Graphs

DSA

Intro

- Our initial part of the course focused on linear data structures—the array, the List, the Queue, the Stack
- Then we saw the concept of trees.
  - trees consist of a set of *nodes*, where all of the nodes share some connection to other nodes.
  - These connections are referred to as *edges*.
    - For example, all nodes in a tree except for one—the root—must have precisely one *parent* node, while all nodes can have an arbitrary number of children. These simple rules ensure that, for any tree, the following statements will hold:
  - Starting from any node, any other node in the tree can be reached. That is, there exists no node that can't be reached through some simple path.
  - There are no *cycles*. A cycle exists when, starting from some node *v*, there is some path that travels through some set of nodes *v*1, *v*2, ..., *v*k that then arrives back at *v*.
- Binary trees are trees whose nodes have at most two children

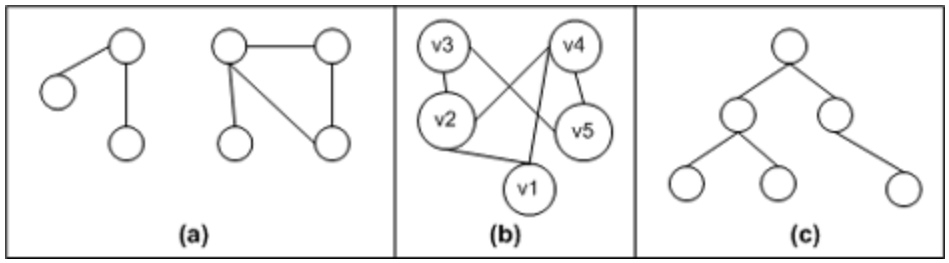
Intro

- Now we're going to examine *graphs*. Graphs are composed of a set of nodes and edges, just like trees, but with graphs there are no rules for the connections between nodes.
- With graphs there is no concept of a root node, nor is there a concept of parents and children. Rather, a graph is just a collection of interconnected nodes.

Realize that all trees are graphs. A tree is a special case of a graph, one whose nodes are all reachable from some starting node and one that has no cycles

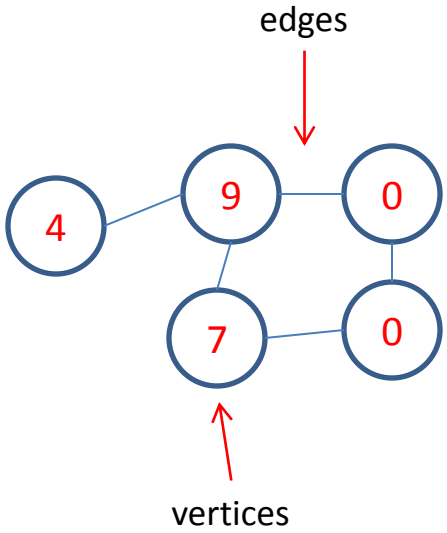
Some examples

- Notice that graphs, unlike trees, can have sets of nodes that are disconnected from other sets of nodes.
  - For example, graph (a) has two distinct, unconnected set of nodes.
  - Graphs can also contain cycles. Graph (b) has several cycles. One such is the path from v1 to v2 to v4 and back to v1. Another one is from v1 to v2 to v3 to v5 to v4 and back to v1. (There are also cycles in graph (a).)
  - Graph (c) does not have any cycles, as one less edge than it does number of nodes, and all nodes are reachable. Therefore, it is a tree.



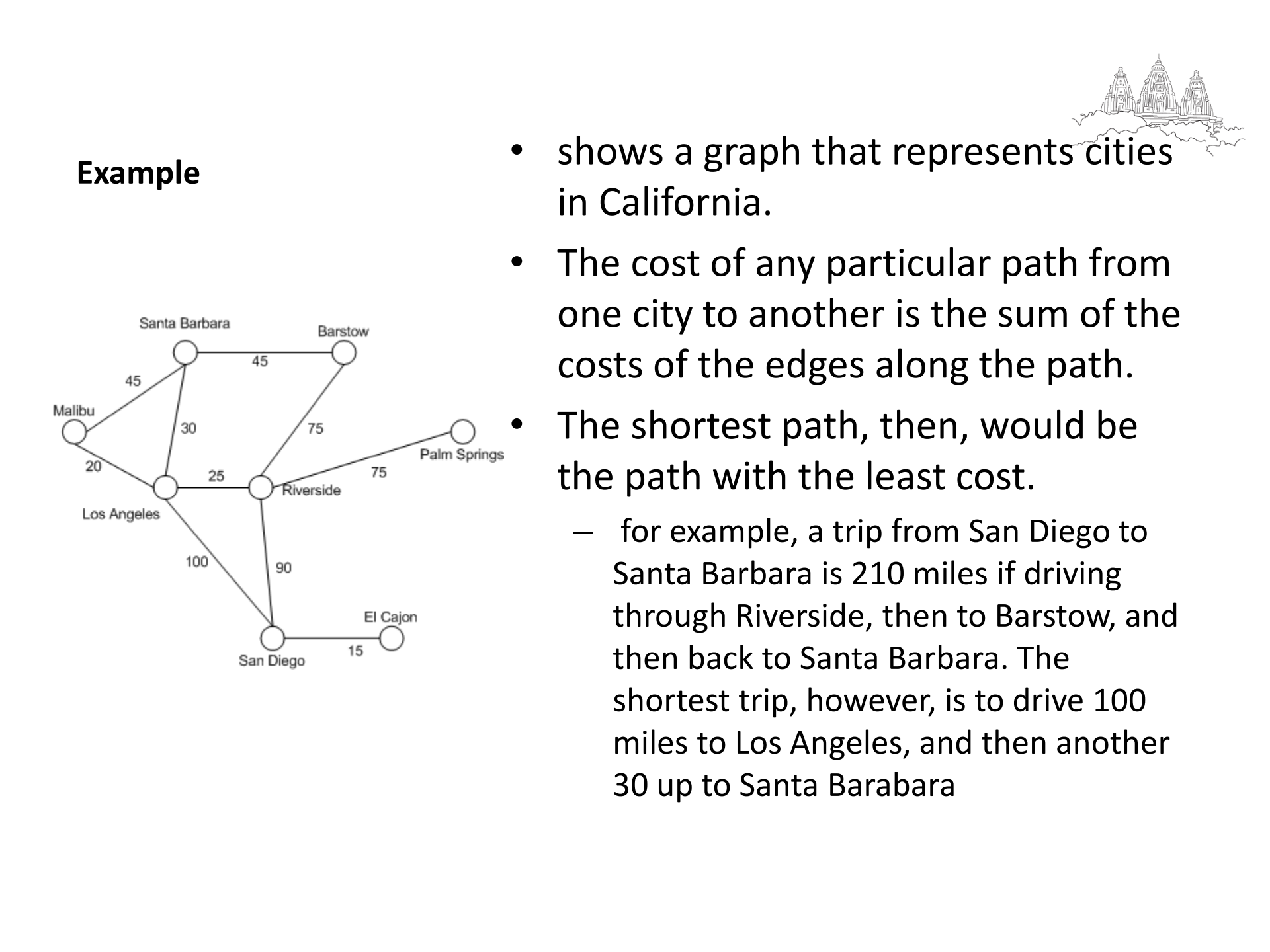
Graphs

- a **graph** is an abstract representation of a set of objects where some pairs of the objects are connected by links.



Some applications

- Many real-world problems can be modeled using graphs.
  - For example, search engines model the Internet as a graph, where Web pages are the nodes in the graph and the links among Web pages are the edges.
  - driving directions from one city to another use graphs, modeling cities as nodes in a graph and the roads connecting the cities as edges.

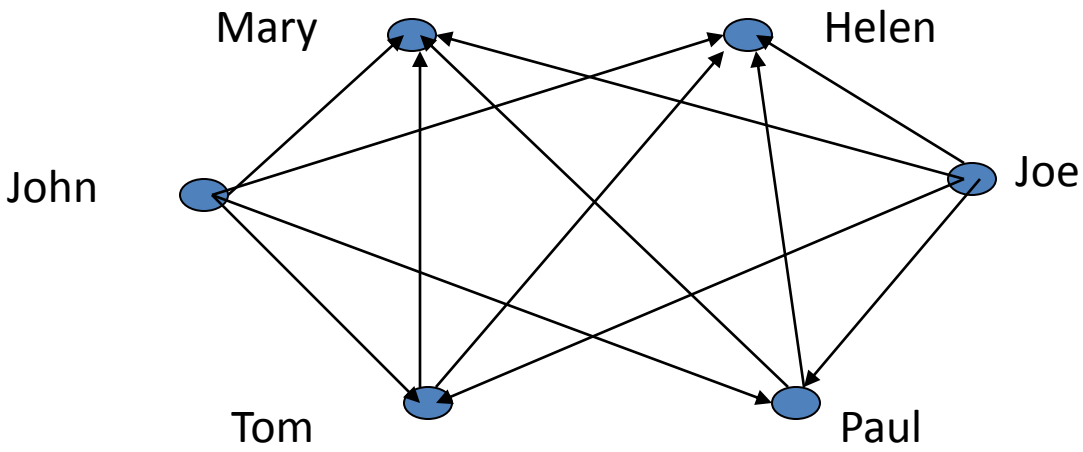


Example

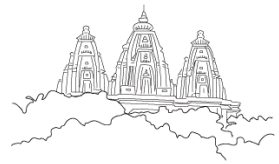
- shows a graph that represents cities in California.
- The cost of any particular path from one city to another is the sum of the costs of the edges along the path.
- The shortest path, then, would be the path with the least cost.
  - for example, a trip from San Diego to Santa Barbara is 210 miles if driving through Riverside, then to Barstow, and then back to Santa Barbara. The shortest trip, however, is to drive 100 miles to Los Angeles, and then another 30 up to Santa Barabara

A "Real-life" Example of a Graph

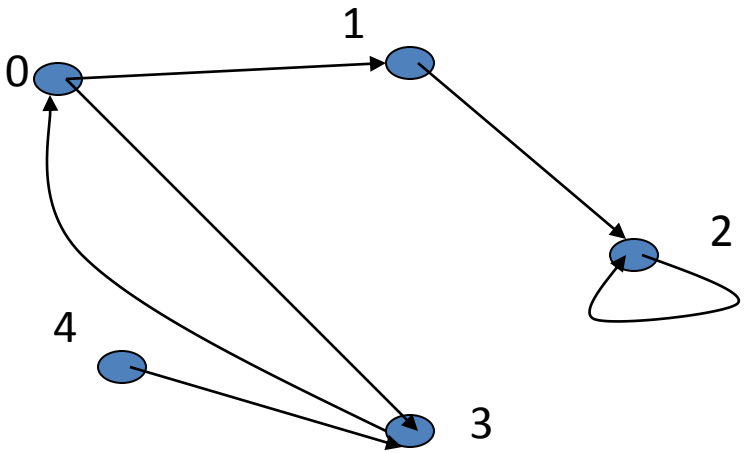
- V = set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively
- E ={(x,y) | if x is younger than y}



Examples of Graphs



- V={0,1,2,3,4}
- E={(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)}



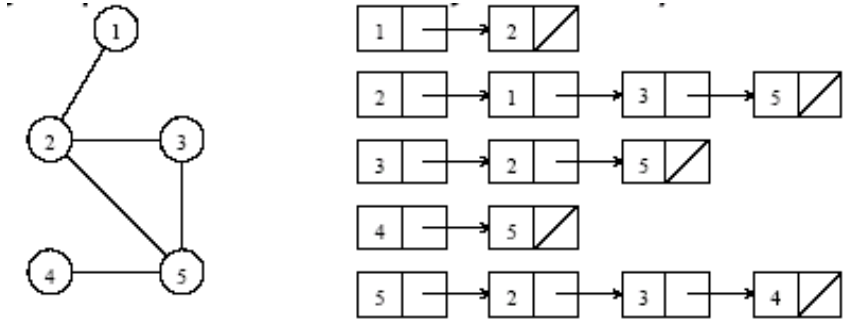
When (x,y) is an edge,
we say that x is *adjacent to* y, and y is *adjacent from* x.

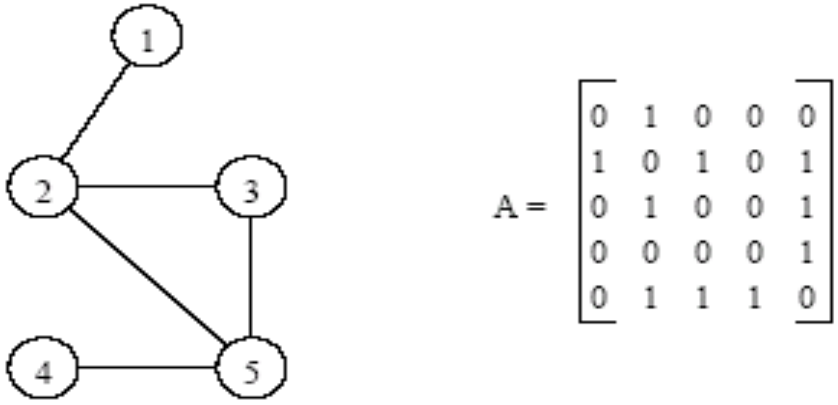
0 is adjacent to 1.
1 is not adjacent to 0.
2 is adjacent from 1.

Intuition Behind Graphs

- The nodes represent entities (such as people, cities, computers, words, etc.)
- Edges (x,y) represent relationships between entities x and y, such as:
  - "x likes y"
  - "x hates y"
  - "x is a friend of y" (note that this not necessarily reciprocal)
  - "x considers y a friend"
  - "x is a child of y"
  - "x is a half-sibling of y"
  - "x is a full-sibling of y"
- In those examples, each relationship is a different graph

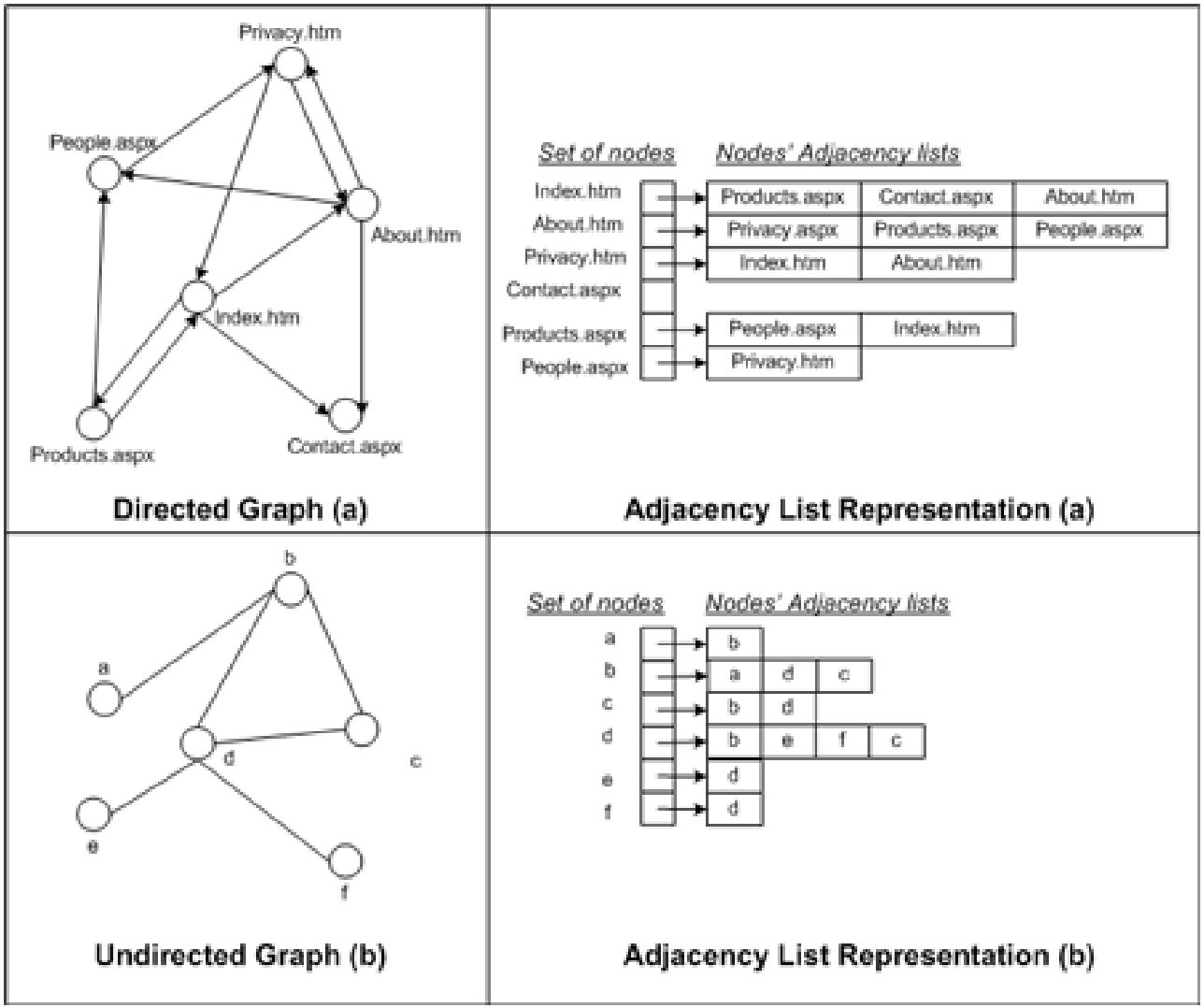
Definitions and Representation



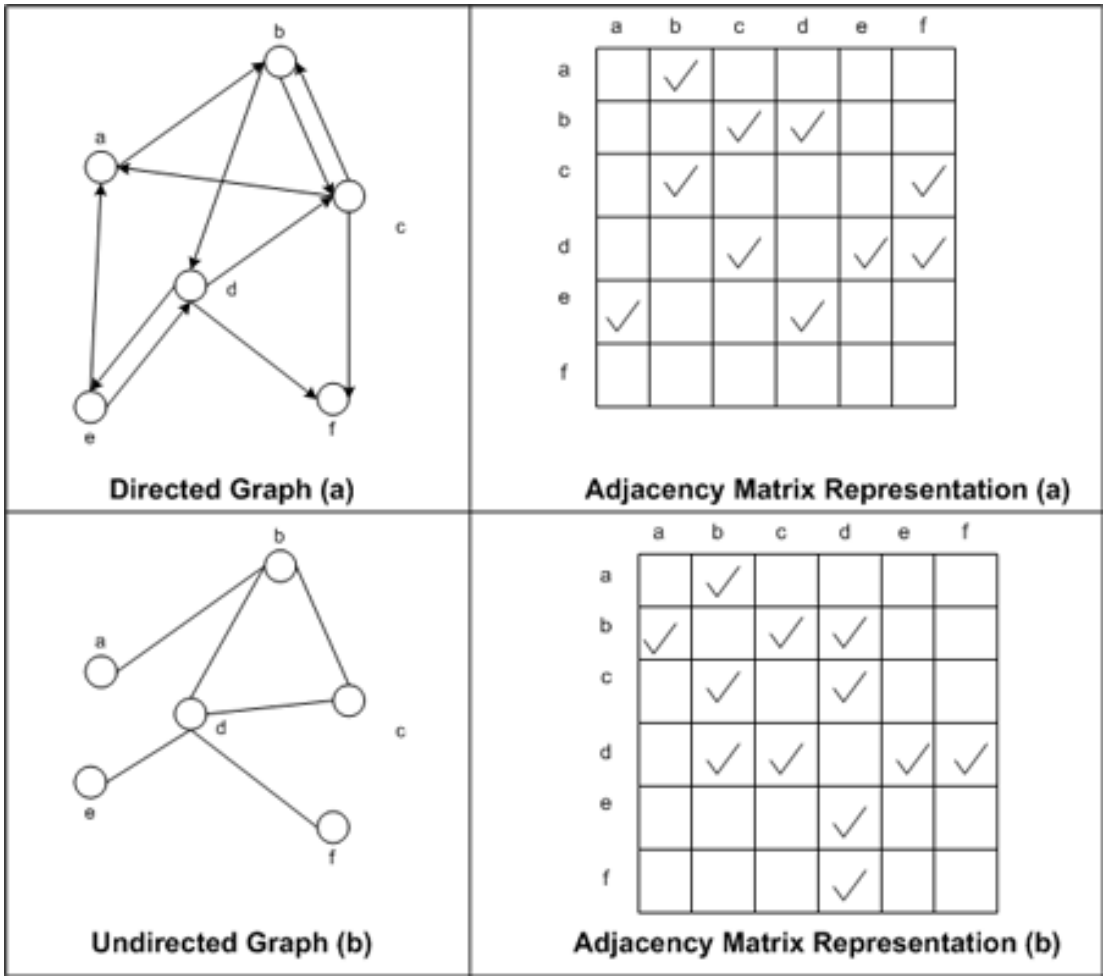


- graphs are typically modeled in many ways - two are shown:
  - As an adjacency list
  - As an adjacency matrix
- An undirected graph and its adjacency list representation
- An undirected graph and its adjacency matrix representation.

Example - List

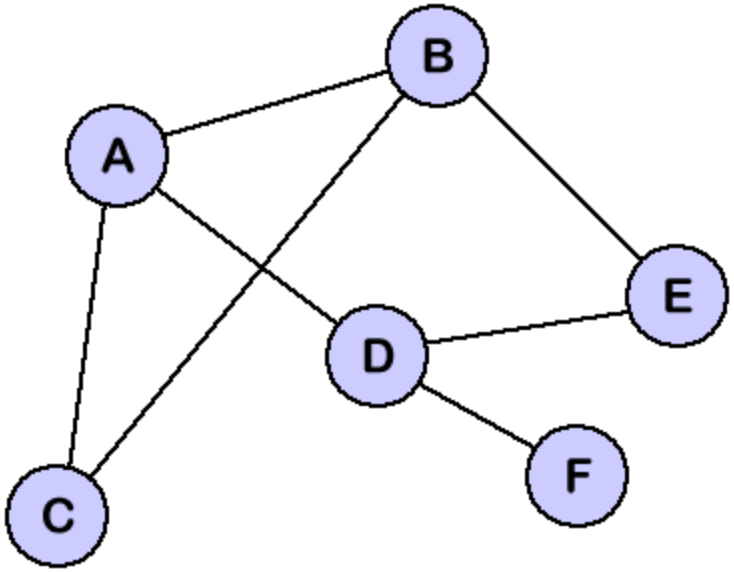


Matrix


a b c d e f
Directed Graph (a)
Adjacency Matrix Representation (a)
Undirected Graph (b)
Adjacency Matrix Representation (b)

Def

- A graph is a finite set of nodes with edges between nodes
- Formally, a graph G is a structure (V,E) consisting of
  - a finite set V called the set of nodes, and
  - a set E that is a subset of VxV. That is, E is a set of pairs of the form (x,y) where x and y are nodes in V



- Notice that the matrix shown has six rows and six columns labeled with the nodes from the graph.
- mark a '1' in a cell if there exists an edge from two nodes that index that cell.
  - For example, since we have a edge between A and B, we mark a '1' in the cells indexed by A and B.

	A	B	C	D	E	F
A	-	1	1	1	-	-
B	1	-	1	-	1	-
C	1	1	-	-	-	-
D	1	-	-	-	1	1
E	-	1	-	1	-	-
F	-	-	-	1	-	-

Weighted Edges

- Very often the edges of a graph have weights associated with them.
  - distance from one vertex to another
  - cost of going from one vertex to an adjacent vertex.
  - To represent weight, we need additional field, weight, in each entry.
  - A graph with weighted edges is called a *network*.

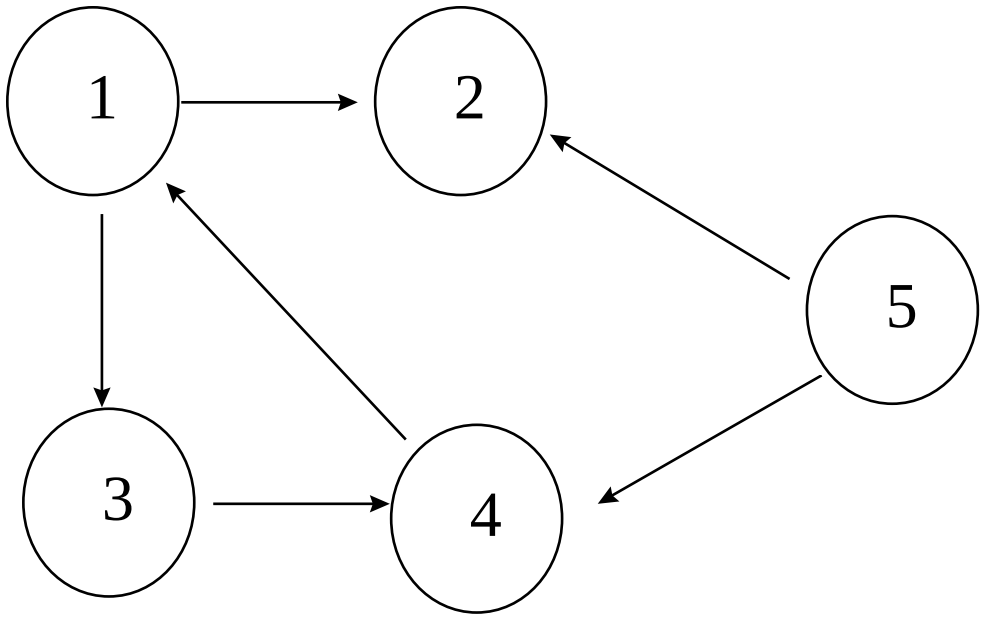
Types

- Directed graphs: G=(V,E) where E is composed of ordered pairs of vertices; i.e. the edges have direction and point from one vertex to another.
- Undirected graphs: G=(V,E) where E is composed of unordered pairs of vertices; i.e. the edges are bidirectional.

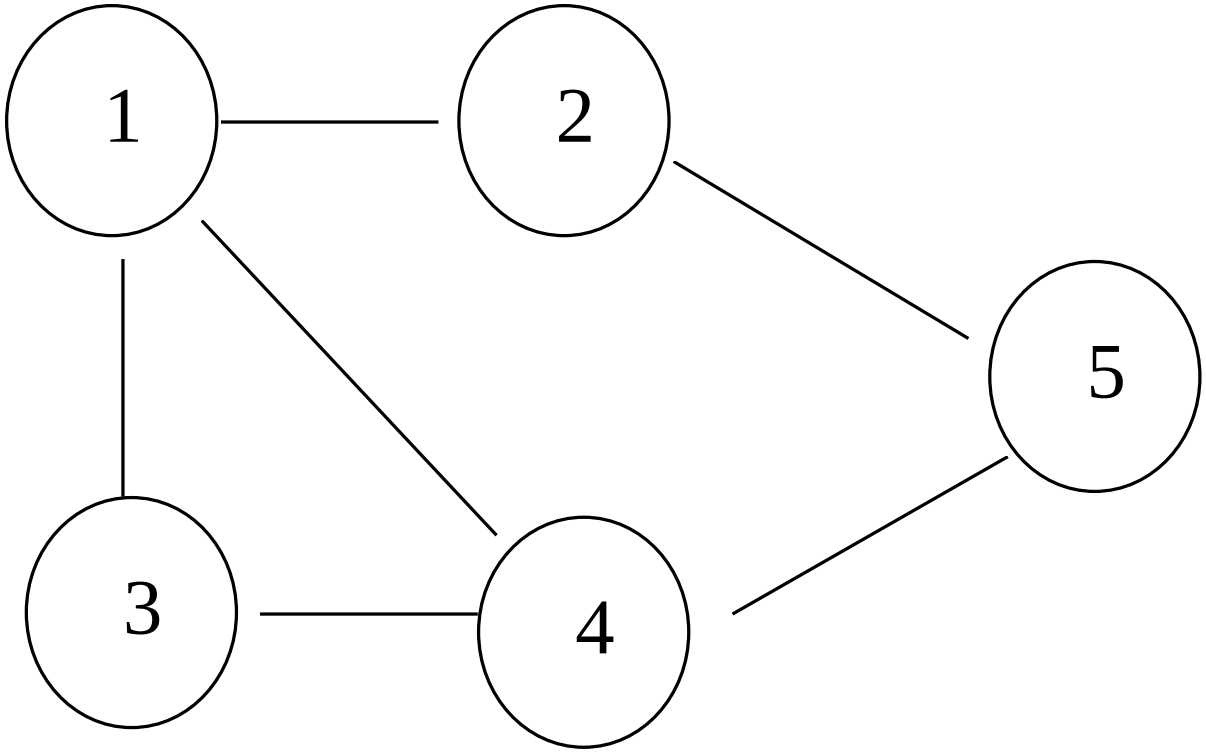
- *A graph G is often denoted G=(V,E) where V is the set of vertices and E the set of edges.*

Complete graph
Complete digraph

Directed Graph



Undirected Graph



Graph Terminology

- The **degree** of a vertex in an undirected graph is the number of edges that leave/enter the vertex.
- The degree of a vertex in a directed graph is the same, but we distinguish between in-degree and out-degree. Degree = in-degree + out-degree.
- The running time of a graph algorithm expressed in terms of E and V, where E = |E| and V=|V|; e.g. G=O(EV) is |E| * |V|

Graph Operations

- A general operation on a graph G is to visit all vertices in G that are reachable from a vertex v.
  - Depth-first search
  - Breath-first search

Operations (ADT)

- The basic operations provided by a graph data structure *G* usually include:
  - adjacent(*G*, *x*, *y*):
    - tests whether there is an edge from node *x* to node *y*.
  - neighbors(*G*, *x*):
    - lists all nodes *y* such that there is an edge from *x* to *y*.
  - add(*G*, *x*, *y*):
    - adds to *G* the edge from *x* to *y*, if it is not there.
  - delete(*G*, *x*, *y*):
    - removes the edge from *x* to *y*, if it is there.
  - get_node_value(*G*, *x*):
    - returns the value associated with the node *x*.
  - set_node_value(*G*, *x*, *a*):
    - sets the value associated with the node *x* to *a*.

Example Implementation

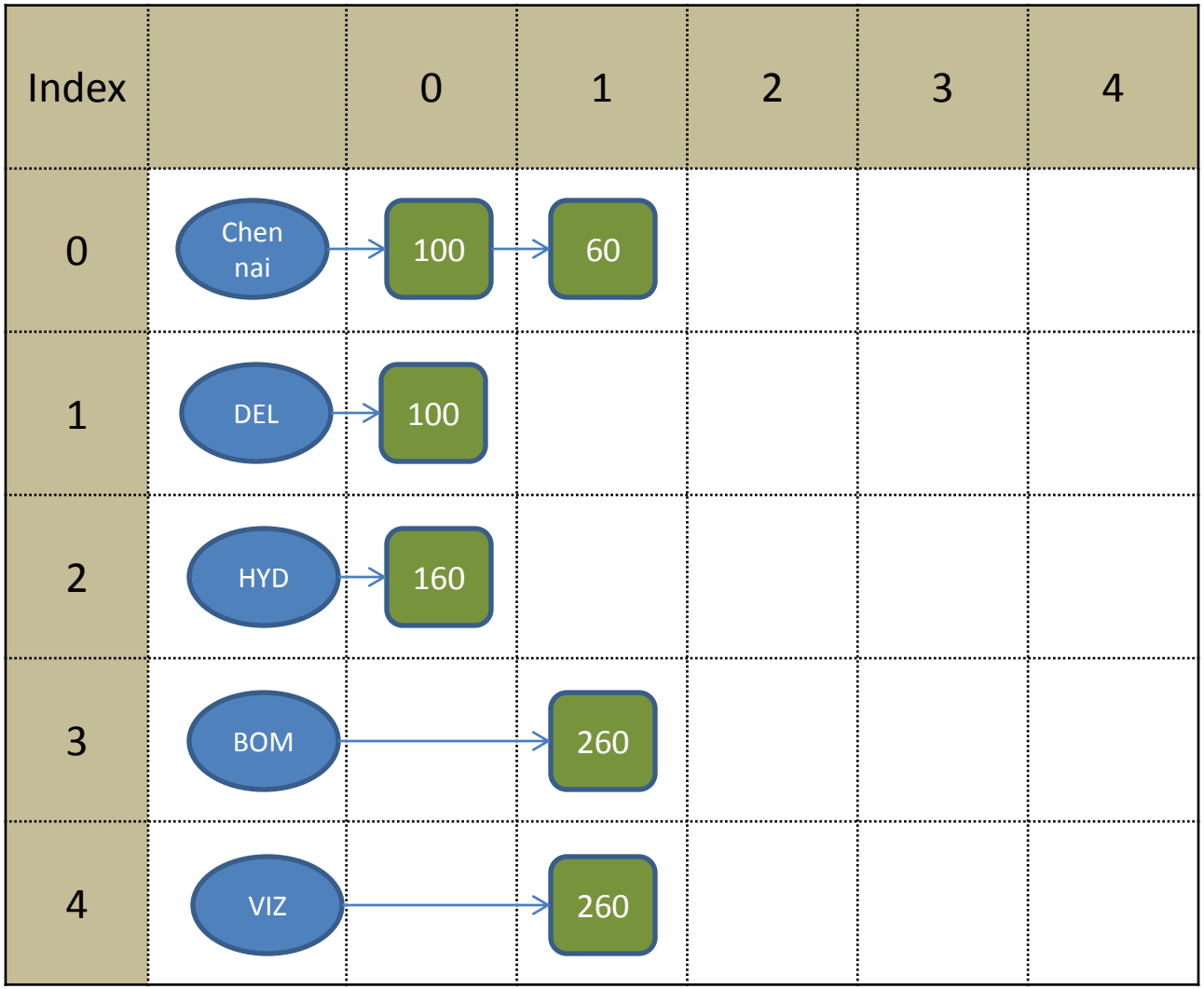
```
typedef struct CELL LIST;
struct CELL {
        int nodeKey;
        int distance;
        struct CELL *next;
};
```

```
struct {
        char city[50];
        struct CELL *adjList;
} cities[100];
```

2 elements – a vertex and a node.
For vertices, we need to store the element at that vertex.
For an edge, we just need to store a reference to the vertex it is connected to.
Thus, vertex nodes and edge nodes are different things.

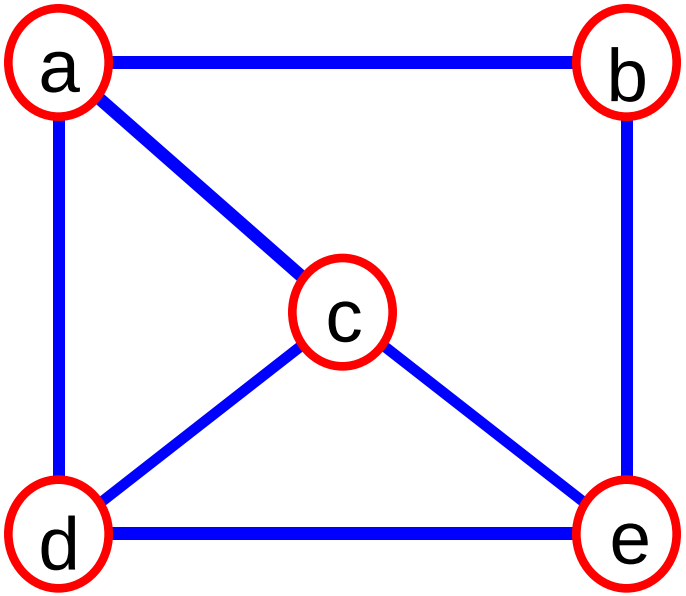
example

g1.c



Summary - 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:

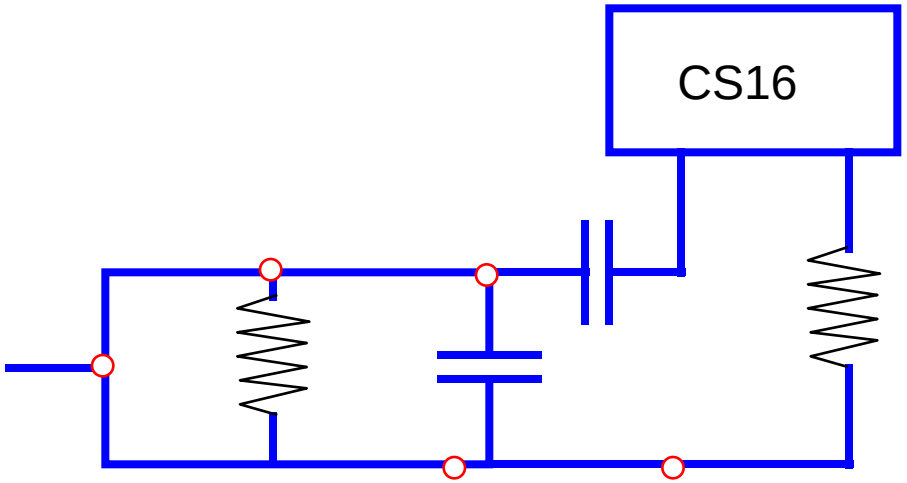


V= {a,b,c,d,e}

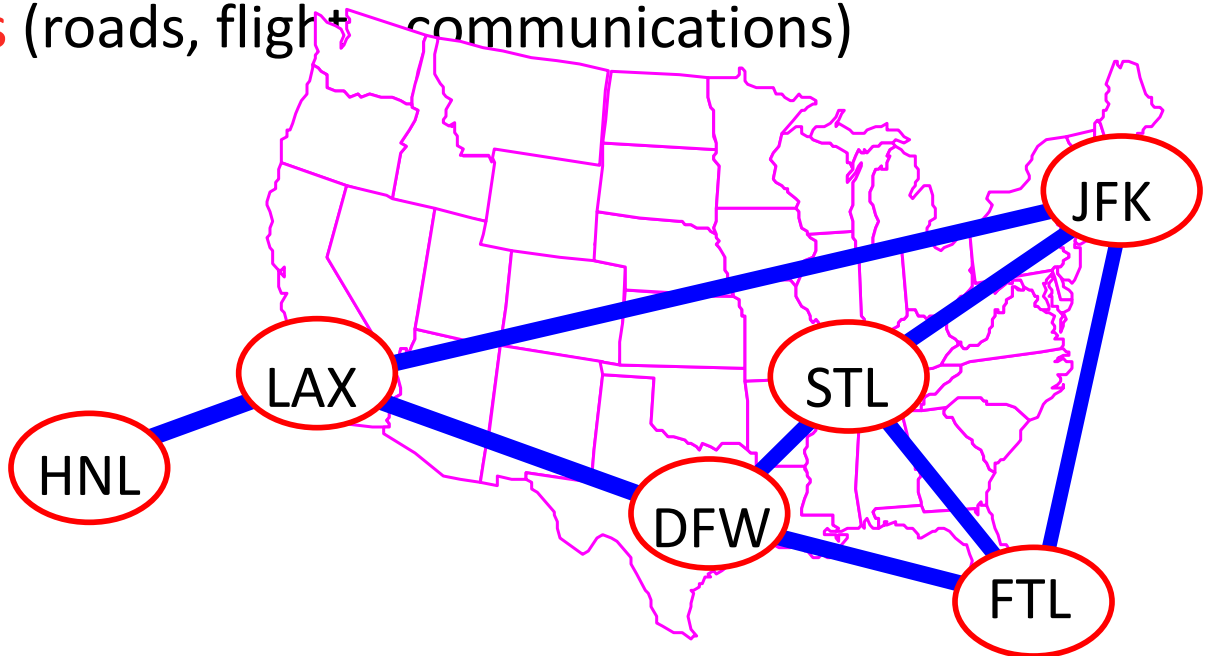
E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Summary - Some Applications

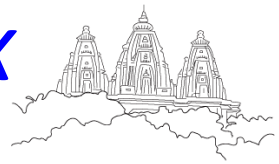
- electronic circuits



- networks (roads, flight, communications)



Terminology: Degree of a Vertex

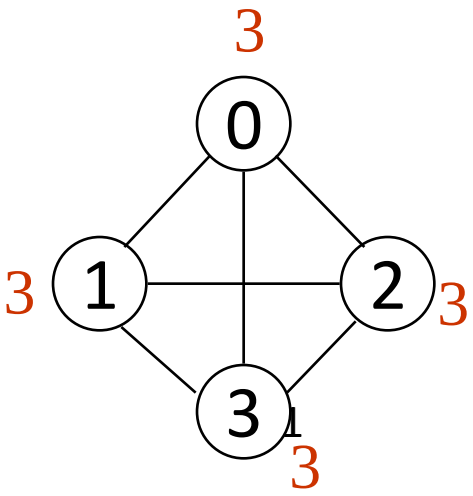


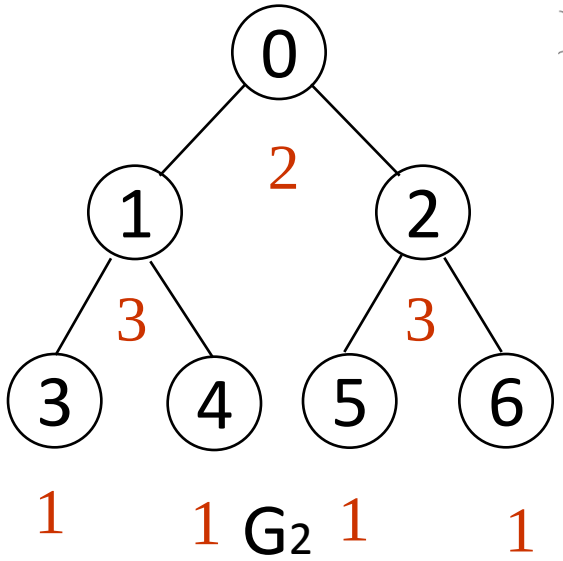
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
  - if di is the degree of a vertex i in a graph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

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

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

Examples



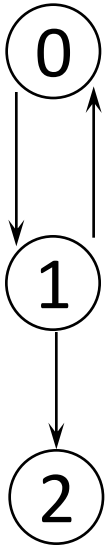


G_2

directed graph

in-degree

out-degree



in:1, out: 1

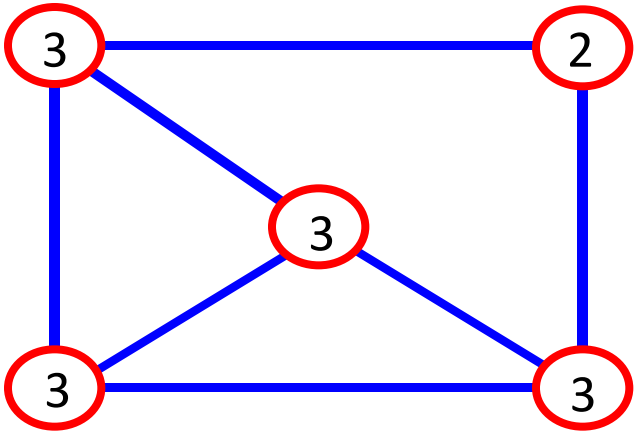
in: 1, out: 2

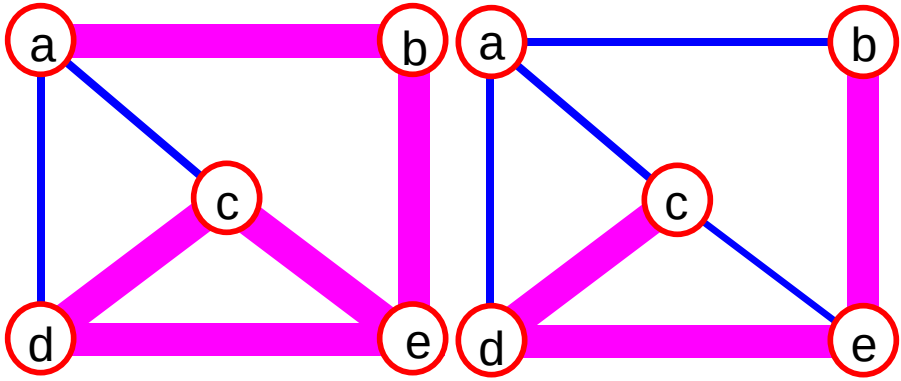
in: 1, out: 0

G_3

Terminology: Path

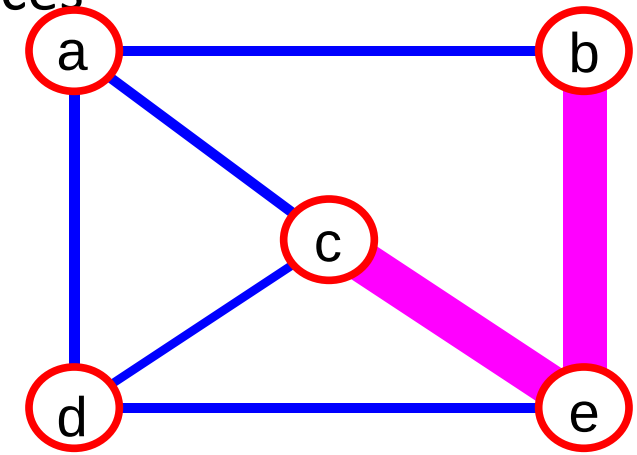
- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.





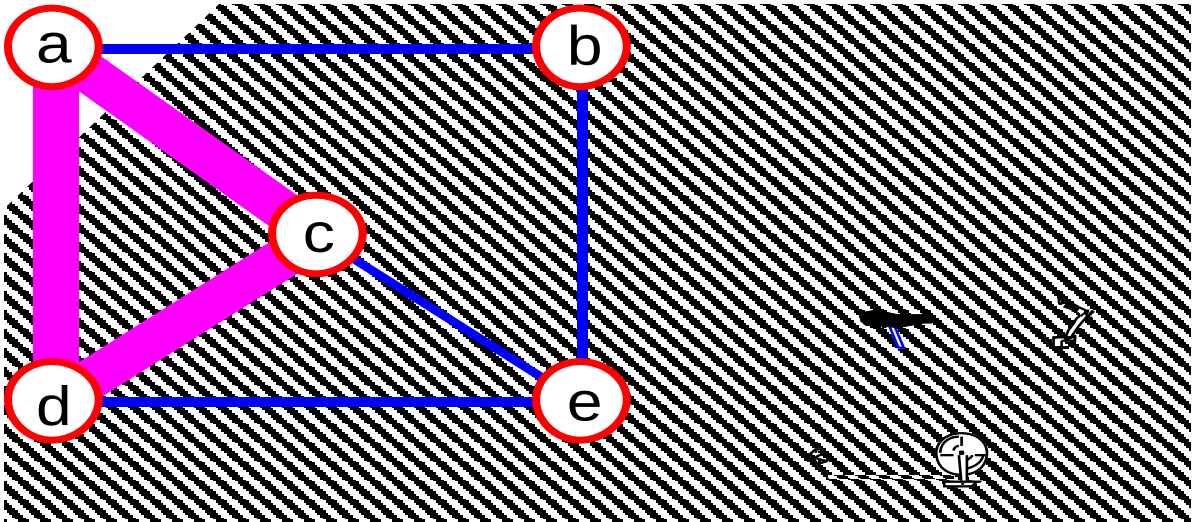
More Terminology

- simple path: no repeated vertices



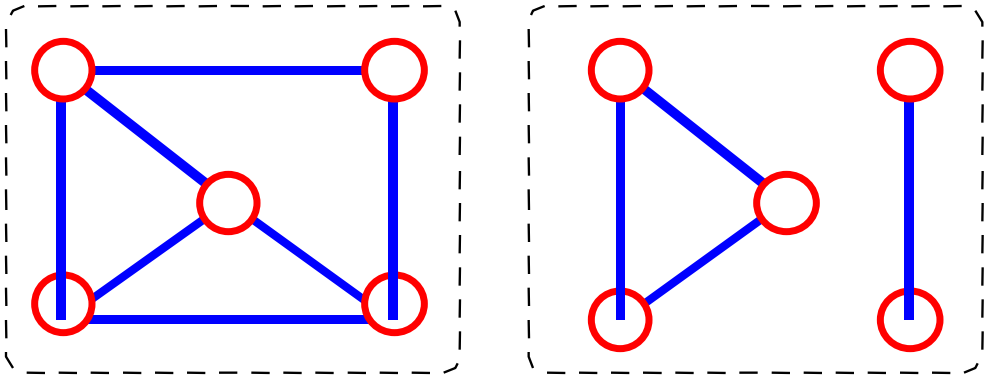
b e c

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



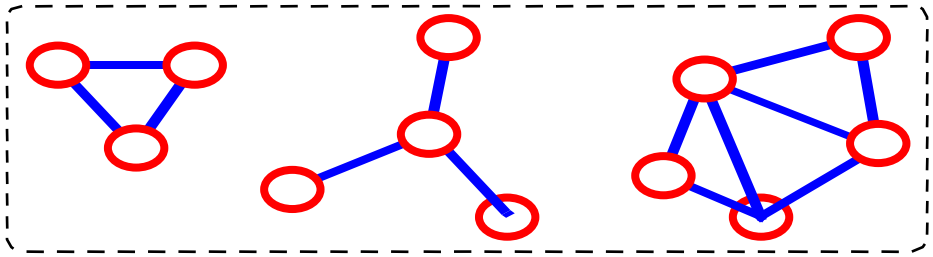
Even More Terminology

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



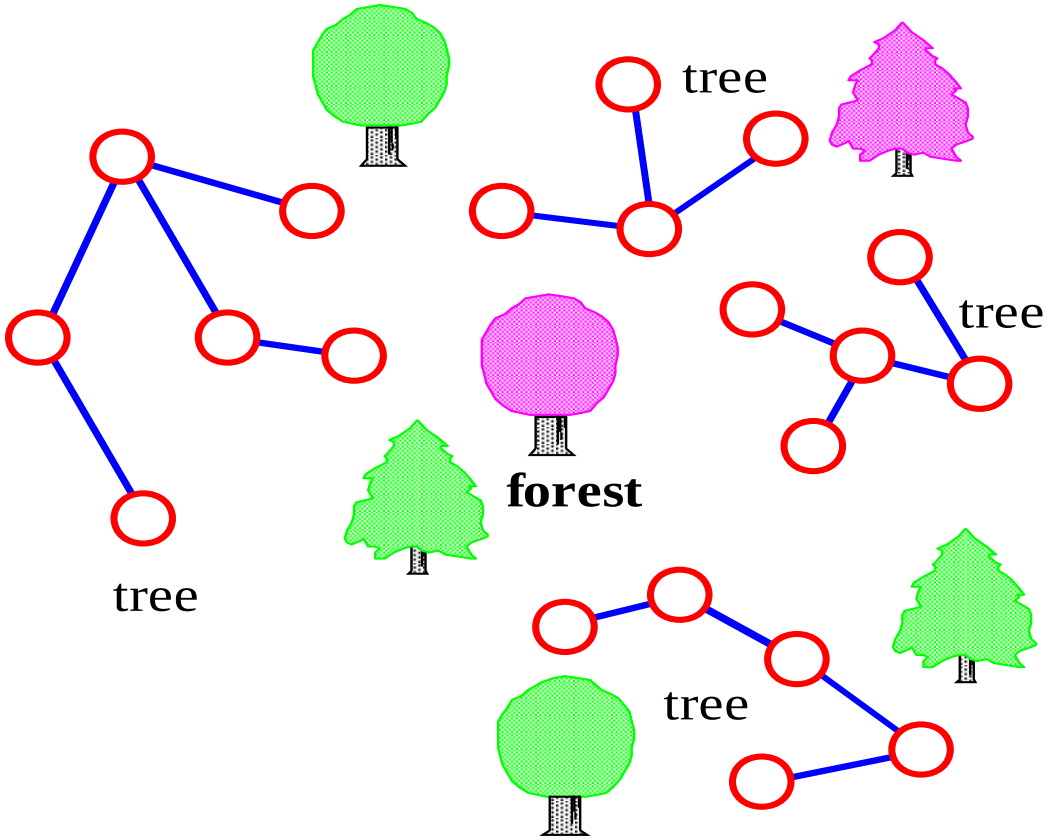
connected not connected

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



More…

- tree - connected graph without cycles
- forest - collection of trees



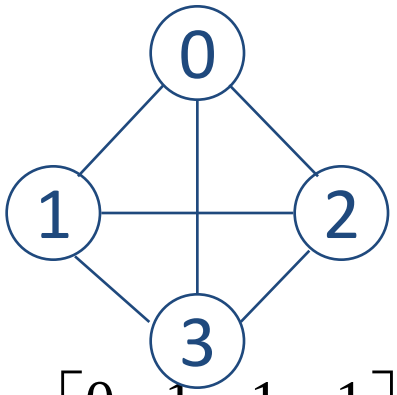
Graph Representations

- Adjacency Matrix
- Adjacency Lists

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

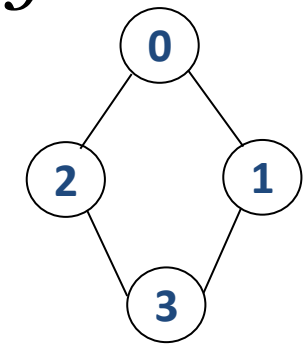
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

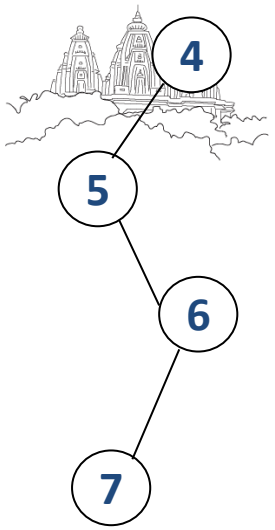
G_2

symmetric

undirected: $n^2/2$

directed: n^2





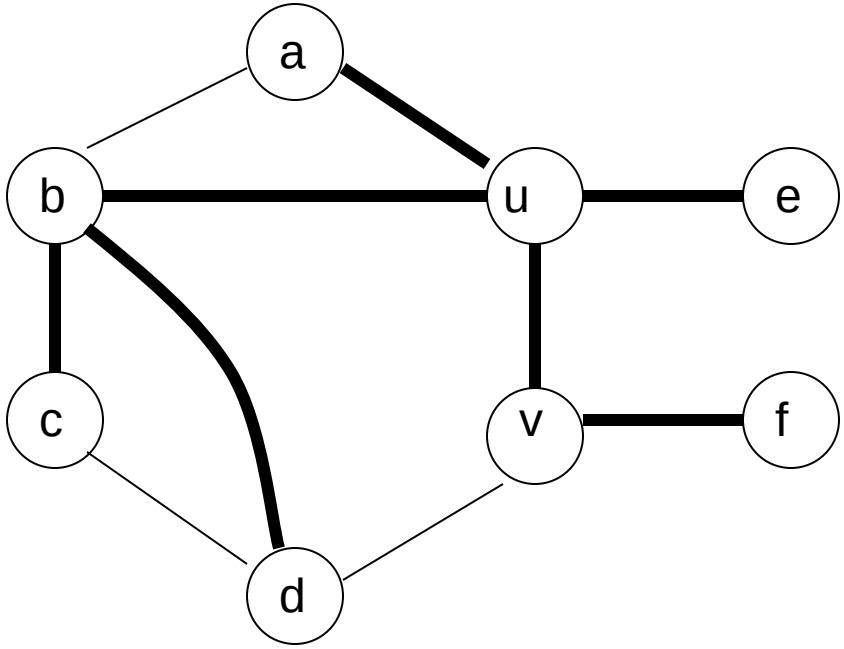
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

SPANNING TREE

What is A Spanning Tree?

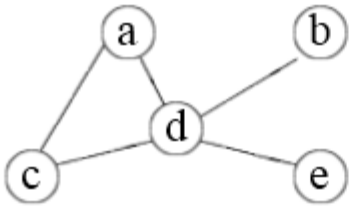
- A *spanning* tree for an undirected graph G=(V,E) is a subgraph of G that is a tree and contains all the vertices of G
- Can a graph have more than one spanning tree?
- Can an unconnected graph have a spanning tree?



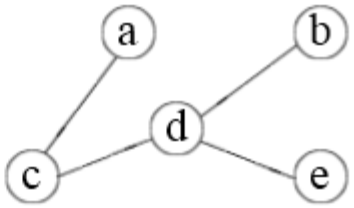
Assignment: Implement Merge-sort algorithm

Example

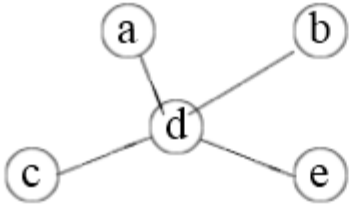
Example:



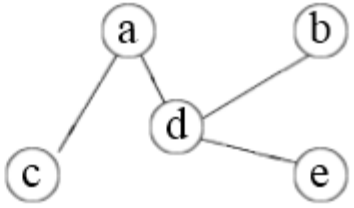
Graph



spanning tree 1



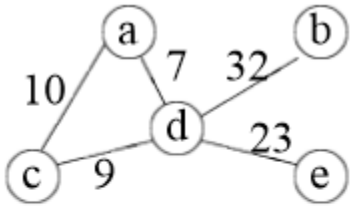
spanning tree 2



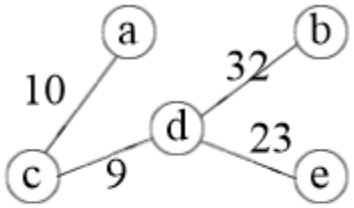
spanning tree 3

- Every connected graph has a spanning tree
- A Weighted graph is a graph in which each Edge has some Weight associated between the connected Vertices
- Weight of a Graph: Sum of the weights of the edges

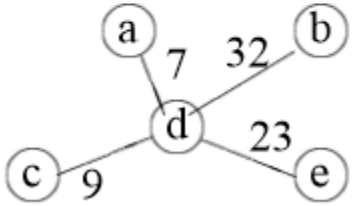
Weights



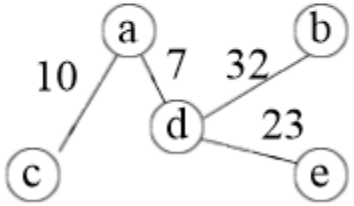
weighted graph



Tree 1. w=74



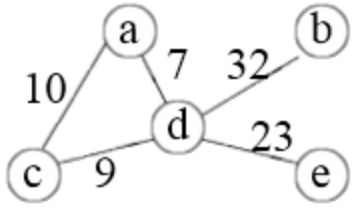
Tree 2, w=71



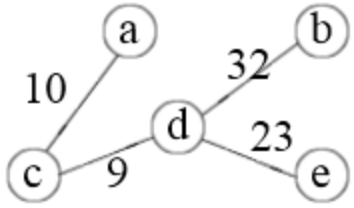
Tree 3, w=72

Minimum Spanning Tree

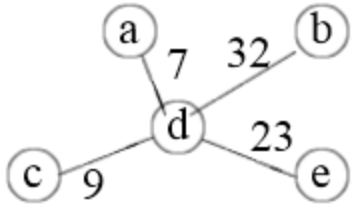
- A MST in an Undirected Connected Weighted graph is a spanning tree of minimum weight



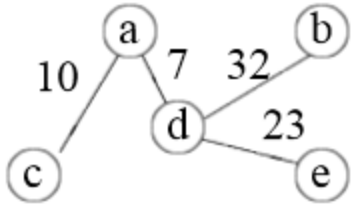
weighted graph



Tree 1. w=74



Tree 2, w=71

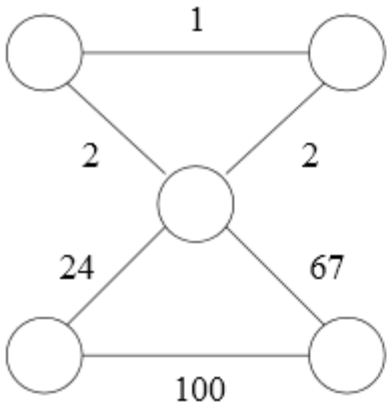


Tree 3, w=72

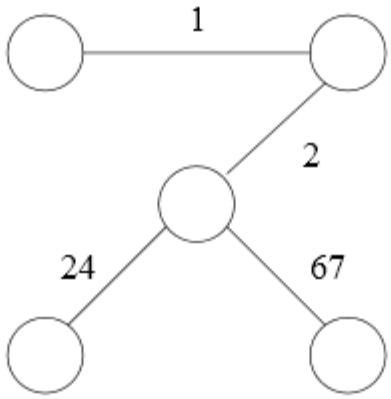
Minimum spanning tree

MST

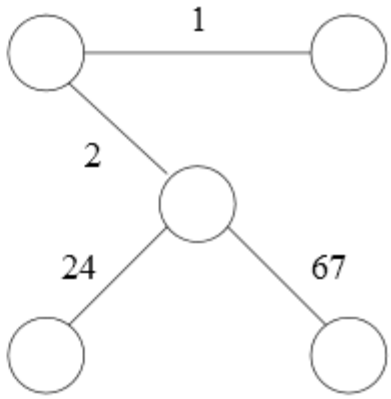
- MST may not be unique



weighted graph



MST1



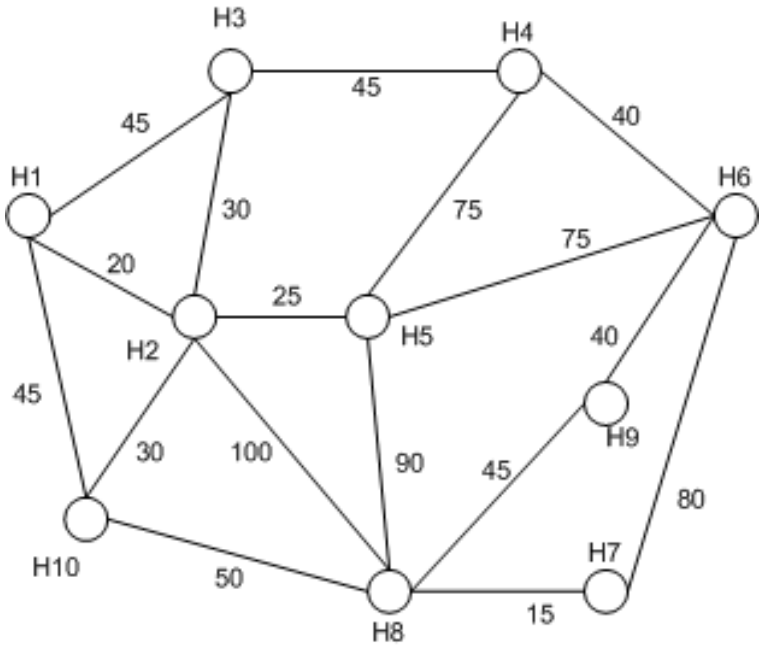
MST2

Minimal Cost Spanning Tree

- The *cost* of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.
- A *minimum-cost spanning tree* is a spanning tree of least cost.
- Three greedy-method algorithms available to obtain a minimum-cost spanning tree of a connected, undirected graph.
  - Kruskal's algorithm
  - Prim's algorithm
  - Sollin's algorithm

Spanning - Example

- Each node is a house, and the edges are the means by which one house can be wired up to another.
- The weights of the edges dictate the distance between the homes.
- Objective is to wire up all ten houses using the least amount of telephone wiring possible.



Example

- A telecommunications company laying cable in a new neighborhood
  - If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths.
  - Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights.
  - A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A *minimum spanning tree* would be one with the lowest total cost.

Spanning - Example

- Several pins of an electronic circuit must be connected using the least amount of wire.

Modeling the Problem

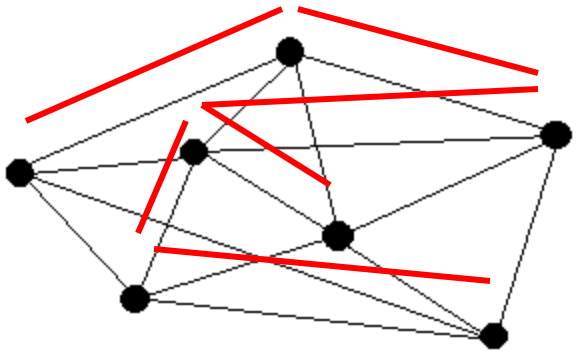
- The graph is a complete, undirected graph $G = (V, E, W)$, where V is the set of pins, E is the set of all possible interconnections between the pairs of pins and $w(e)$ is the length of the wire needed to connect the pair of vertices.
- Find a minimum spanning tree.

Way to solve (finding a MST)

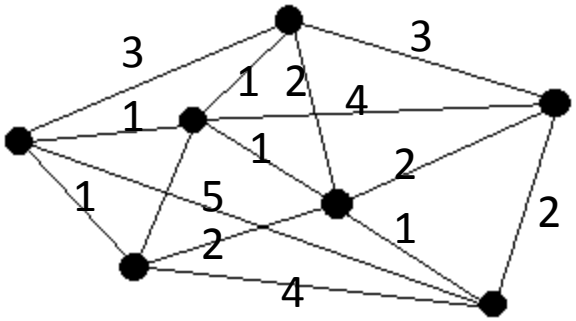
- A tree is an acyclic graph
- Start with an empty graph try to add edges always ensuring it is acyclic

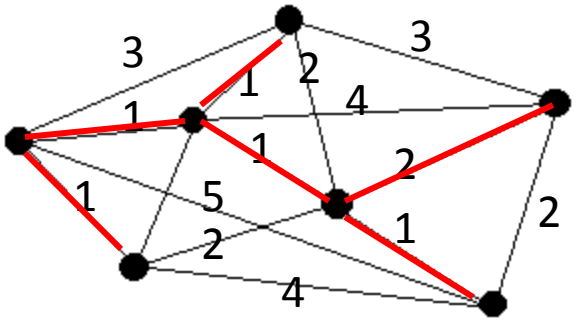
Some definitions

A spanning tree of a graph G is a subgraph of G that is a tree containing all the vertices of G.



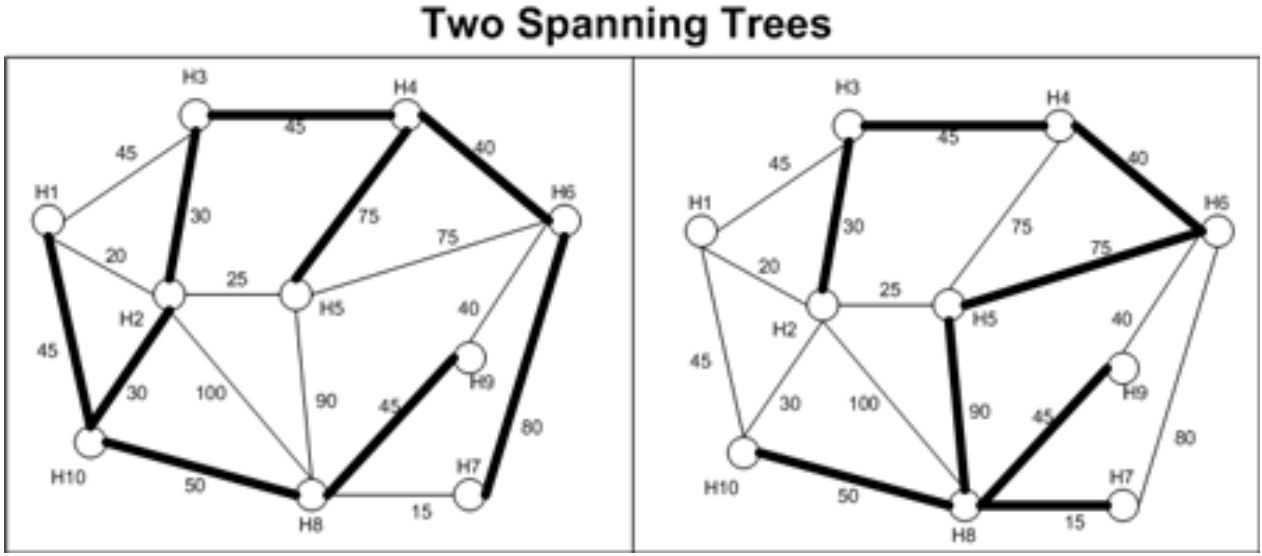
In a weighted graph, a minimum spanning tree is a spanning tree whose sum of edge weights is as small as possible. It is the most economical tree of a graph with weighted edges.





Spanning

- For a connected, undirected graph, there exists some subset of the edges that connect all the nodes and does not introduce a cycle.
  - Such a subset of edges would form a tree (because it would comprise one less edge than vertices and is acyclic), called a *spanning tree*.
  - There are typically many spanning trees for a given graph.



Approach

- There are two basic approaches to solving the minimum spanning tree problem.

Edges with Min Weight

- One approach is build up a spanning tree by choosing the edges with the minimum weight, so long as adding that edge does not create a cycle among the edges chosen thus far

- The other approach builds up the spanning tree by dividing the nodes of the graph into two disjoint sets: the nodes currently in the spanning tree and those nodes not yet added. At each iteration, the least weighted edge that connects the spanning tree nodes to a node not in the spanning tree is added to the spanning tree. To start off the algorithm, some random start node must be selected.

KRUSKALS

Kruskal's Algorithm

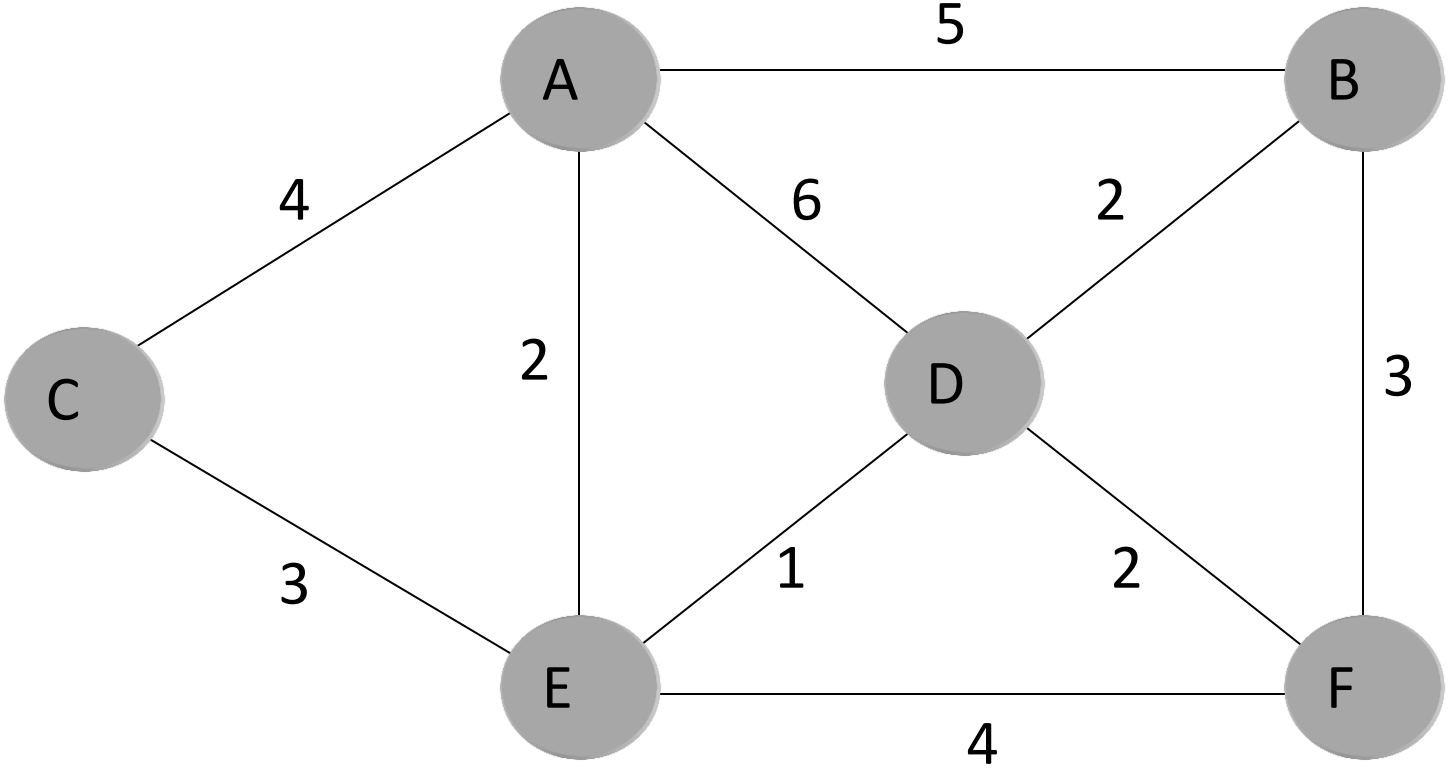
1. Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

3. Continue until *n-1* edges were selected

1.Each vertex is in its own cluster

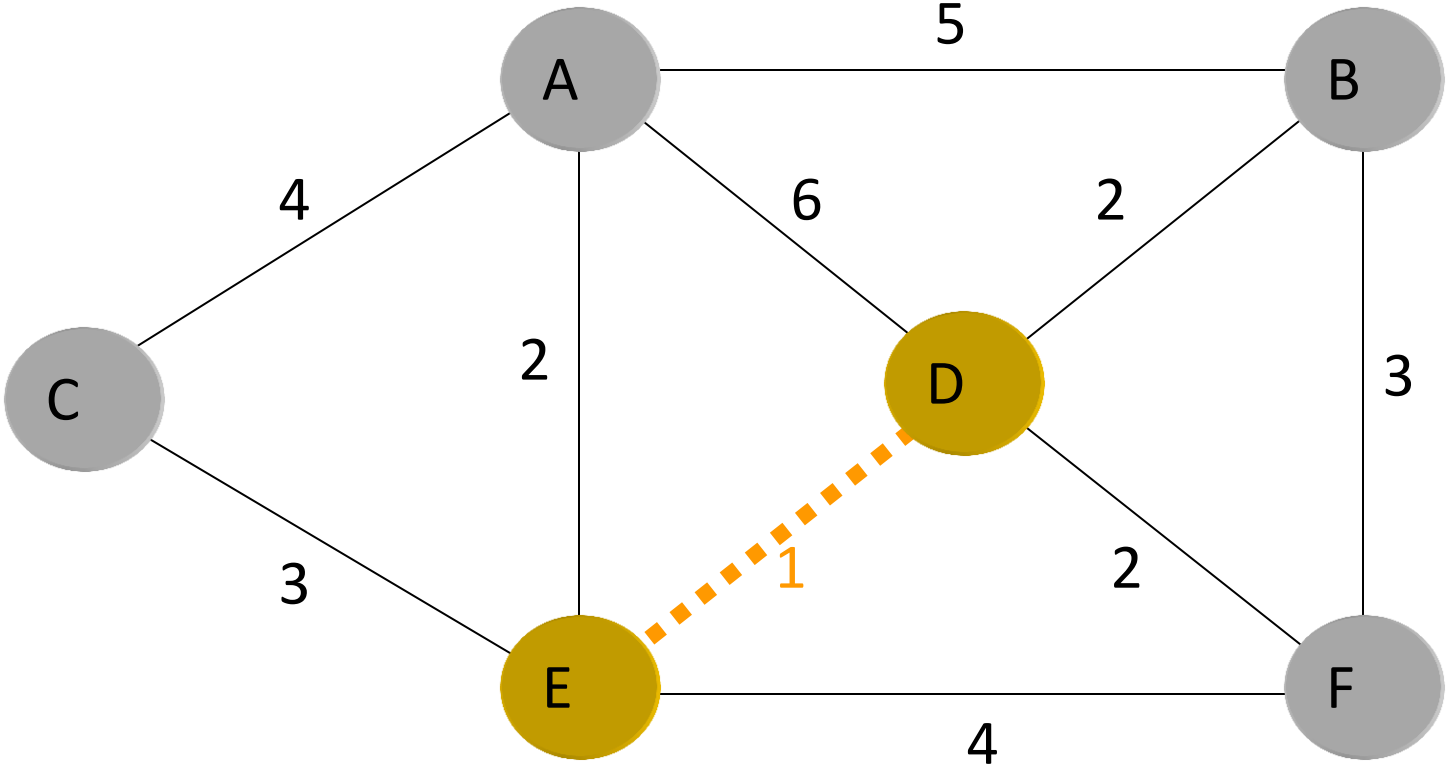
2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it
3. Continue until *n-1* edges were selected



1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

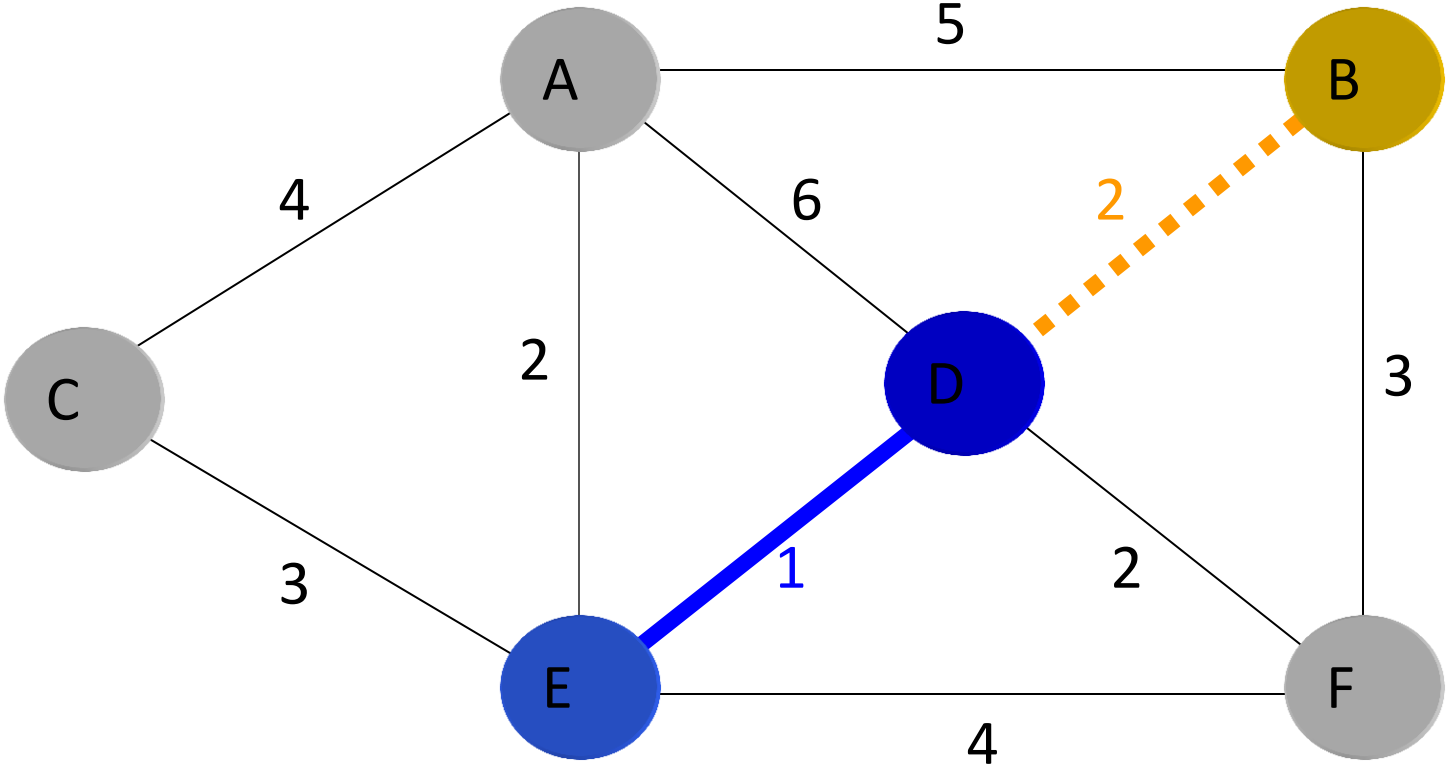
3. Continue until *n-1* edges were selected



1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

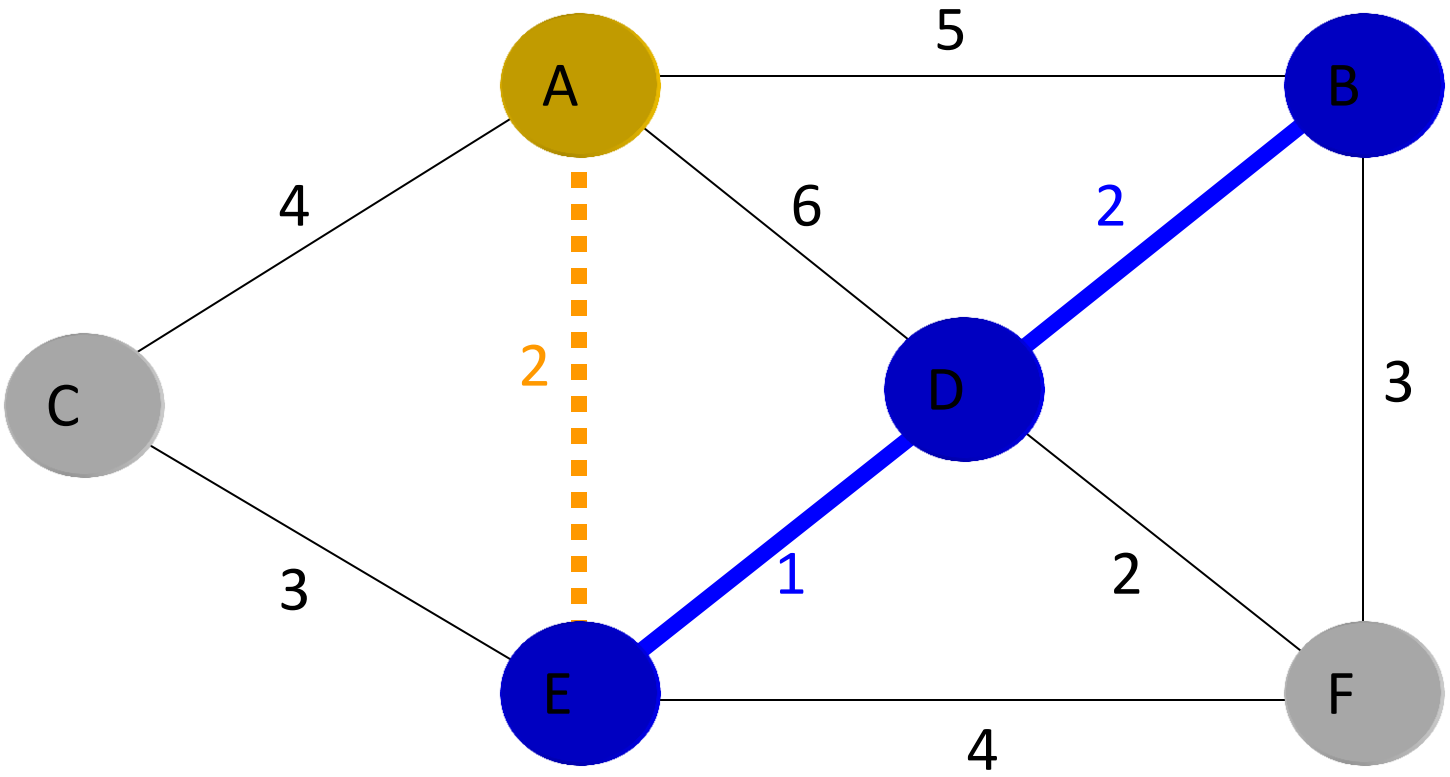
3. Continue until *n-1* edges were selected



1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

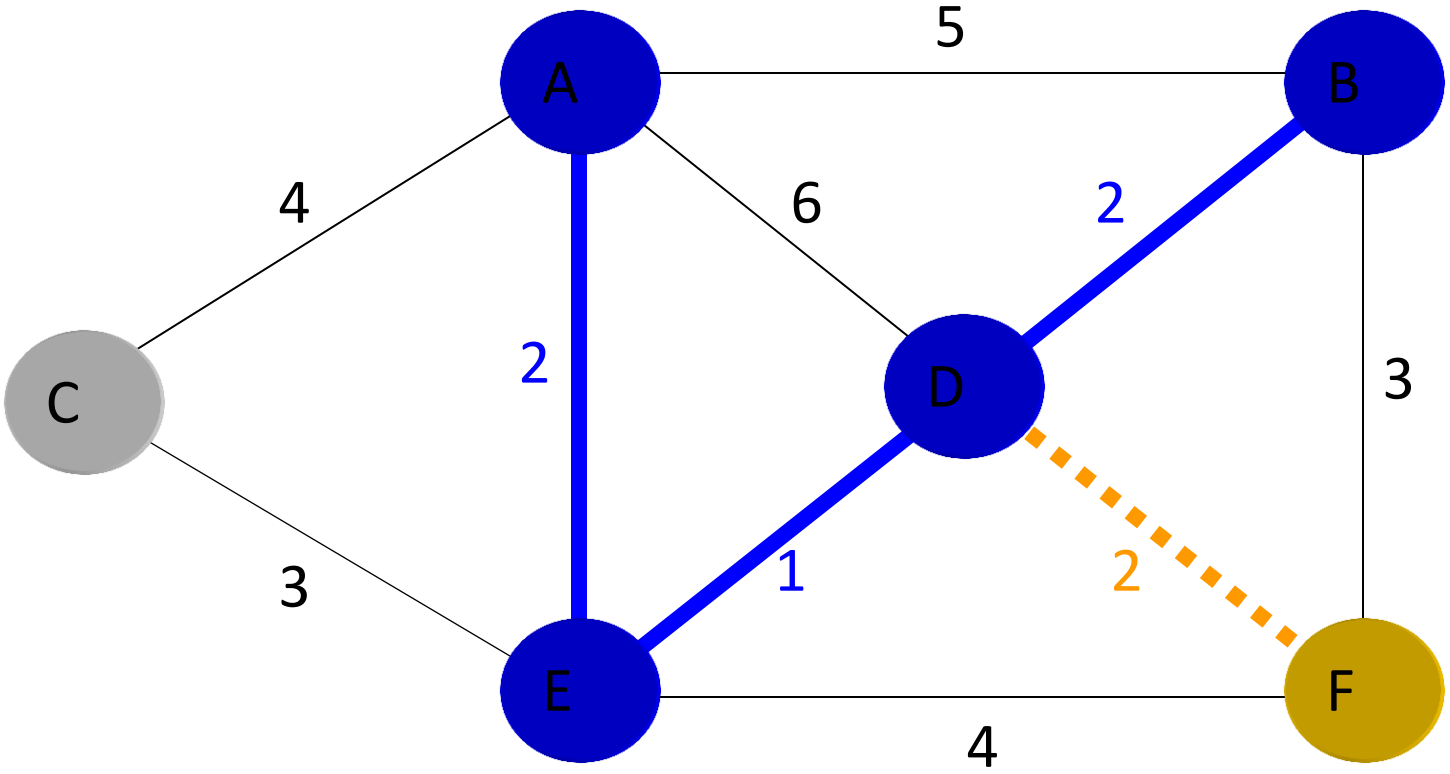
3. Continue until *n-1* edges were selected



1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

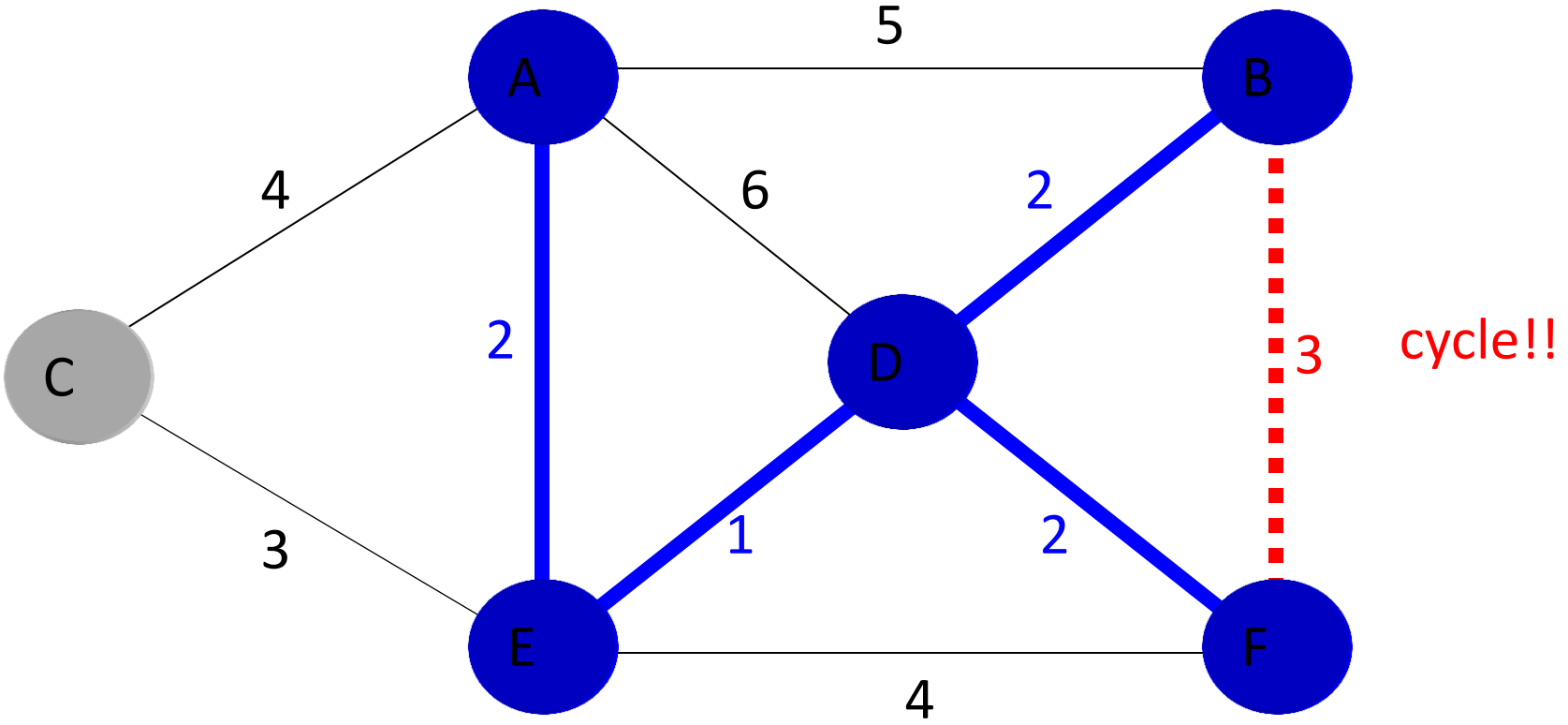
3. Continue until *n-1* edges were selected



1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

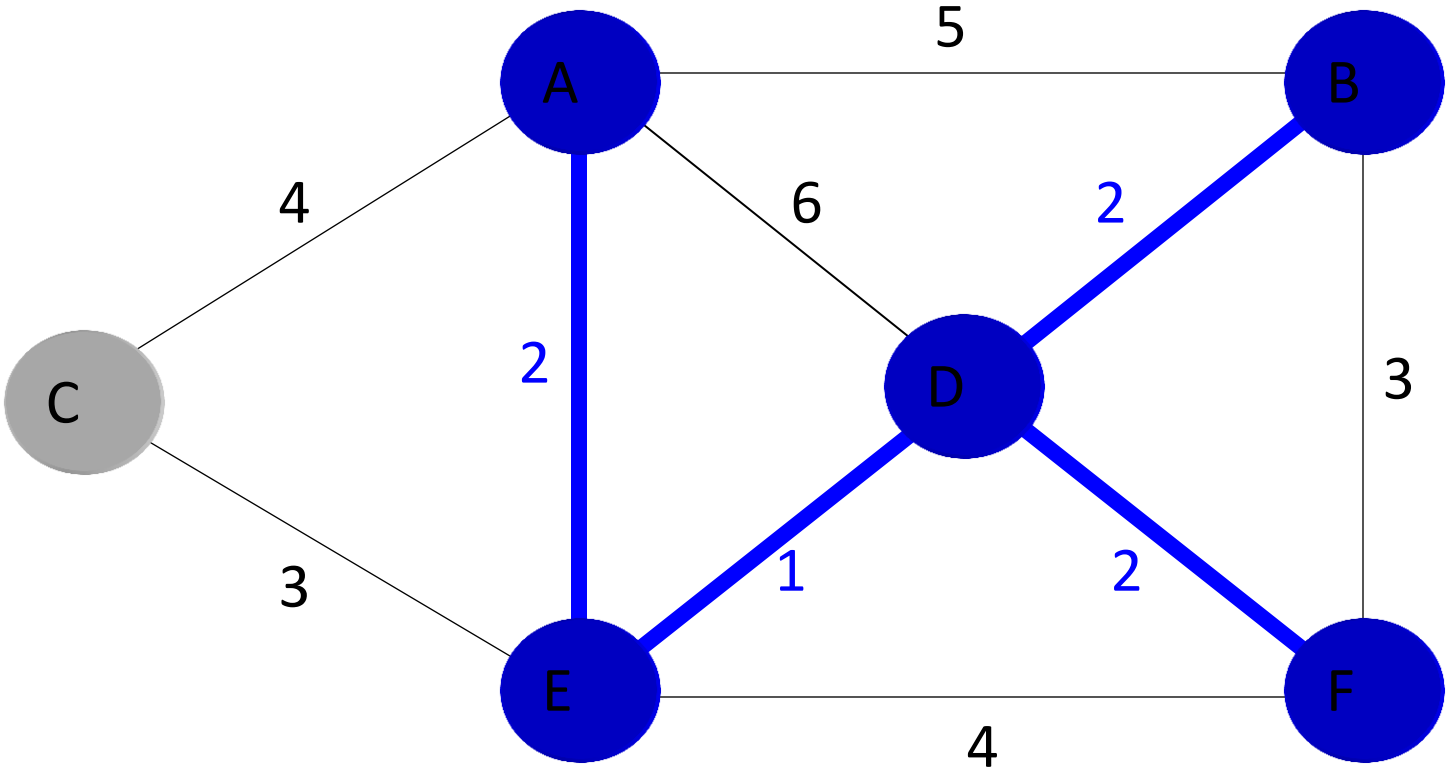
3. Continue until *n-1* edges were selected



1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

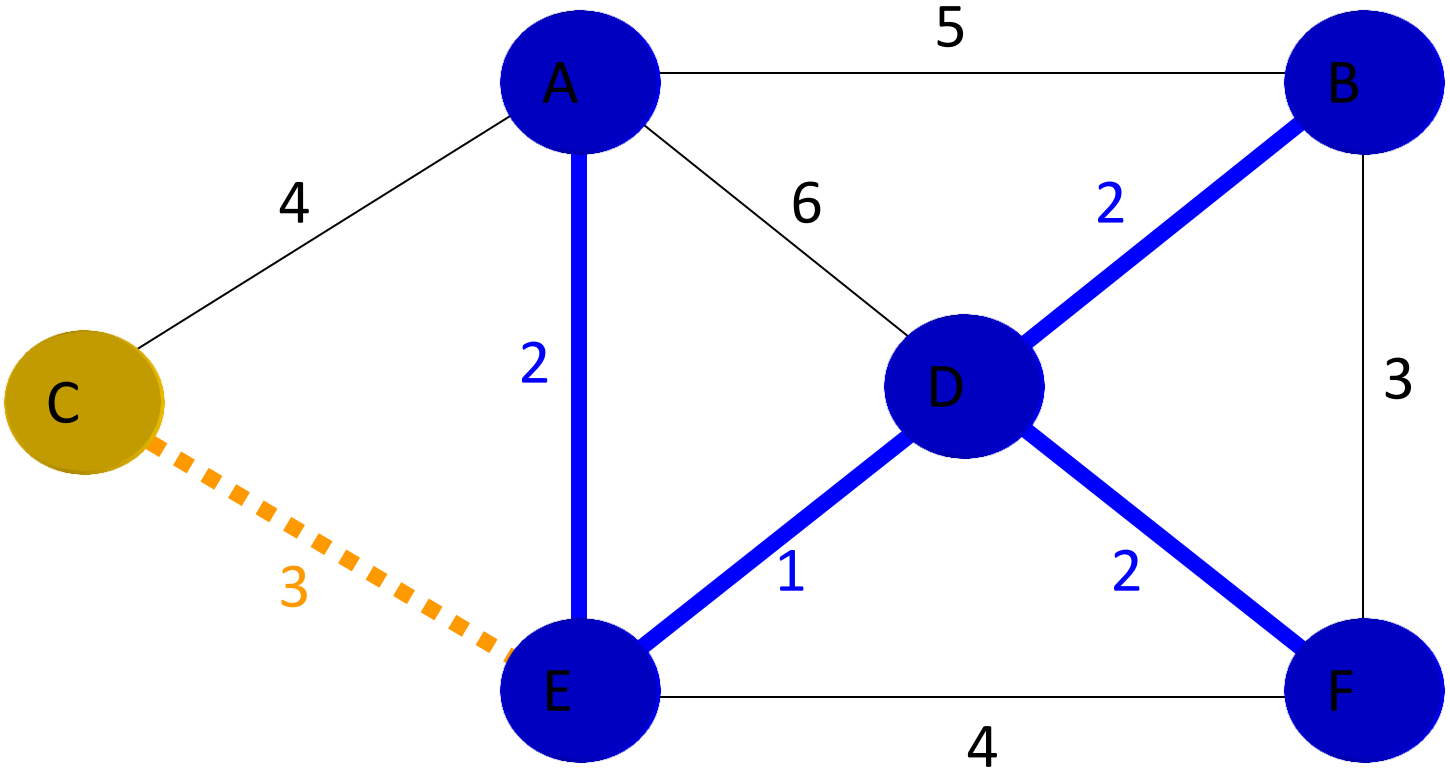
3. Continue until *n-1* edges were selected

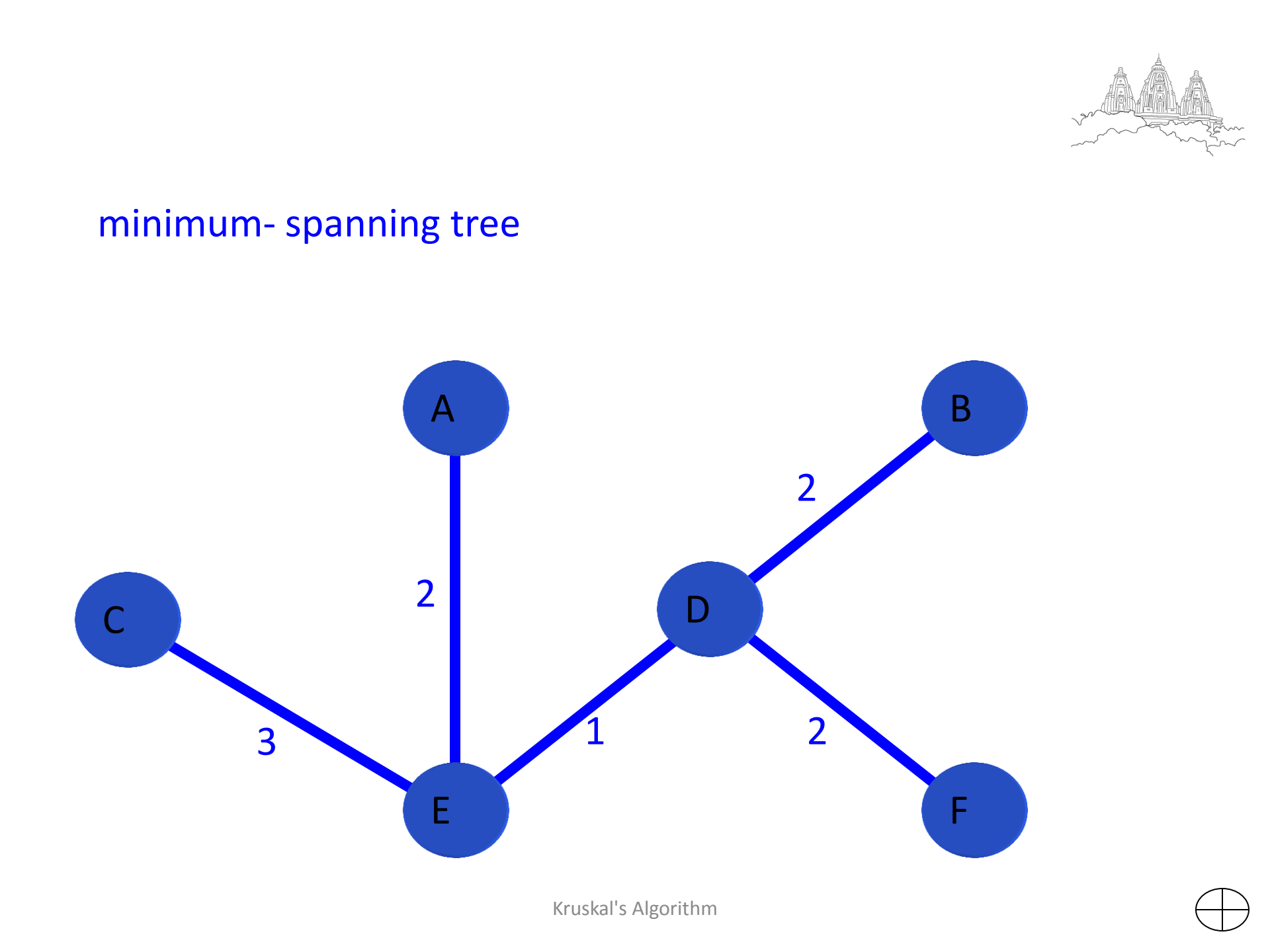


1.Each vertex is in its own cluster

2. Take the edge *e* with the smallest weight
   - if *e* connects two vertices in different clusters, then *e* is added to the MST and the two clusters, which are connected by *e*, are merged into a single cluster
   - if *e* connects two vertices, which are already in the same cluster, ignore it

3. Continue until *n-1* edges were selected





minimum- spanning tree

The correctness of Kruskal's Algorithm

⇨ Crucial Fact about MSTs

Running time: $O(m \log n)$

By implementing queue Q as a heap, Q could be initialized in $O(m)$ time and a vertex could be extracted in each iteration in $O(\log n)$ time

Steps - summary

- scan all edges in increasing weight order; if an edge is safe, keep it (i.e. add it to the set A).
  - **1.** Sort all the edges in non-decreasing order of their weight.
  - **2.** Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
  - **3.** Repeat step#2 until there are (V-1) edges in the spanning tree.

Code Fragment

Input: A weighted connected graph G with n vertices and m edges
Output: A minimum-spanning tree T for G

```
for each vertex v in G do
    Define a cluster C(v) = {v}.
Initialize a priority queue Q to contain all edges in G, using weights as keys.
T = NULL
while Q != NULL do
        Extract (and remove) from Q an edge (v,u) with smallest weight.
        Let C(v) be the cluster containing v, and let C(u) be the cluster containing u.
        if C(v) != C(u) then
            Add edge (v,u) to T.
            Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).
return tree T
```

Kruskals

- Starts with a “FOREST” which consists of “TREES”
- Each TREE is only one node
- In Every step, two different trees are connected to a bigger tree
- Until we end up in a tree which is the minimum genetic tree

- *How many edges does a minimum spanning tree has?*
  - (V – 1) edges where V is the number of vertices in the graph

Prim‘s Algorithm

1. All vertices are marked as not visited

2. Any vertex *v* you like is chosen as starting vertex and is marked as visited (define a cluster *C*)

3. The smallest- weighted edge *e = (v,u)*, which connects one vertex *v* inside the cluster *C* with another vertex *u* outside of *C*, is chosen and is added to the MST.

4. The process is repeated until a spanning tree is formed

Prim's Algo

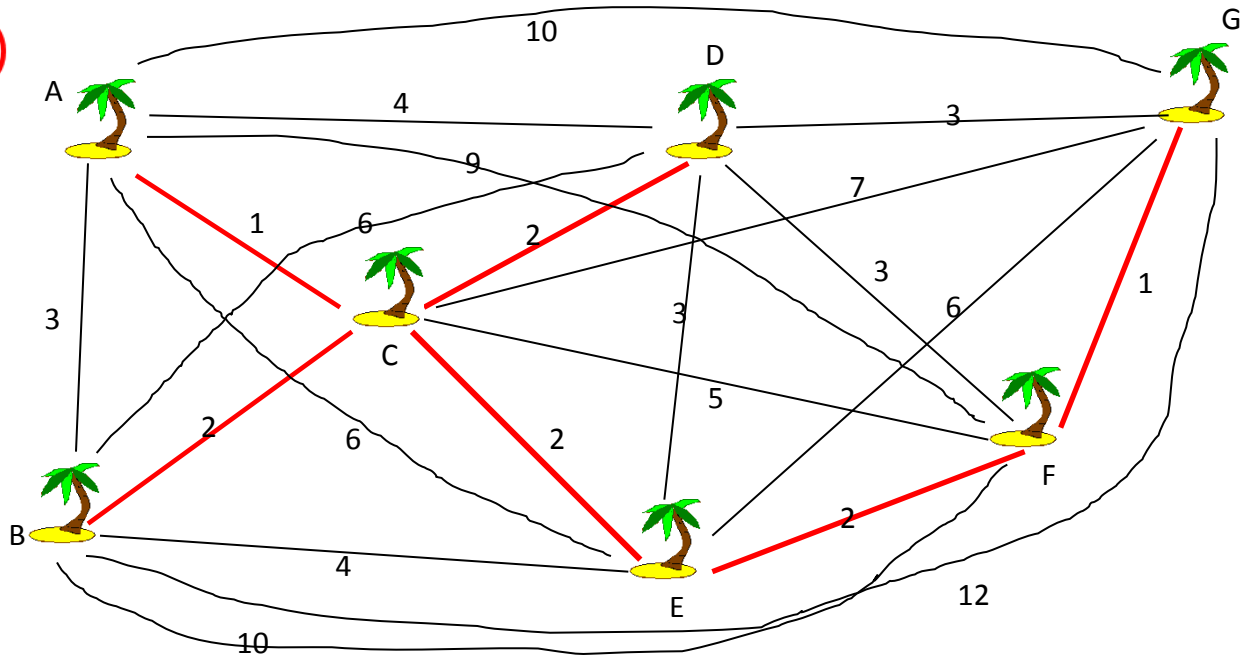
- Determines a minimal spanning tree in a connected graph

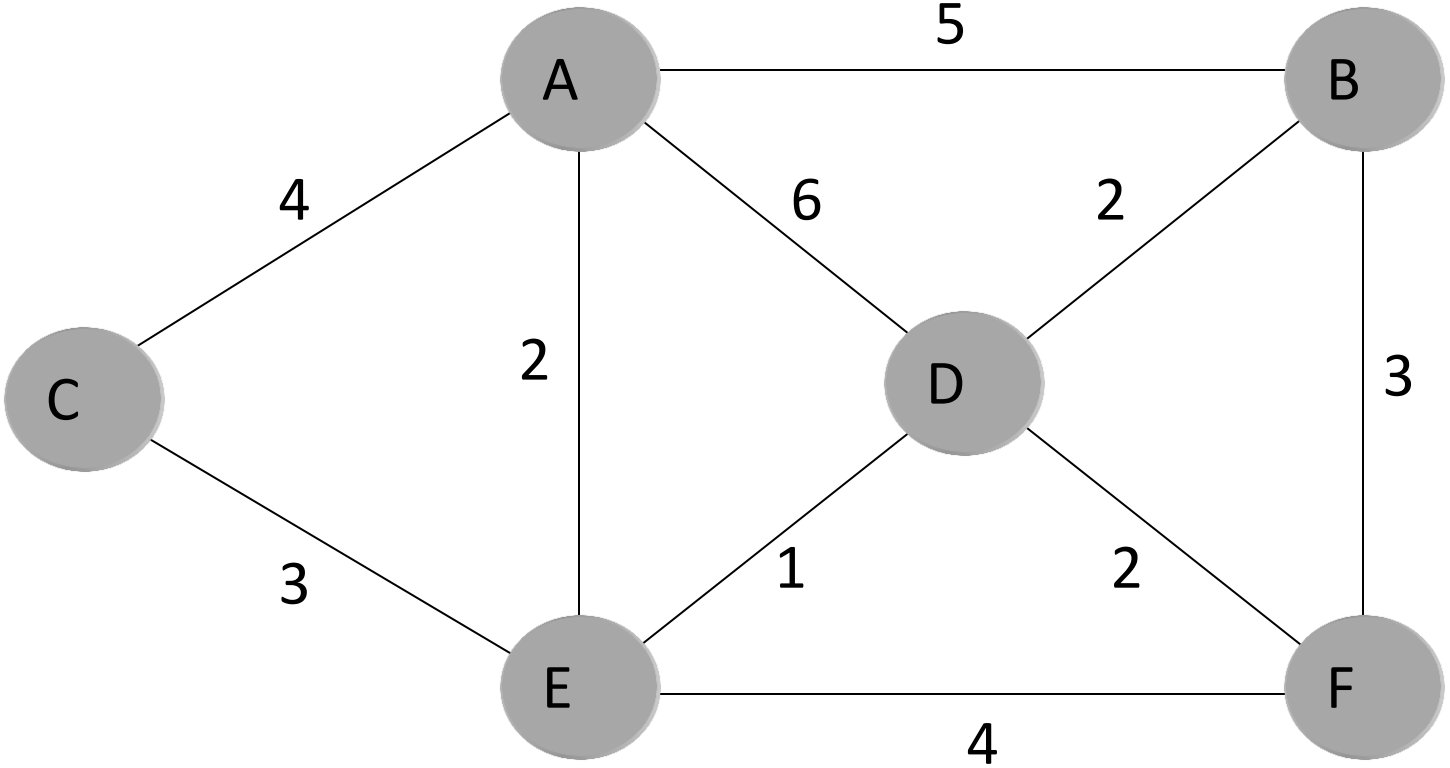
Prim's

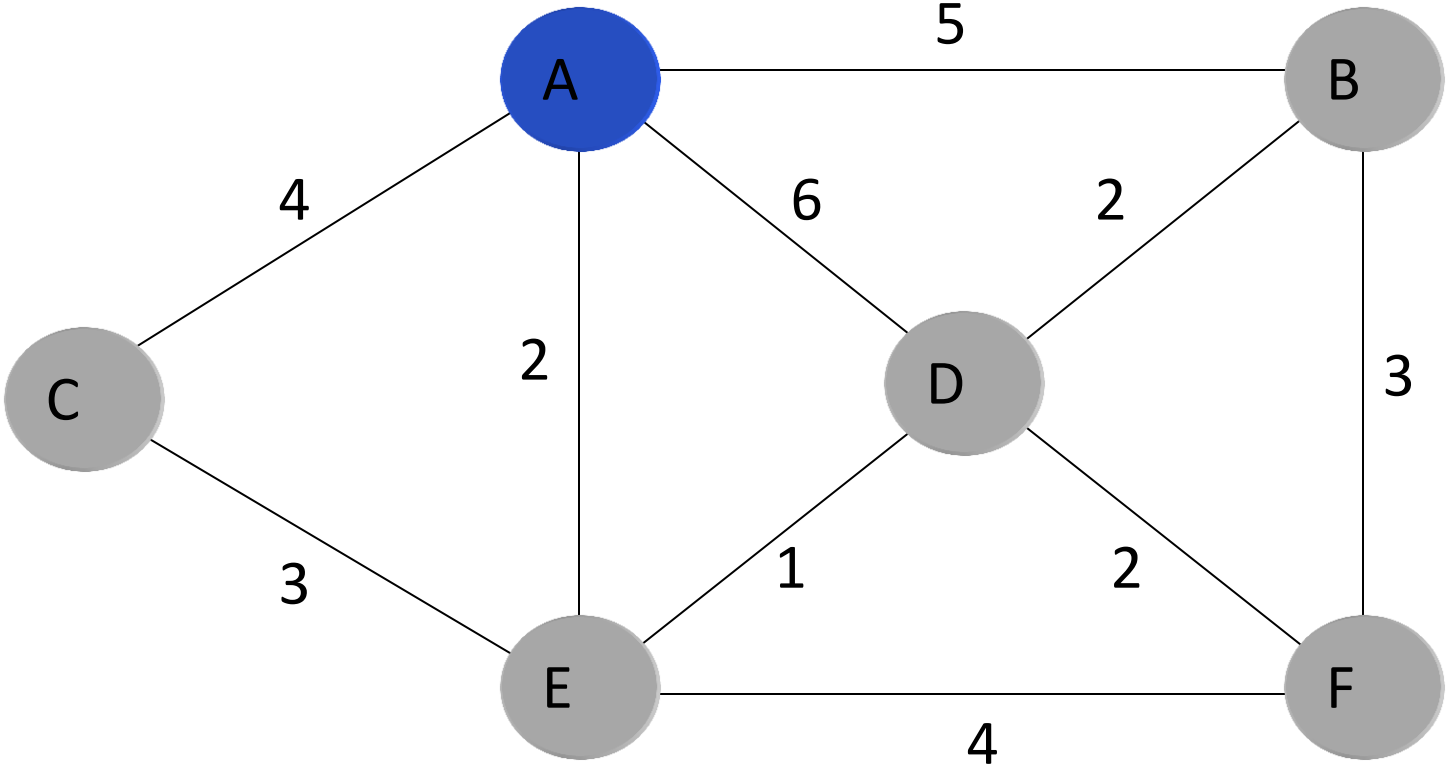
	A	B	C	D	E	F	G
A	*	3	1	4	6	9	10
B	3	*	2	6	4	10	12
C	1	2	*	2	2	5	7
D	4	6	2	*	3	3	3
E	6	4	2	3	*	2	6
F	9	10	5	3	2	*	1
G	10	12	7	3	6	1	*

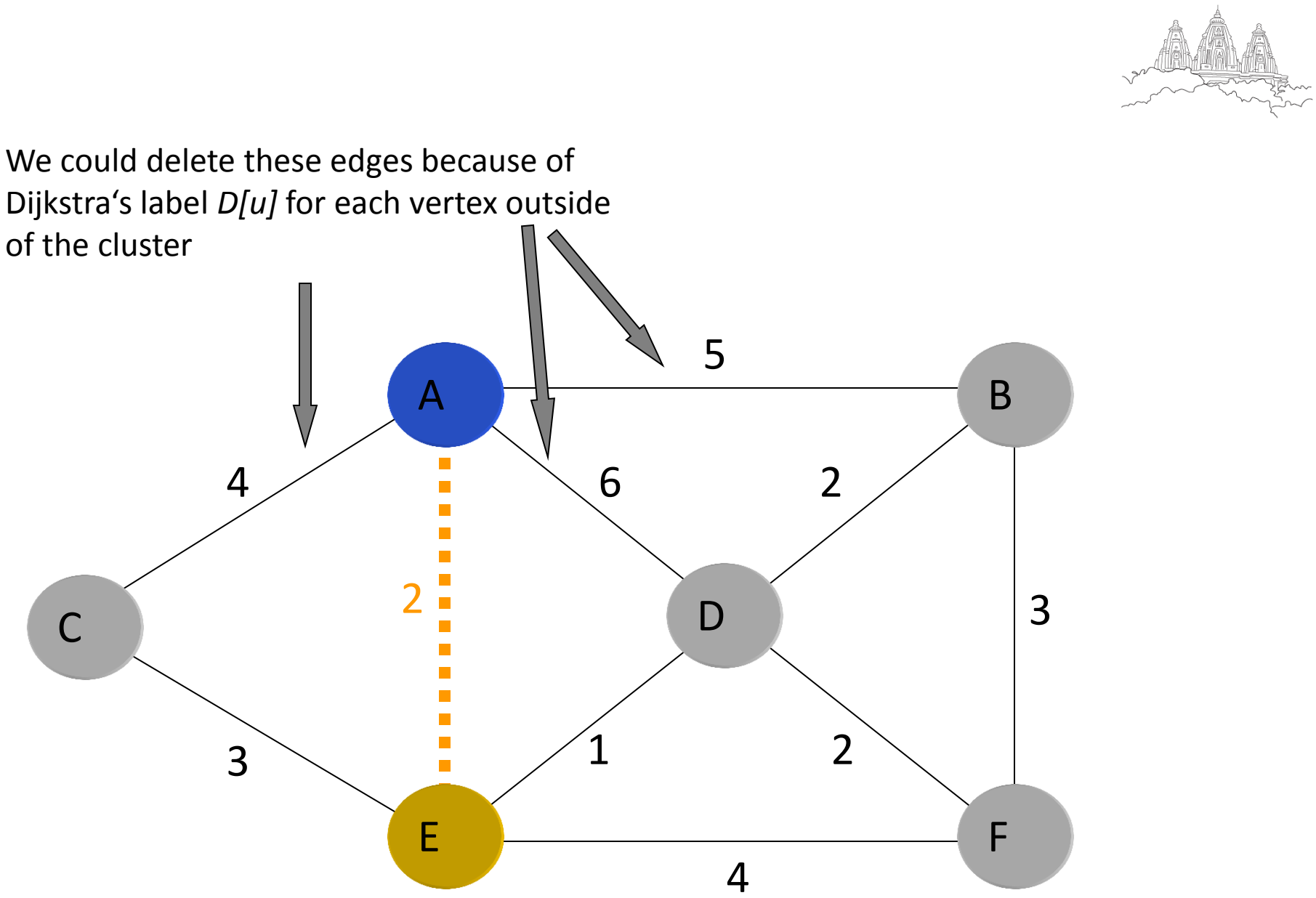
- Prim's algorithm states: Setting n= to the number of vertices, repeat the following step until the tree T has n-1 edges: Add to T the shortest edge between a vertex in T and a vertex not in T (initially pick any edge of the shortest length).

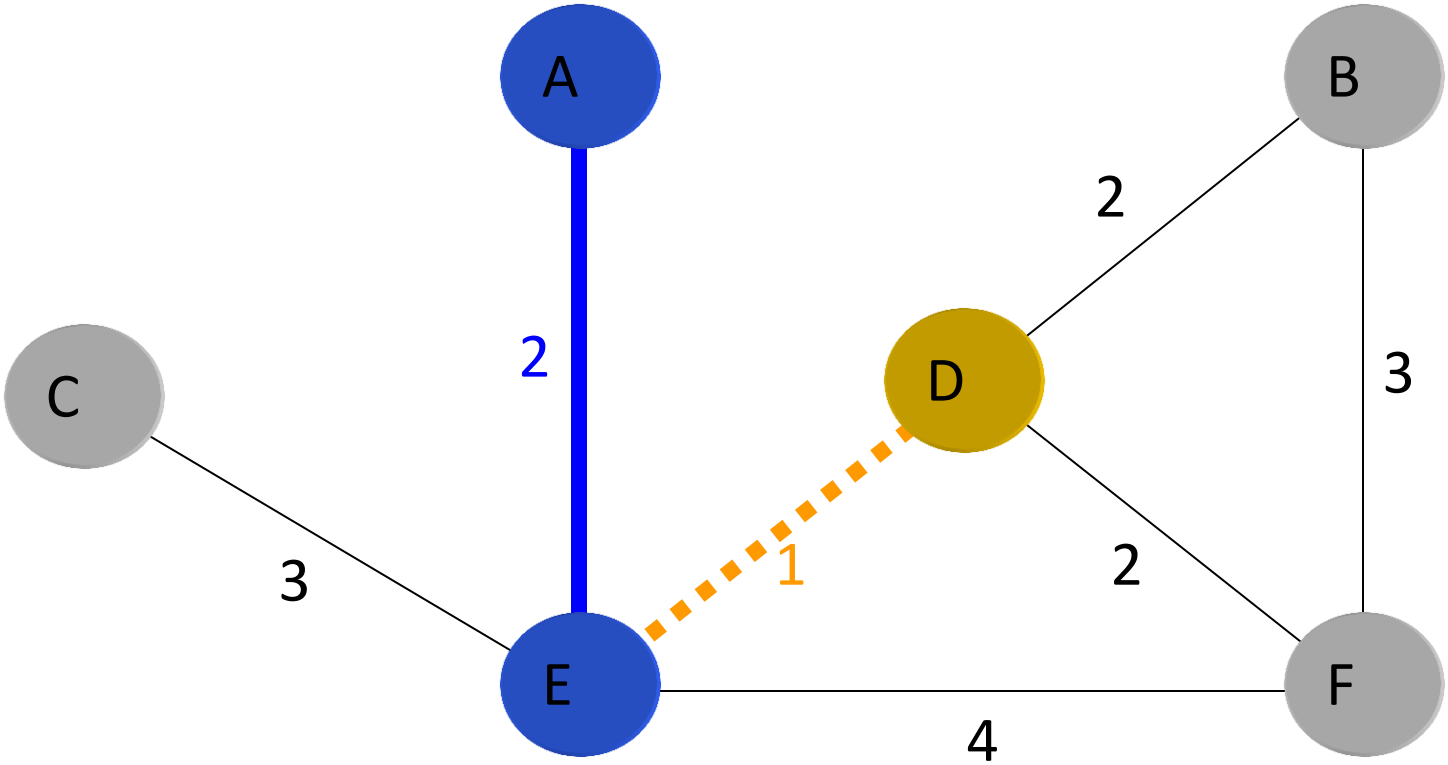
Here n = 7; T will have 6 edges

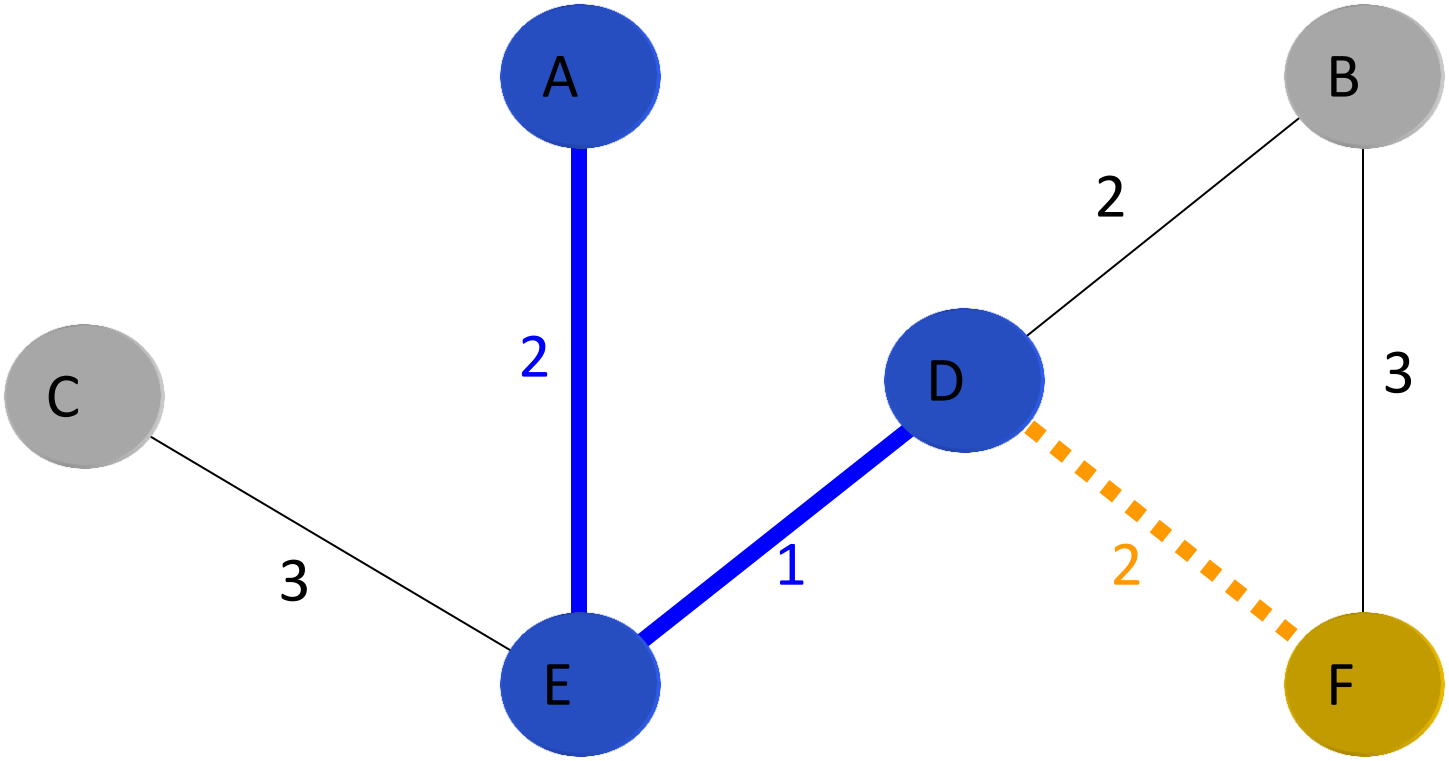


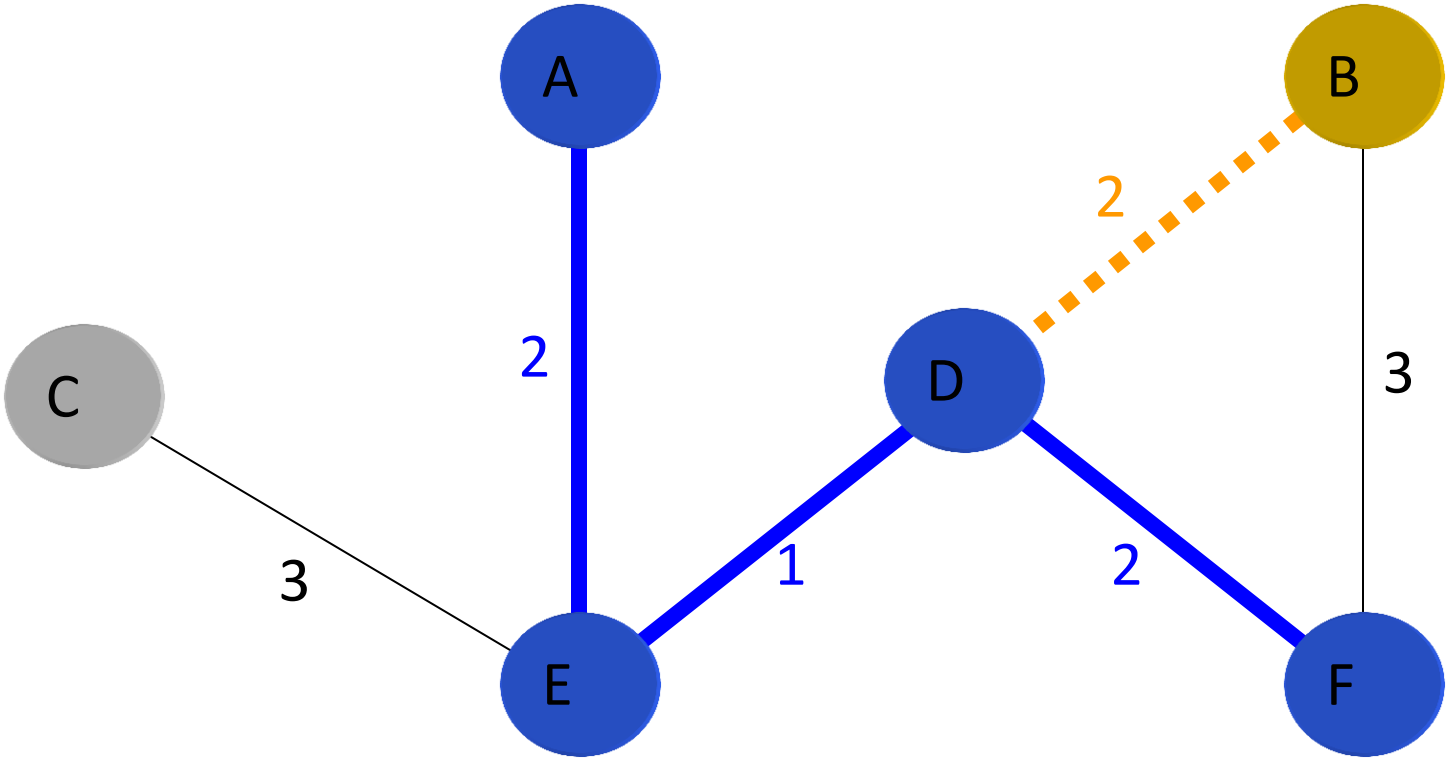

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

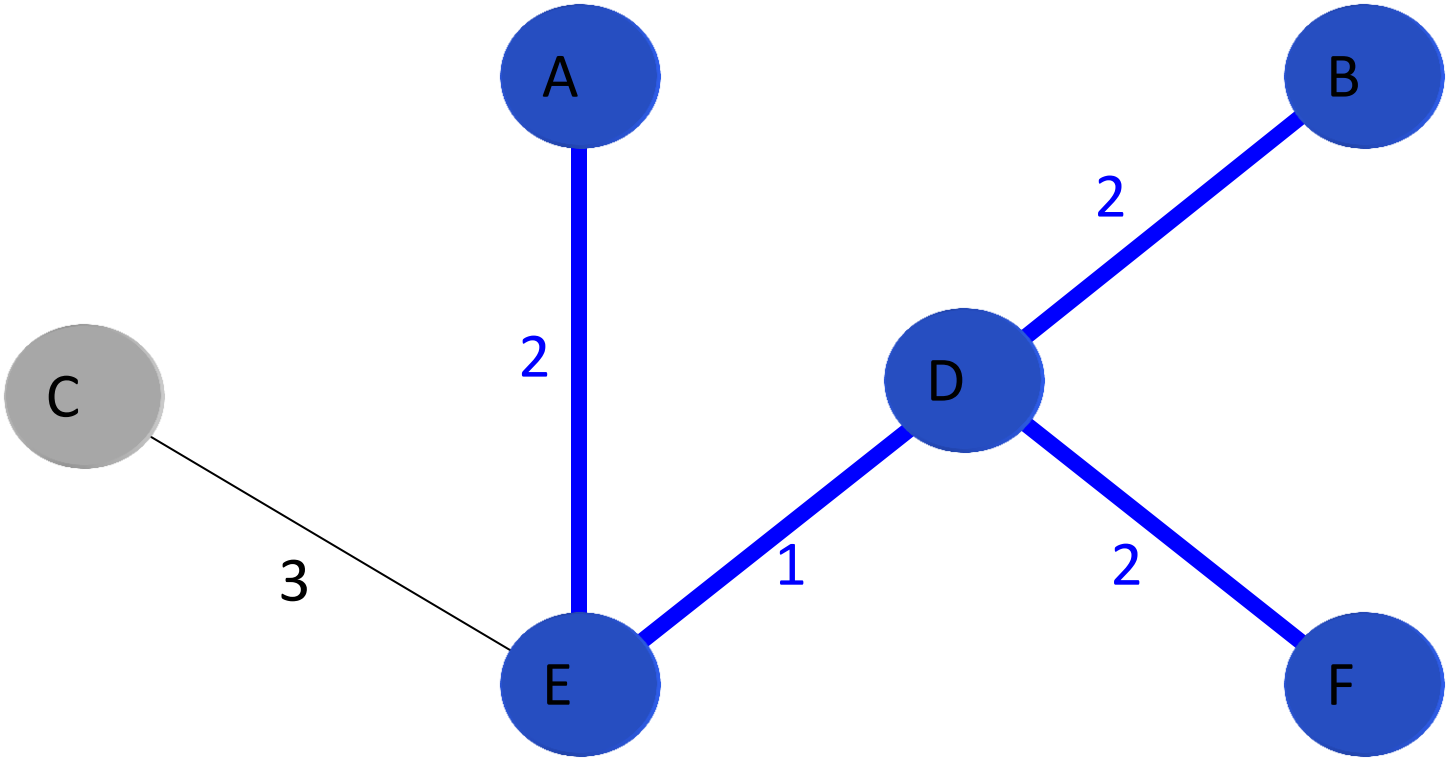

A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

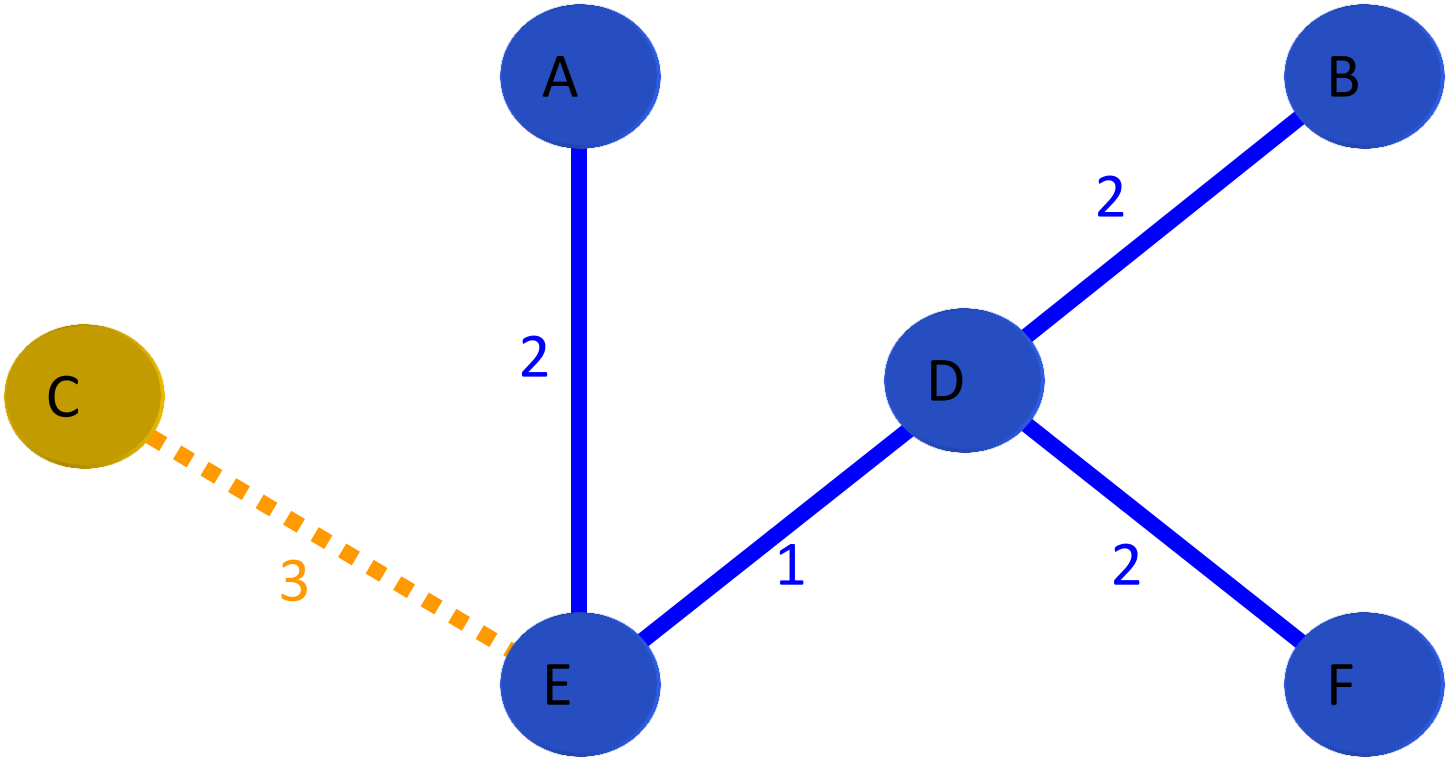

We could delete these edges because of Dijkstra‘s label *D[u]* for each vertex outside of the cluster
A
B
C
D
E
F
5
4
6
2
2
3
3
1
2
4

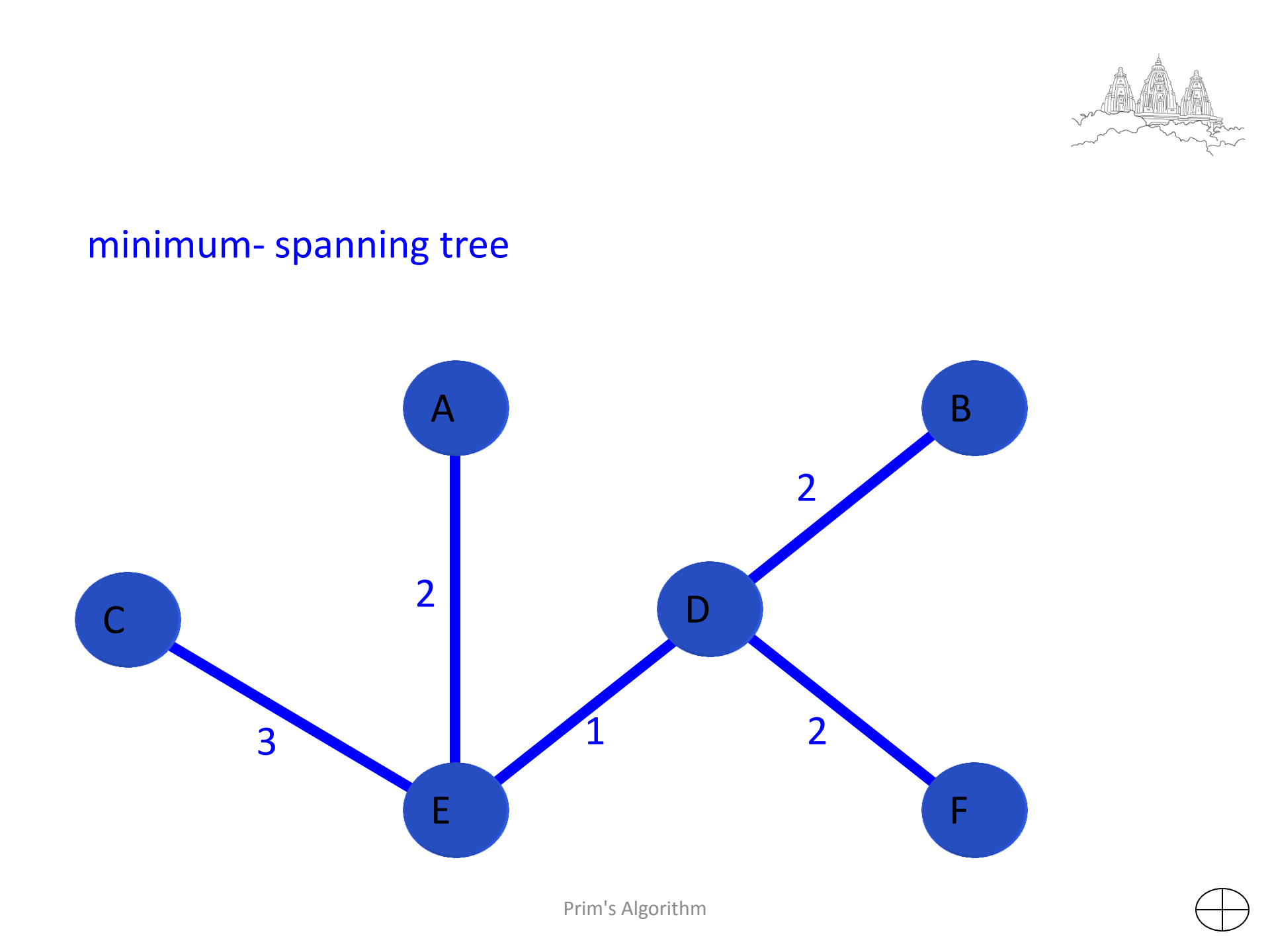

A
B
C
D
E
F
2
2
3
3
1
2
4


A
B
C
D
E
F
2
2
3
3
1
2

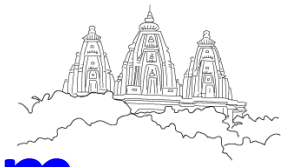

A
B
C
D
E
F
2
2
3
3
1
2


A
B
C
D
E
F
2
2
3
1
2


A
B
C
D
E
F
2
2
3
1
2



minimum- spanning tree



The correctness of Prim‘s Algorithm

⇨ Crucial Fact about MSTs

Running time: *O (m log n)*

By implementing queue *Q* as a heap, *Q* could be initialized in *O (m)* time and a vertex could be extracted in each iteration in *O (log n)* time

Another example - Prim's

- Approach:
  - Choose an arbitrary start node v
  - At any point in time, we have connected component N containing v and other nodes V-N
  - Choose the minimum weight edge from N to V-N

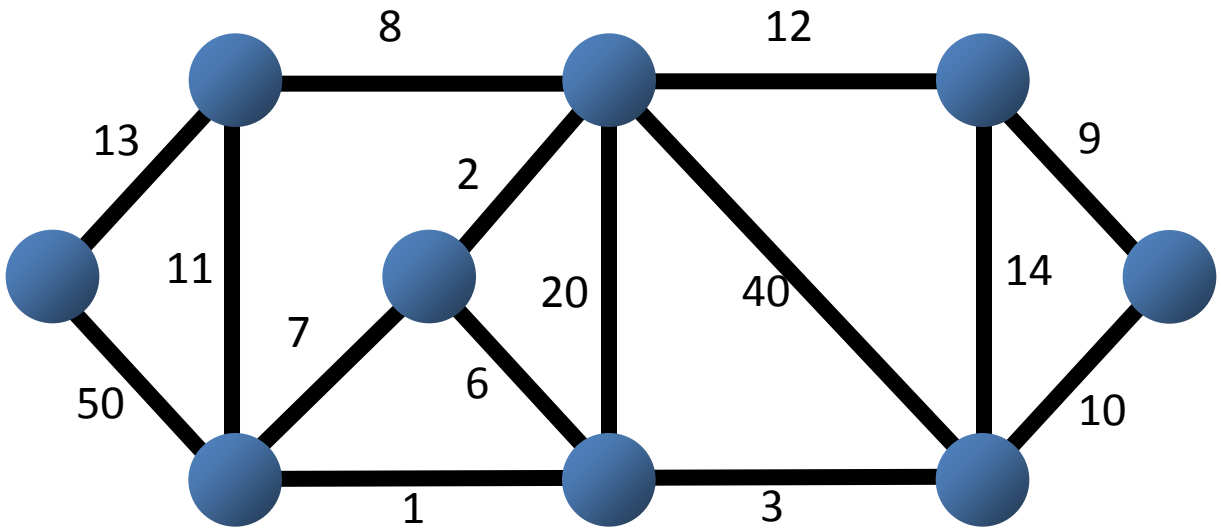
Algorithm:

```
let T be a single vertex x
while (T has fewer than n vertices)
{
    find the smallest edge connecting T to G-T
    add it to T
}
```

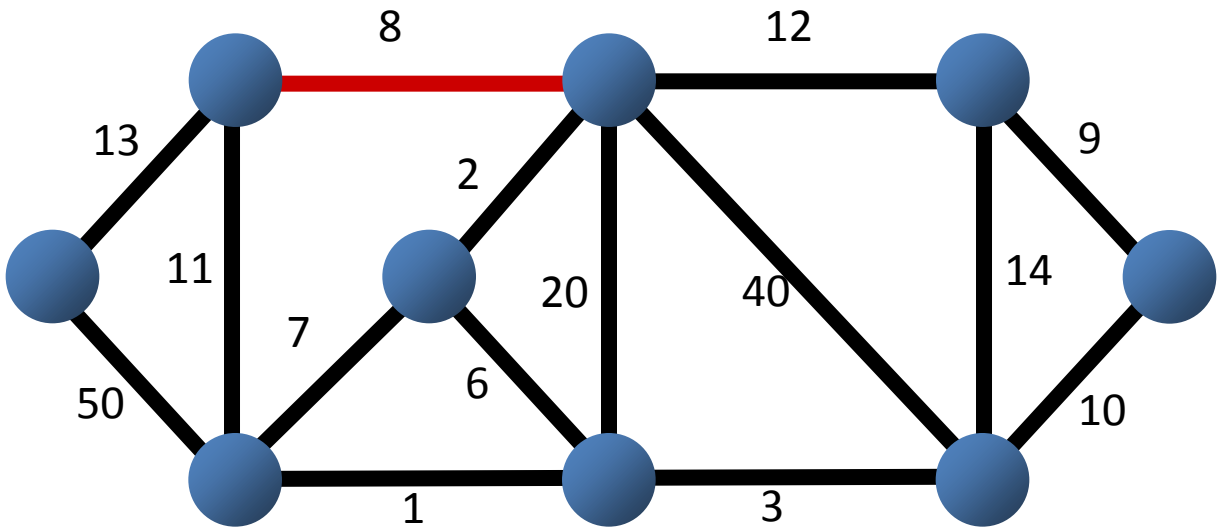
- 1. Pick some arbitrary start node s. Initialize tree T = {s}.
- 2. Repeatedly add the shortest edge incident to T (the shortest edge having one vertex in T and one vertex not in T) until the tree spans all the nodes

- Choose any vertex to start
- Look at all Edges connecting to the vertex
- Choose one with the lowest weight and add to this tree
  - Look at all edges connected to the tree
  - Choose one with the lowest weight and add to this tree
- Repeat until all vertices are covered

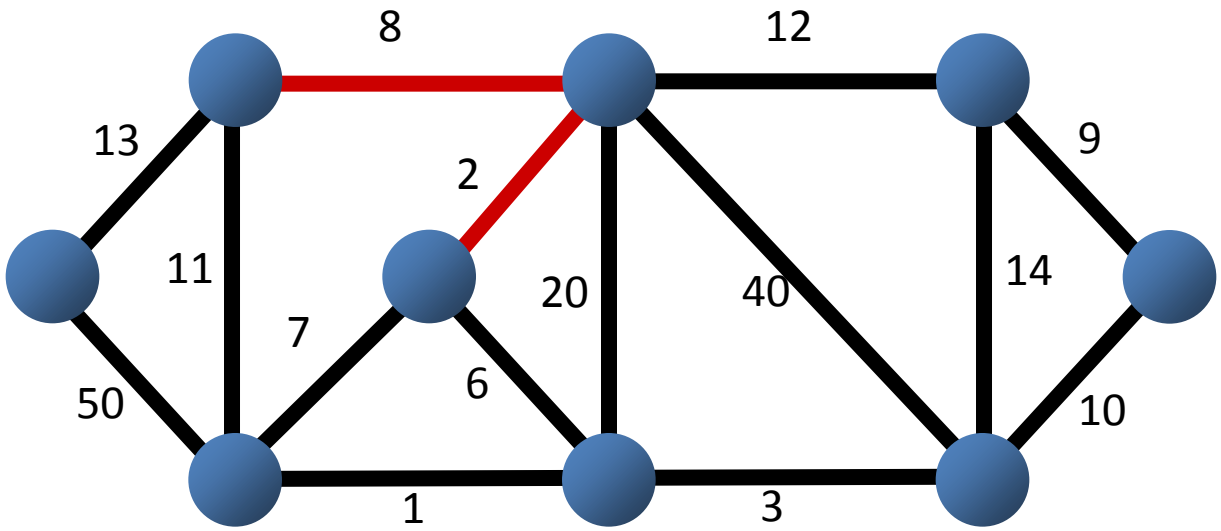
Prim's Algorithm - Example



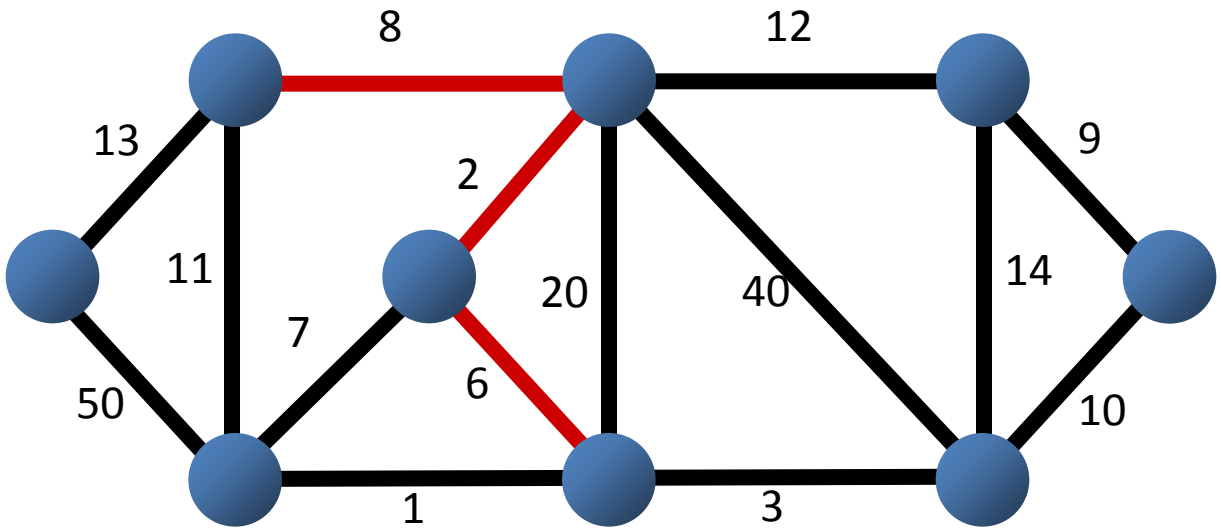
Prim's Algorithm - Example



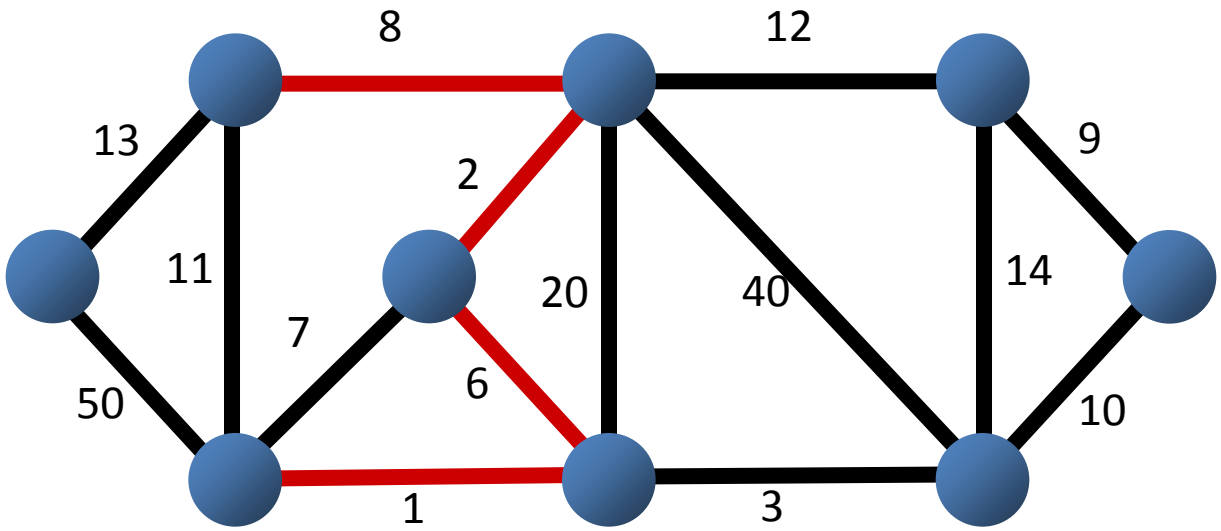
Prim's Algorithm - Example



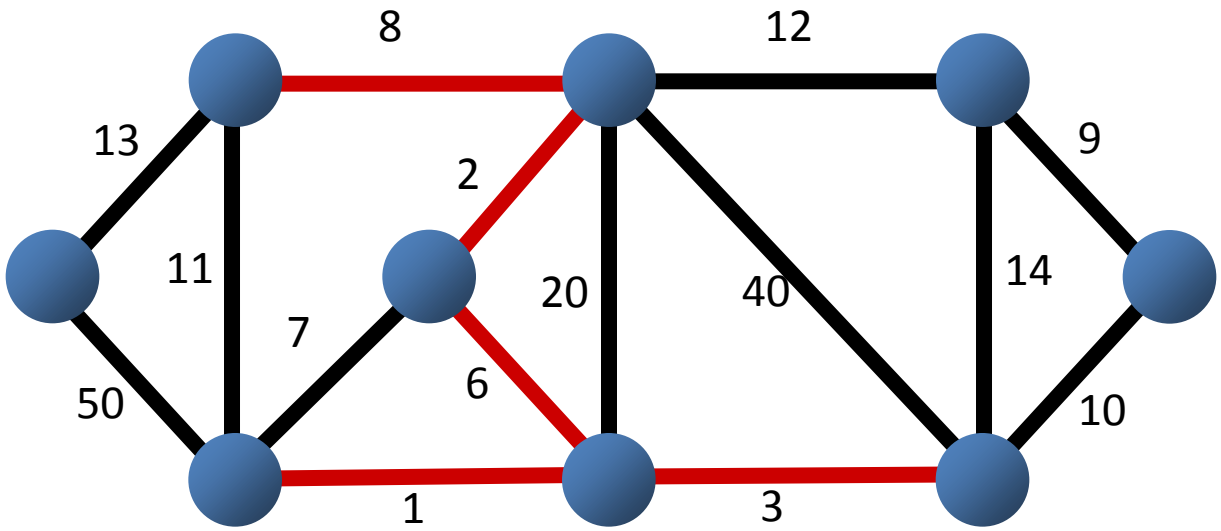
Prim's Algorithm - Example



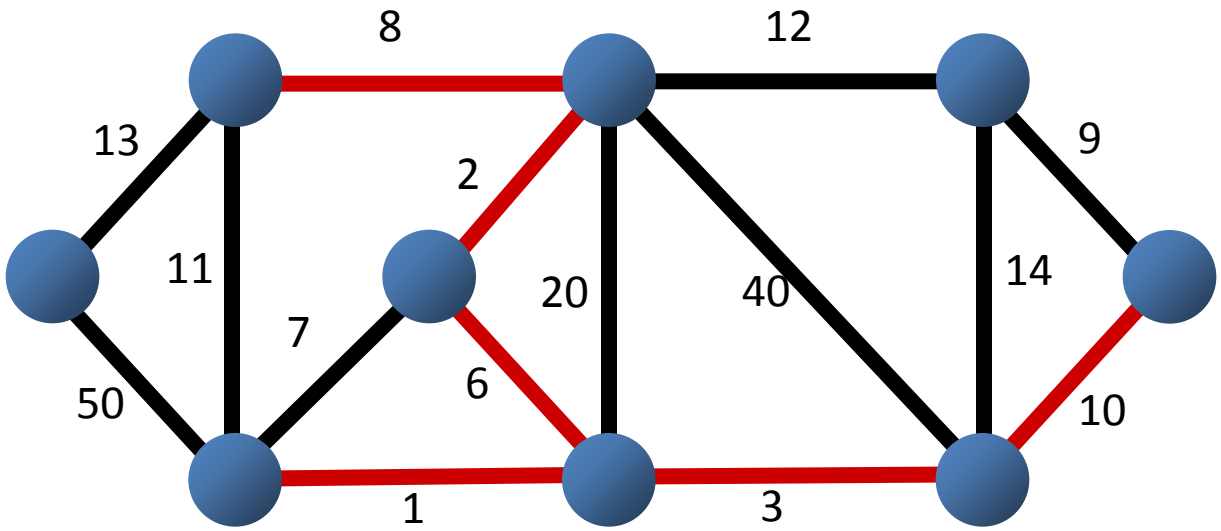
Prim's Algorithm - Example



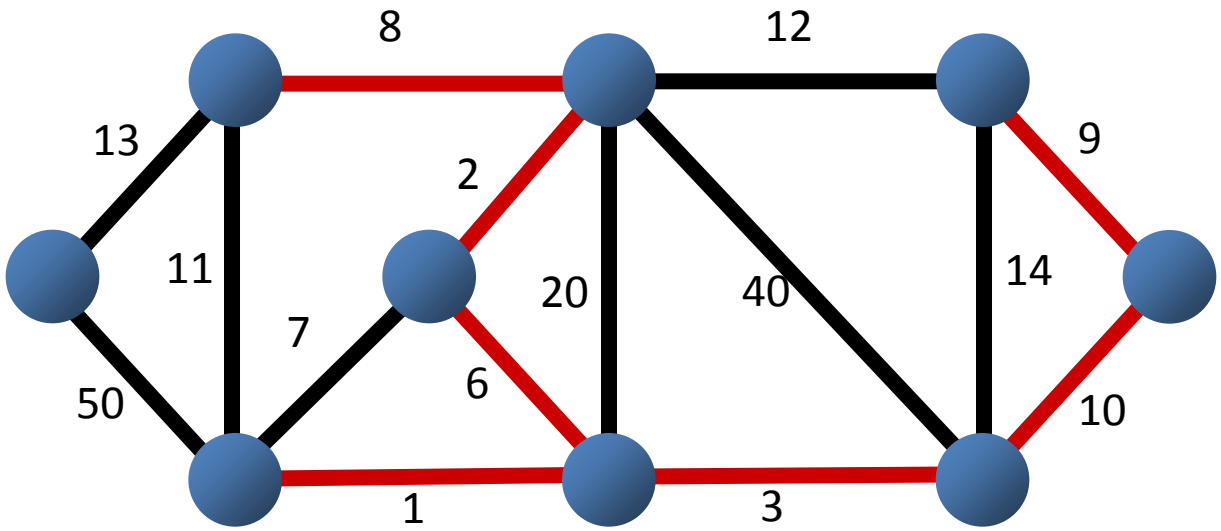
Prim's Algorithm - Example



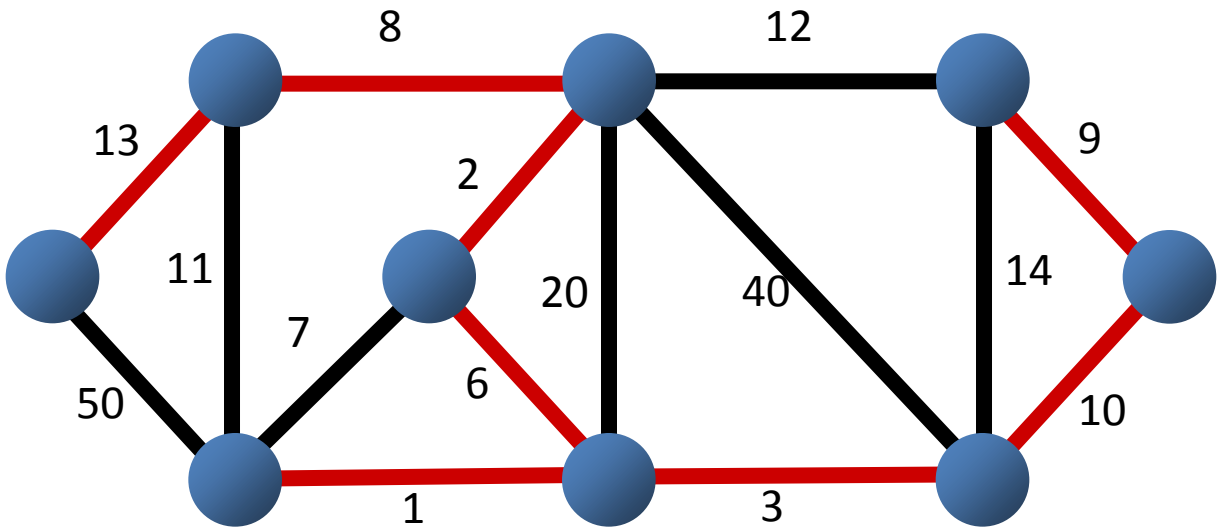
Prim's Algorithm - Example

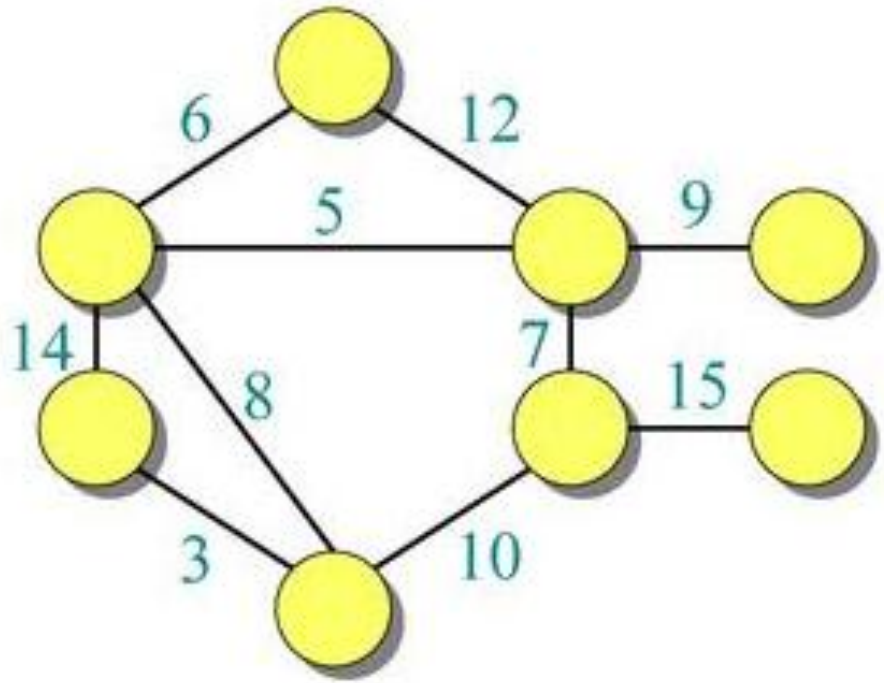


Prim's Algorithm - Example



Prim's Algorithm - Example




6
12
5
9
14
7
8
15
3
10

TOPOLOGICAL SORT

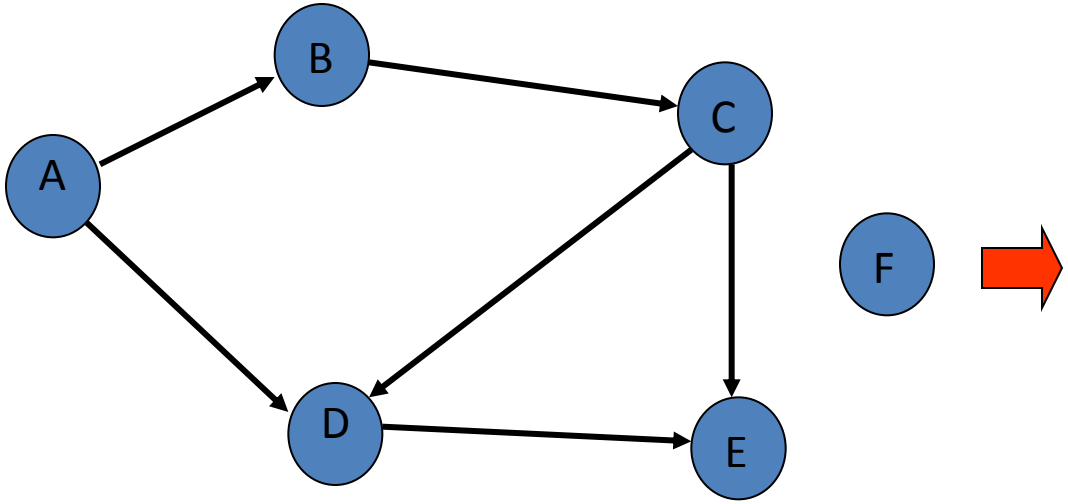
Topological Sort

- a **topological sort (ordering)** is a linear ordering of its vertices such that, for every edge *uv*, *u* comes before *v* in the ordering
  - Consider a Project Management chart in the form of a graph, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks
- A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Graph Algorithms: Topological Sort

The topological sorting problem: given a directed, acyclic graph $G = (V, E)$, find a linear ordering of the vertices such that

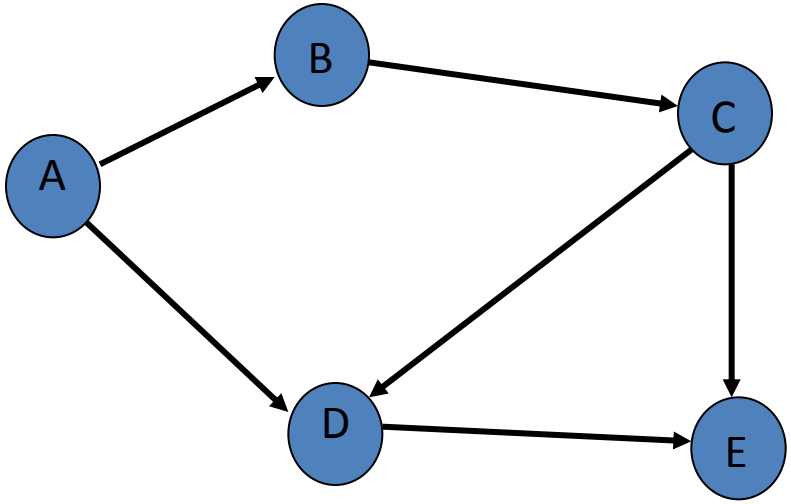
for all $(v, w) \in E$, v precedes w in the ordering.

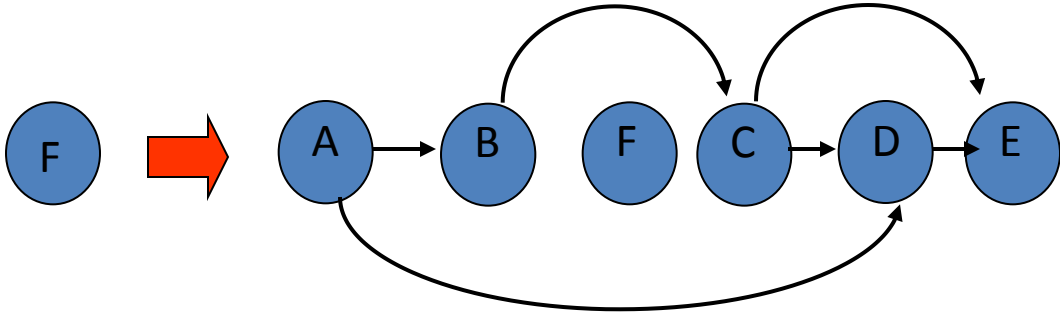


Graph Algorithms: Topological Sort

The topological sorting problem: given a directed, acyclic graph $G = (V, E)$, find a linear ordering of the vertices such that

for all $(v, w) \in E$, v precedes w in the ordering.

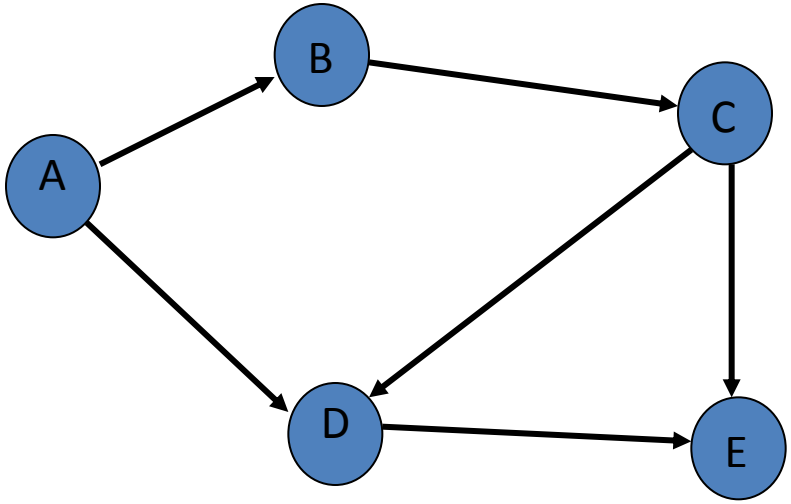




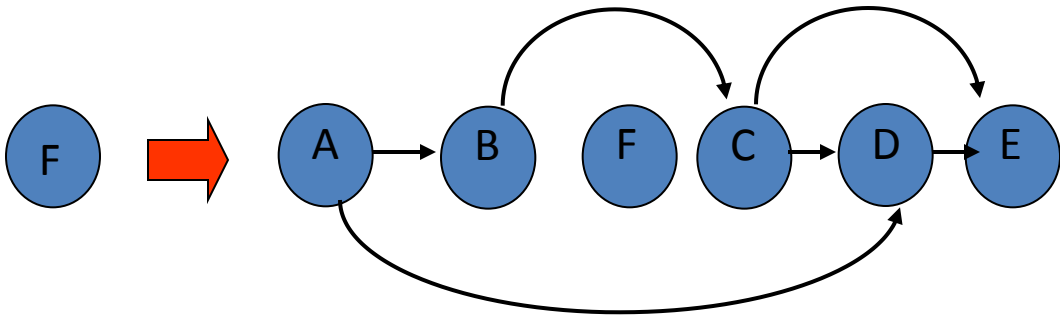
Graph Algorithms: Topological Sort

The topological sorting problem: given a directed, acyclic graph $G = (V, E)$, find a linear ordering of the vertices such that

for all $(v, w) \in E$, v precedes w in the ordering.



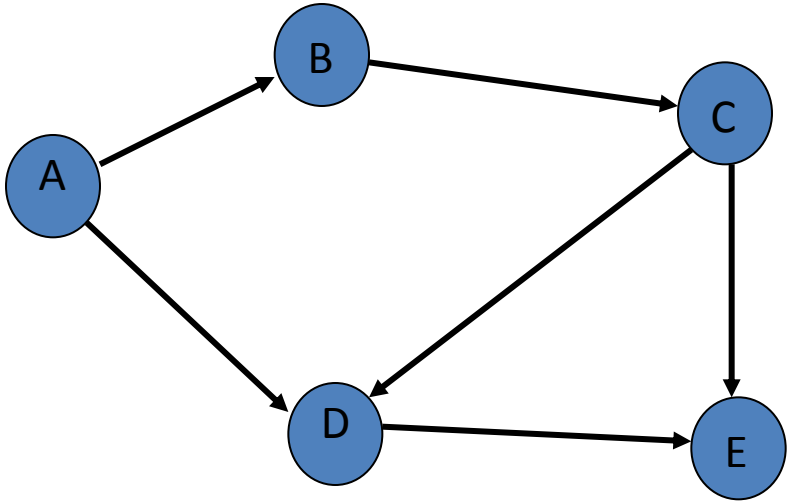
Any linear ordering in which all the arrows go to the right.



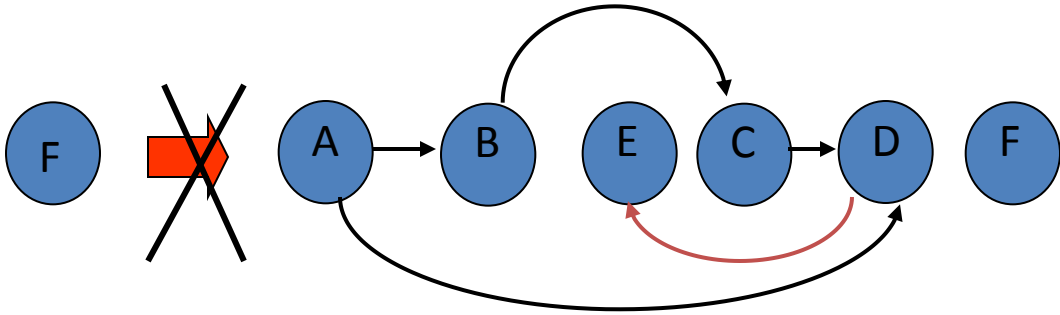
Graph Algorithms: Topological Sort

The topological sorting problem: given a directed, acyclic graph $G = (V, E)$, find a linear ordering of the vertices such that

for all (v, w) E, v precedes w in the ordering.



Any linear ordering in which all the arrows go to the right.

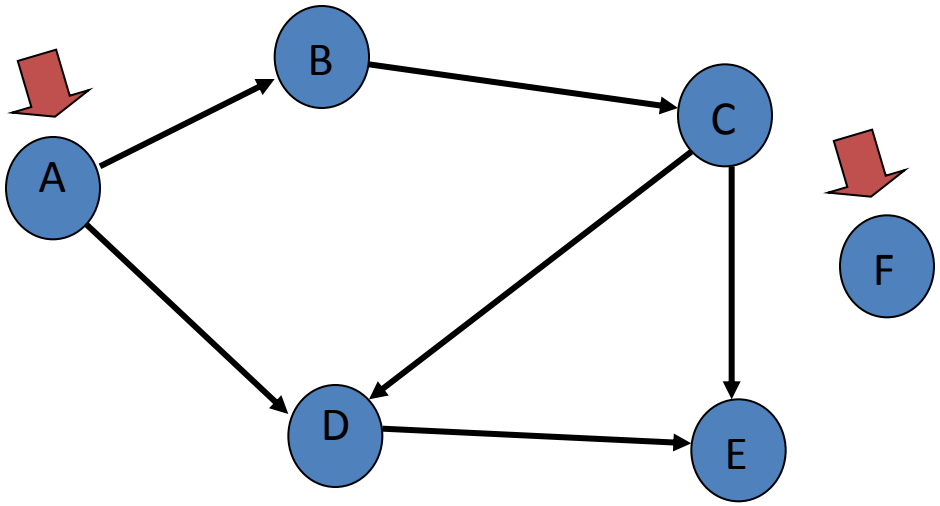


This is not a topological ordering.

Graph Algorithms: Topological Sort

The topological sorting algorithm:

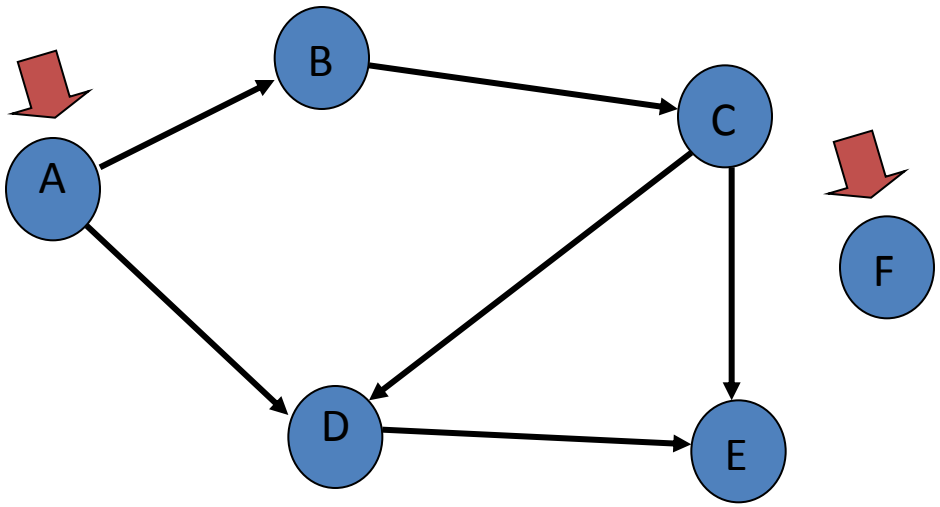
Identify the subset of vertices that have no incoming edge.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

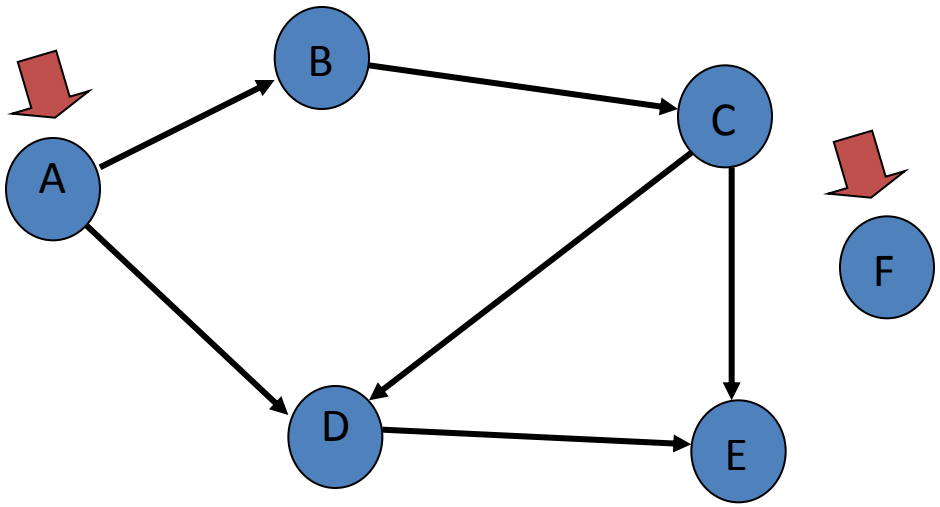
Identify the subset of vertices that have no incoming edge. (In general, this subset must be nonempty—why?)



Graph Algorithms: Topological Sort

The topological sorting algorithm:

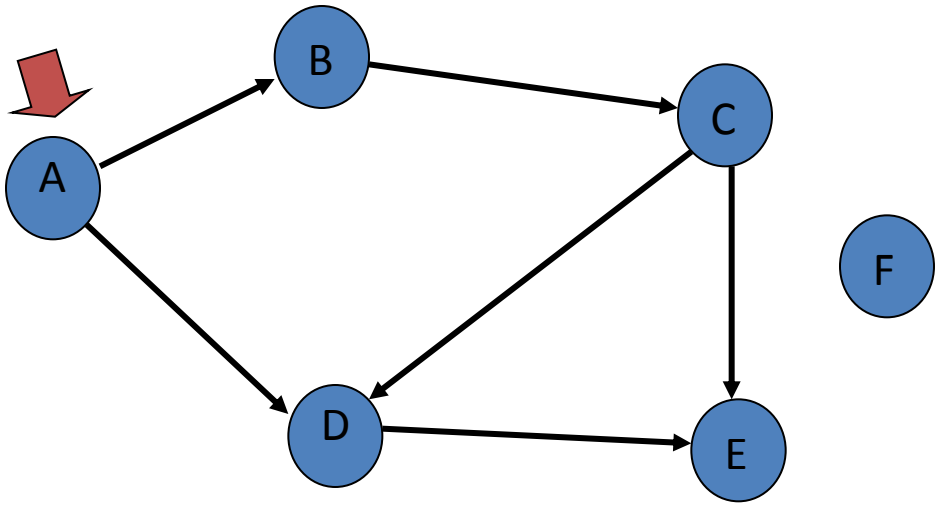
Identify the subset of vertices that have no incoming edge. (In general, this subset must be nonempty—because the graph is acyclic.)



Graph Algorithms: Topological Sort

The topological sorting algorithm:

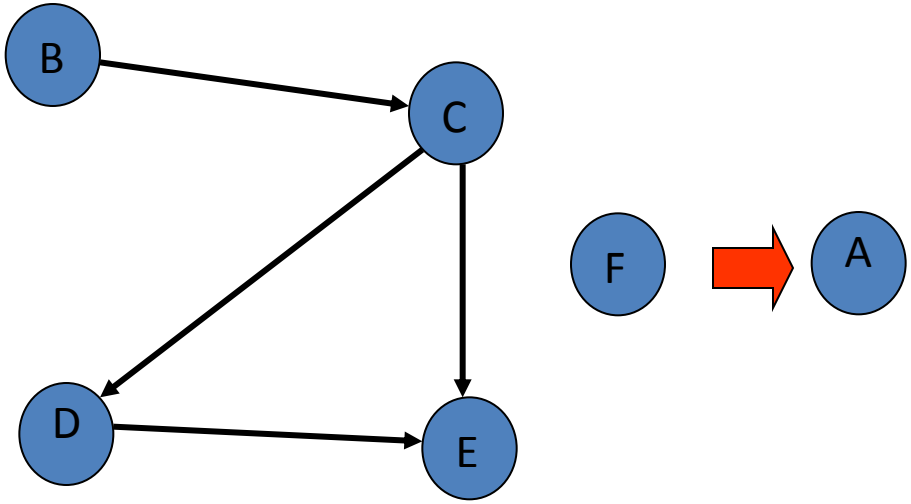
Identify the subset of vertices that have no incoming edge. Select one of them.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

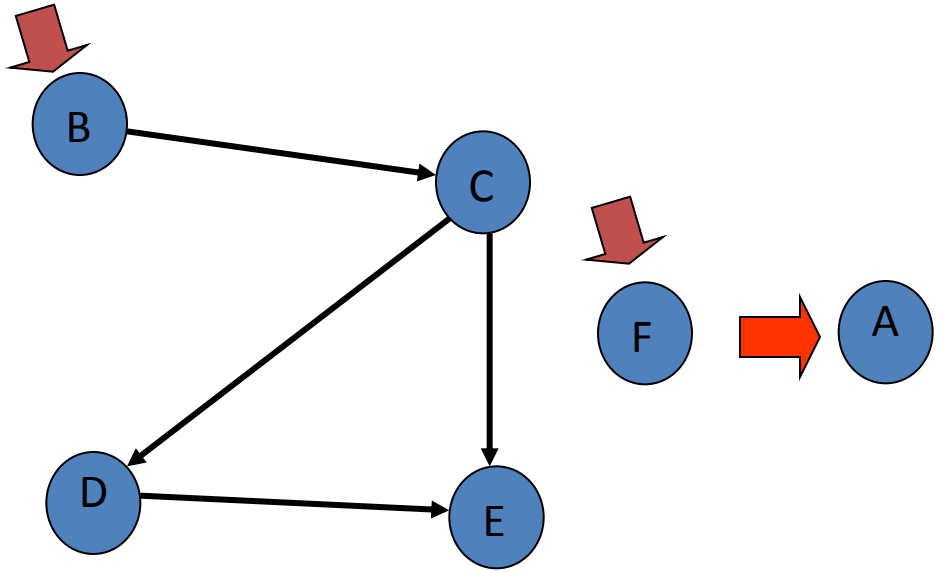
Remove it, and its outgoing edges,
and add it to the output.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

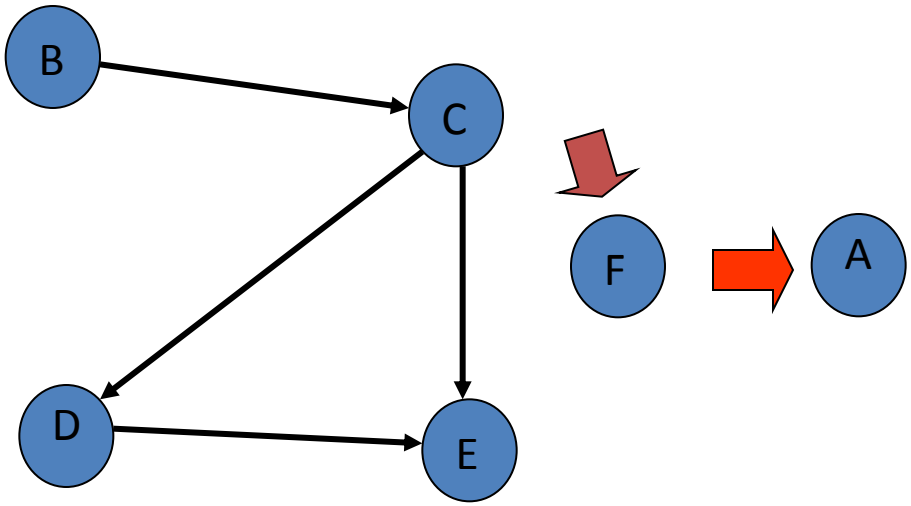
Again, identify the subset of vertices that have no incoming edge, . . .



Graph Algorithms: Topological Sort

The topological sorting algorithm:

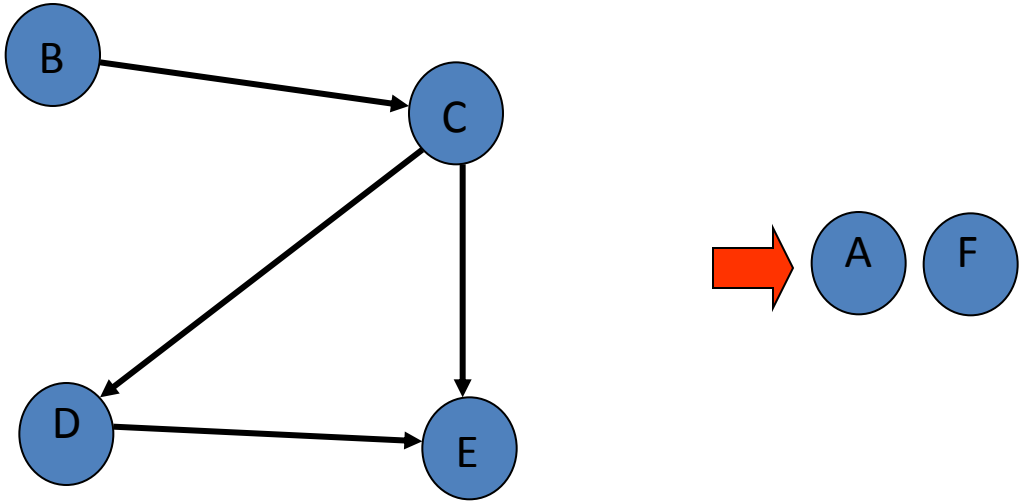
Again, identify the subset of vertices that have no incoming edge, select one of them, . . .



Graph Algorithms: Topological Sort

The topological sorting algorithm:

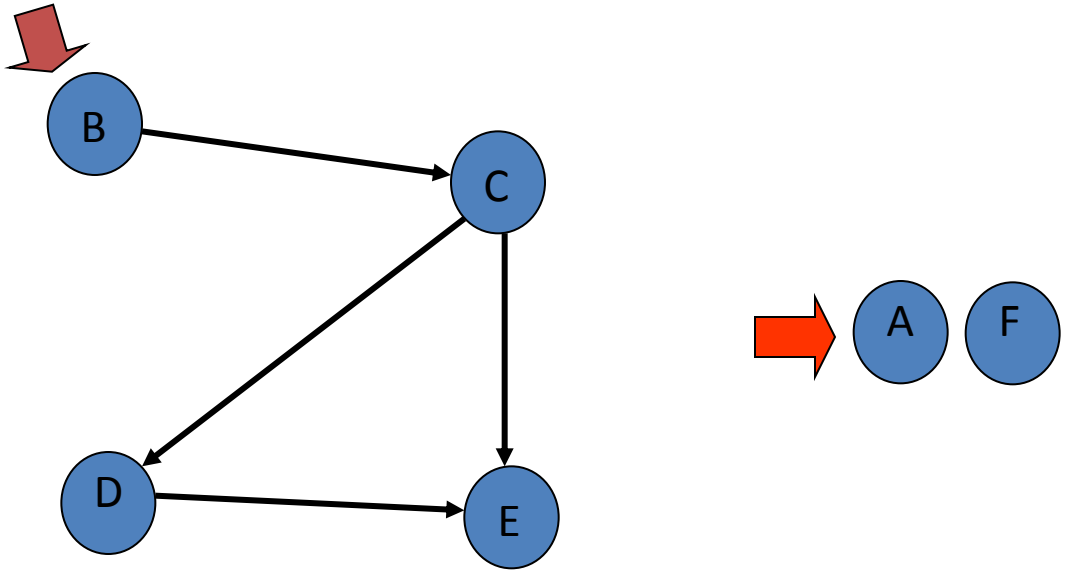
Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

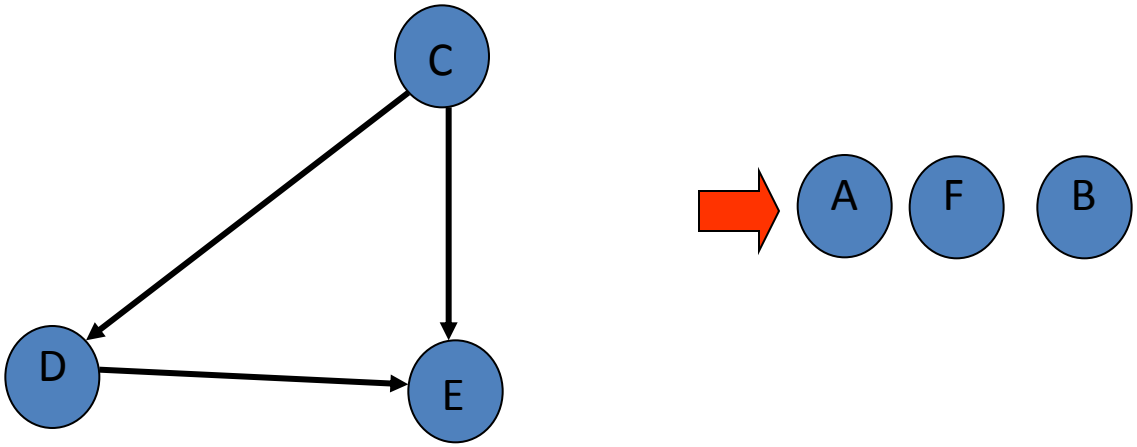
Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

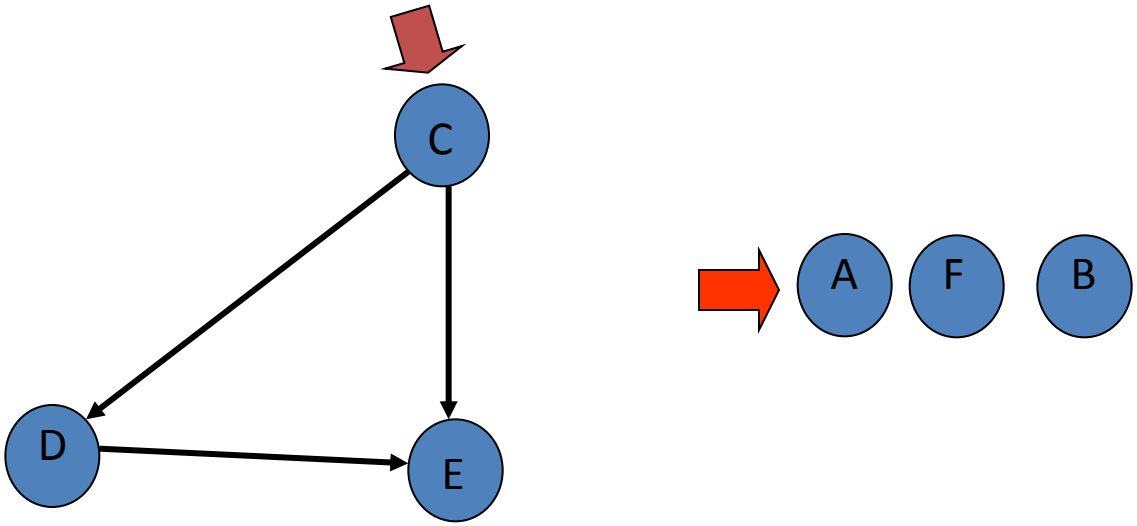
Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

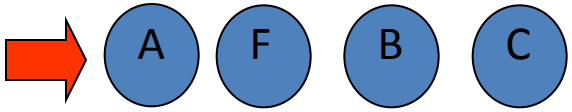
Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.



Graph Algorithms: Topological Sort

The topological sorting algorithm:

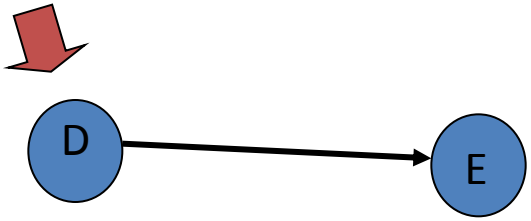
Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.

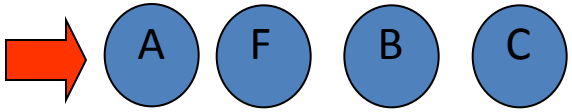


Graph Algorithms: Topological Sort

The topological sorting algorithm:

Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.

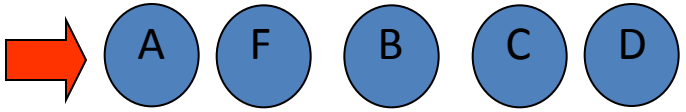


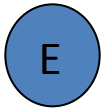


Graph Algorithms: Topological Sort

The topological sorting algorithm:

Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.

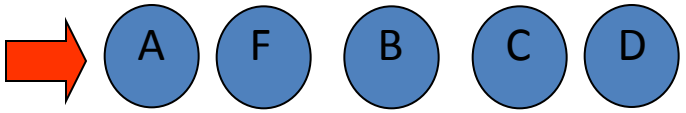


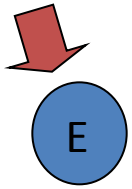


Graph Algorithms: Topological Sort

The topological sorting algorithm:

Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.

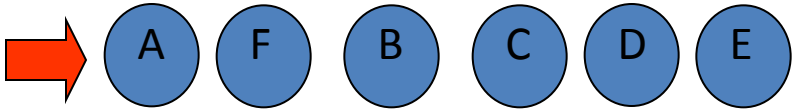




Graph Algorithms: Topological Sort

The topological sorting algorithm:

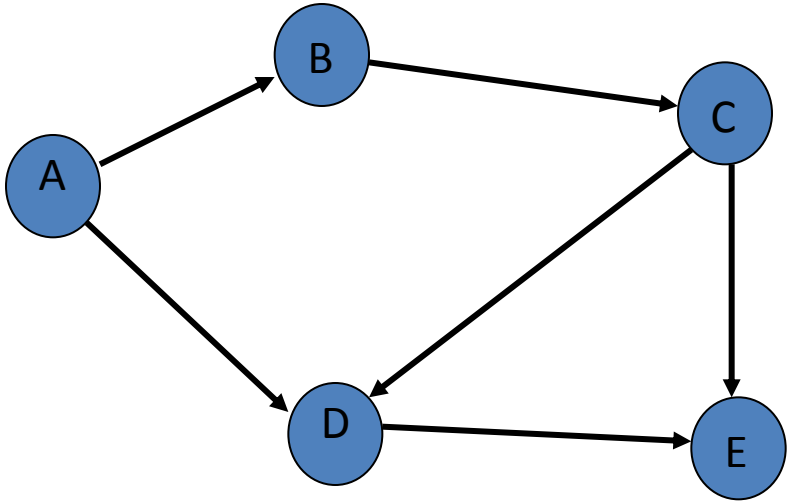
Again, identify the subset of vertices that have no incoming edge, select one of them, remove it and any outgoing edges, and put it in the output.

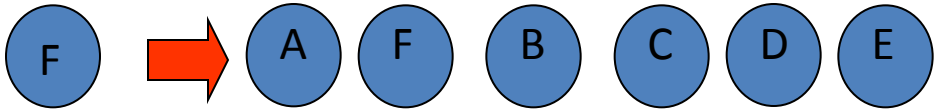


Graph Algorithms: Topological Sort

The topological sorting algorithm:

finished!

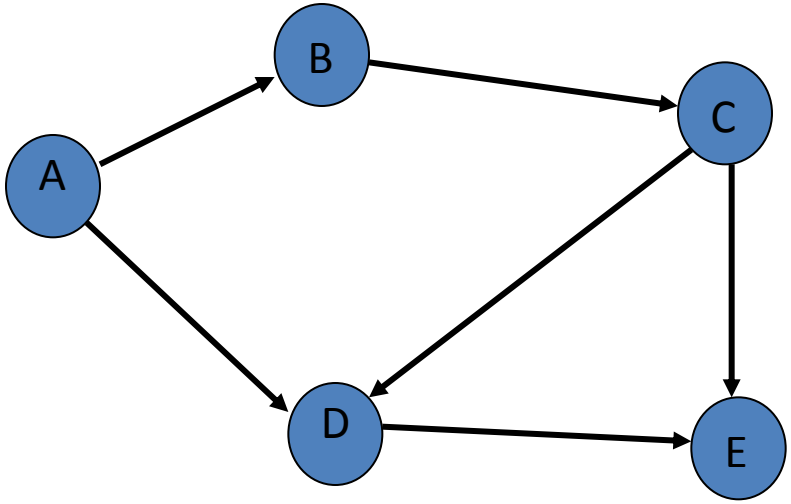


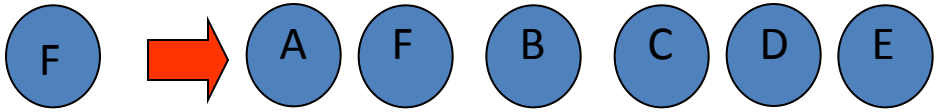


Graph Algorithms: Topological Sort

The topological sorting algorithm:

Time bound?





Graph Algorithms: Topological Sort

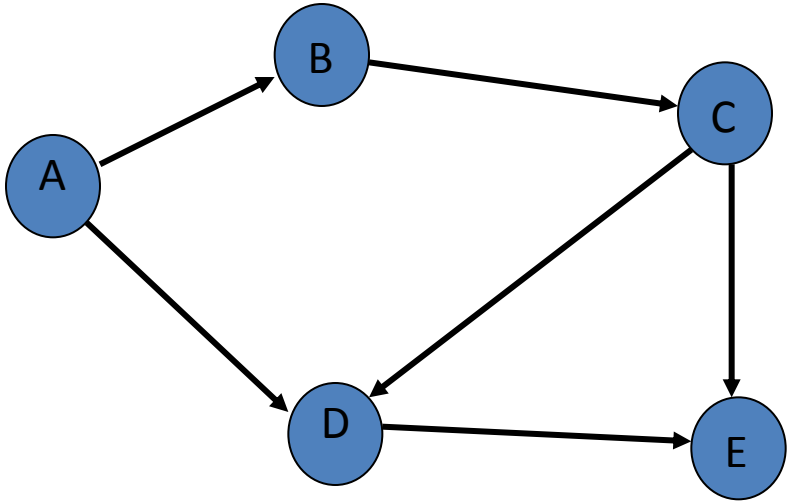
The topological sorting algorithm:

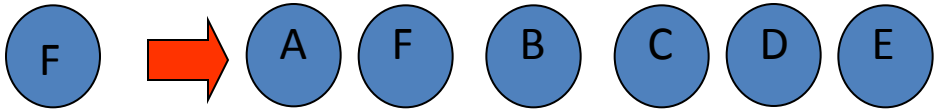
Time bound: *Break down into total time to:*

Find vertices with no predecessors: ?

Remove edges: ?

Place vertices in output: ?





Graph Algorithms: Topological Sort

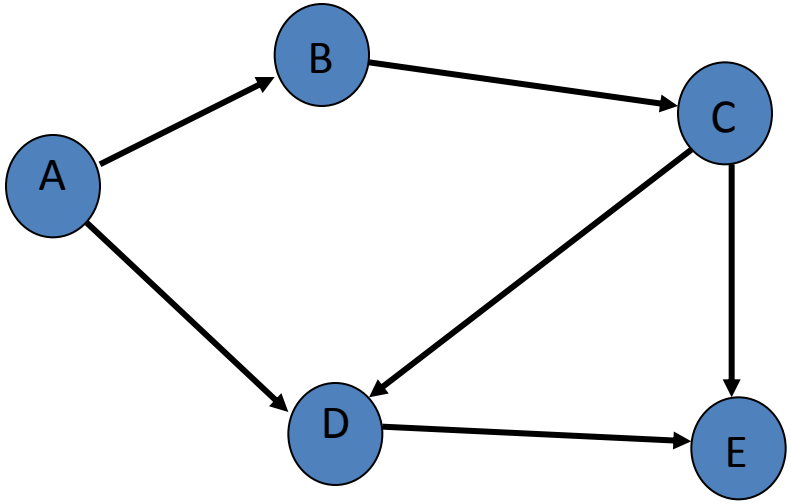
The topological sorting algorithm:

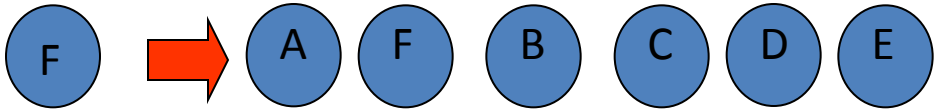
Time bound: *Break down into total time to:*

Find vertices with no predecessors: ?

Remove edges: O(|*E*|)

Place vertices in output: ?





Graph Algorithms: Topological Sort

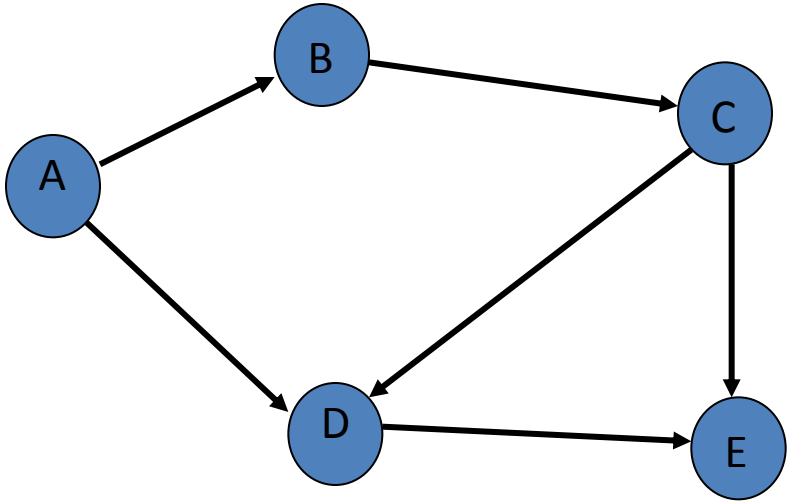
The topological sorting algorithm:

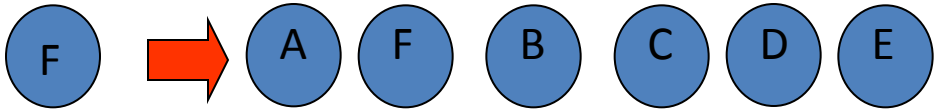
Time bound: *Break down into total time to:*

Find vertices with no predecessors: ?

Remove edges: O(|*E*|)

Place vertices in output: O(|*V*|)

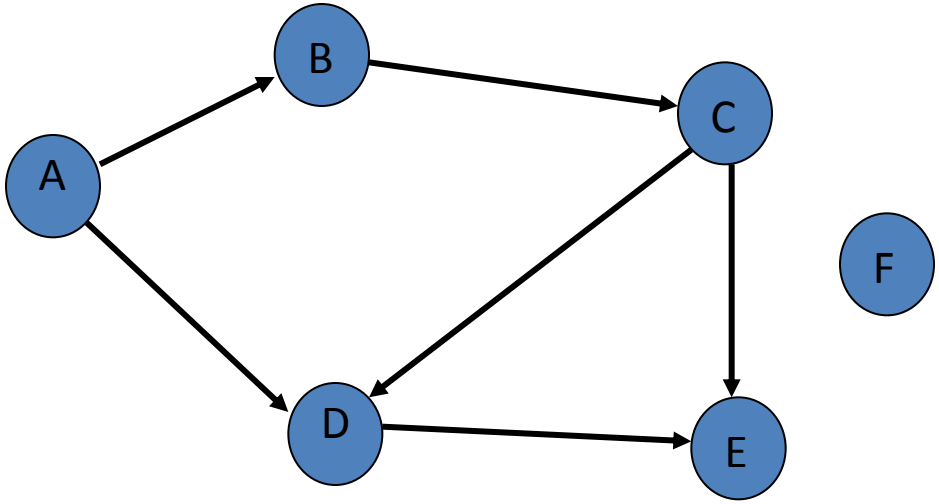


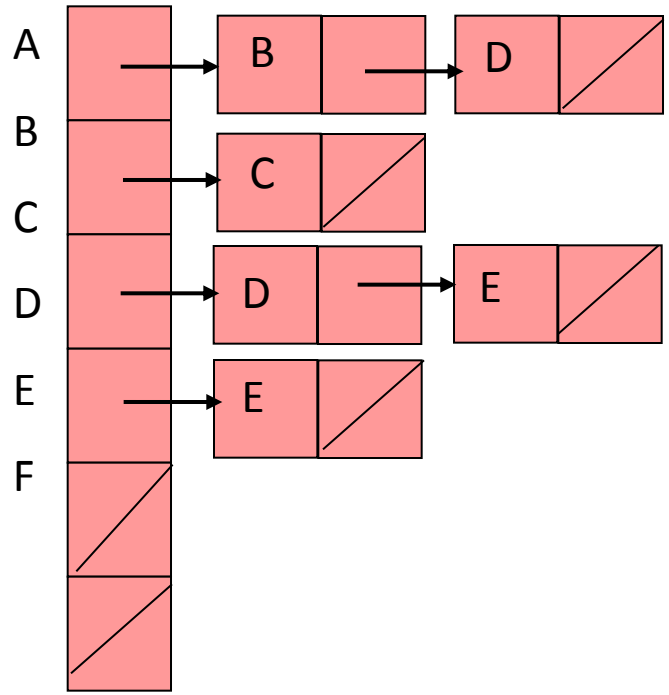


Graph Algorithms: Topological Sort

Find vertices with no predecessors: ?

Assume an adjacency list representation:

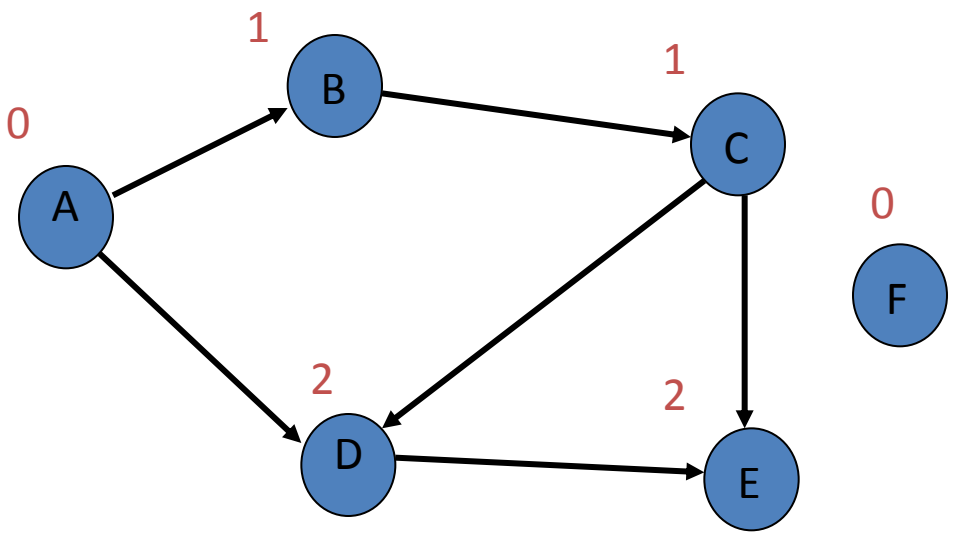


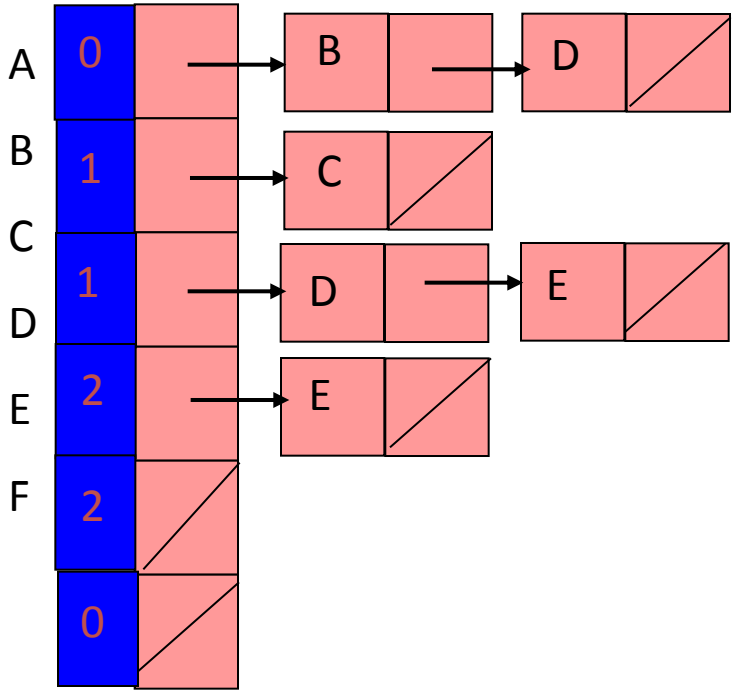


Graph Algorithms: Topological Sort

The topological sorting algorithm:

…and initialize and maintain for each vertex its no. of predecessors.

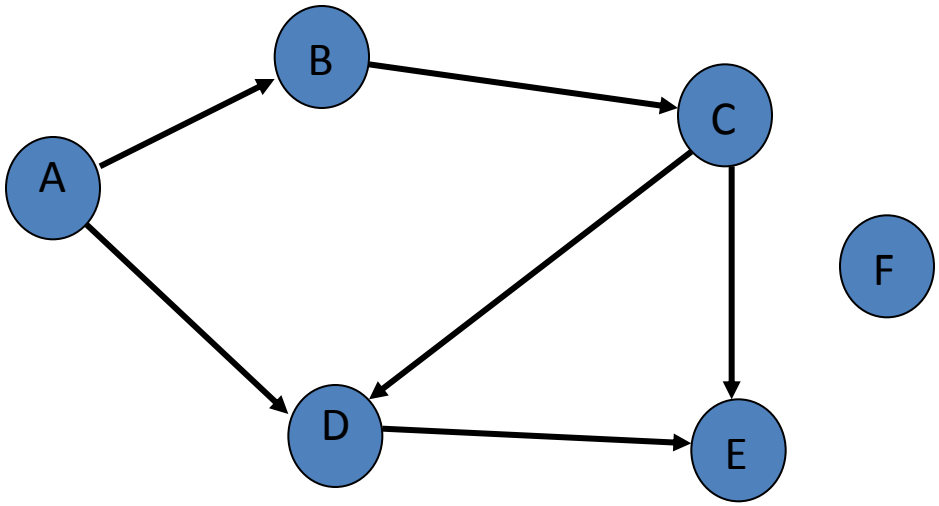


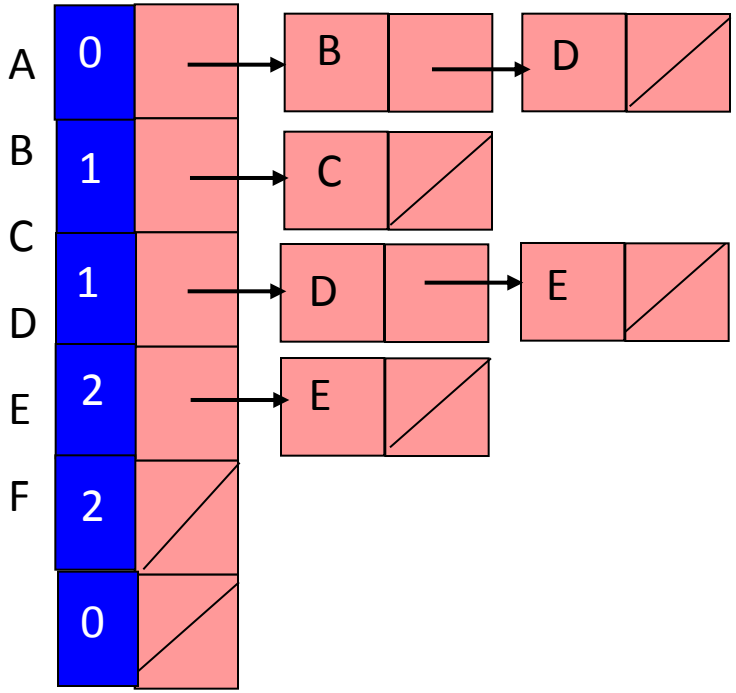


Graph Algorithms: Topological Sort

Find vertices with no predecessors: ?

Time for each vertex: O(|*V*|)

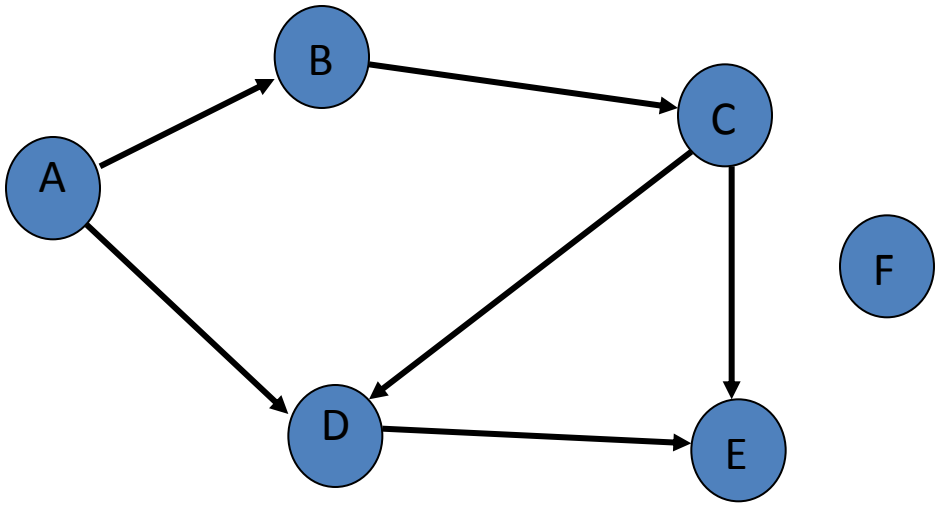


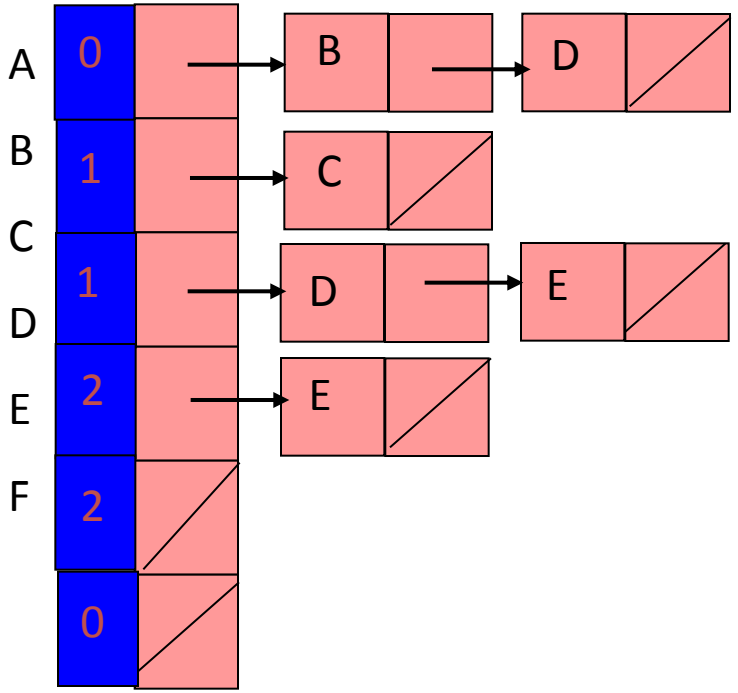


Graph Algorithms: Topological Sort

Find vertices with no predecessors: ?

Total time: O(|*V*| ^{2})





Graph Algorithms: Topological Sort

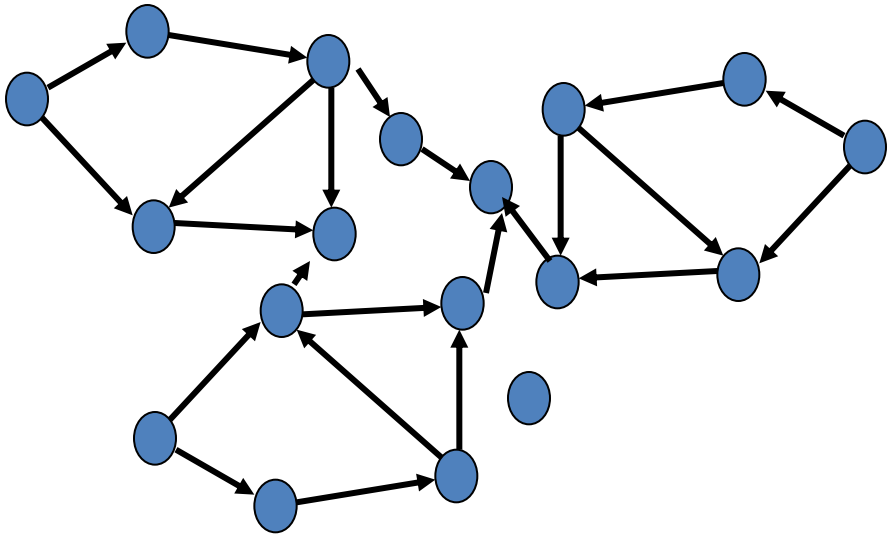
The topological sorting algorithm:

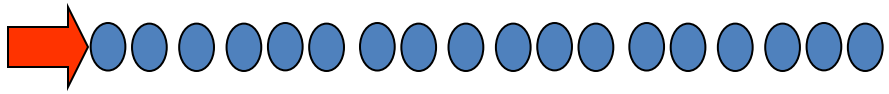
Time bound: *Break down into total time to:*

Find vertices with no predecessors: $O(|V|^2)$

Remove edges: $O(|E|)$

Place vertices in output: $O(|V|)$





Graph Algorithms: Topological Sort

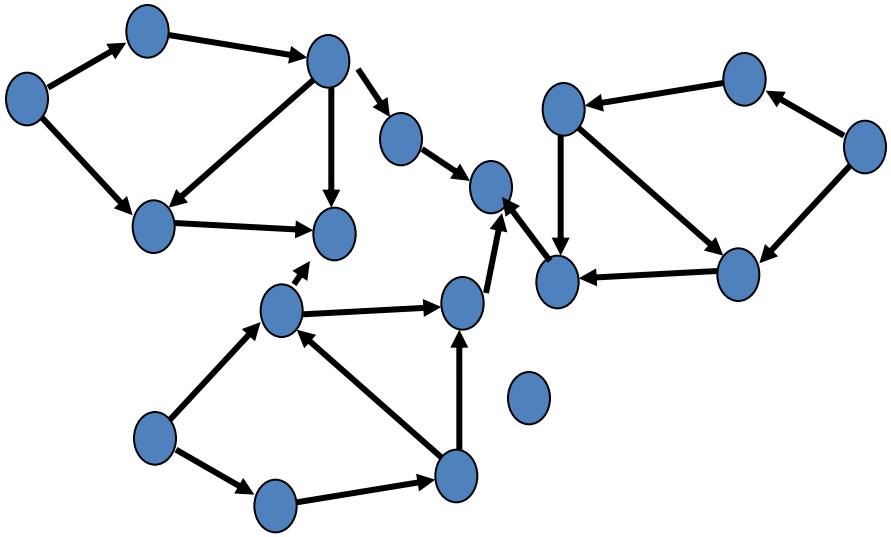
The topological sorting algorithm:

Time bound: *Break down into total time to:*

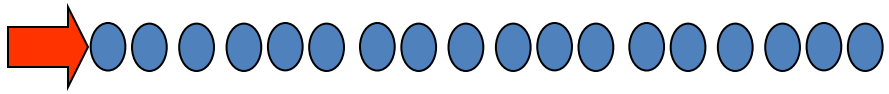
Find vertices with no predecessors: $O(|V|^2)$

Remove edges: $O(|E|)$

Place vertices in output: $O(|V|)$



Total: $O(|V| + |E|^2)$



Graph Algorithms: Topological Sort

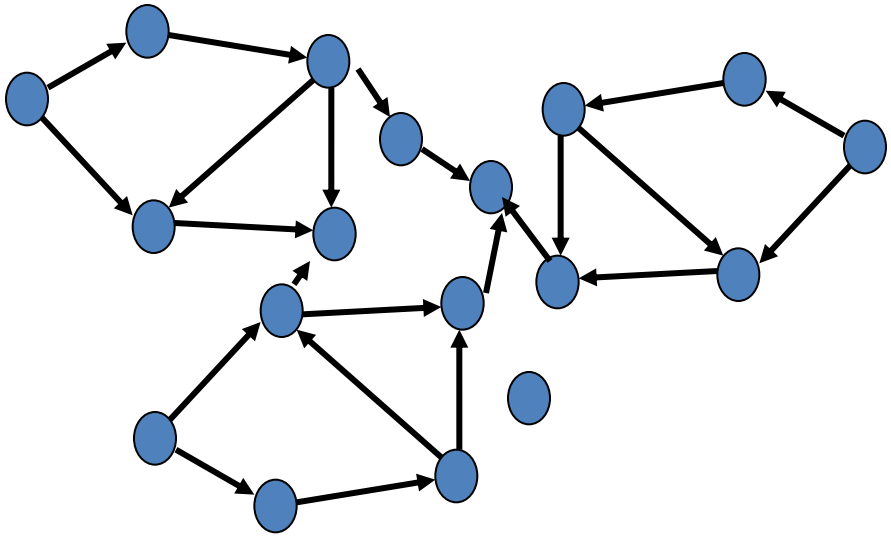
The topological sorting algorithm:

Time bound: *Break down into total time to:*

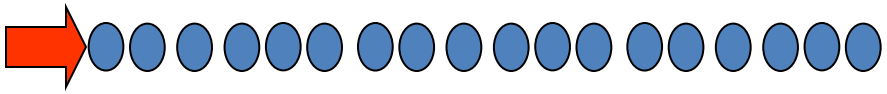
Find vertices with no predecessors: $O(|V|^2)$

Remove edges: $O(|E|)$

Place vertices in output: $O(|V|)$



Total: $O(|V| + |E|^2)$



Too much!

Graph Algorithms: Topological Sort

The topological sorting algorithm:

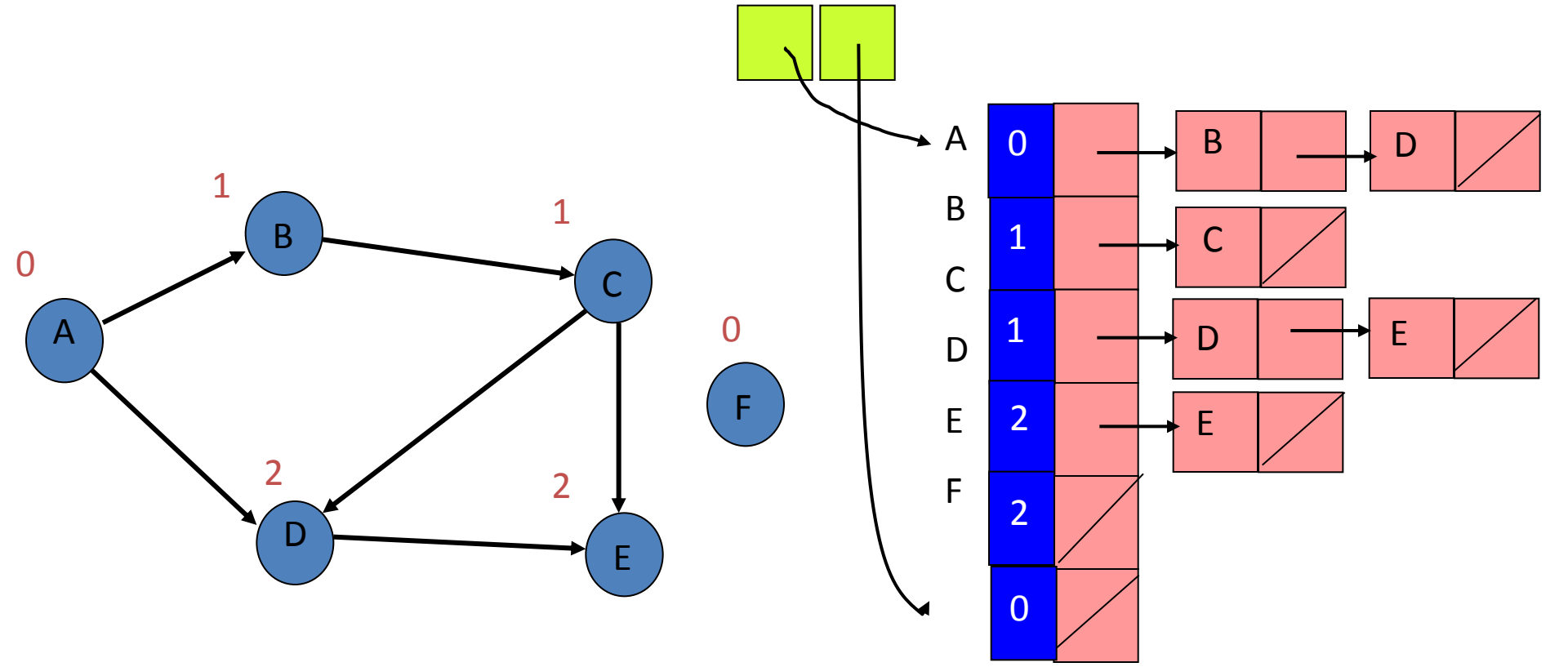
We need a faster way to do this step:

Find vertices with no predecessors.

Graph Algorithms: Topological Sort

The topological sorting algorithm:

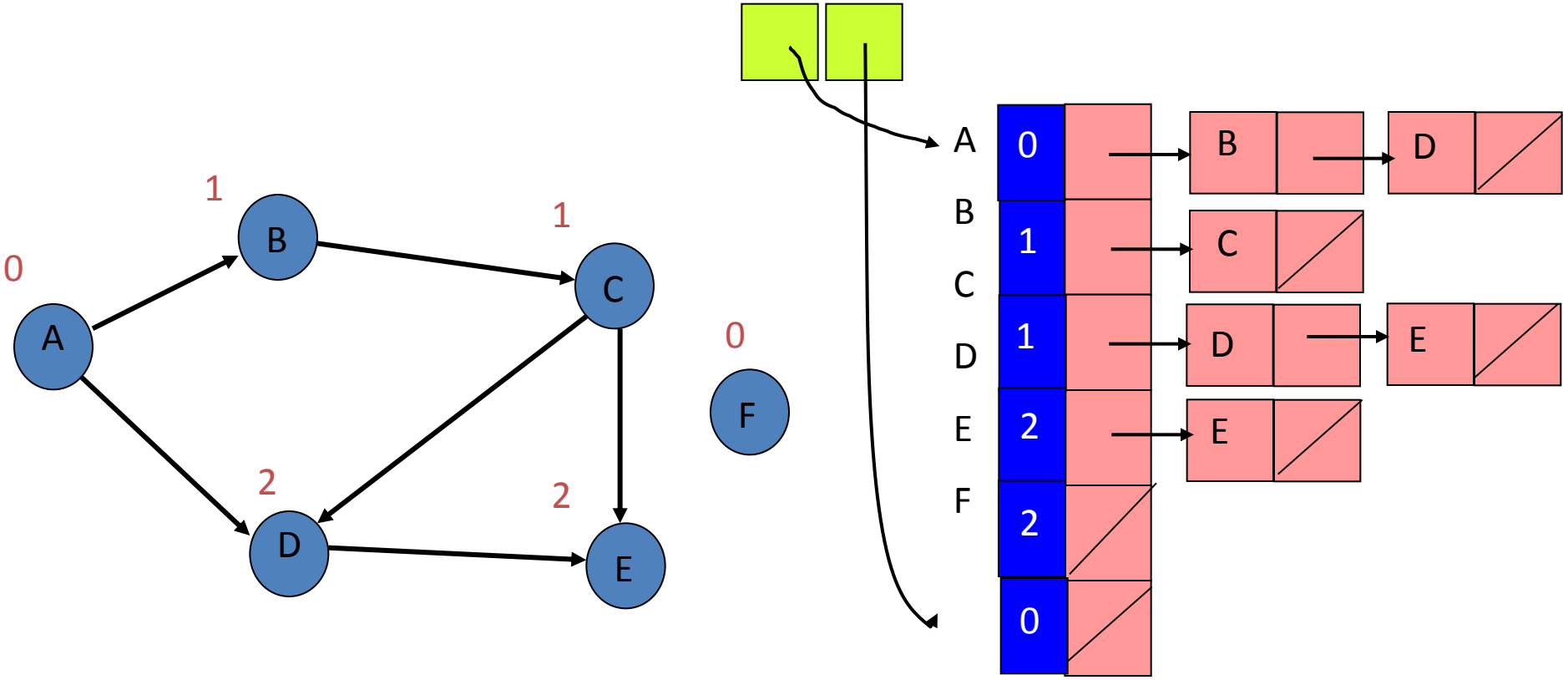
Key idea: initialize and maintain a queue (or stack) holding pointers to the vertices with 0 predecessors



Graph Algorithms: Topological Sort

The topological sorting algorithm:

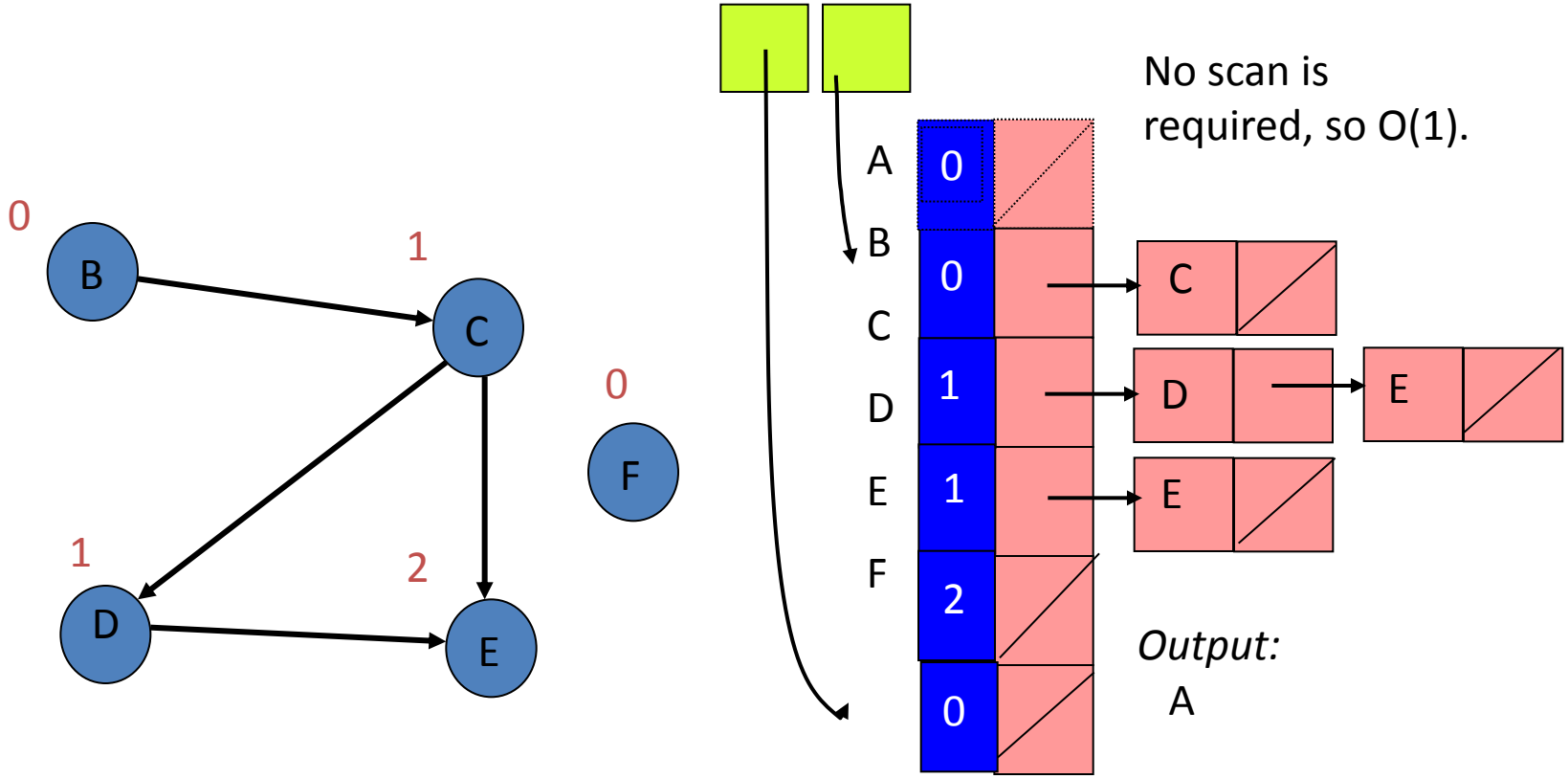
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.



Graph Algorithms: Topological Sort

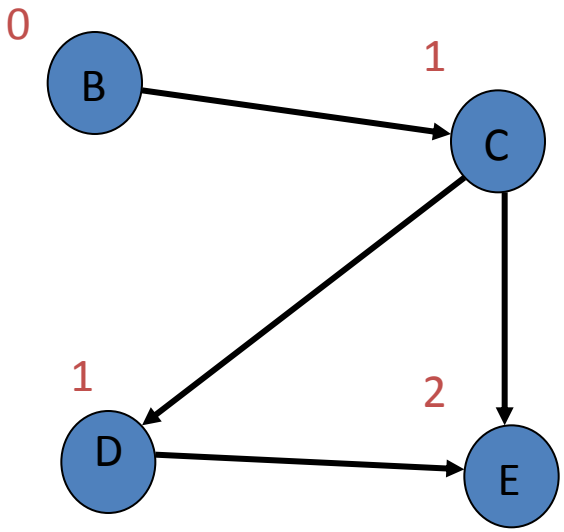
The topological sorting algorithm:

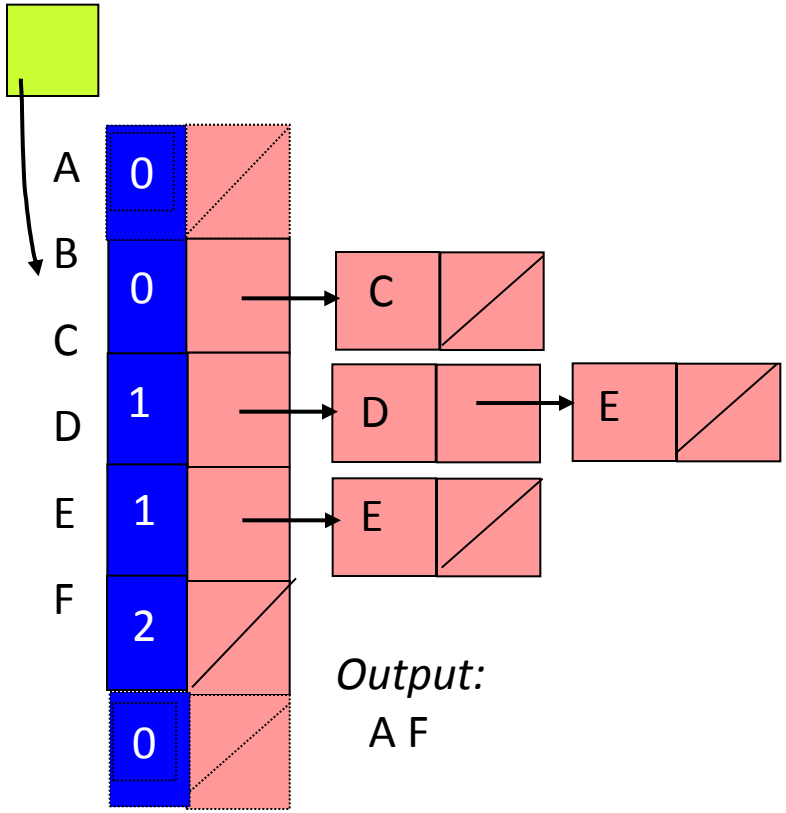
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.



Graph Algorithms: Topological Sort

The topological sorting algorithm:
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.

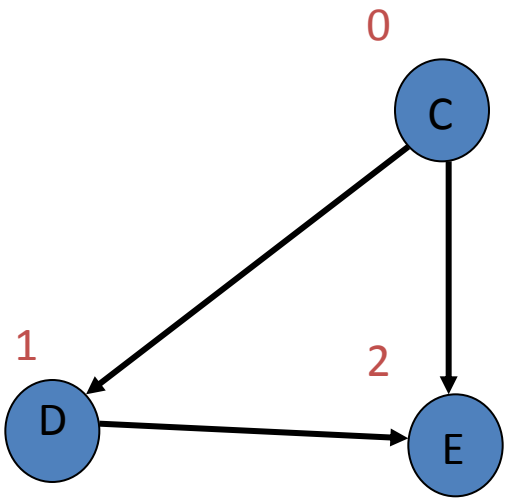


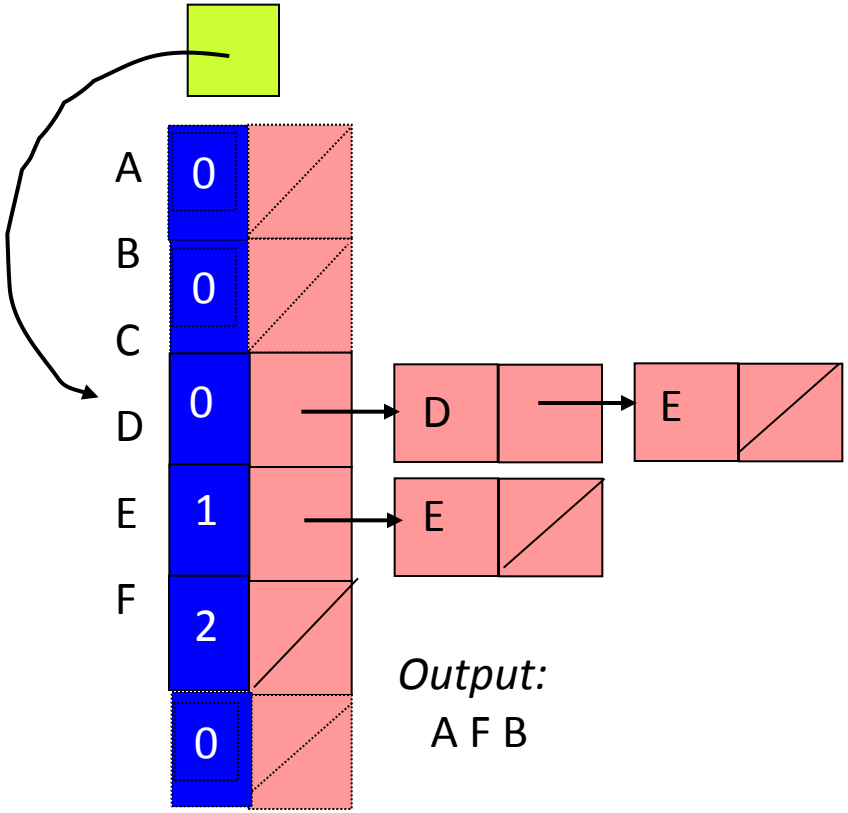


Graph Algorithms: Topological Sort

The topological sorting algorithm:

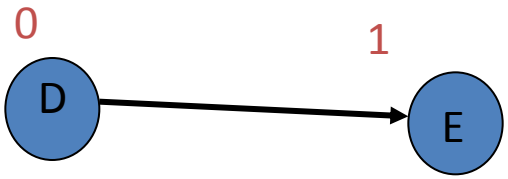
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.

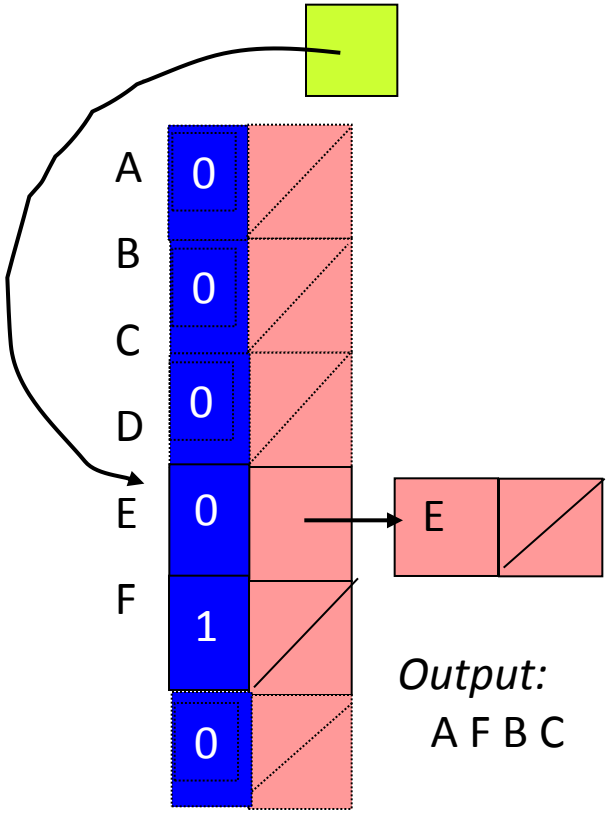




Graph Algorithms: Topological Sort

The topological sorting algorithm:
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.

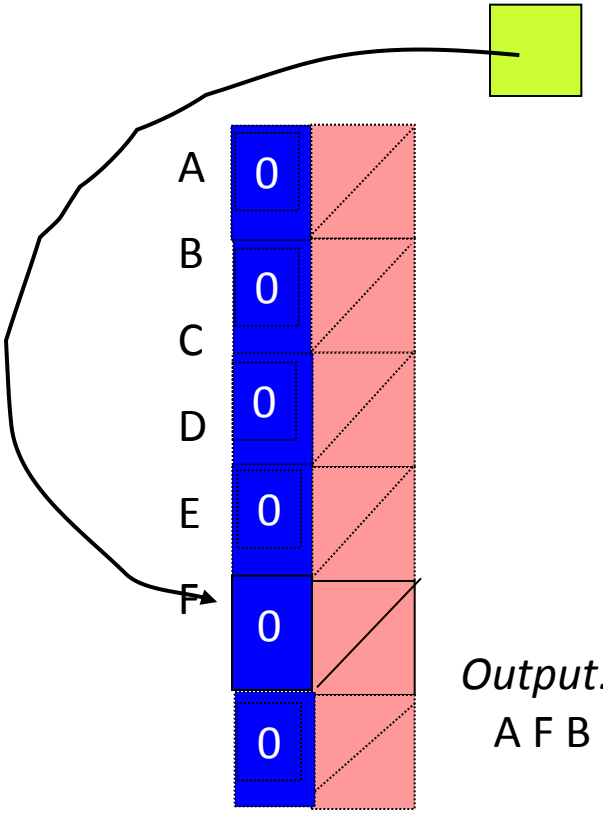




Graph Algorithms: Topological Sort

The topological sorting algorithm:

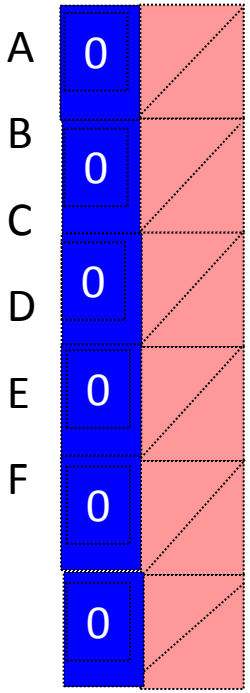
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.



Graph Algorithms: Topological Sort

The topological sorting algorithm:
As each vertex is removed, update the predecessor counts, and for any vertex whose count has become 0, put it in the queue.

Completed!



Output:
A F B C D E

Graph Algorithms: Topological Sort

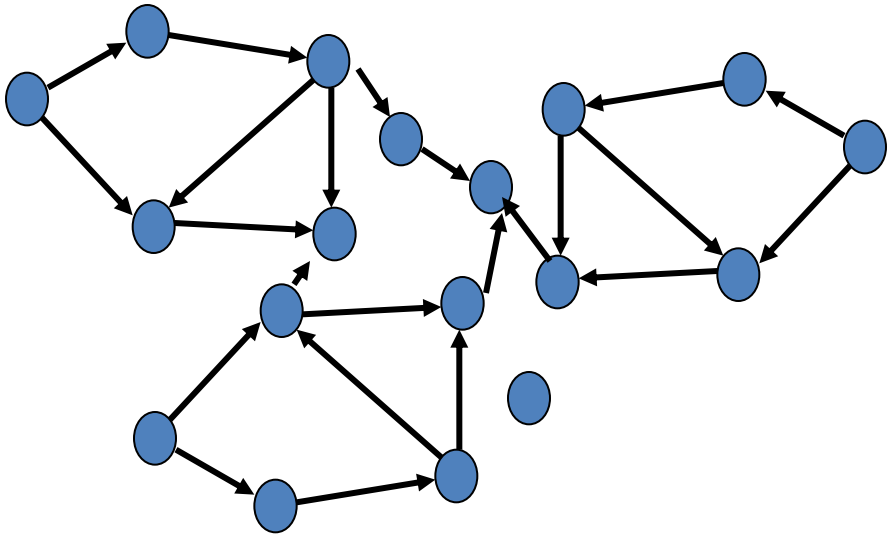
The topological sorting algorithm:

Time bound: *Now the time for each part is*

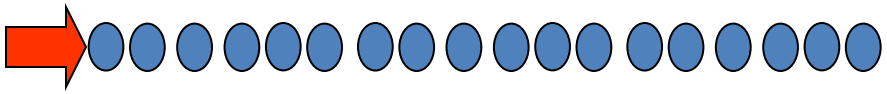
Find vertices with no predecessors: O(|*V*|)

Remove edges: O(|*E*|)

Place vertices in output: O(|*V*|)



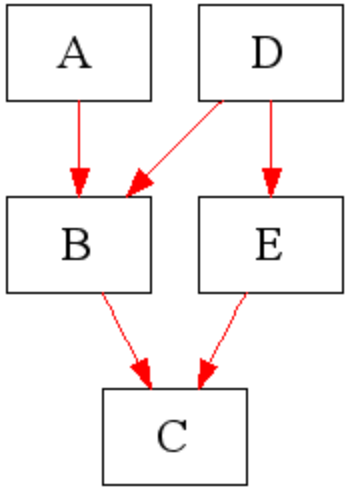
Total: O(|*V*|+|*E*|)

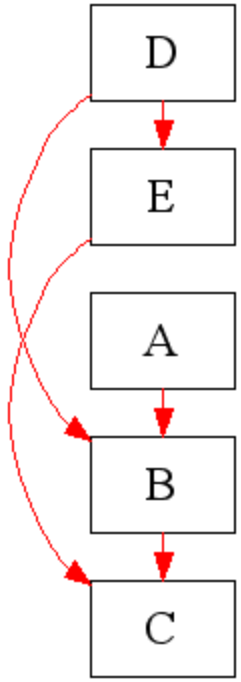


Linear in |*V*|+|*E*|.

Much better!

Examples

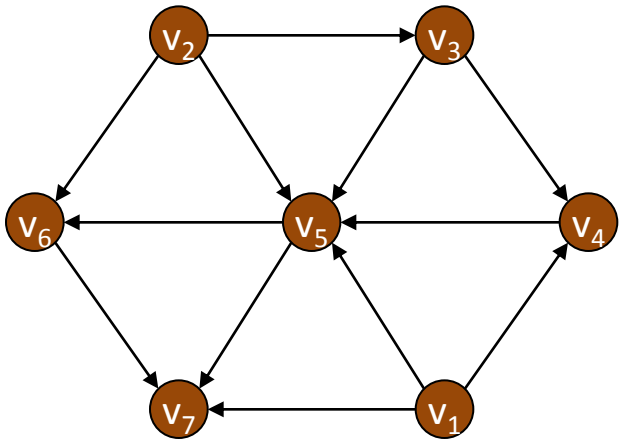


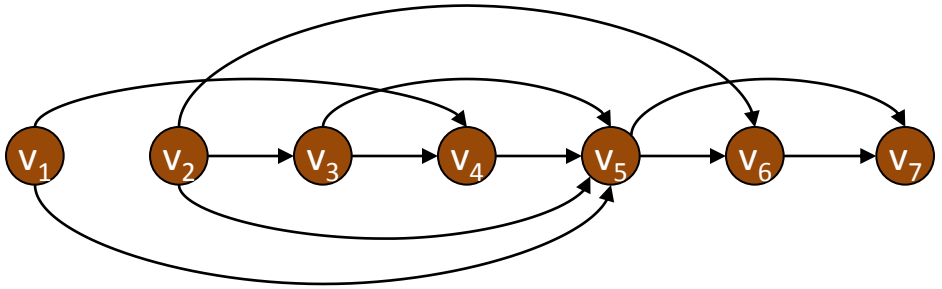


ADBEC

What about the following
ADEBC?
CBAED?
ABCDE?
DECAB?

One more example - Topological Ordering





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Algo

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

Algo – depth-first

```
L ← Empty list that will contain the sorted nodes
S ← Set of all nodes with no outgoing edges
for each node n in S do
    visit(n)
function visit(node n)
    if n has not been visited yet then
        mark n as visited
        for each node m with an edge from m to n do
            visit(m)
        add n to L
```