Chapter 3- Graph Algorithms

2023

Prepared by: Beimnet G.

Graph Algorithms

Representing and Searching graphs

Identifying the least-cost (least-weight) way of connecting vertices

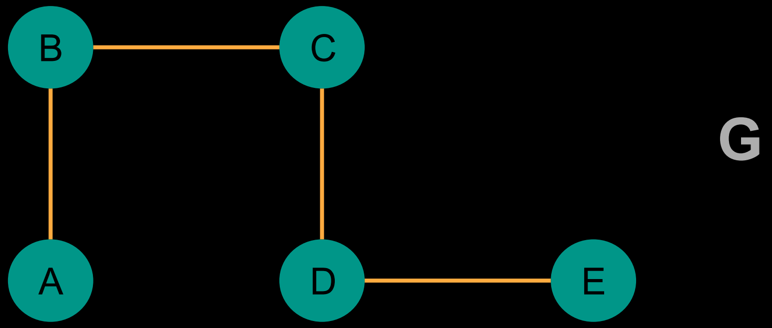
Identifying the shortest path between vertices

Graph

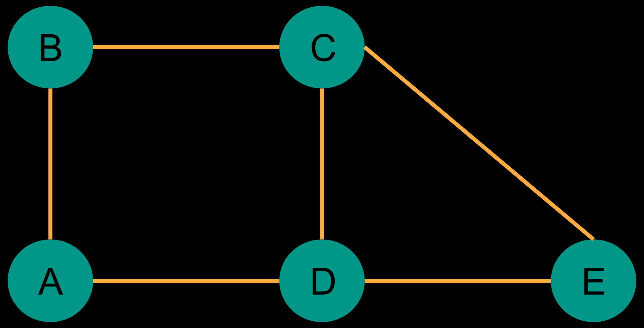
A **graph** (G) is a way of encoding pairwise relationships among a set of objects: it consists of a collection **V** of nodes (vertices) and a collection **E** of edges, each of which ”joins” two of the nodes.

Edge $e \in E$ is represented as a two-element subset of V : $e = u, v$ for some $u, v \in V$, where we call u and v the ends of e

Graph

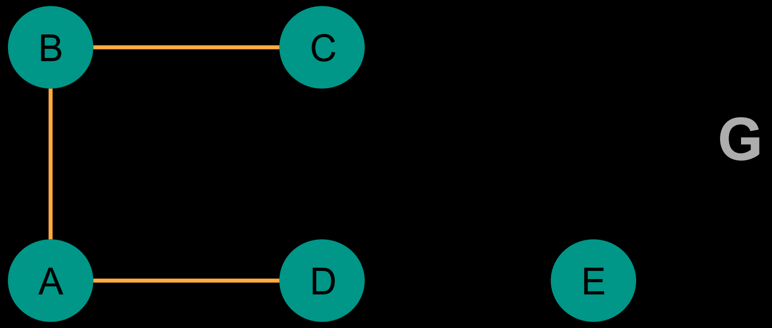


Graph

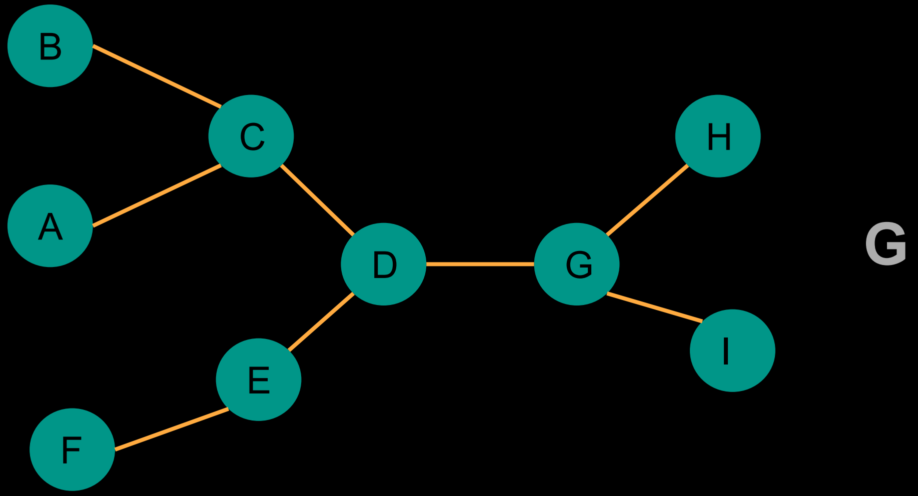


G

Graph



Graph



Edges

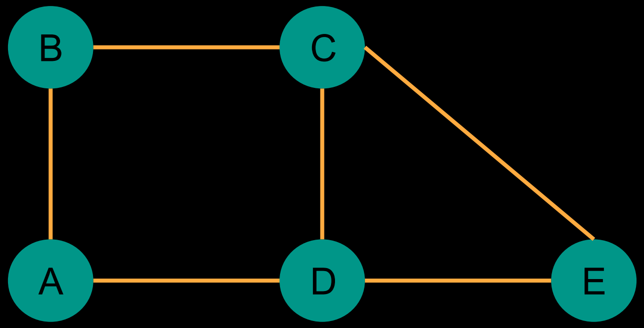
- Can represent symmetric or asymmetric relationships.

Edges

- Can represent symmetric or asymmetric relationships.

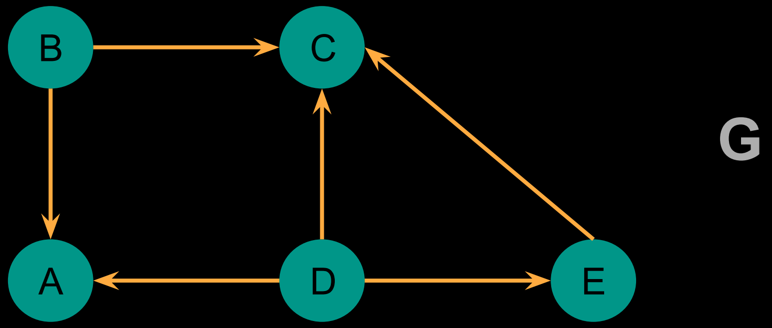
Directed vs Undirected Graphs

Undirected Graph



G

Directed Graph



Directed Relationships

A directed graph G’ consists of a set of nodes V and a set of directed edges E’. Each e’ $\in$ E’ is an **ordered pair** (u, v); in other words, the roles of u and v are not interchangeable.

u is the **tail** of the edge and v the **head**.

You can also say that edge e’ leaves node u and enters node v.

Note: You can generally assume a graph is undirected if it’s not specified otherwise.

Paths

A **path,** in an undirected graph $G = (V, E)$,

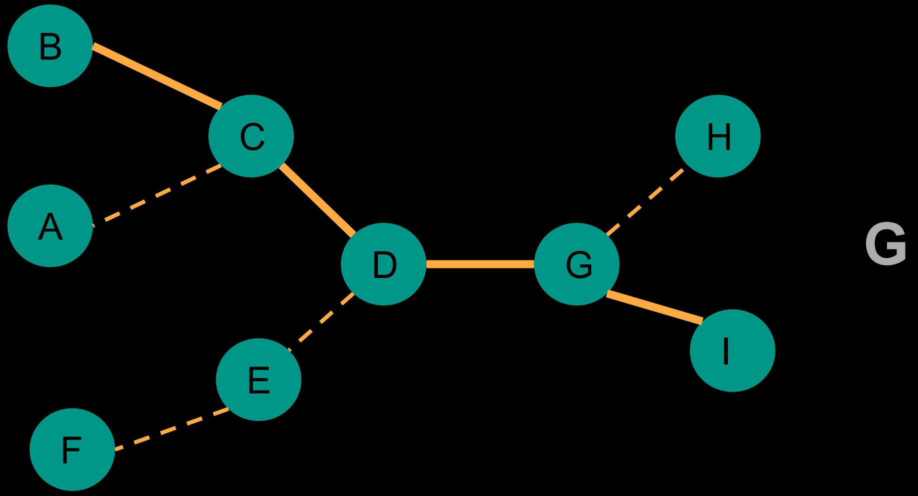
- sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k$ with the property that each consecutive pair v_i , v_{i+1} is joined by an edge in G.
- P is often called a path from v_1 to v_k, or a $v_1 - v_k$ path.

Paths

A path is called **simple** if all its vertices are distinct from one another.

A **cycle** is a path v_1, v_2, ..., v_{k-1}, v_k in which $k > 2$,the first k-1 nodes are all distinct, and $v_1 = v_k$

Simple Path



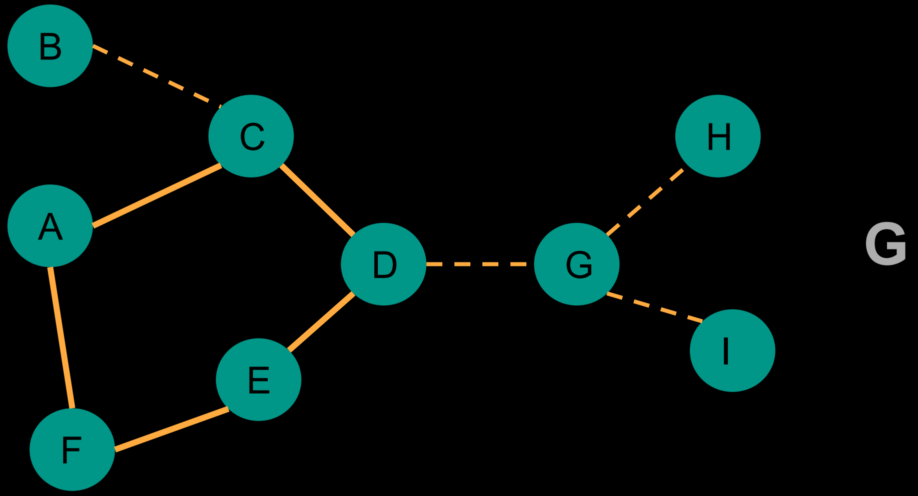
P= B → C→D→G→I

Paths

A path is called **simple** if all its vertices are distinct from one another.

A **cycle** is a path $v_1, v_2, ..., v_{k-1}, v_k$ in which $k > 2$,the first k-1 nodes are all distinct, and $v_1 = v_k$

Cycle



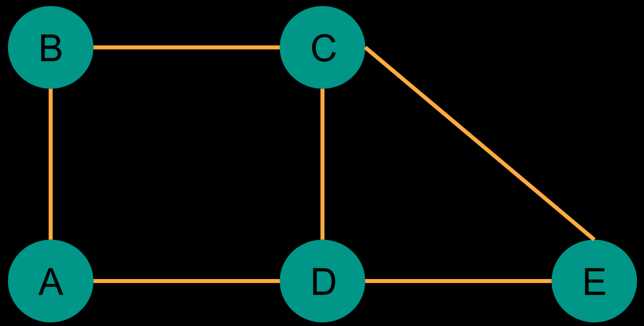
P= A → C→D→E→F→A

Graph

A **undirected graph is connected** if, for every pair of nodes u and v, there is a path from u to v.

A **directed graph is strongly connected** if, for every two nodes u and v, there is a path from u to v and a path from v to u.

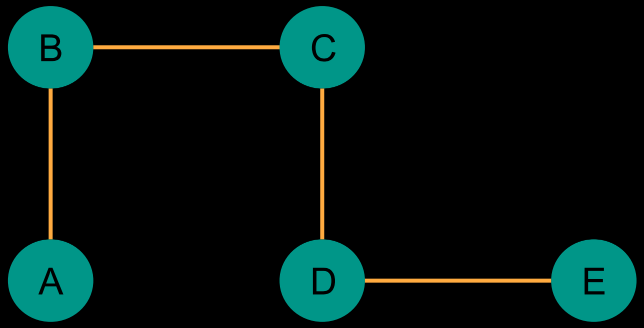
Graph



G

CONNECTED

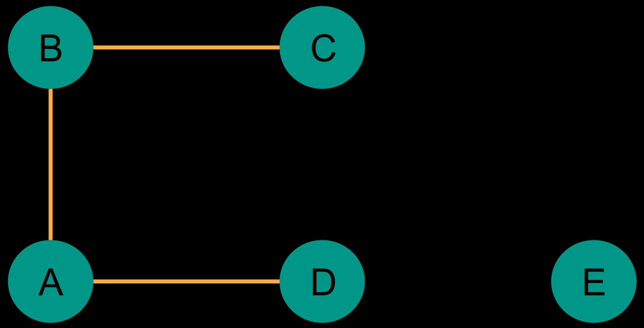
Graph



G

CONNECTED

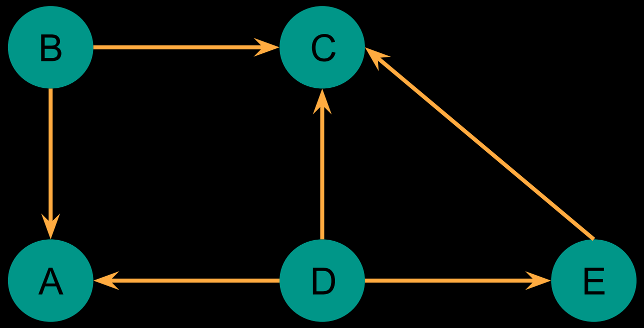
Graph



G

NOT CONNECTED

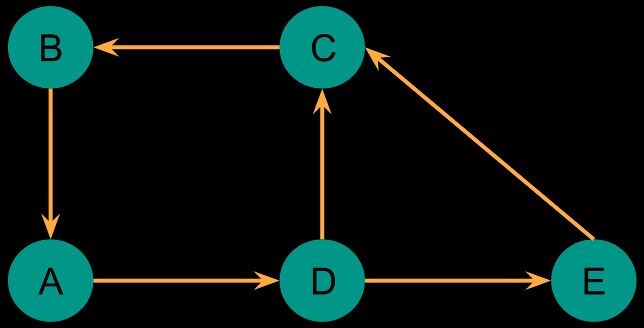
Directed Graph



G

NOT STRONGLY CONNECTED

Directed Graph



G

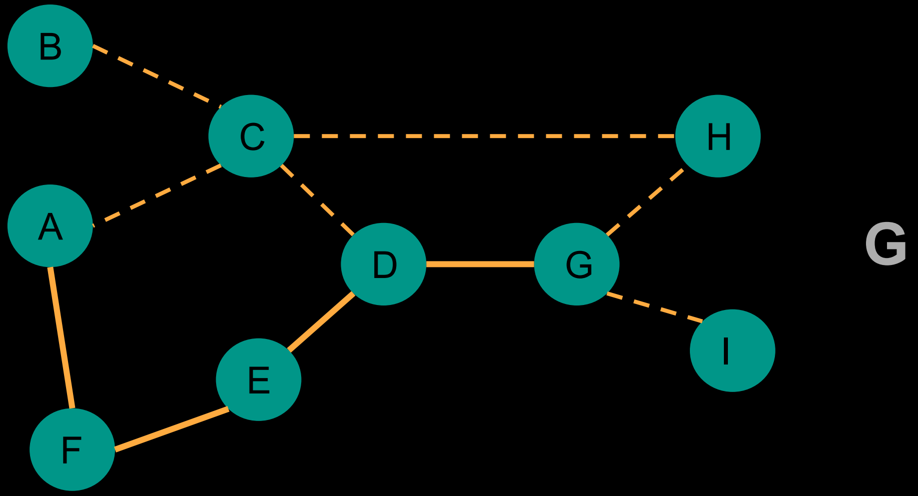
STRONGLY CONNECTED

Graph

Shortest Path is define as the distance between two nodes u and v to be the minimum number of edges in a $u - v$ path.

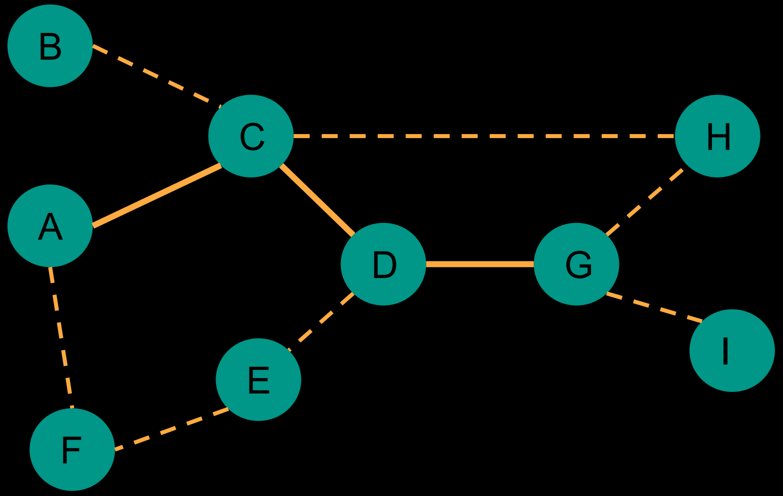
You can designate some symbol like ∞ to denote the distance between nodes that are not connected by a path.

Path



P= A→F→E→D→G

Path



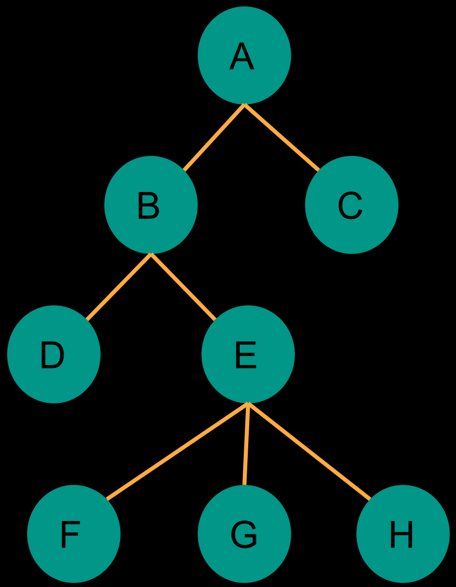
G

P= A→C→D→G

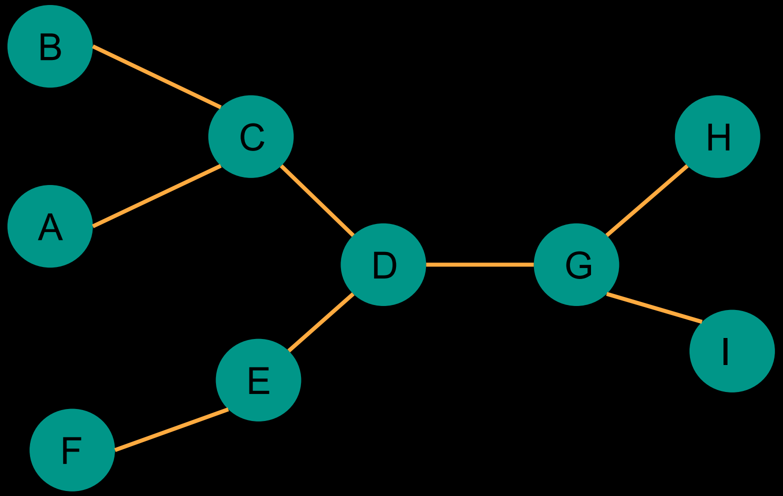
Tree

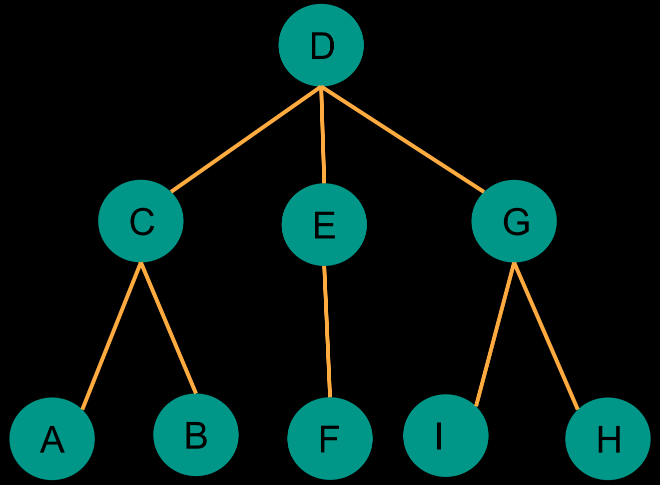
an undirected graph is a tree if it is connected and does not contain a cycle. Trees are the simplest kind of connected graph, deleting any edge from a tree will disconnect it.

Tree concepts: root, parent, child, descendant, leaf.



Introduction





Both figures represent the same graph. The second picture is a tree representation of the graph rooted at node D.

Tree

Statements:

Every n node tree has exactly n-1 edges

Let G be an undirected graph with n nodes. Any two of the following statements implies the third.

1. G is connected.
2. G does not contain a cycle.
3. G has $n - 1$ edges.

Real world examples of Graphs

Transportation Networks

Communication Networks

Information Networks (WWW)

Social Networks

Dependency Networks- University course management, software system

Graph Representation

2 standard graph representations for any graph G = (V, E):

- Adjacency lists
- Adjacency matrix

Adjacency List

provides a compact way to represent **sparse graphs** - those for which |E| is much less than $|V|^2$

It is usually the method of choice.

Adjacency Matrix

prefered when the **graph is dense** - $|E|$ is close to $|V|^2$

Or when we need to be able to tell quickly if there is an edge connecting two given vertices.

Adjacency List

The **adjacency-list representation** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V.

For each $u \in V$, the adjacency list Adj[u] contains all the vertices such that there is an edge $(u,v) \in E$. That is, Adj[u] consists of all the vertices adjacent to u in G.

Since the adjacency lists represent the edges of a graph, in pseudocode we treat the array Adj as an attribute of the graph, just as we treat the edge set E. In pseudocode, therefore, we will see notation such as G.Adj[u].

Adjacency List

If G is a directed graph, the sum of the lengths of all the adjacency lists is |E|, since an edge of the form (u,v) is represented by having v appear in Adj[u].

If G is an undirected graph, the sum of the lengths of all the adjacency lists is 2 |E|, since if (u,v) is an undirected edge, then u appears in v's adjacency list and vice versa.

The adjacency-list representation requires Θ(V+E) memory.

If there is a weight(cost) associated with an edge, the adjacency list representation can be adjusted to accommodate that.

Adjacency Matrix

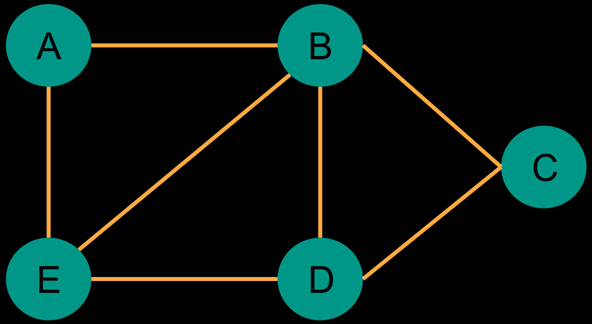
A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u,v) is present in the graph than to search for v in the adjacency list Adj[u].

An adjacency-matrix representation of the graph remedies this disadvantage, but at the cost of using asymptotically more memory.

For the adjacency-matrix representation of a graph G= (V ,E), we assume that the vertices are numbered 1, 2, 3,... |V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a |V| x |V| matrix A=(aij) such that

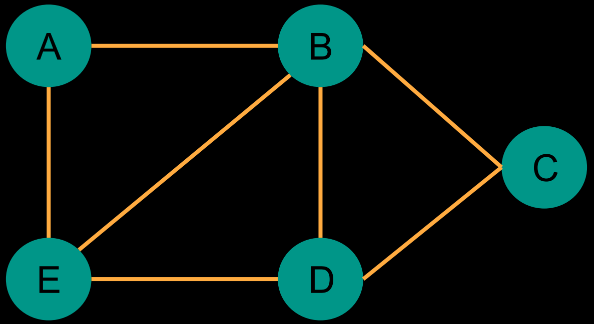
$$a_{ij} = \begin{cases} 1 \text{ if } (i,j) \in E, \\ 0 \text{ otherwise} \end{cases}$$

Graph Representation: Example



Represent this graph using an adjacency list.

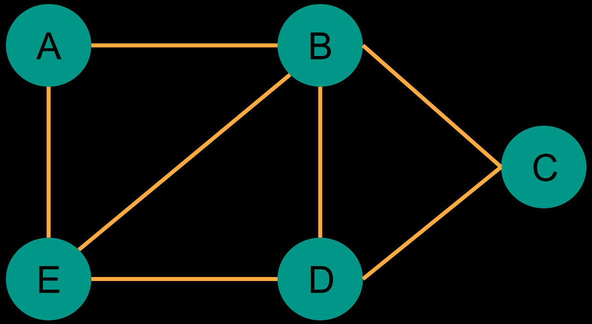
Graph Representation: Example



Adjacency list representation

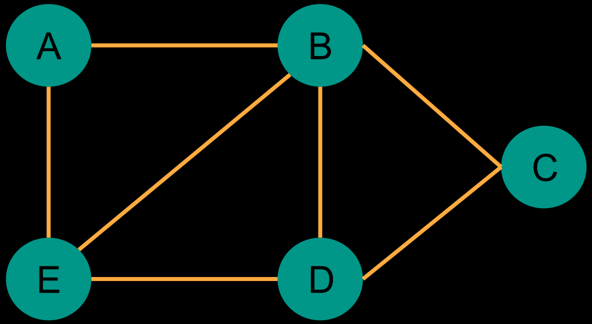
A	B	E		
B	A	C	D	E
C	B	D		
D	B	C	E	
E	A	B	D	

Graph Representation: Example



Represent this graph using an adjacency matrix.

Graph Representation: Example



Adjacency matrix representation.

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	1	1
C	0	1	0	1	0
D	0	1	1	0	1
E	1	1	0	1	0

Graph Representation

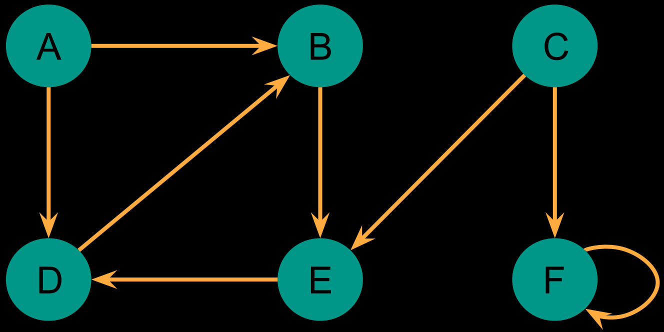
The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A_T$

In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

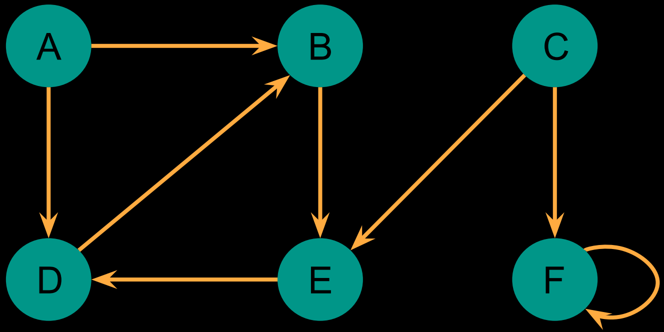
Like the adjacency-list representation of a graph, an adjacency matrix can be modified to represent a weighted graph.

Graph Representation: Example



Represent this graph using an adjacency list.

Graph Representation: Example



Represent this graph using an adjacency matrix.

Representing Attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges.

An attribute d of a vertex v is represented as v.d

For an edge (u,v) with an attribute f , you can denote the attribute as (u, v).f.

Graph Traversal

Graph Traversal

- For determining **node-to-node connectivity**.

Graph Traversal

Suppose we are given a graph $G = (V, E)$ and two particular nodes s and t. We'd like to find an efficient algorithm that answers the question:

- Is there a path from s to t in G? We will call this the problem of determining $s - t$ connectivity.

Two natural algorithms: **breadth-first search** (BFS) and **depth-first search** (DFS).

Breadth First Search (BFS)

Breadth First Search (BFS)

Given a graph G=(V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s.

It computes the distance (smallest number of edges) from s to each reachable vertex.

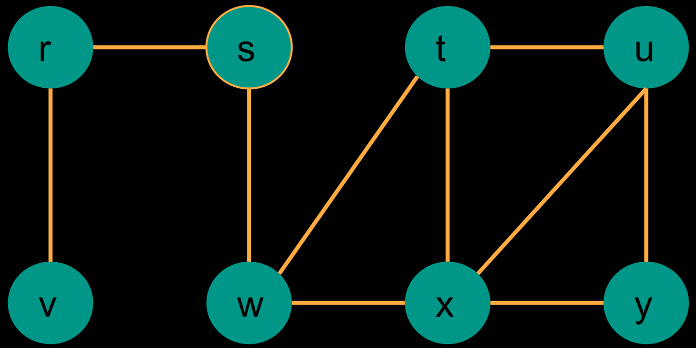
It also produces a “breadth-first tree” with root s that contains all reachable vertices. **For any vertex v reachable from s, the simple path in the breadth-first tree from s to corresponds to a “shortest path” from s to v in G**.

Breadth First Search (BFS)

```
BFS(G,s)
    for each vertex u E G.V
        u.color= WHITE
        u.d=∞
        u.π= NIL
    s.color=GRAY
    s.d=0
    s.π=NIL
    Q= ∅
    ENQUEUE(Q,s)
    while Q != ∅
        u=DEQUEUE(Q)
        for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.d= u.d +1
                v.π= u
               ENQUEUE(Q,v)
        u.color=BLACK
```

Breadth First Search (BFS)

Given the following graph, run the BFS procedure on it starting from node s.



Breadth First Search (BFS): Connected Component

The set of nodes discovered by the BFS algorithm is **the connected component of G containing s**;

BFS is just one possible way to produce this component. We can build the component R by ”exploring” G in any order, starting from s.

Breadth First Search (BFS): Shortest Path

Breadth-first search finds the distance to each reachable vertex in a graph $G(V,E)$ from a given source vertex $s \in V$.

We define the shortest-path distance as $\delta(s,v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v then $\delta(s,v)= \infty$

For a graph $G=(V,E)$ and an arbitrary vertex $s \in V$: for any edge $(u,v) \in E$, $\delta(s,v) <= \delta(s,u)+1$.

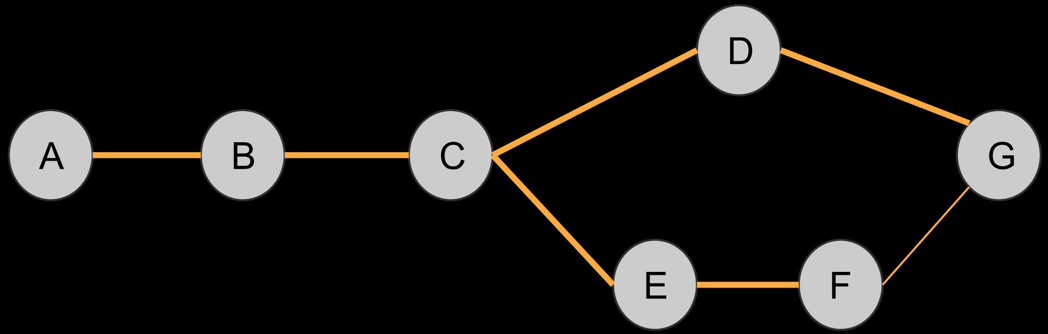
If vertices V_i and V_j are enqueued during the execution of BFS, and that V_i is enqueued before V_j . Then $V_i.d <= V_j.d$ at the time that V_j is enqueued.

Breadth First Search (BFS)

Model Problems

1. Single_Pair_Reachability (G,s,t)
    - Is there a path in G from s to t?
2. Single_Pair_Shortest_Path(G,s,t)
    - What is the shortest distance from s to t in G? And what is the path?
3. Single_Source_Reachability(G,s)
    - What is the distance from s to all vertices? And what is the path?

Breadth First Search (BFS)



Breadth First Tree

After you've determined the distance from s to a vertice, how do you store it?

And how do you store the path?

Solution: **Breadth first tree** (shortest path tree).

In a breadth first tree, you can construct the shortest path from each node to s by recursively identifying the parent of each vertex.

Question:

- How are we sure it's a tree?
- What happens if you add a path from A to D?

Depth First Search (DFS)

Depth First Search (DFS)

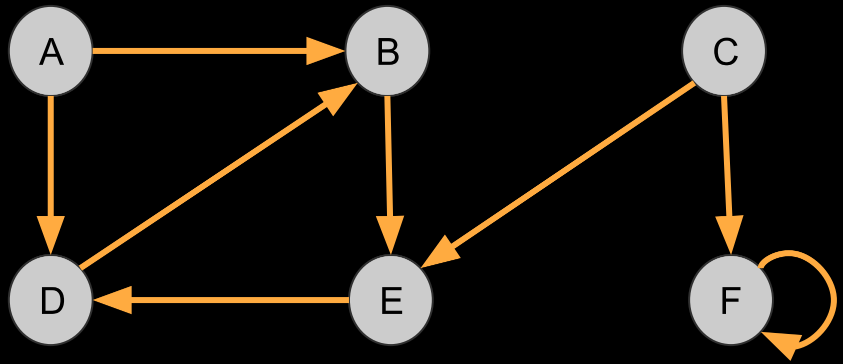
A recursive algorithm.

It searches “deeper” in the graph whenever possible and backtracks when it can’t go any deeper.

Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it **until it has covered every vertex**.

A depth first search may result in **several trees.**

Depth First Search (DFS)



Depth First Search (DFS): Algorithm

```
DFS_VISIT(u)
    time=time+1
    u.d=time
    u.color=GRAY
    for each v in Adj[u]:
        if v.π ==NIL
            V.π = u
            DFS_VISIT(v)
    u.color=BLACK
    time=time+1
    u.f=time
```

```
DFS(G)
    for each u in G.V
        u.color=WHITE
        u.π = NIL
    time=0
    for each u in G.V
        if u.π ==NIL
            DFS_VISIT(u)
```

Depth First Search (DFS)

Will not find the shortest path!

DFS has other applications.

DFS is useful in **edge classification**.

Edge Classification

Every edge in a graph G gets visited at least once in a DFS (twice if it’s an undirected graph)

What happens when you visit an edge?

Depth First Search (DFS): Edge Classification

- Tree Edge: a visit may lead to something unvisited resulting in a tree edge.
- Forward Edge: connects a node to its descendant
- Backward Edge: connects a node to its ancestor
- Cross Edge: any other node that connects edges that have no herricical relation

Question: Which one of these edges do you think exist in an undirected graph?

Depth First Search (DFS): Edge Classification

Applications of Edge Classification:

- Cycle detection
- Topological Sort

Depth First Search (DFS): Cycle Detection

A graph G has a cycle ⇔ DFS (G) has a back edge.

Proof:

⇐ side proof: a back edge implies a cycle by definition.

⇒ side proof: there is a cycle in a path spanning vertices v1,v2,v3… vk if there exists an edge (vk,v1). In a DFS edge (vk,v1) will be a back edge

Depth First Search (DFS): Topological Sort

Job Scheduling: given a directed acyclic graph (DAG) , order the vertices so that all edges point from lower order to higher order vertices.

Topological Sort solves this problem

Topological sort uses DFS to order these nodes in a graph.

Algorithm:

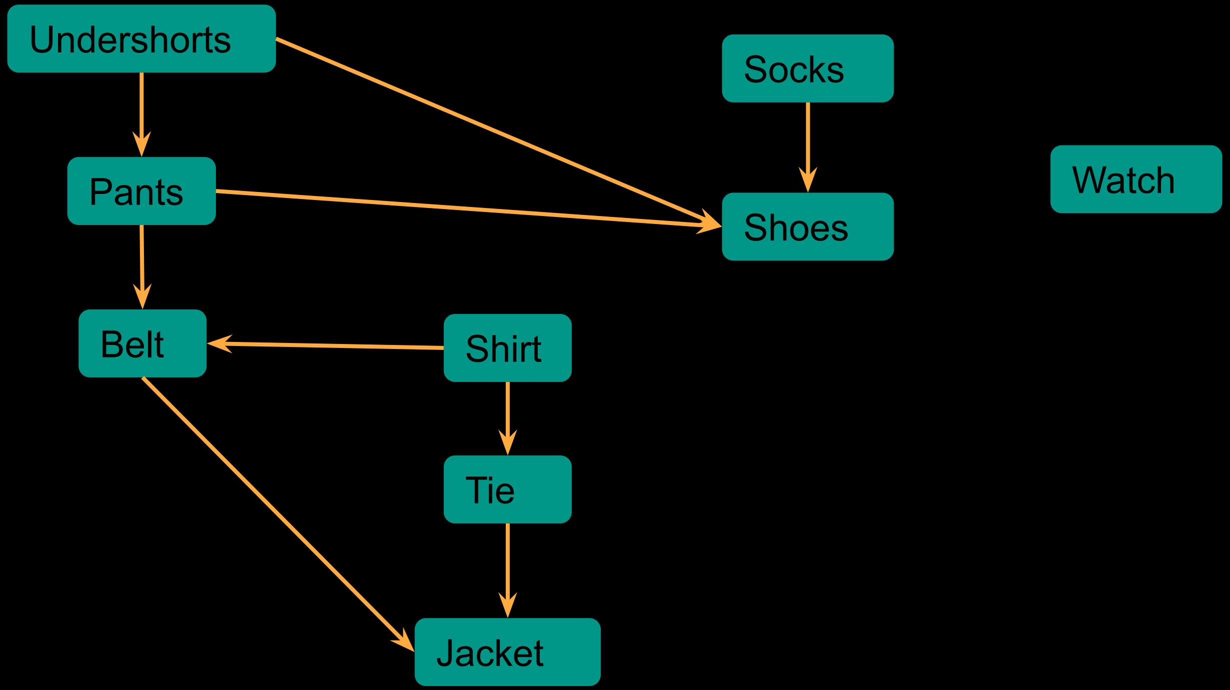
Topological-Sort (G):

run DFS(G)

Output the reverse of finishing time (i.e when a node finishes add it to the front of a list)

Depth First Search (DFS): Topological Sort

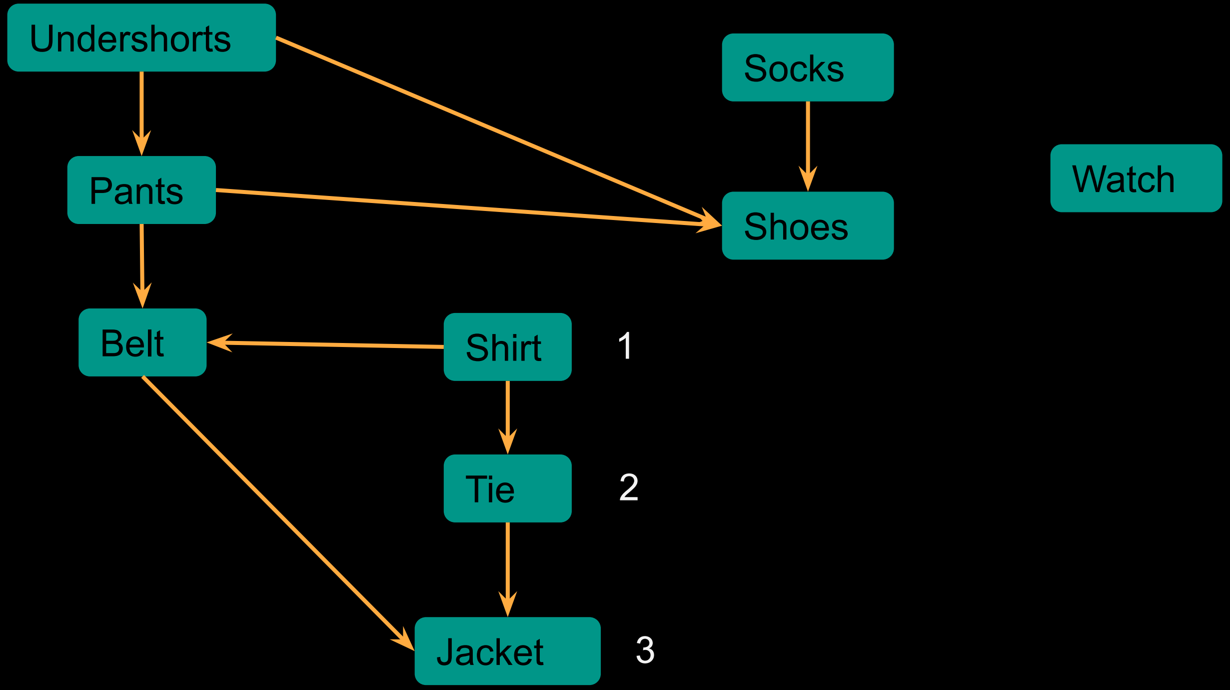
Example: This is how Professor Bumstead gets dressed in the morning.



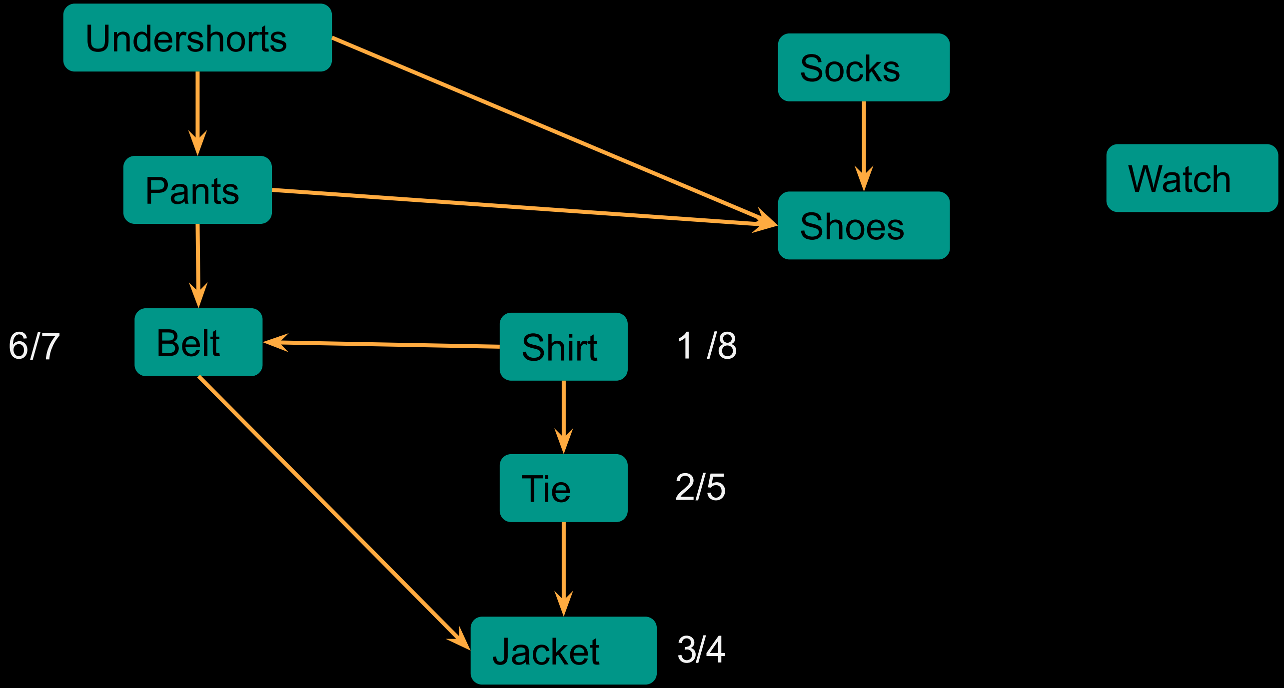
If there is an edge (u,v) then u appears before v in the ordering. That is, u must be done before v can be done.

$u \longrightarrow v$

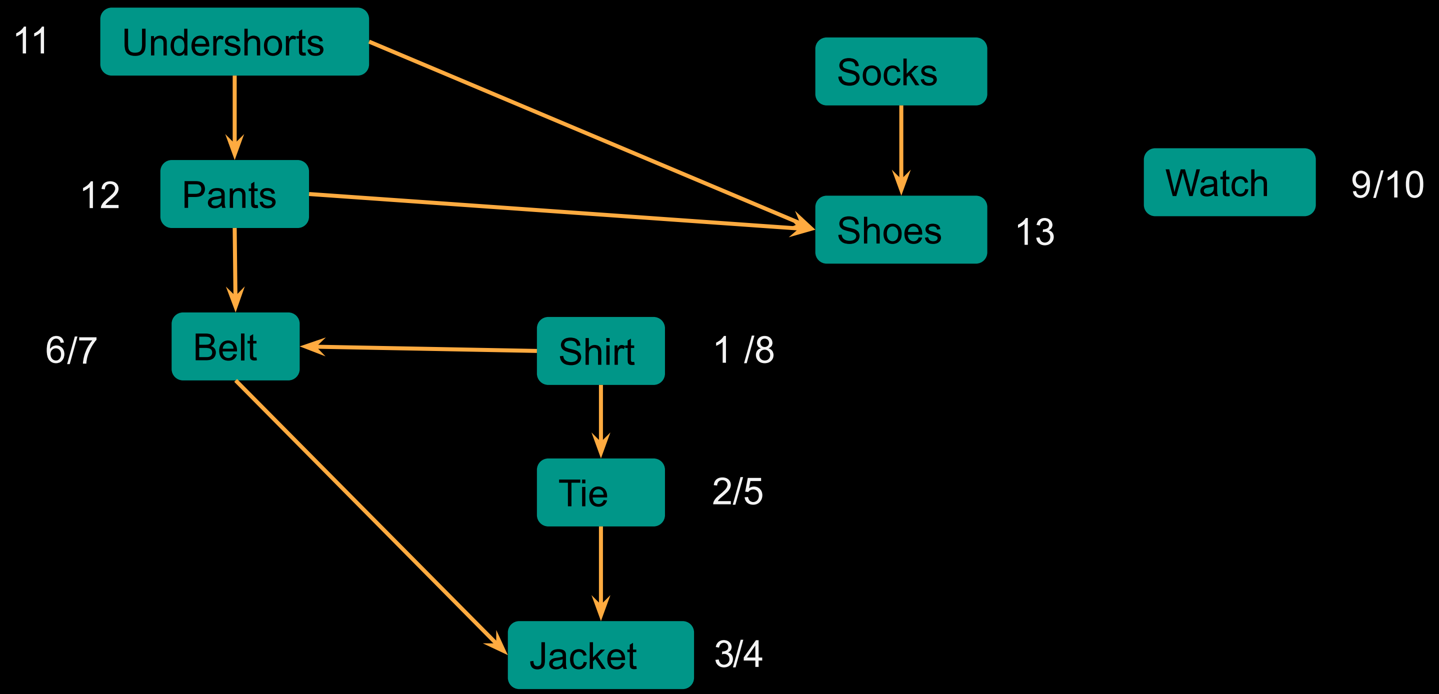
Depth First Search (DFS): Topological Sort



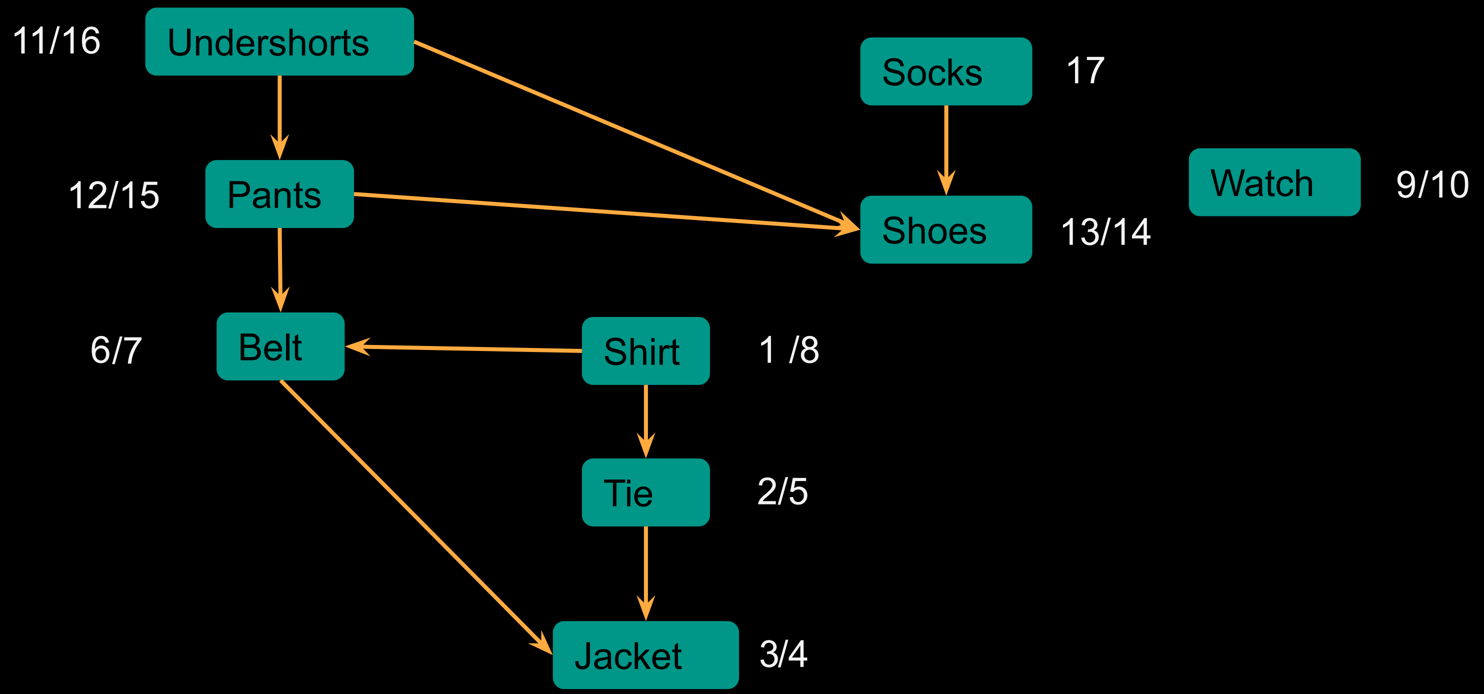
Depth First Search (DFS): Topological Sort



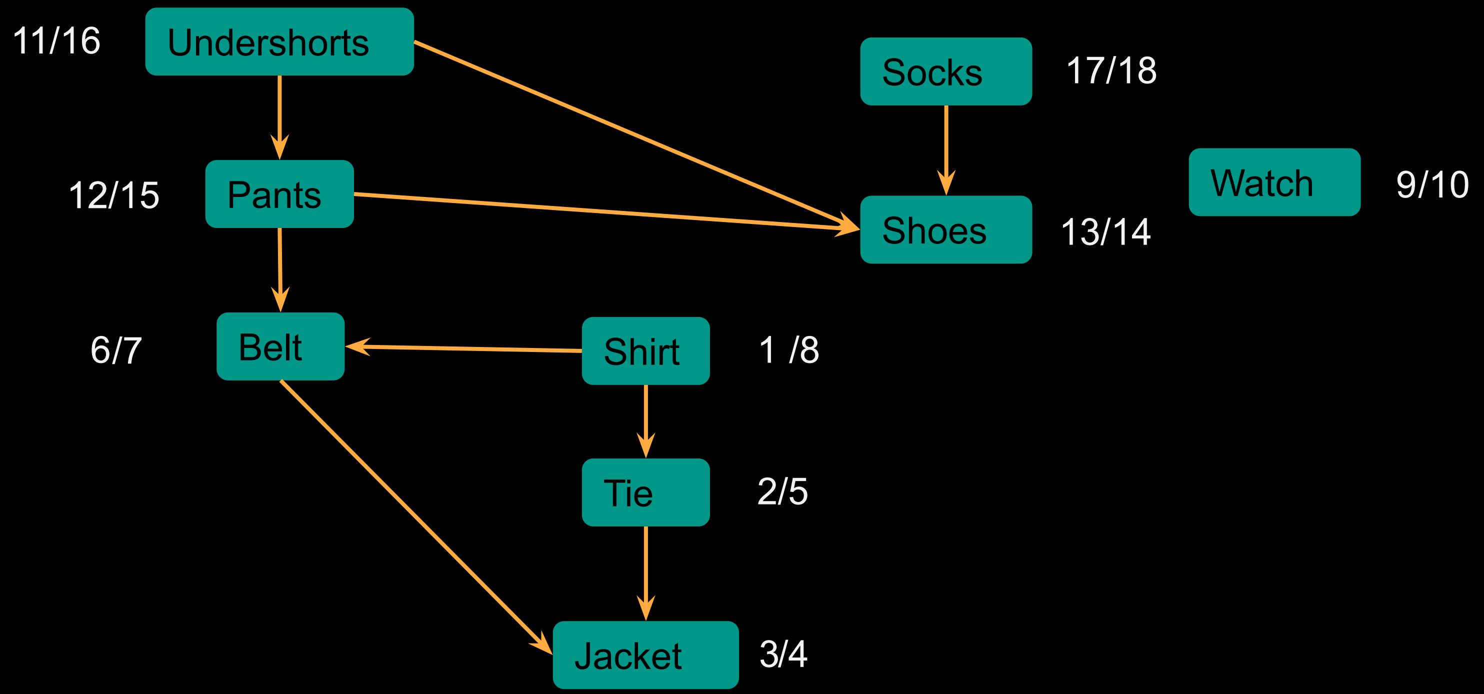
Depth First Search (DFS): Topological Sort



Depth First Search (DFS): Topological Sort



Depth First Search (DFS): Topological Sort



Topological Sort: Proof

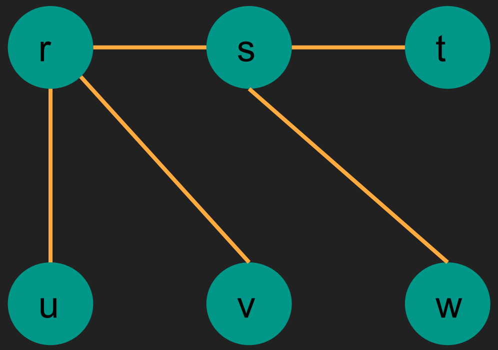
Proof that topological sort gives a valid job scheduling solution.

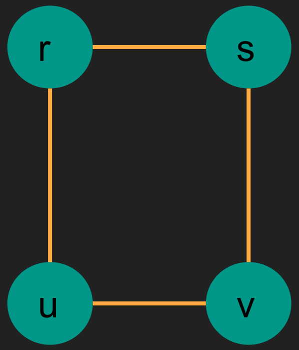
For it to be correct:

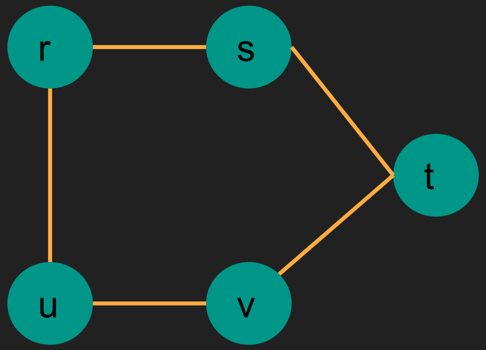
for any edge (u,v), v finishes before u finishes.

Case 1: u starts before v

Case 2: v starts before u


r
s
t
u
v
w


r
s
u
v


r
s
t
u
v