minimal Spanning Tree

There are two algorithms presented in our textbook for determining the minimal spanning tree of an undirected graph that is connected and weighted.

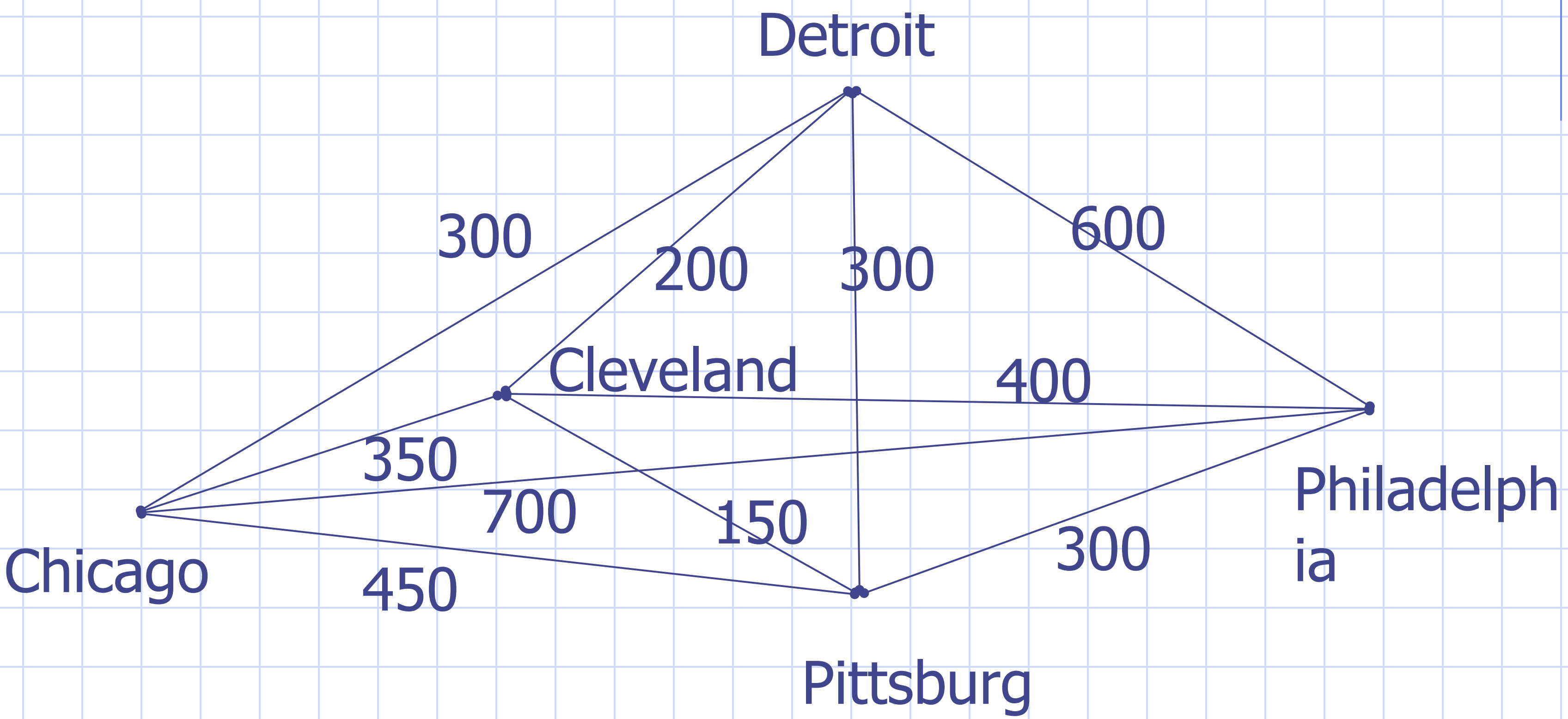
- Prim’s Algorithm: process of stepping from vertex to vertex
- Kruskal’s Algoritm: searching through edges for minimum weights

Minimum Spanning Tree: Prim's Algorithm

- Prim's algorithm for finding an MST is a greedy algorithm.
- The goal is to one at a time include a new vertex by adding a new edge without creating a cycle

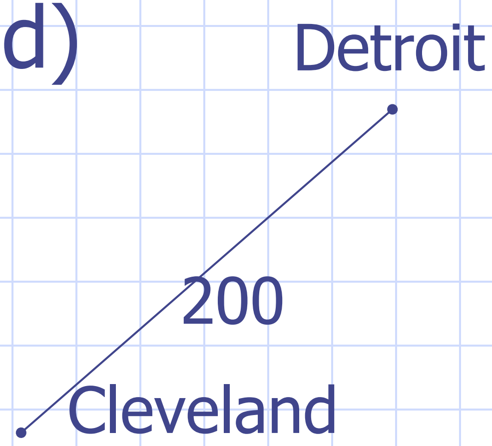
Prim’s Algorithm

- Start by selecting an arbitrary vertex, include it into the current MST.
- From it, pick the edge with the lowest weight.
- Grow the current MST by inserting into it the vertex closest to one of the vertices already in current MST.
- As you add vertices, you will add possible edges to follo`w to new vertices.
- Pick the edge with the lowest weight to go to a new vertex without creating a cycle.

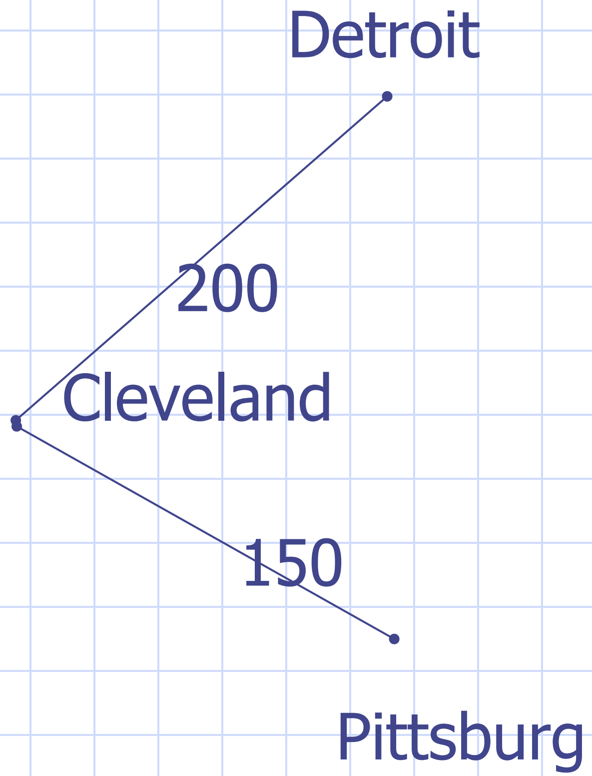


Detroit
300
200
300
600
Cleveland
400
350
700
150
300
Philadelph
ia
Chicago
450
Pittsburg

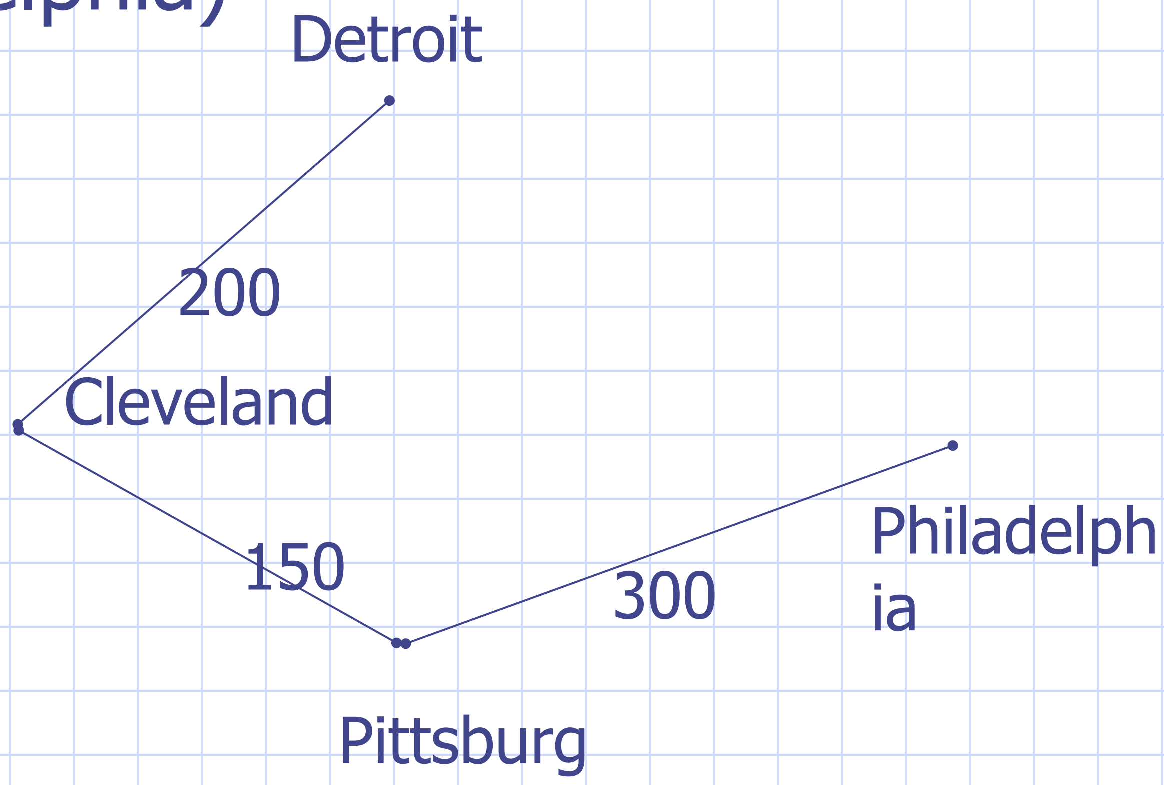
- Pick any starting point: Detroit.
- Pick edge with lowest weight: 200 (Cleveland)



Pick any edge connected to Cleveland or Detroit that doesn't create a cycle: 150 (Pittsburg)



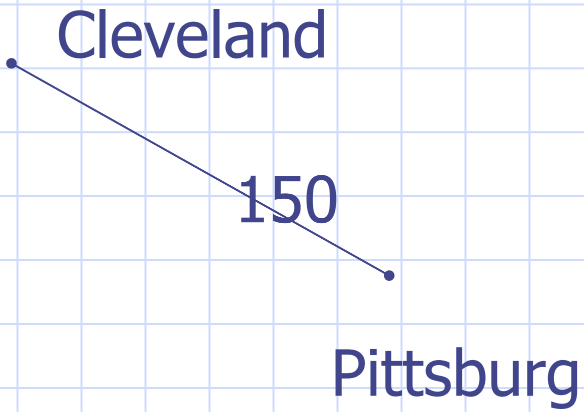
Pick any edge connected to Cleveland, Detroit, or Pittsburg that doesn’t create a cycle: 300 (Philadelphia)



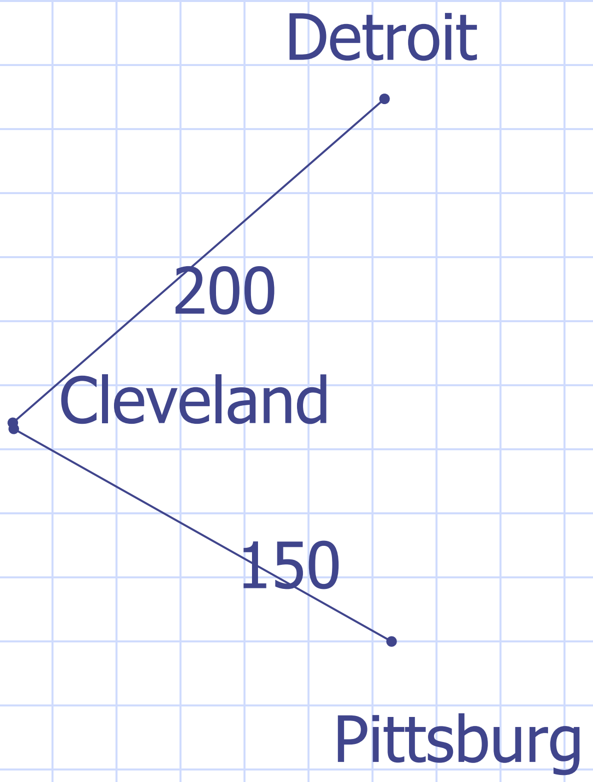
Kruskal's Algorithm in English

- The goal is to one at a time include a new edge without creating a cycle.
- Start by picking the edge with the lowest weight.
- Continue to pick new edges without creating a cycle. Edges do not necessarily have to be connected.
- Stop when you have n-1 edges.

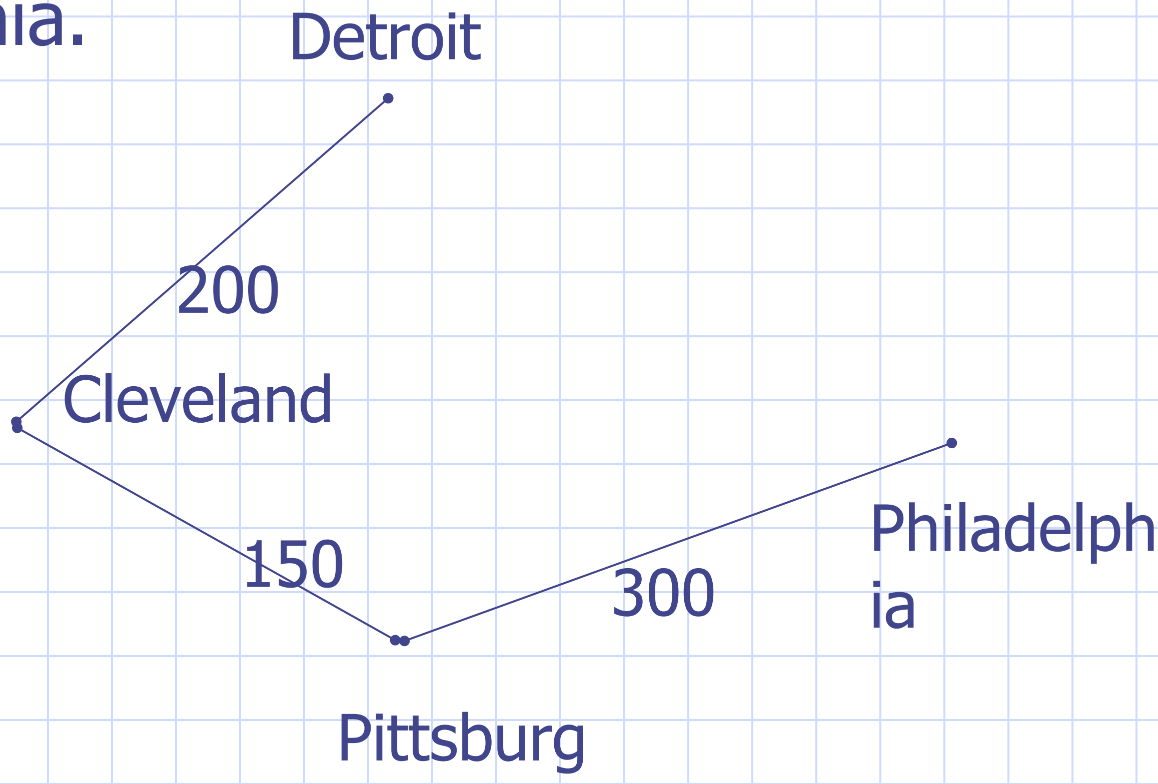
First edge picked is lowest value of 150 (Cleveland to Pittsburg)



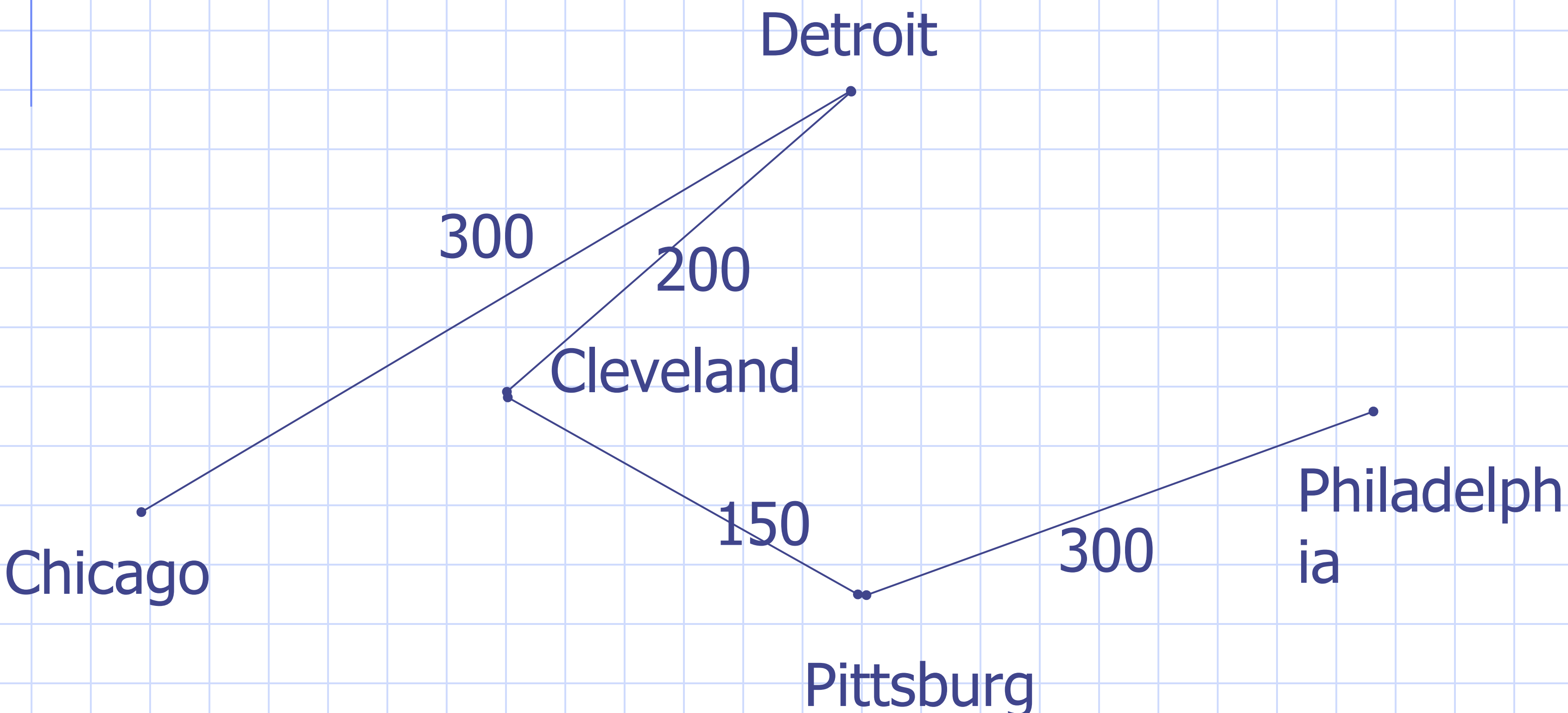
Next lowest edge is 200 (Cleveland to Detroit)



There are a few edges of 300, but Detroit to Pittsburg cannot be selected because it would create a cycle. We go with Pittsburg to Philadelphia.

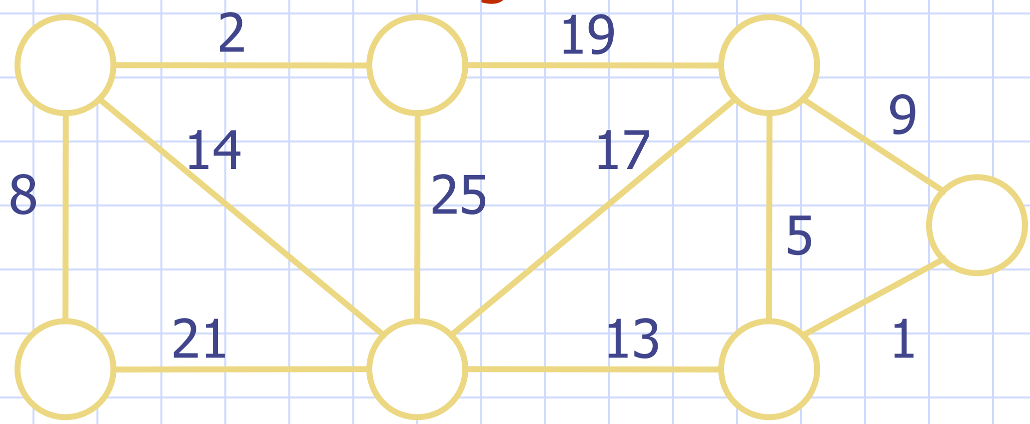


The next lowest edge that doesn't create a cycle is 300 edge from Detroit to Chicago. This is the 4th edge!



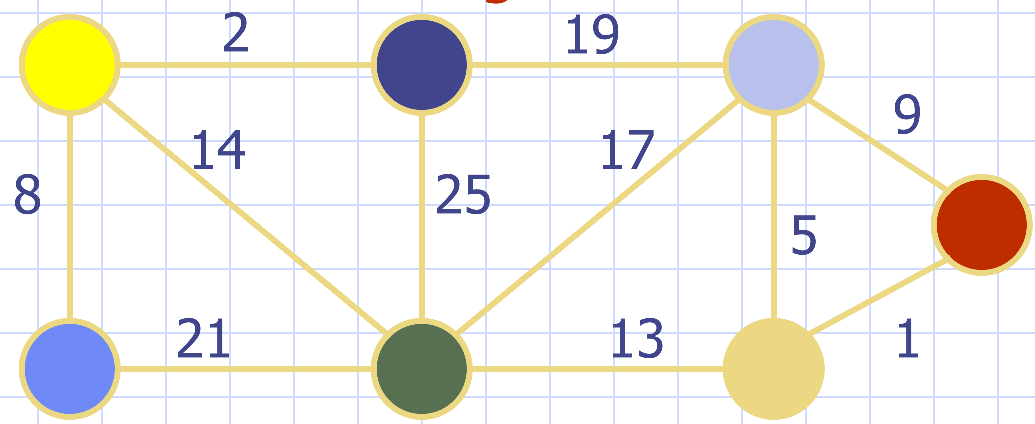
Kruskal’s Algorithm

Run the algorithm:



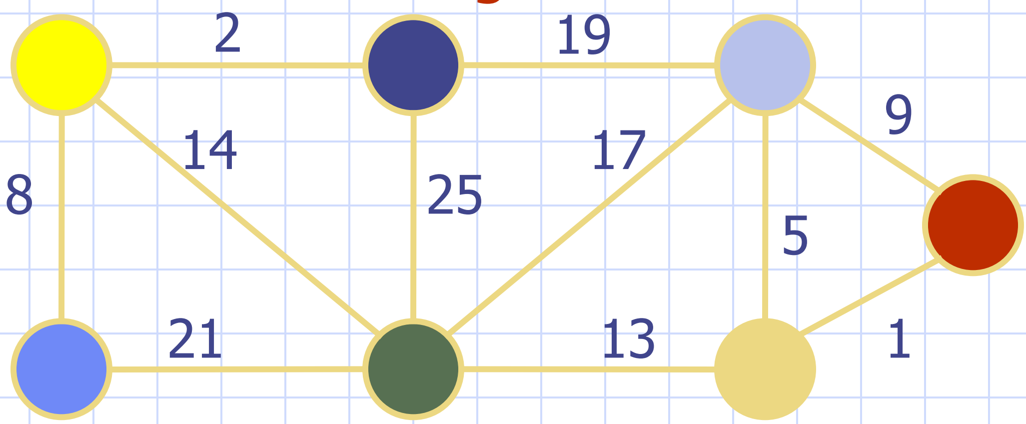
Kruskal's Algorithm

Run the algorithm:



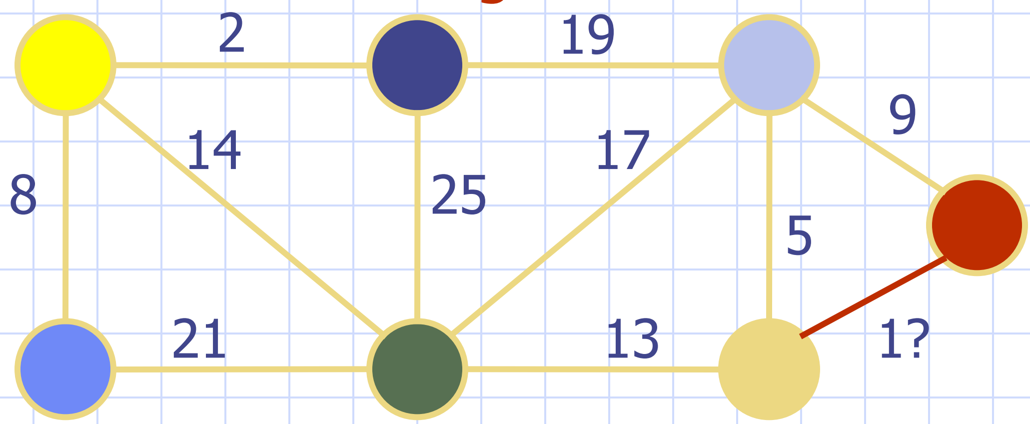
Kruskal’s Algorithm

Run the algorithm:



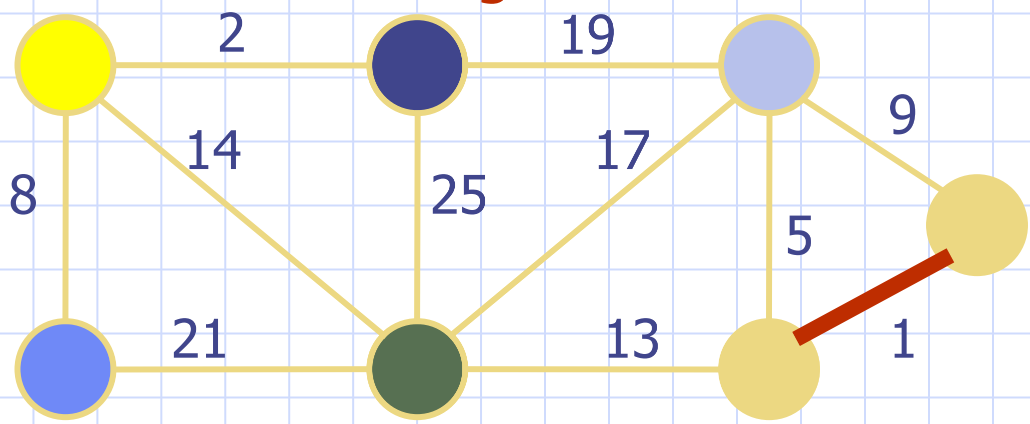
Kruskal’s Algorithm

Run the algorithm:



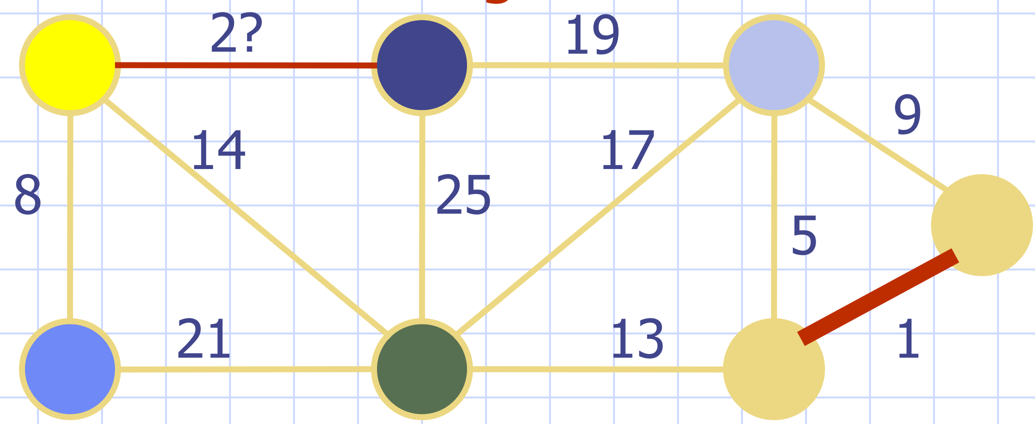
Kruskal’s Algorithm

Run the algorithm:



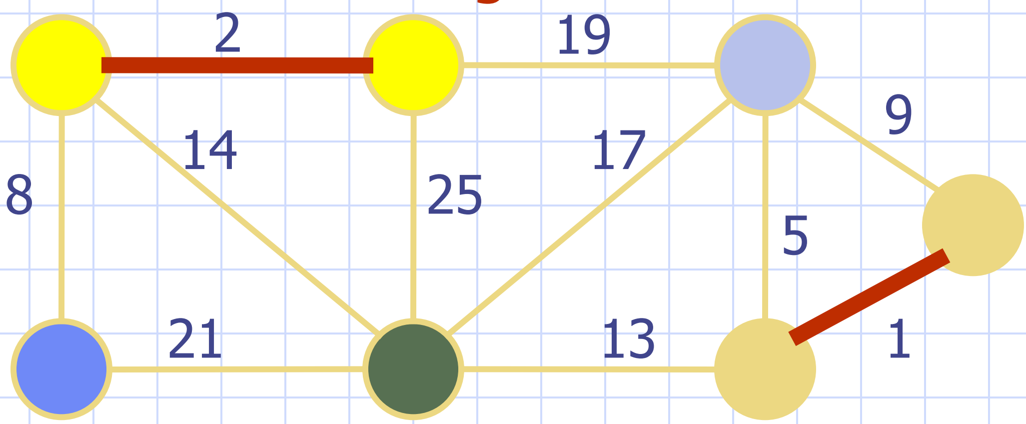
Kruskal's Algorithm

Run the algorithm:



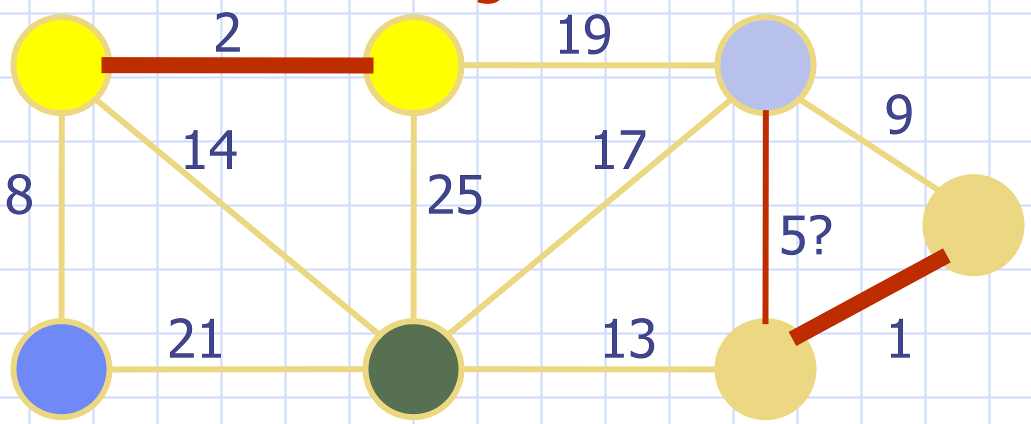
Kruskal's Algorithm

Run the algorithm:



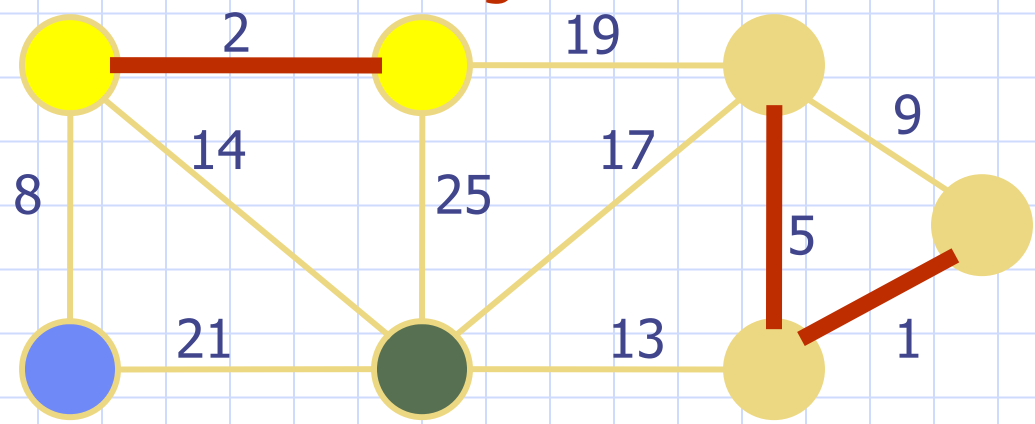
Kruskal's Algorithm

Run the algorithm:



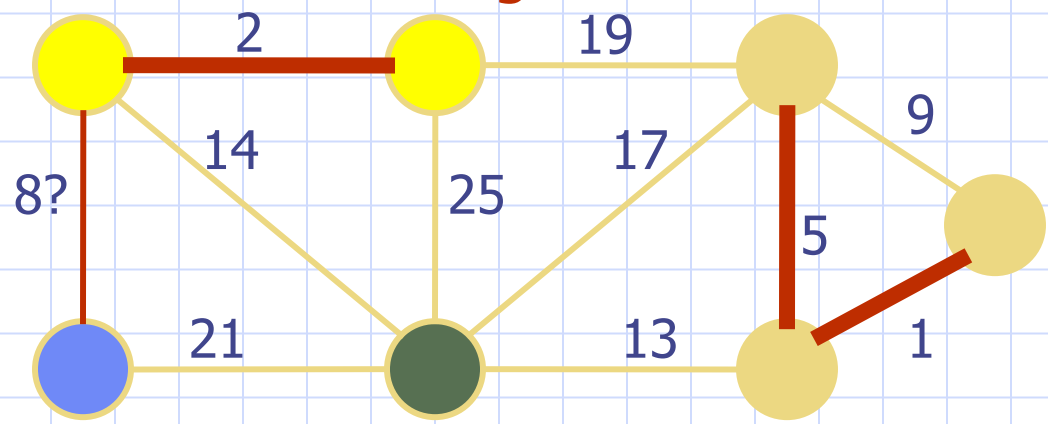
Kruskal’s Algorithm

Run the algorithm:



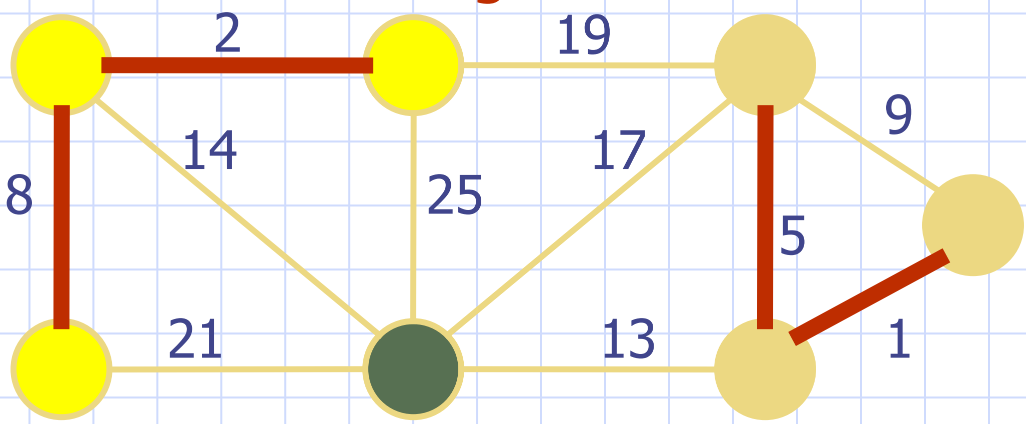
Kruskal's Algorithm

Run the algorithm:



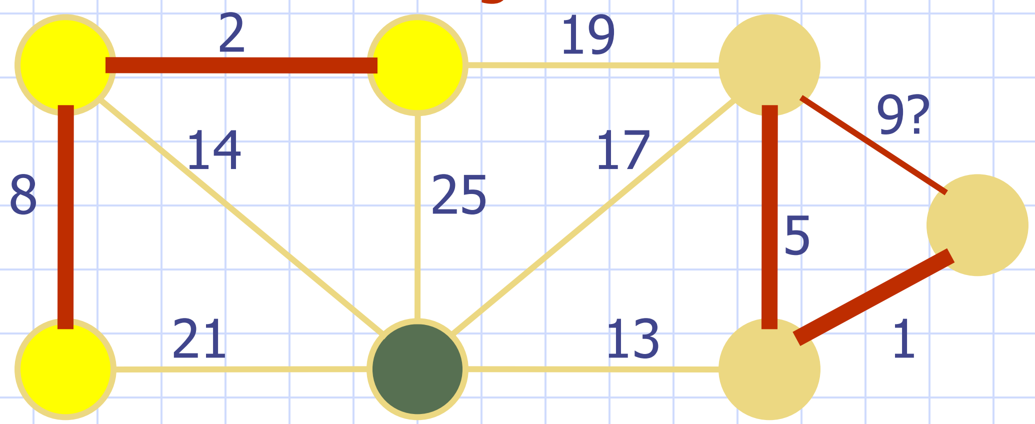
Kruskal’s Algorithm

Run the algorithm:



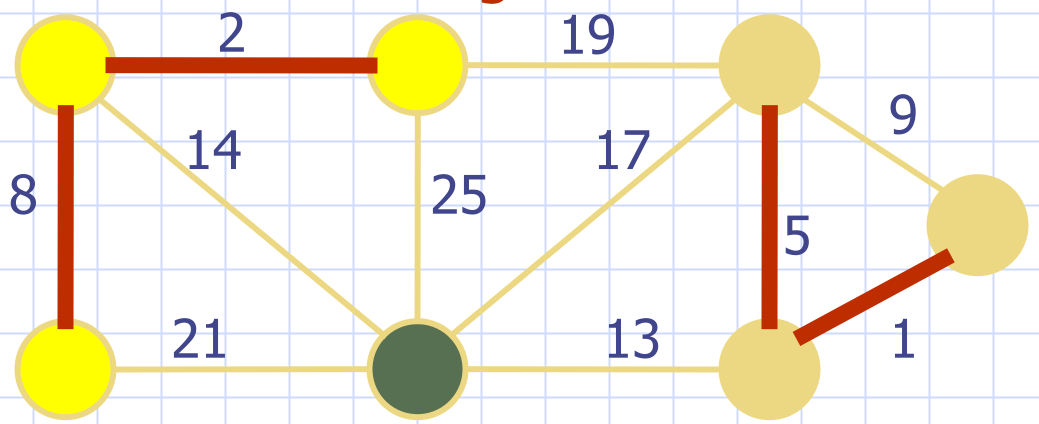
Kruskal's Algorithm

Run the algorithm:



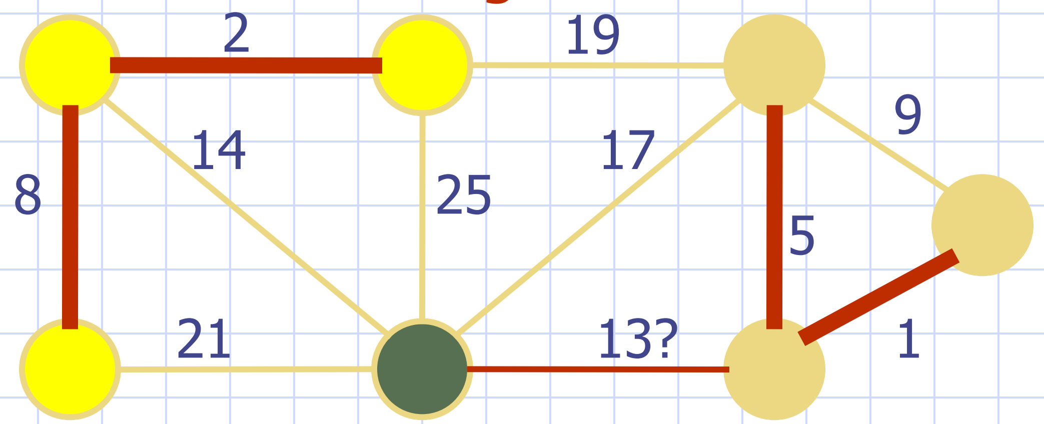
Kruskal's Algorithm

Run the algorithm:



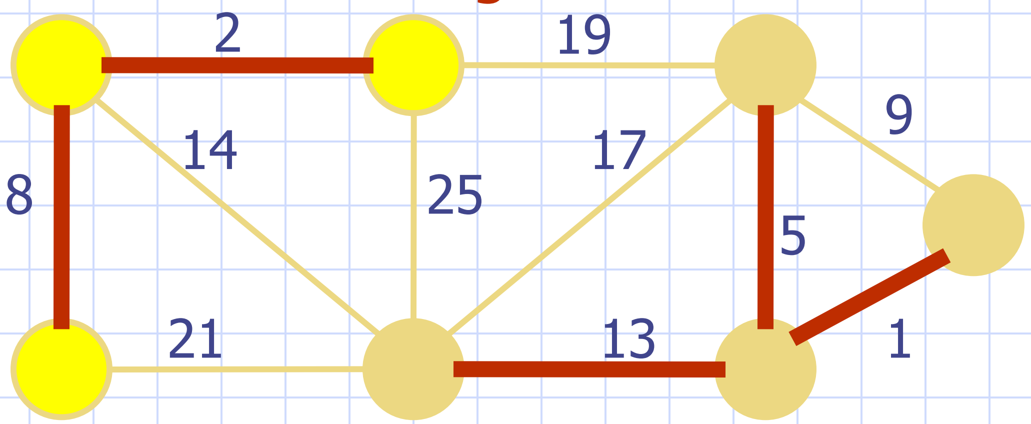
Kruskal’s Algorithm

Run the algorithm:



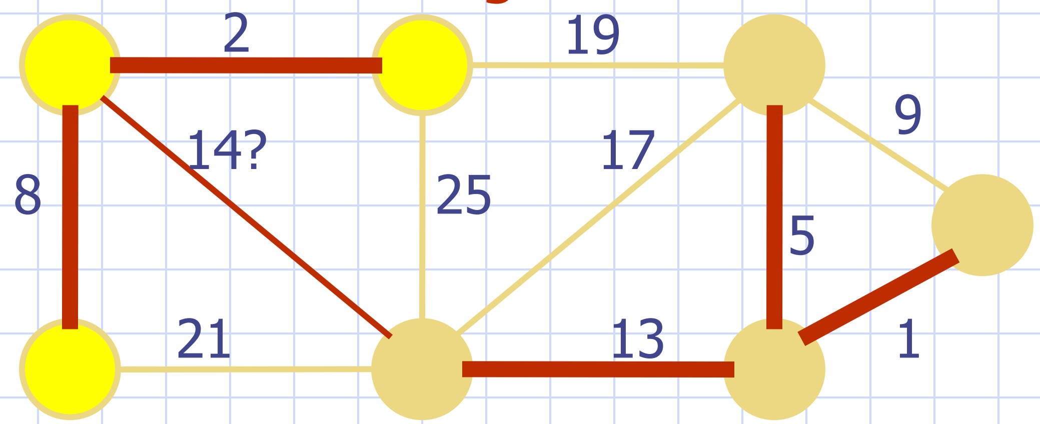
Kruskal’s Algorithm

Run the algorithm:



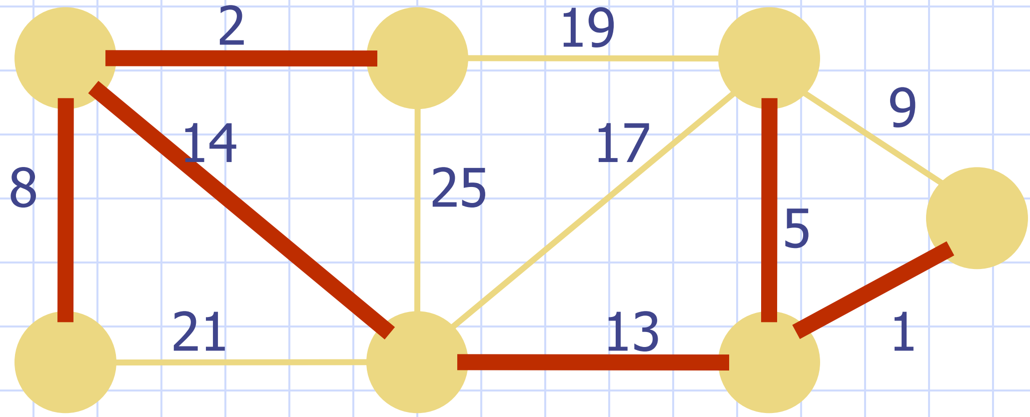
Kruskal's Algorithm

Run the algorithm:



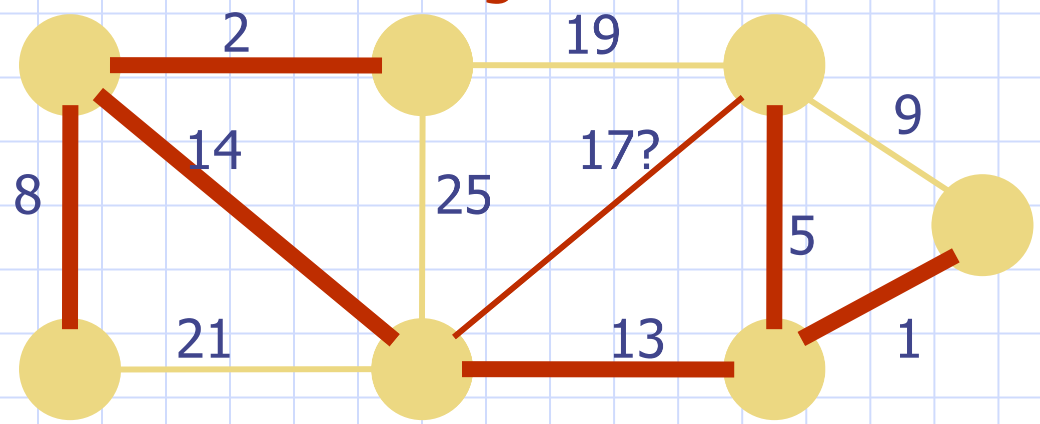
Kruskal’s Algorithm

Run the algorithm:



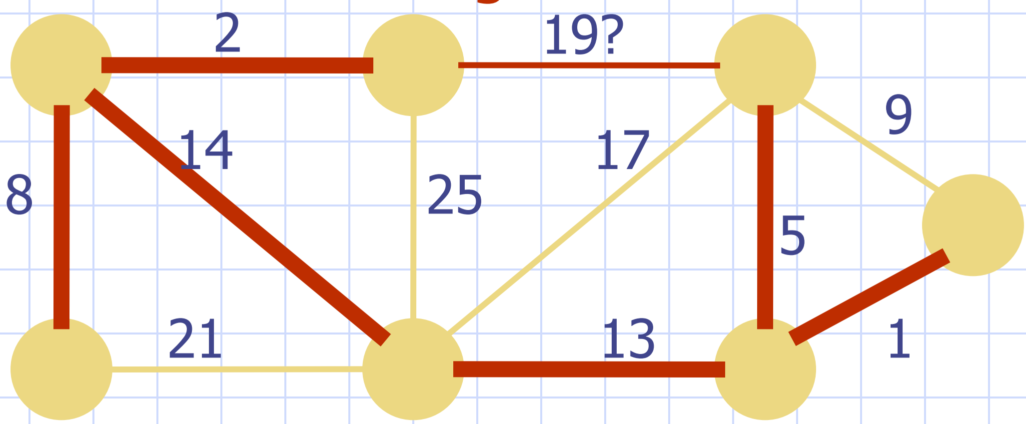
Kruskal’s Algorithm

Run the algorithm:



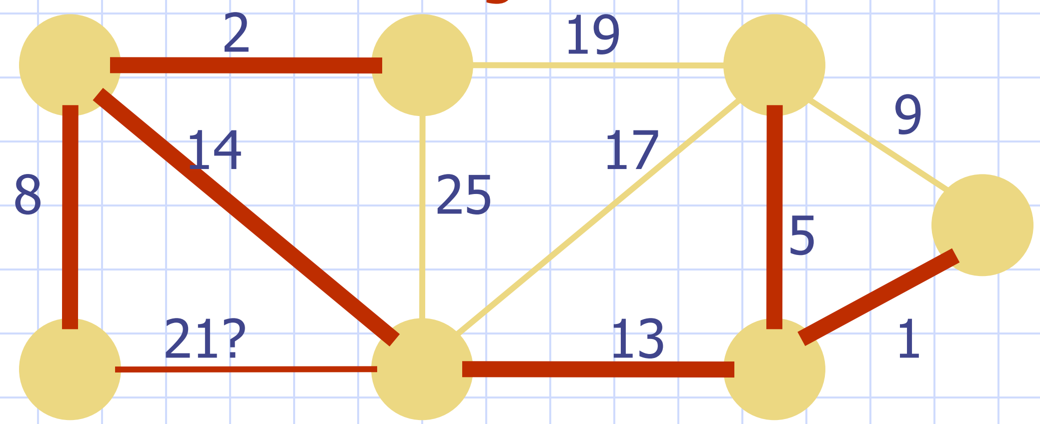
Kruskal's Algorithm

Run the algorithm:



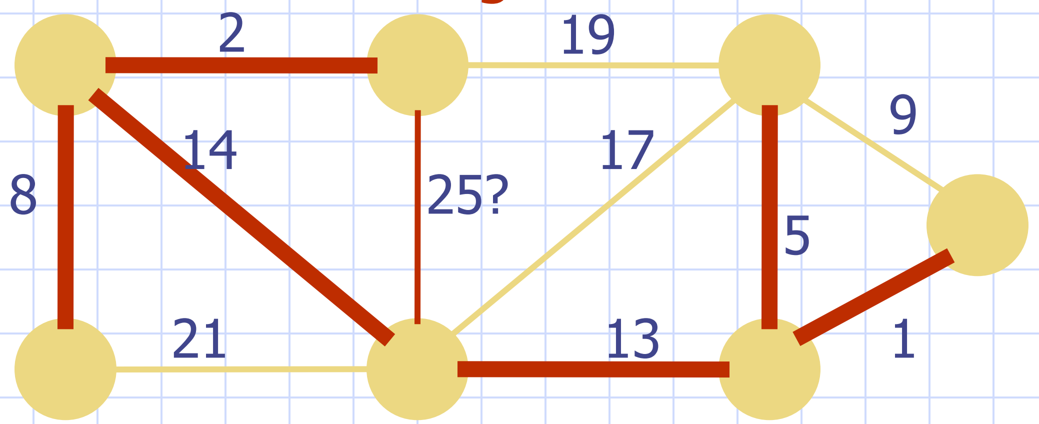
Kruskal’s Algorithm

Run the algorithm:



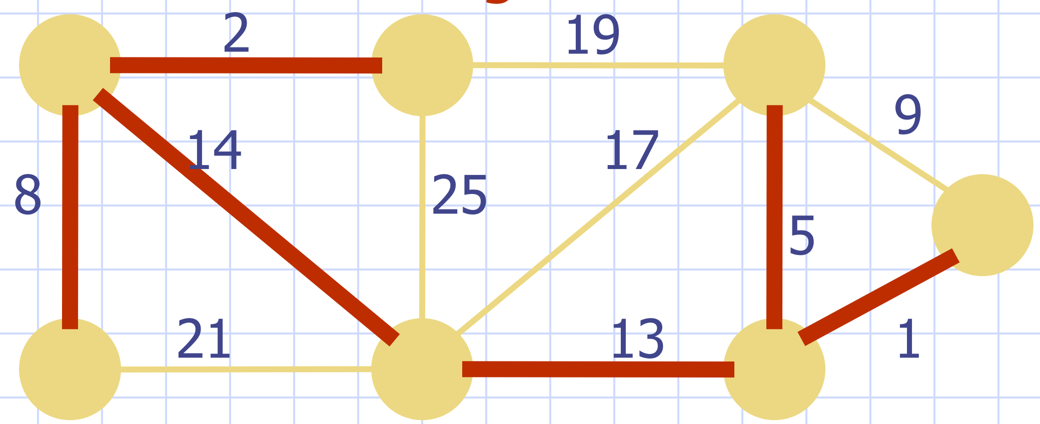
Kruskal's Algorithm

Run the algorithm:



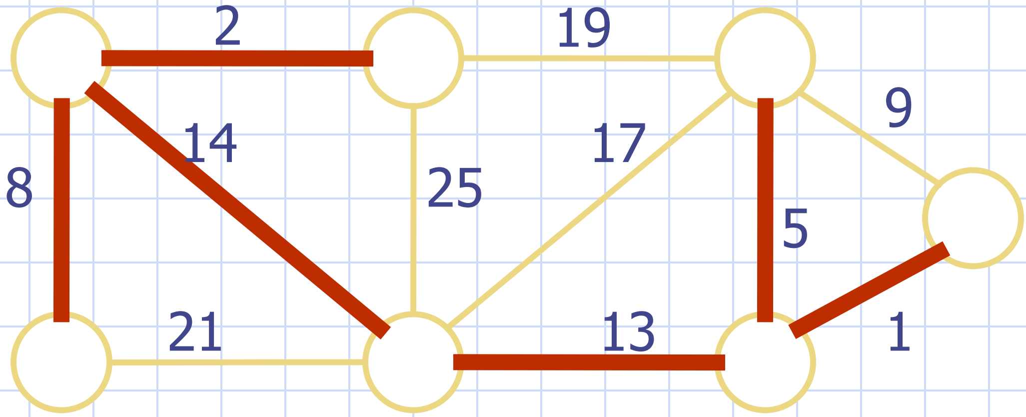
Kruskal’s Algorithm

Run the algorithm:



Kruskal's Algorithm

Run the algorithm:



Minimum Connector Algorithms

Kruskal's algorithm

Select the shortest edge in a network

Select the next shortest edge which does not create a cycle

Repeat step 2 until all vertices have been connected

Prim's algorithm

Select any vertex

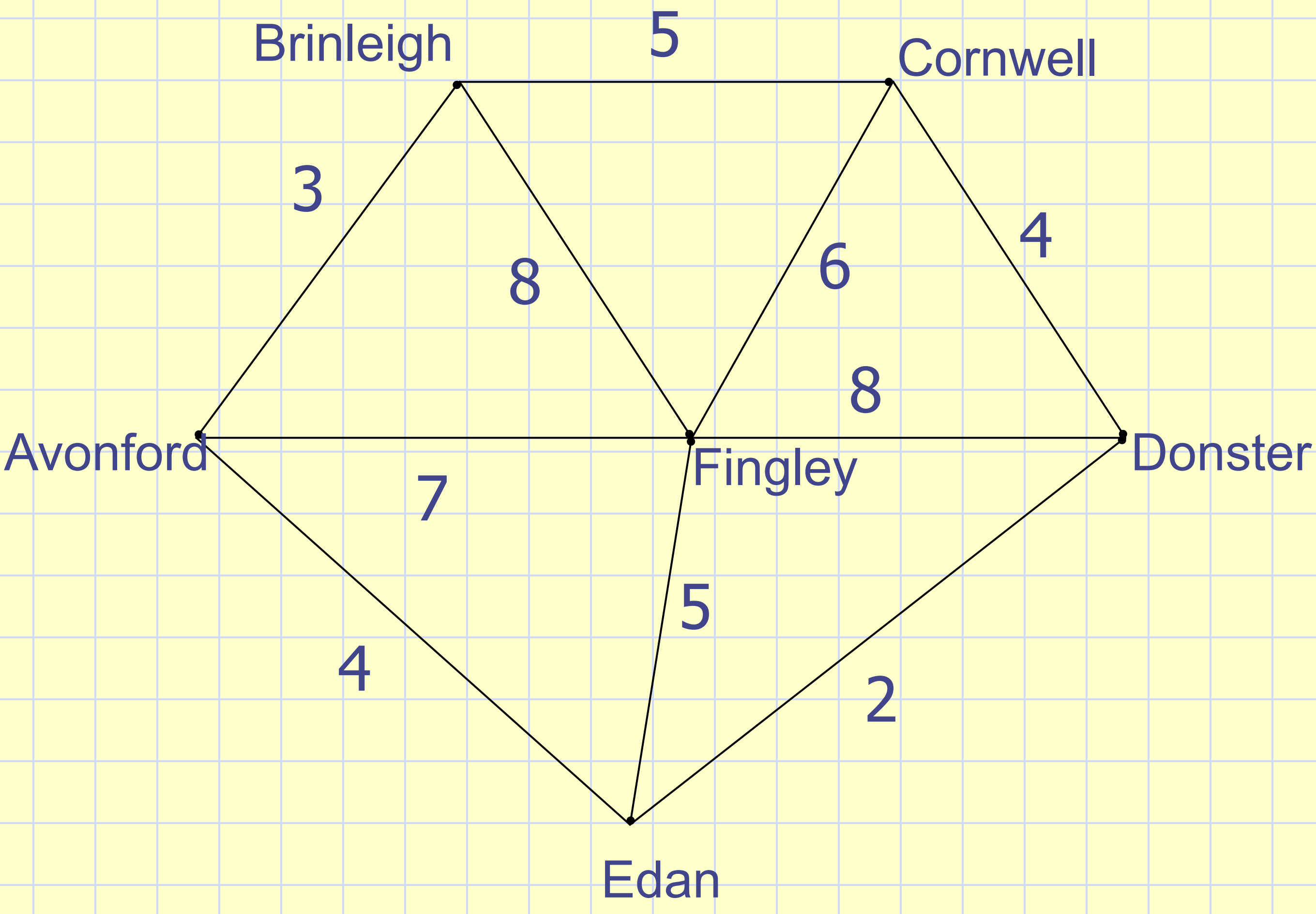
Select the shortest edge connected to that vertex

Select the shortest edge connected to any vertex already connected

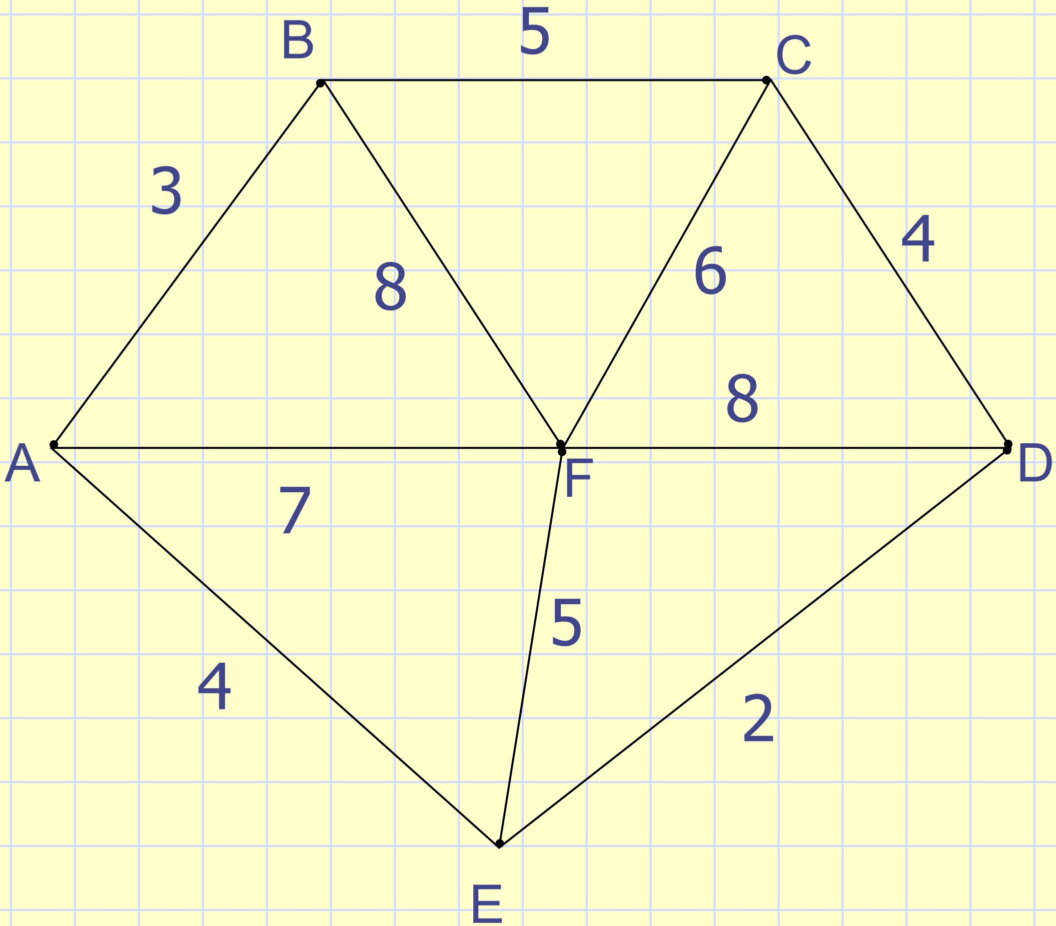
Repeat step 3 until all vertices have been connected

Example

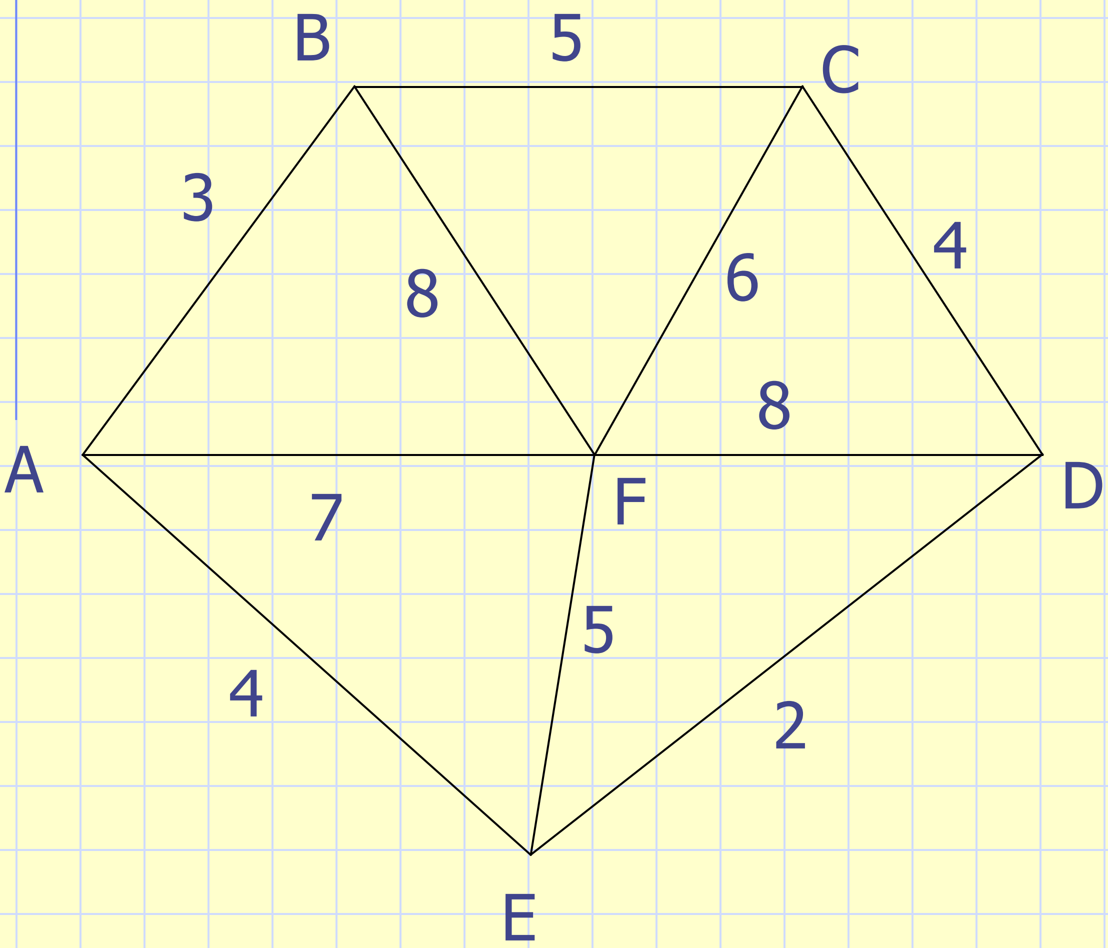
A cable company want to connect five villages to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



We model the situation as a network, then the problem is to find the minimum connector for the network



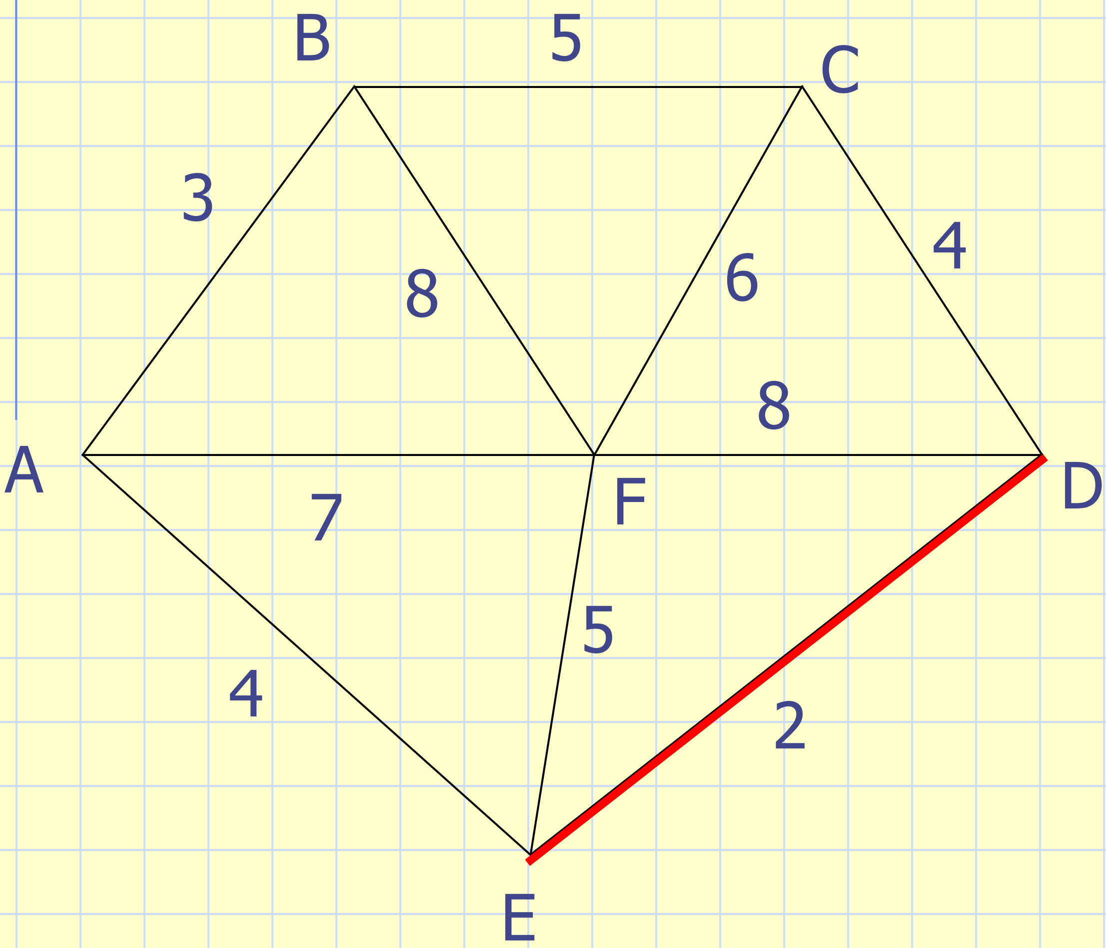
Kruskal's Algorithm



List the edges in order of size:

ED 2
AB 3
AE 4
CD 4
BC 5
EF 5
CF 6
AF 7
BF 8
CF 8

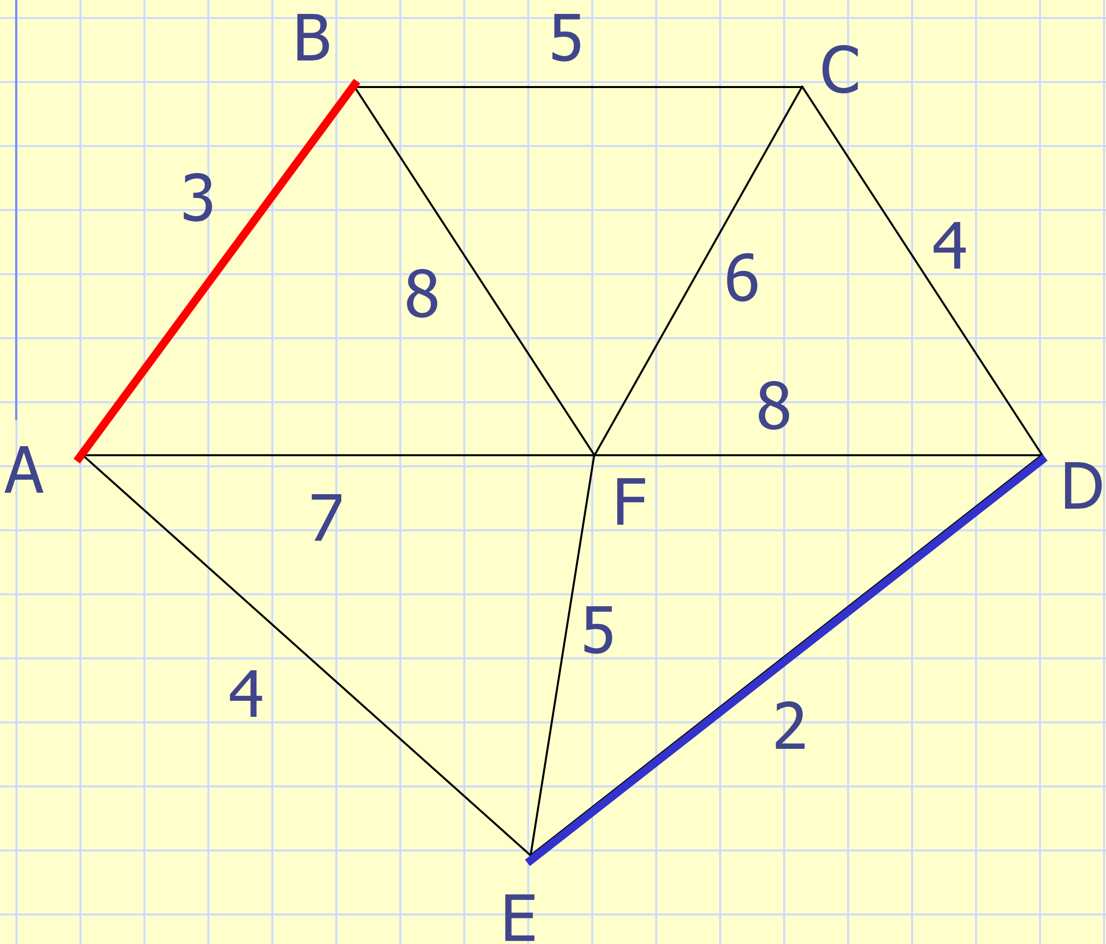
Kruskal's Algorithm



Select the shortest edge in the network

ED 2

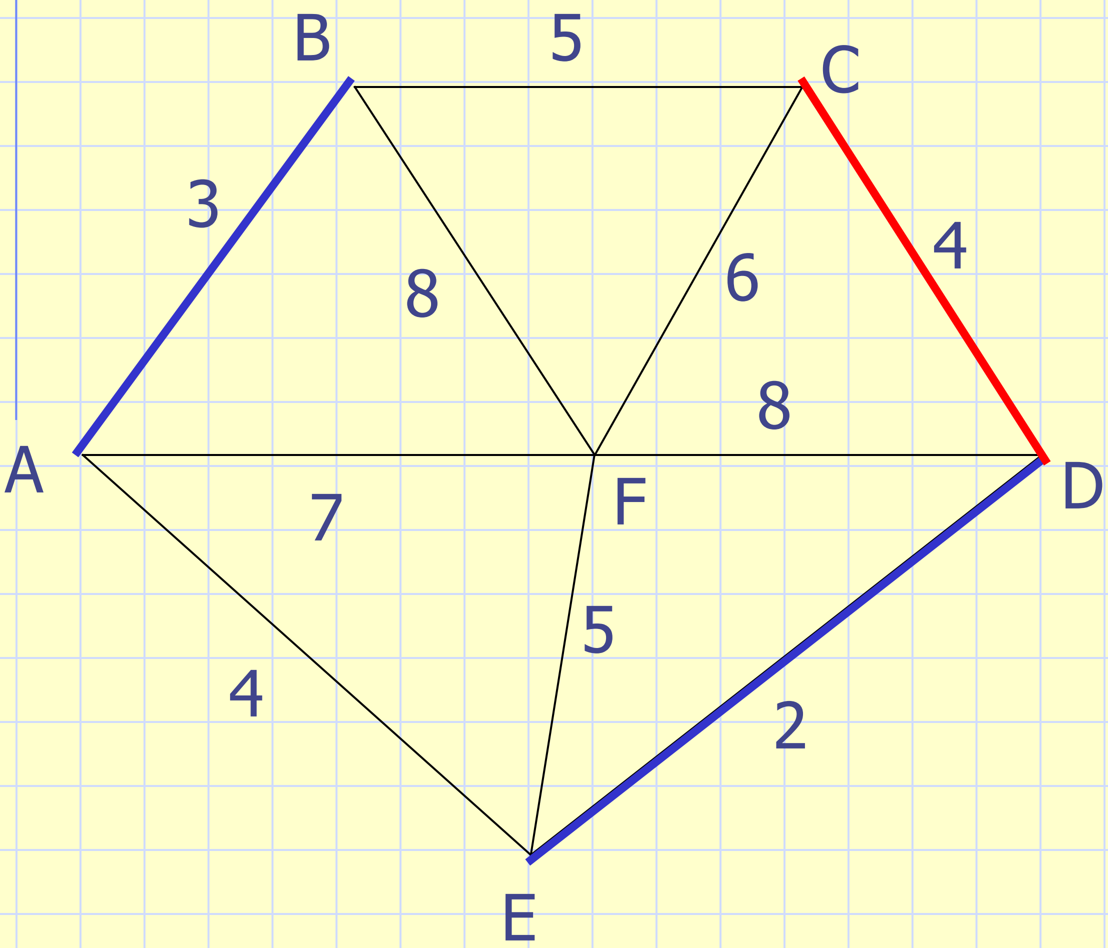
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3

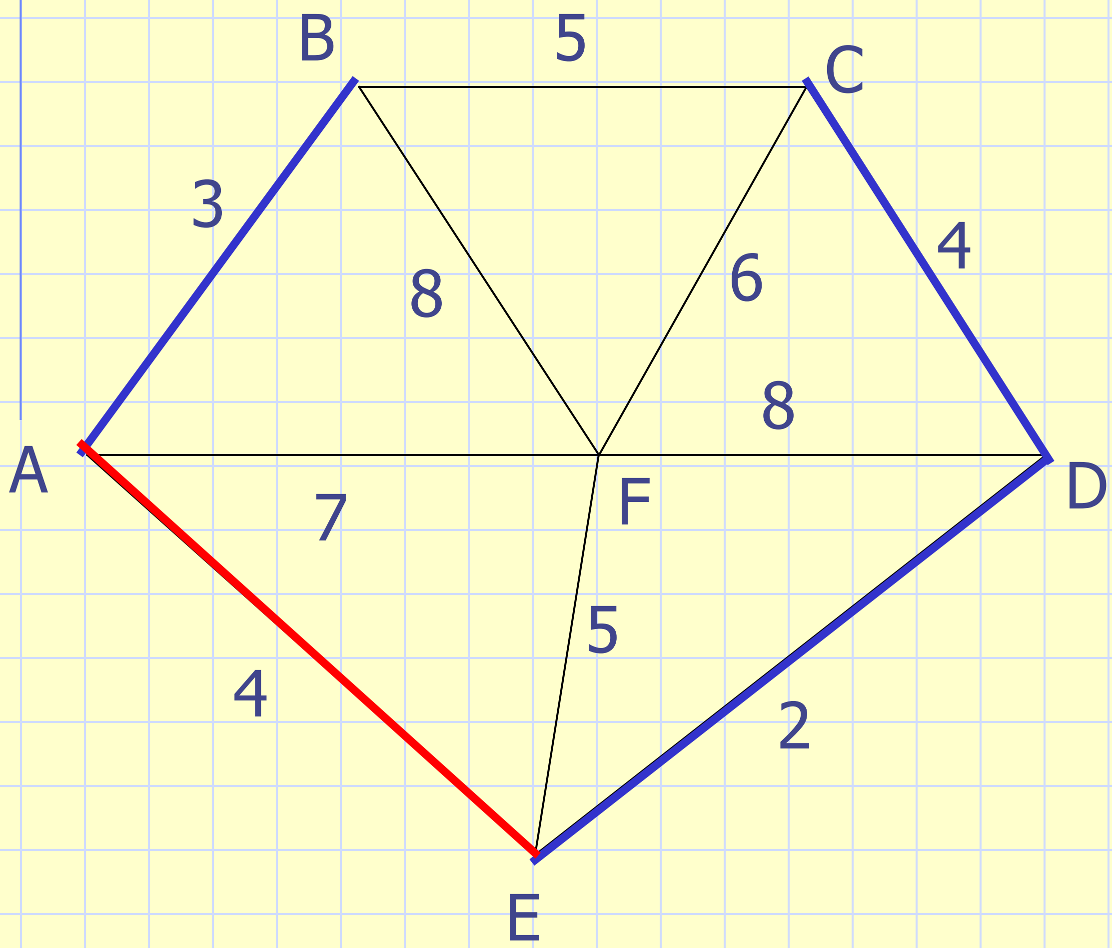
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4 (or AE 4)

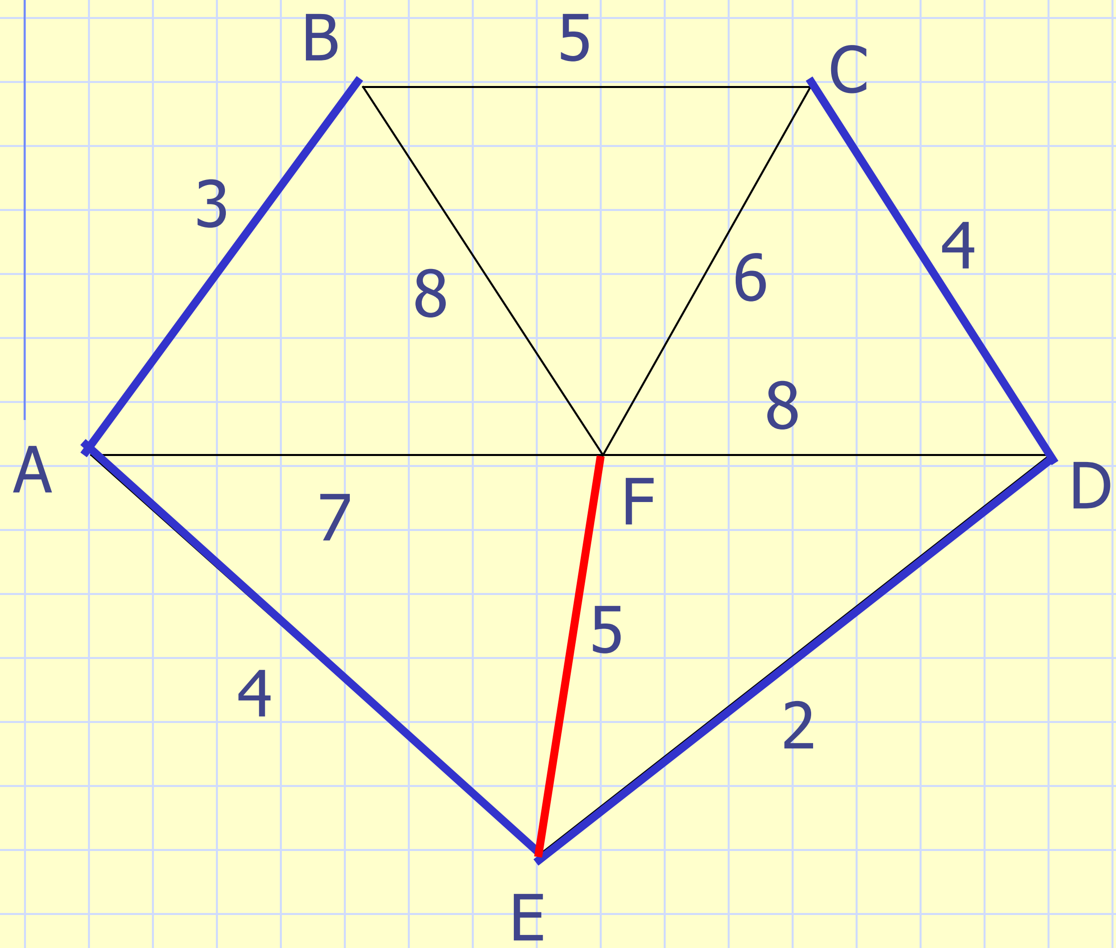
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4

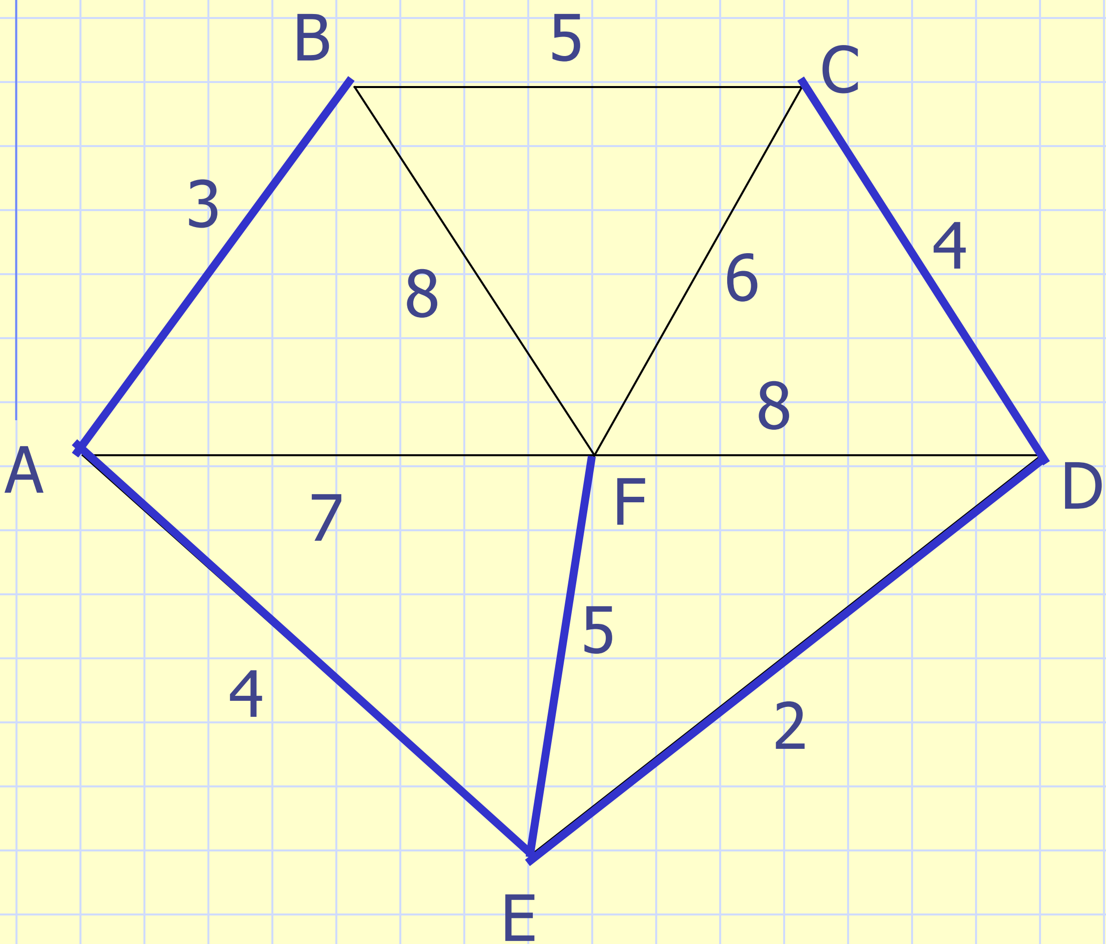
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4
BC 5 – forms a cycle
EF 5

Kruskal's Algorithm



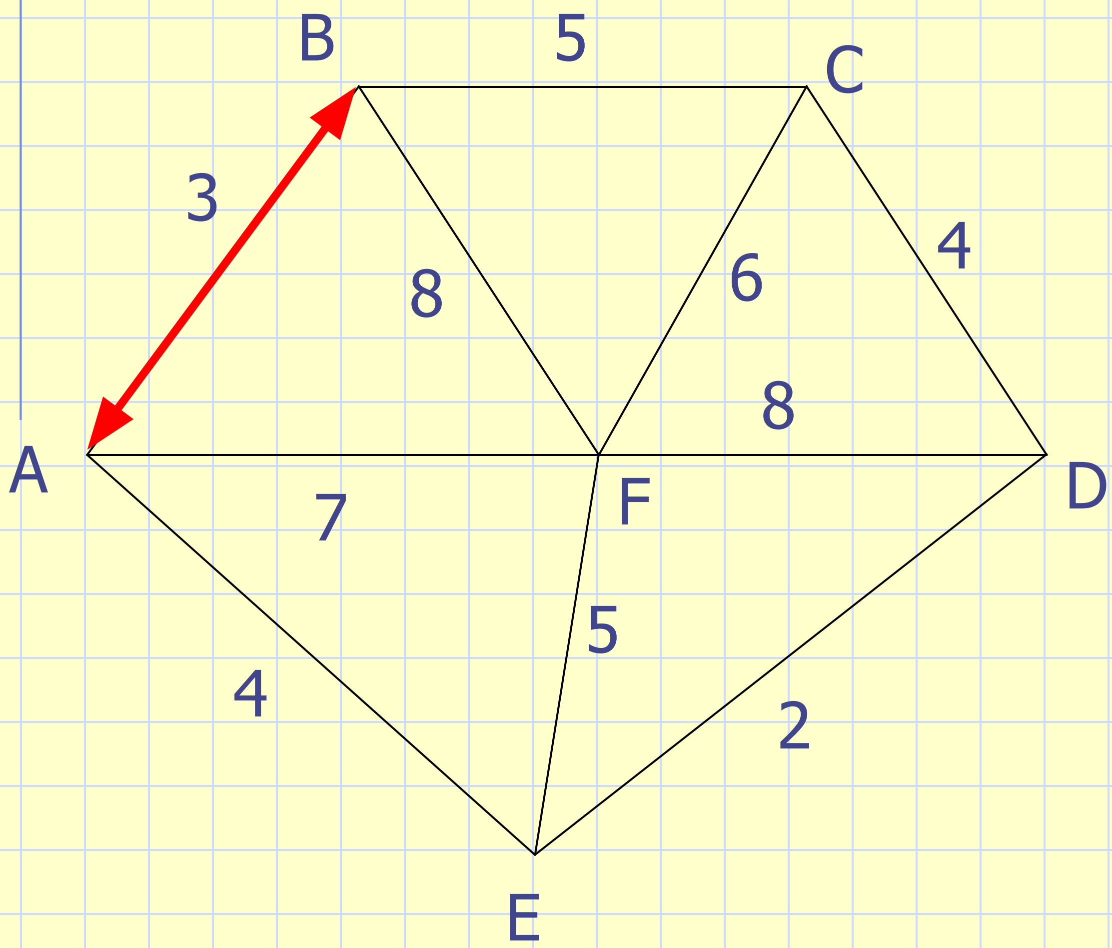
All vertices have been connected.

The solution is

ED 2
AB 3
CD 4
AE 4
EF 5

Total weight of tree: 18

Prim's Algorithm



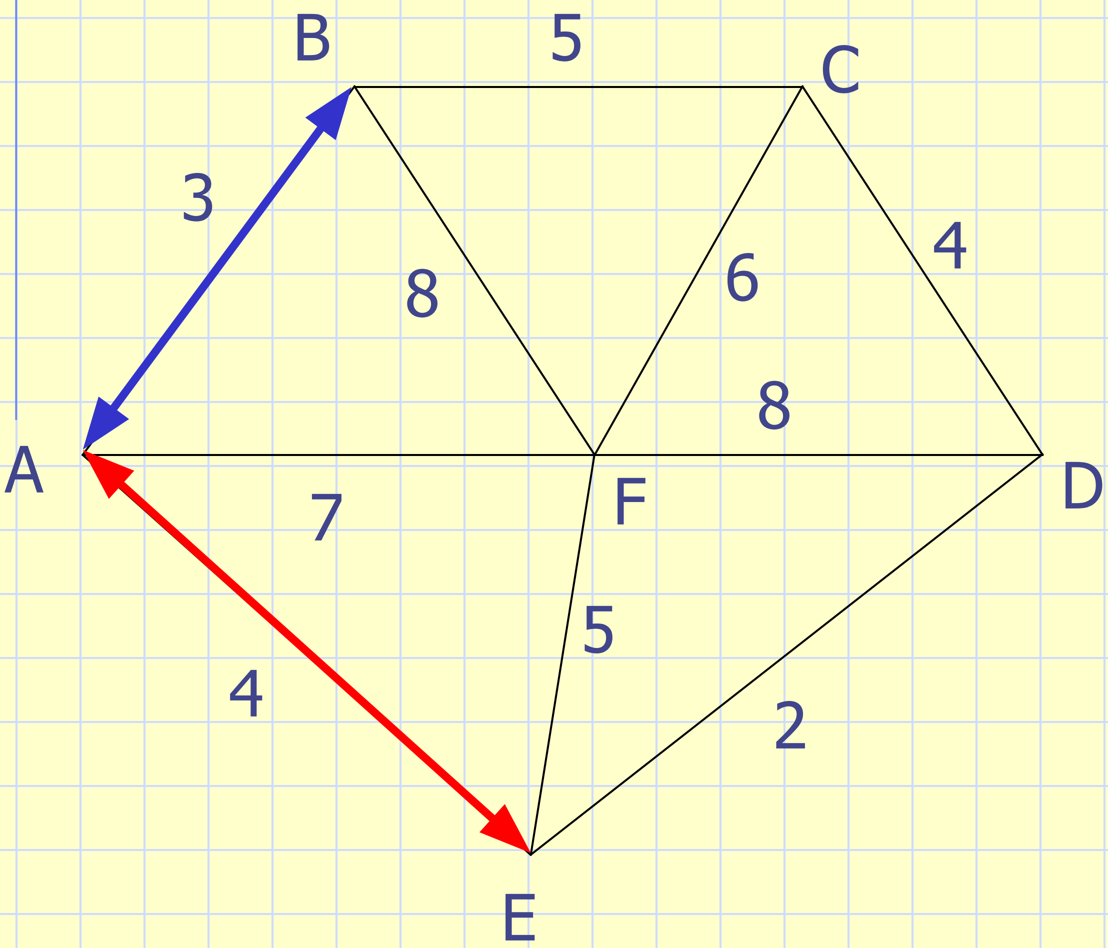
Select any vertex

A

Select the shortest edge connected to that vertex

AB 3

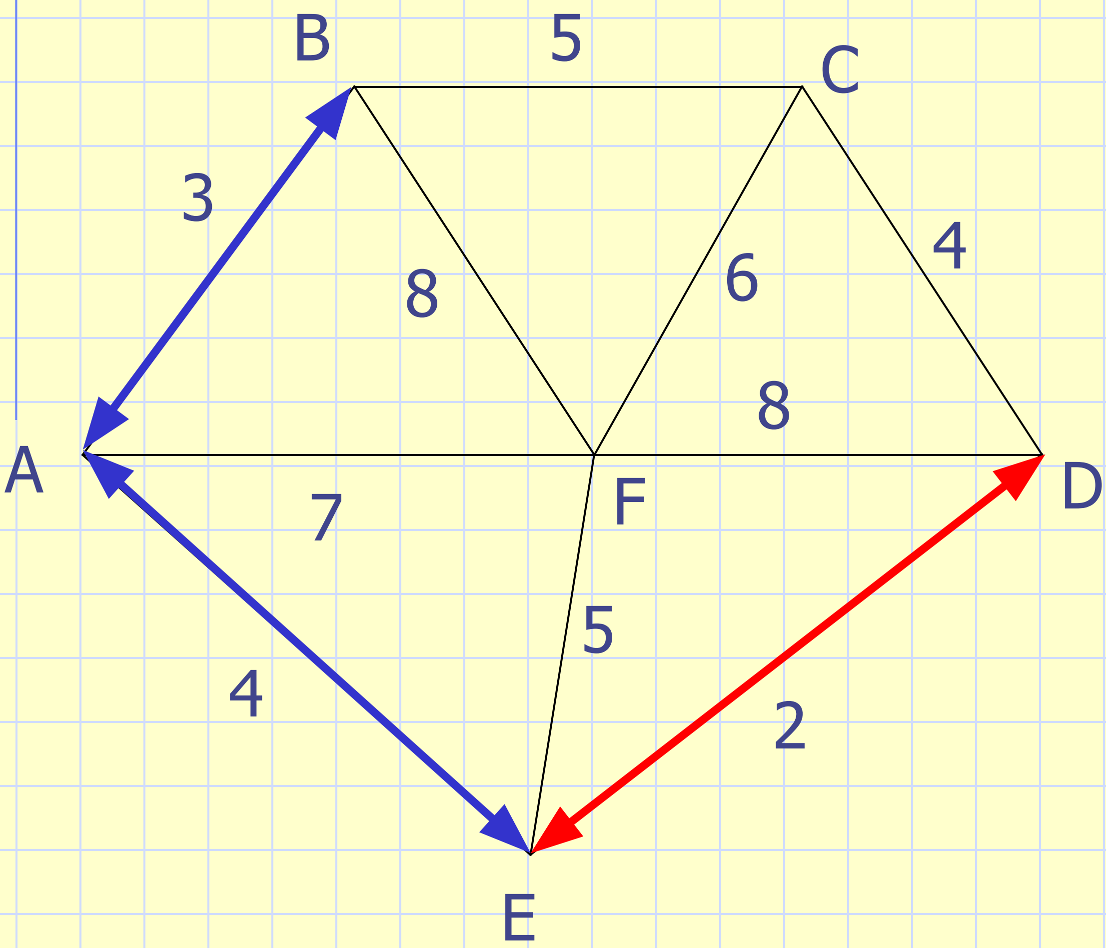
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

AE 4

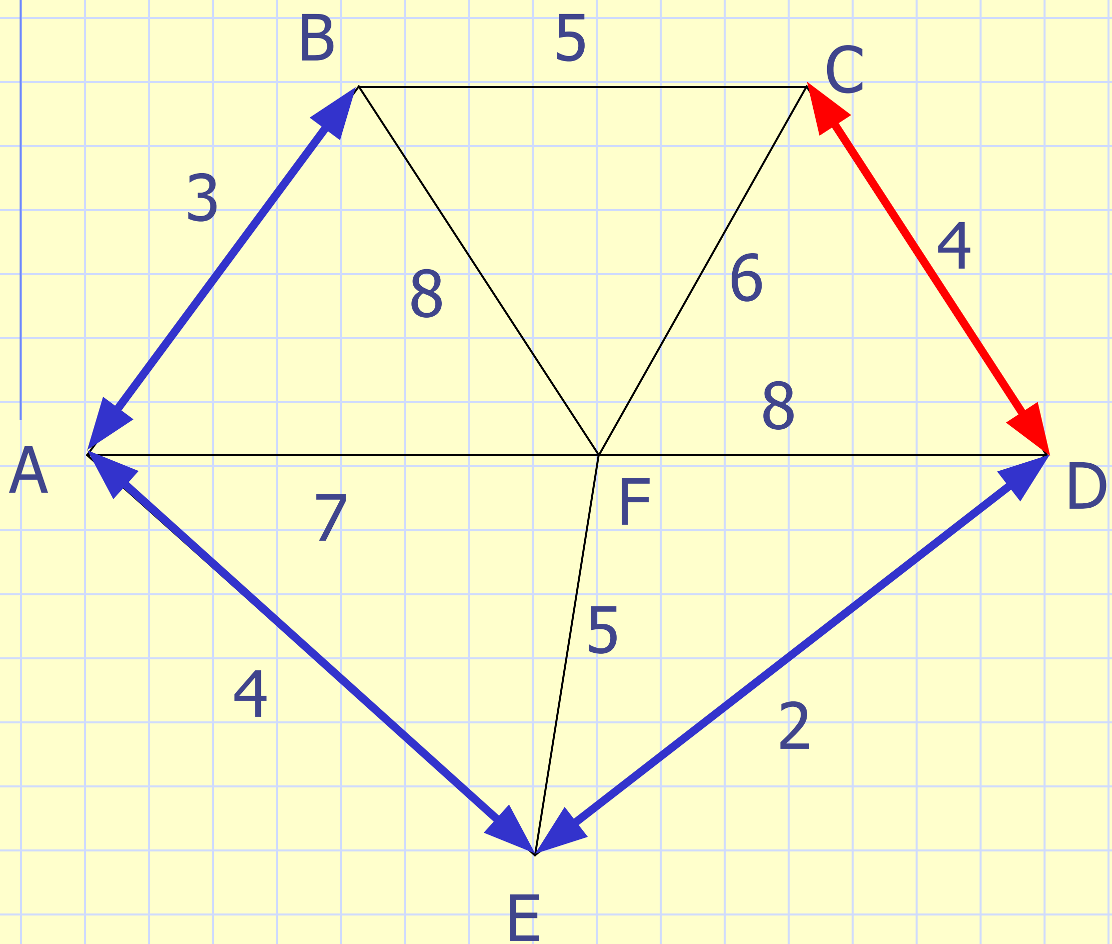
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

ED 2

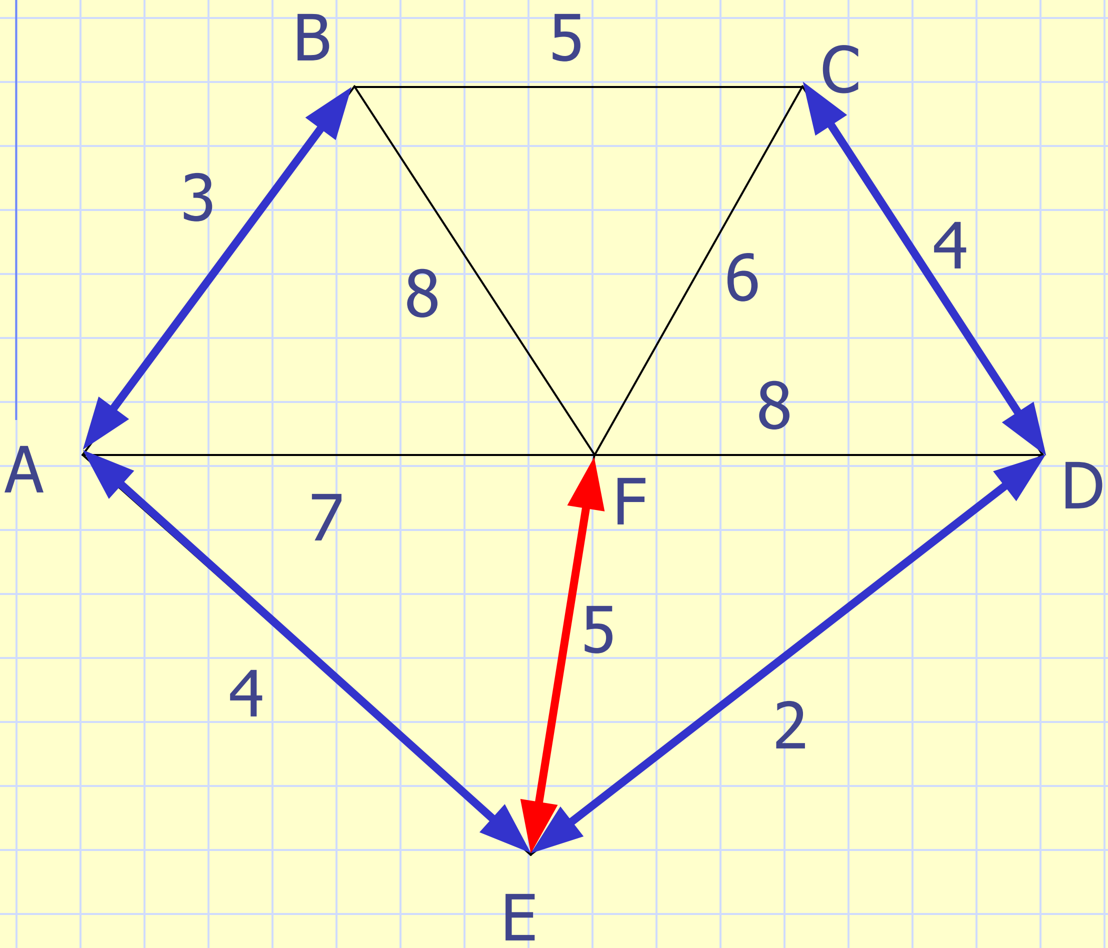
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

DC 4

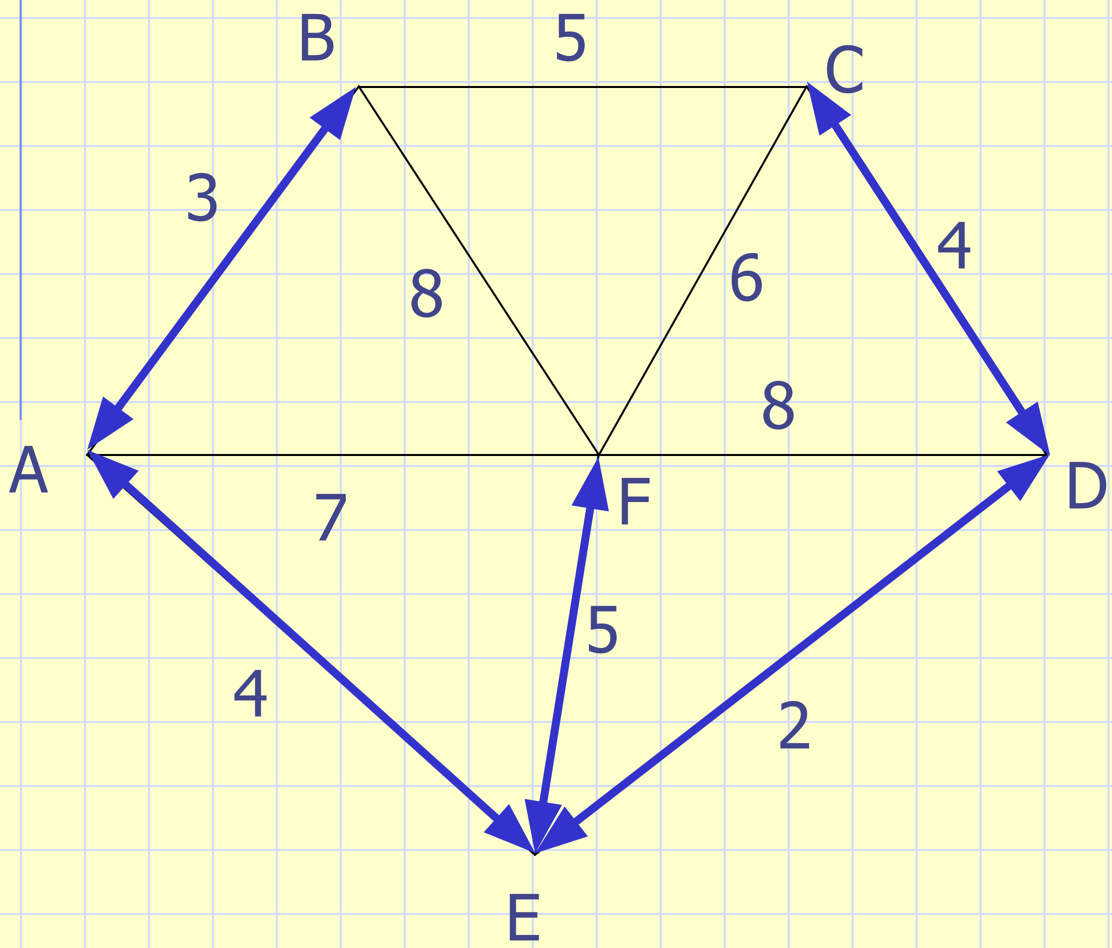
Prim's Algorithm



Select the shortest edge connected to any vertex already connected.

EF 5

Prim's Algorithm



All vertices have been connected.

The solution is

AB 3
AE 4
ED 2
DC 4
EF 5

Total weight of tree: 18

Some points to note

- Both algorithms will always give solutions with the same length.
- They will usually select edges in a different order – you must show this in your workings.
- Occasionally they will use different edges – this may happen when you have to choose between edges with the same length. In this case there is more than one minimum connector for the network.

Shortest Path Problems

Dijkstra's Algorithm

Warshall's Algorithm

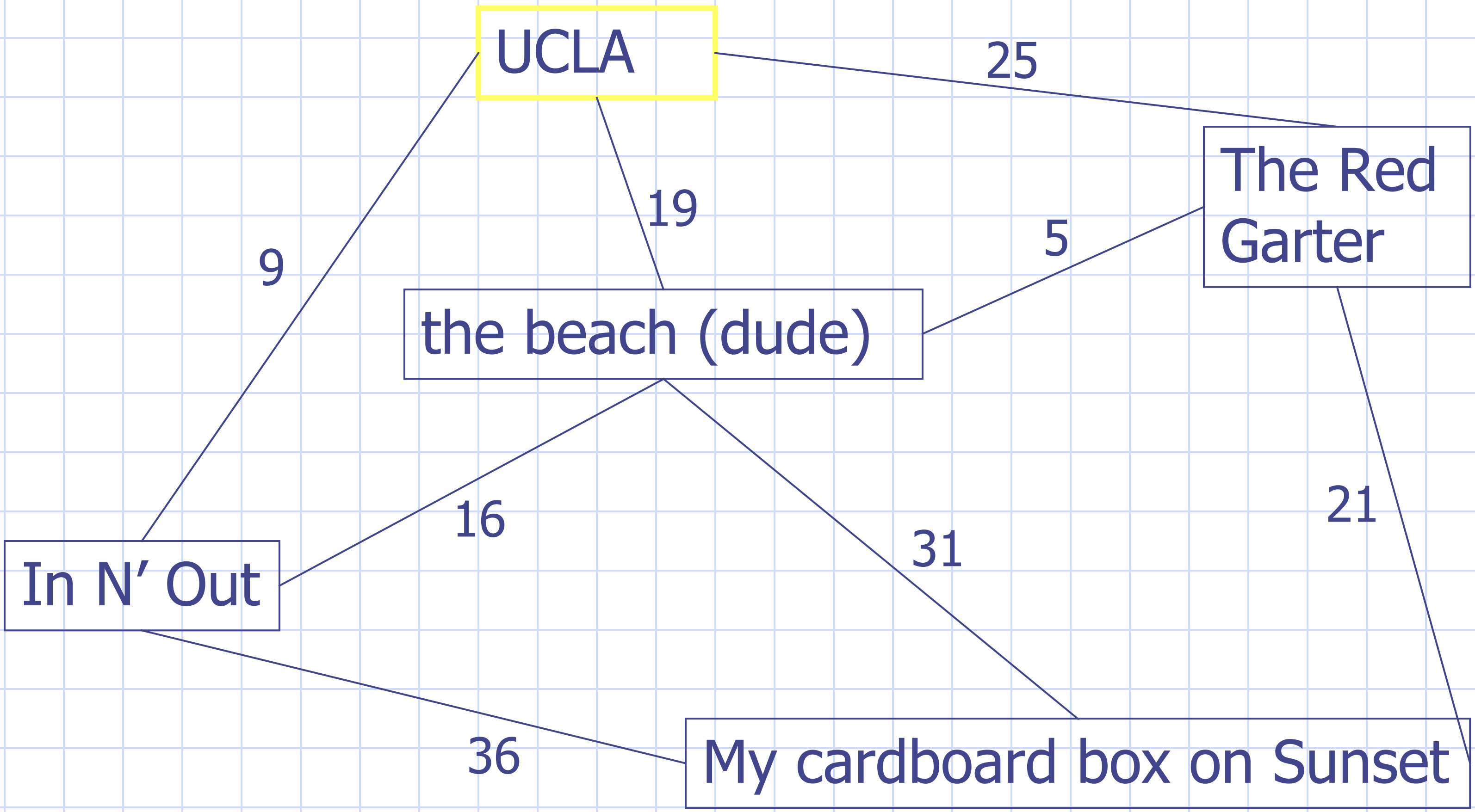
Shortest-Path Problems

- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.

Introduction

- Many problems can be modeled using graphs with weights assigned to their edges:
  - Airline flight times
  - Telephone communication costs
  - Computer networks response times

Optimal driving time



Dijkstra's algorithm

- Dijkstra's algorithm determines the distances (costs) between a given vertex and all other vertices in a graph.

Dijkstra's algorithm

- The algorithm begins at a specific vertex and extends outward within the graph, until all vertices have been reached
- The only distinction is that Prim's algorithm stores a minimum cost edge whereas Dijkstra's algorithm stores the total cost from a source vertex to the current vertex.
- More simply, Dijkstra's algorithm stores a summation of minimum cost edges whereas Prim's algorithm stores at most one minimum cost edge

Dijkstra's algorithm

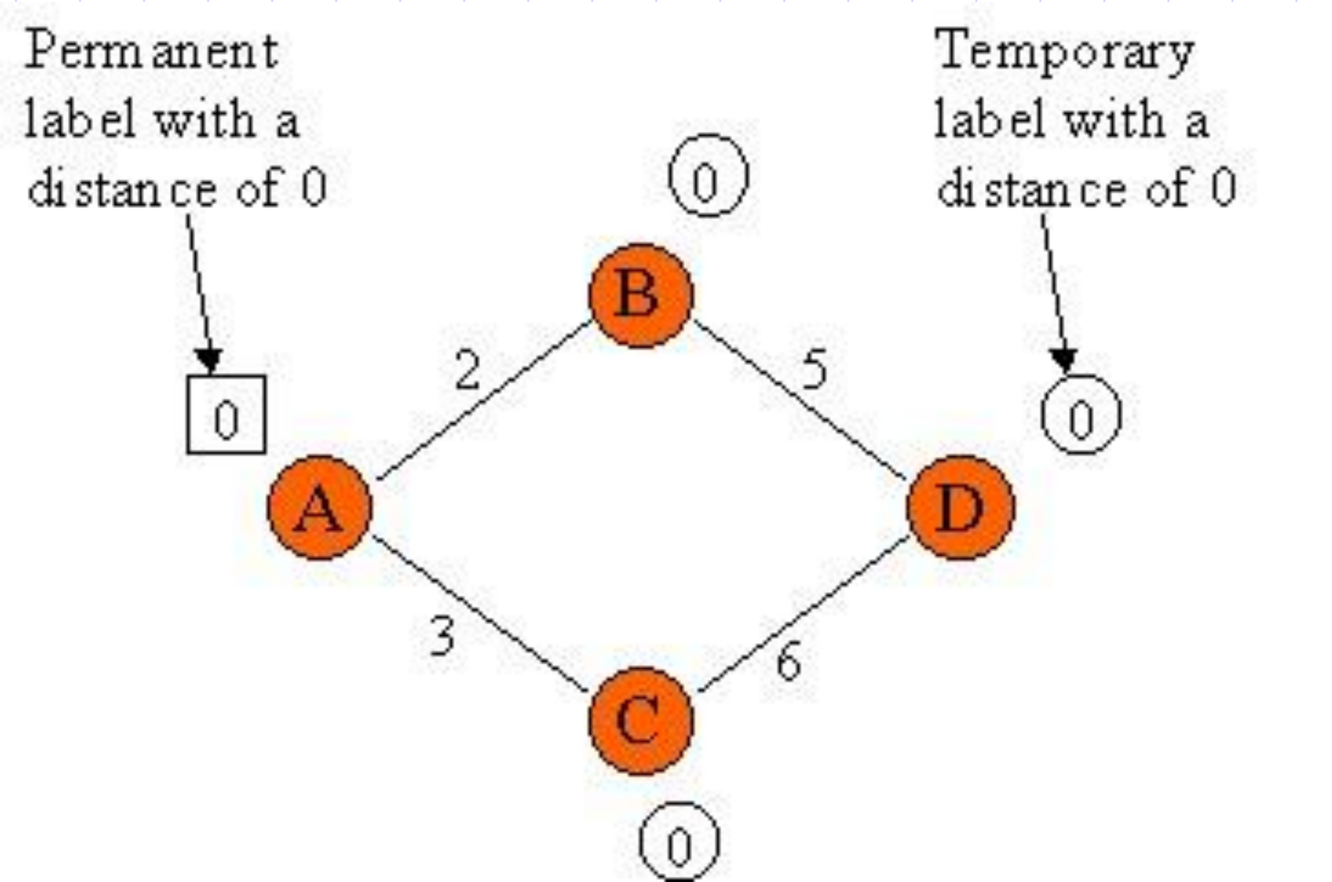
- Dijkstra's algorithm creates labels associated with vertices that represent the distance (cost) from the source vertex to that particular vertex.
- Temporary Labels are given to vertices that have not been reached
- Permanent Labels are given to vertices that have been reached

Vertex A has a temporary label with a distance of 0

Vertex B has a permanent label with a distance of 5

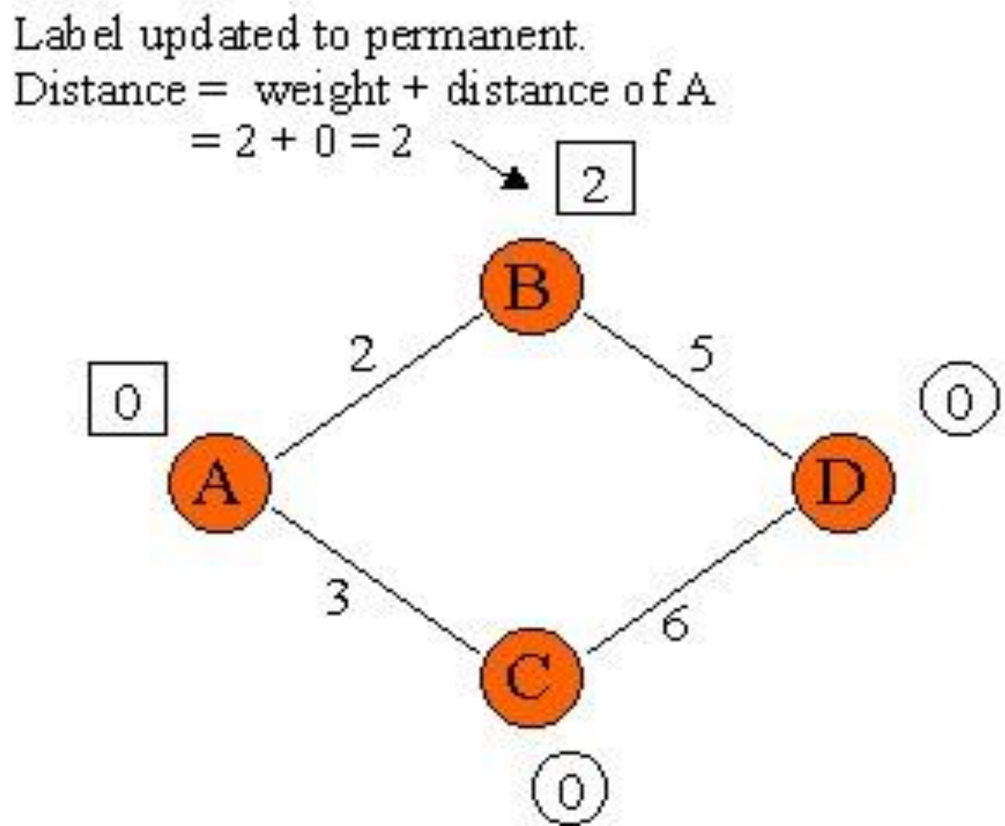
Dijkstra's algorithm

The algorithm begins by initializing any vertex in the graph (vertex A, for example) a permanent label with the value of 0, and all other vertices a temporary label with the value of 0.



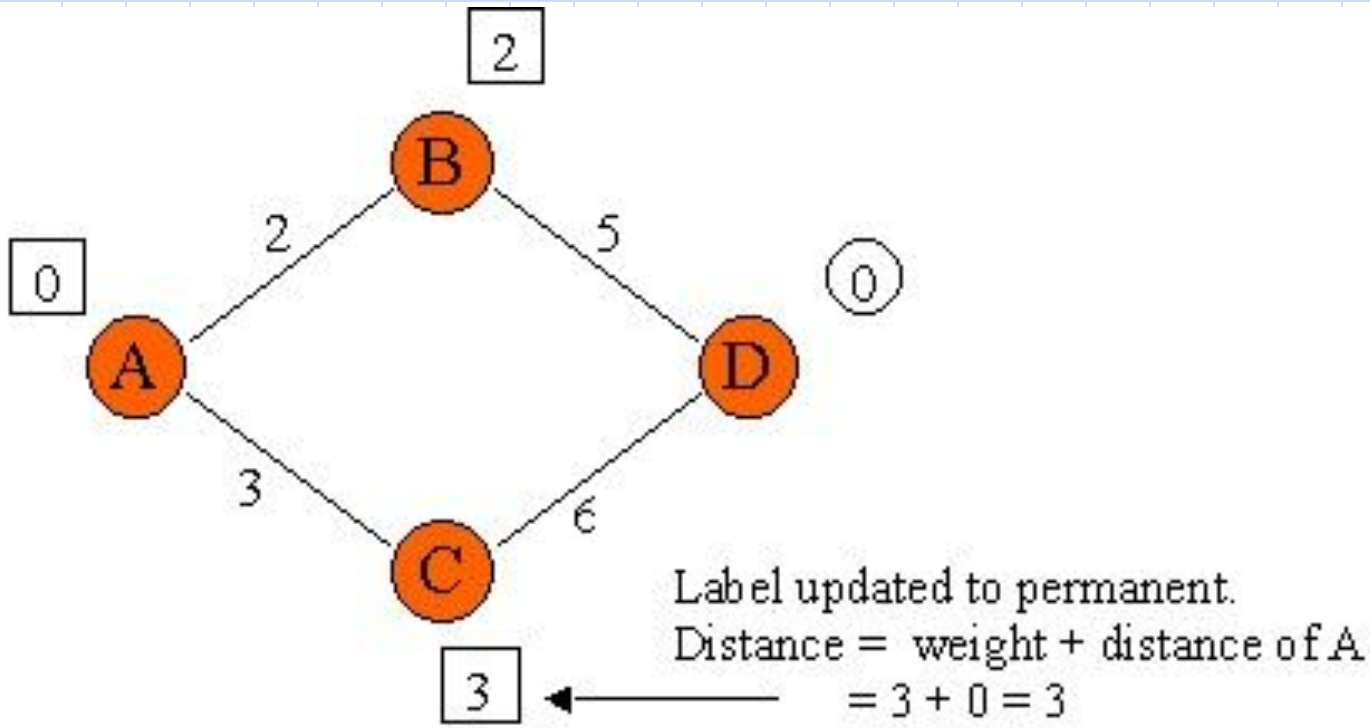
Dijkstra's algorithm

The algorithm then proceeds to select the least cost edge connecting a vertex with a permanent label (currently vertex A) to a vertex with a temporary label (vertex B, for example).

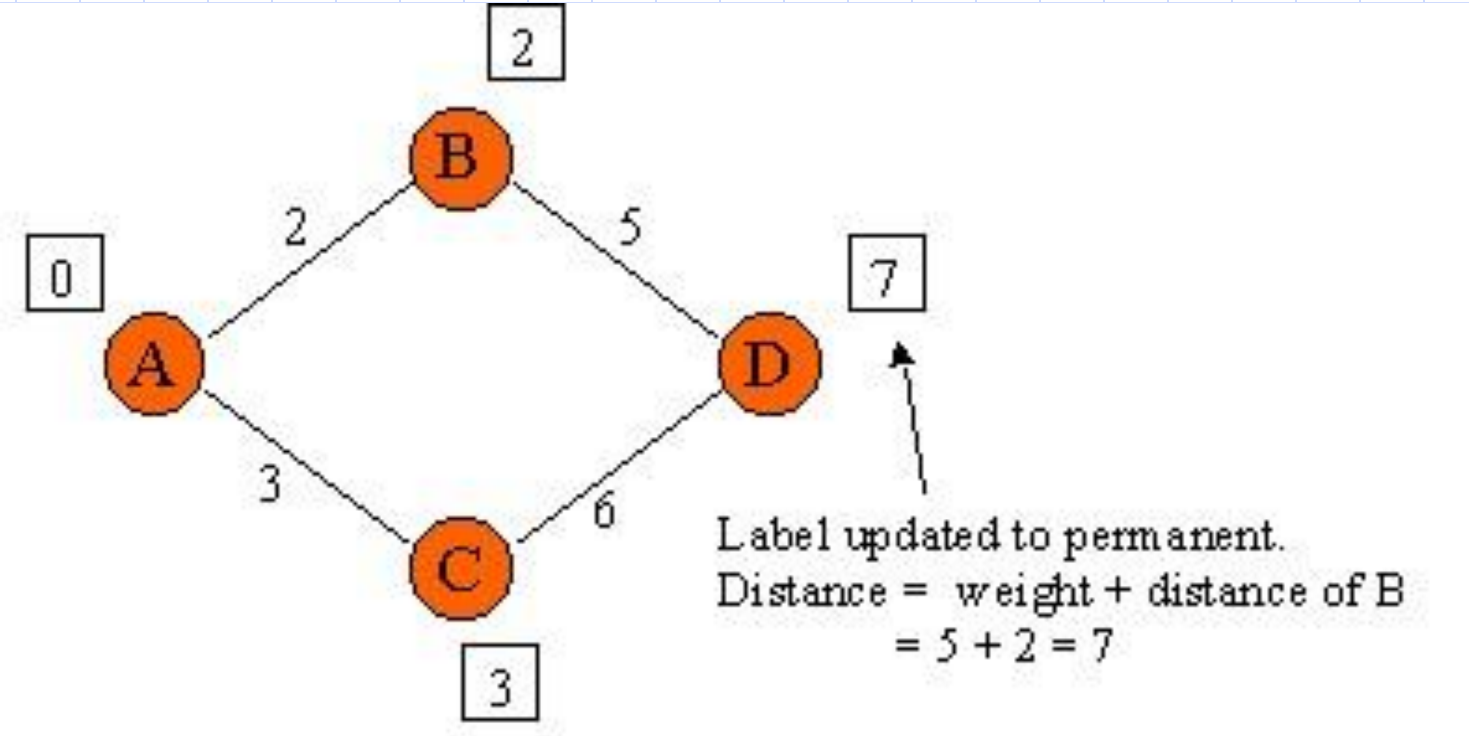


Dijkstra's algorithm

The next step is to find the next least cost edge extending to a vertex with a temporary label from either vertex A or vertex B (vertex C, for example),



Dijkstra's algorithm

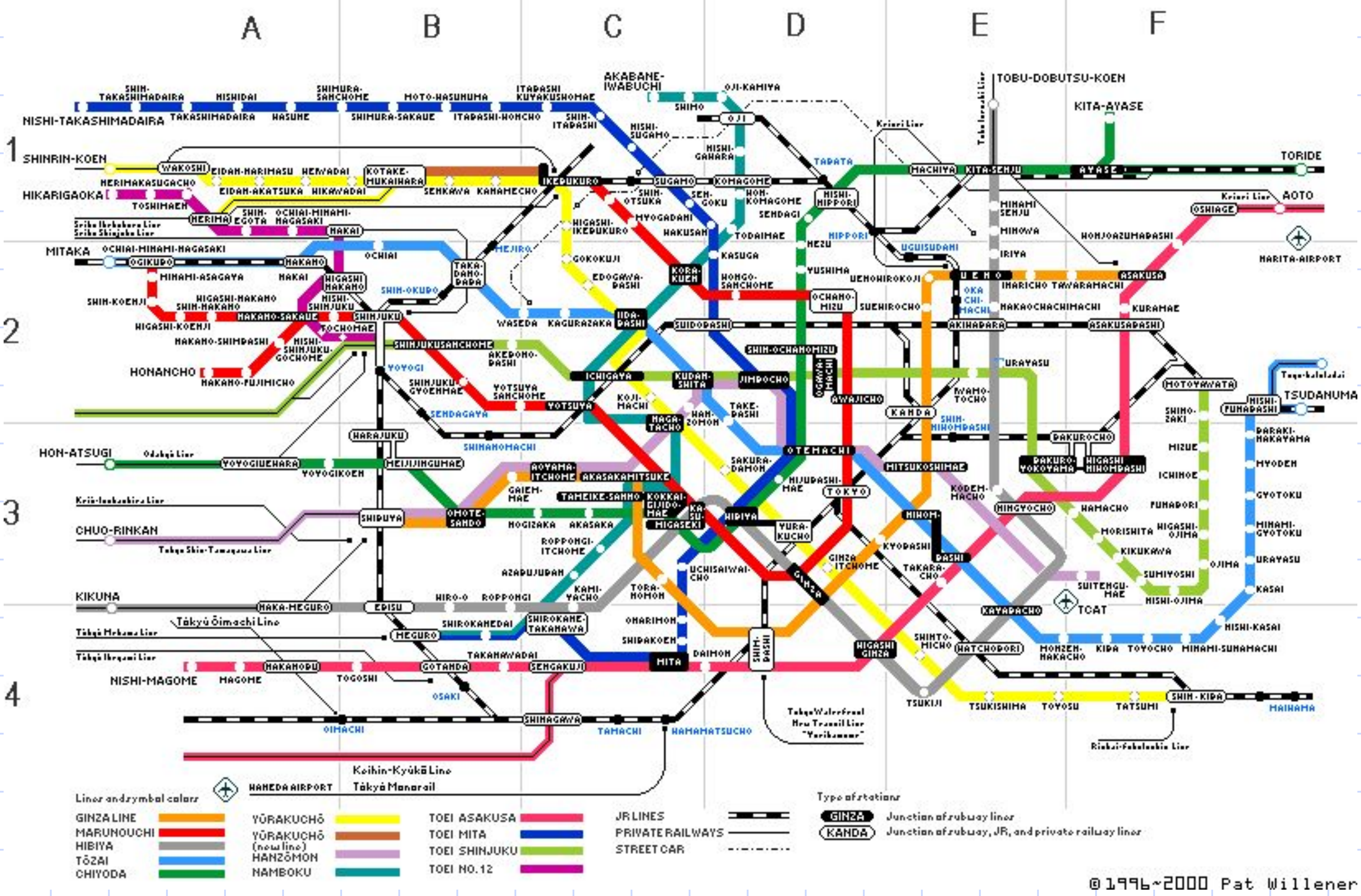


This process is repeated until the labels of all vertices in the graph are permanent

Dijkstra's algorithm

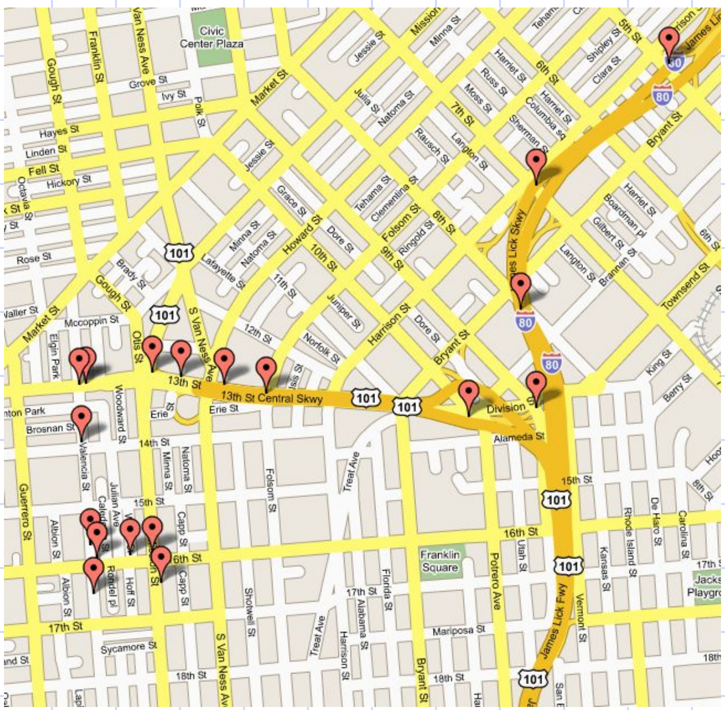
- useful to determine alternatives in decision making.
- For example, a telephone company may forgo the decision to install a new telephone cable in a rural area when presented with the option of installing the same cable in a city, reaching twice the people at half the cost.

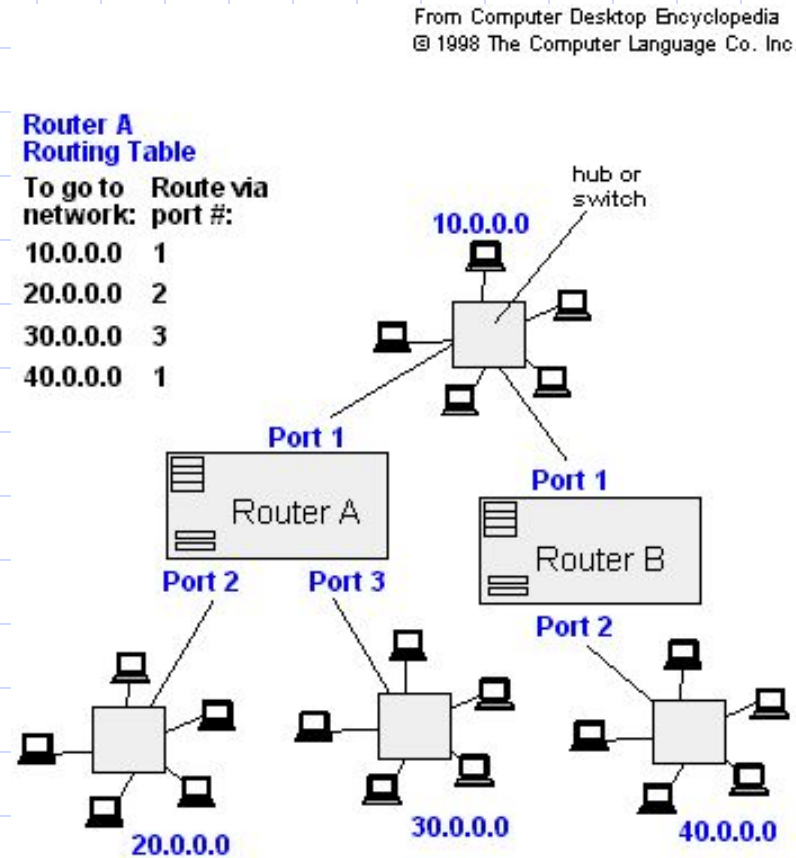
Tokyo Subway Map



Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems





Warshall's algorithm of path matrix

Step 1: [Initialization of path matrix]

Repeat through step 2 for i=0,1,2,…n-1

Repeat through step 2 for j=0,1,2,…n-1

Step 2: [Test the condition and assign accordingly value to path matrix accordingly]

If (a[i][j] == 0)

p[i][j]=0

Else

p[i][j]=1

Step 3: [Evaluate path matrix]

Repeat through step 4 for k=0,1,2,….,n-1

Repeat through step 4 for i=0,1,2,….,n-1

Repeat through step 4 for j=0,1,2,….,n-1

Step 4:p[i][j]=p[i][j] V (p[i][k] Λ p[k][j]);

Step 5: Exit

Modified Warshall's algorithm

Step 1: [Initialization matrix m]

Repeat through step 2 for i=0,1,2,…n-1

Repeat through step 2 for j=0,1,2,…n-1

Step 2: [Test the condition and assign the required value to matrix m]

If (a[i][j] == 0)

m[i][j]=Infinity

Else

m[i][j]=a[i][j]

Step 3: [Shortest path Evaluation]

Repeat through step 4 for k=0,1,2,….,n-1

Repeat through step 4 for i=0,1,2,….,n-1

Repeat through step 4 for j=0,1,2,….,n-1

Step 4: If m[i][j] < m[i][k] + m[k][j]

m[i][j]=m[i][j]

Else

m[i][j] = m[i][k] + m[k][j]

Step 5: Exit

Dijkstra's Algorithm

Step 1: Assign a temporary label (vi)=∞ to all vertices except v_s

Step 2: [Mark v_s as permanent by assigning 0 label to it]

$l(v_s)=0$

Step 3: [Assign value of v_s to v_r where v_r is last vertex to be made permanent]

$v_r = v_s$

Step 4: if $l(v_i) > l(v_k) + w(v_k, v_i)$

$l(v_i) = l(v_k) + w(v_k, v_i)$

Step 5: $v_r = v_i$

Step 6: If v_t has temporary label, repeat step 4 to step 5 otherwise the value of v_t is permanent label and is equal to the shortest path v_s to v_t

Step 7: Exit