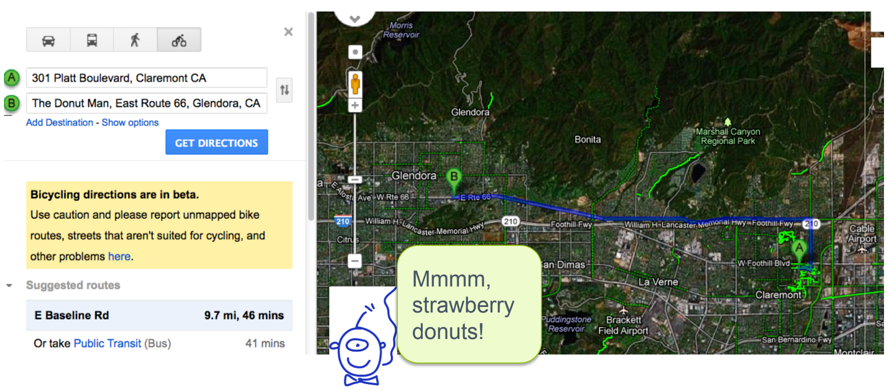
Graph Algorithms!

Slides adapted from Ran Libeskind-Hadas,David Kauchak

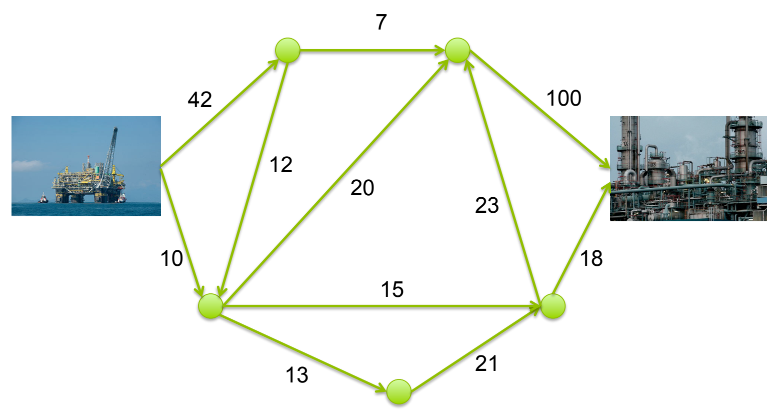
Why Graphs?

- Shortest paths (Google Maps)
- Networks (social networks, computer networks)
- Problems that don't even "look" graph theoretic!
  - Clustering of data (recommendation systems)
  - Optimization problems (Space shuttle payloads)

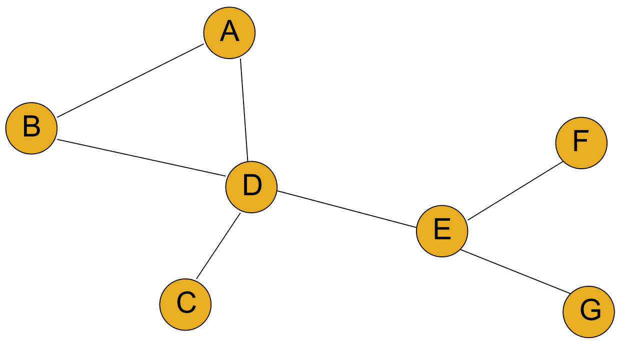
Shortest Paths



Network Flow



What is a graph?



Types / Characteristics

- Undirected:
- Directed:
- Weighted:

Terminology

- Path:
- Cycle:
- Connected:
- Strongly-Connected

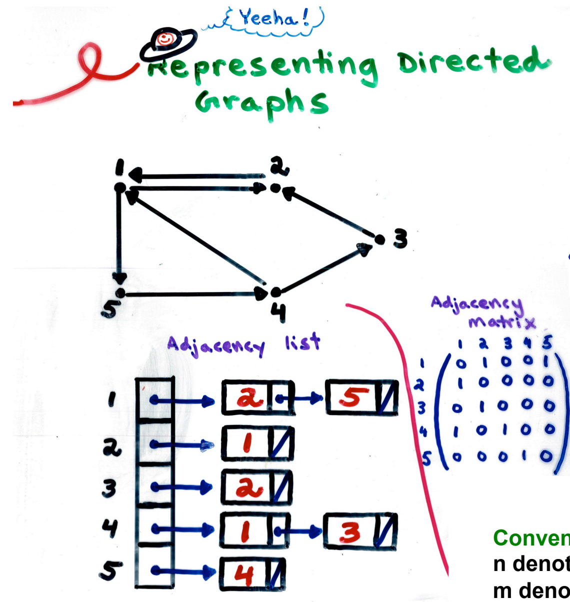
Terminology

- Tree:
- Dag:
- Complete:
- Bipartite:

Representing graphs

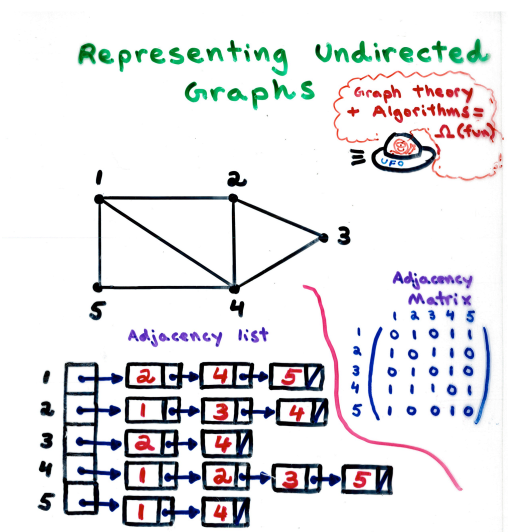
- Adjacency List:

- Adjacency Matrix:



What are the relative merits of these two representations?

Convention:
n denotes the number of vertices
m denotes the number of edges



Convention:
n denotes the number of vertices
m denotes the number of edges

Adjacency list vs. adjacency matrix

Adjacency list	Adjacency matrix

Other Representations

- Sparse:

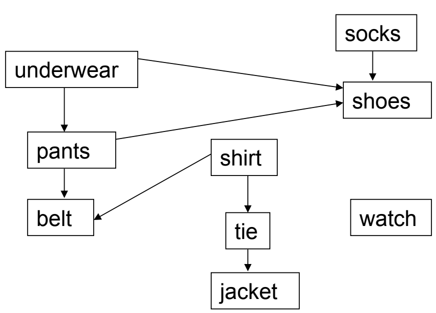
- Weighted:

Graph algorithms/questions

- Graph traversal (BFS, DFS)
- Shortest path from a to b
  - unweighted
  - weighted positive weights
  - negative/positive weights
- Minimum spanning trees

DAGs

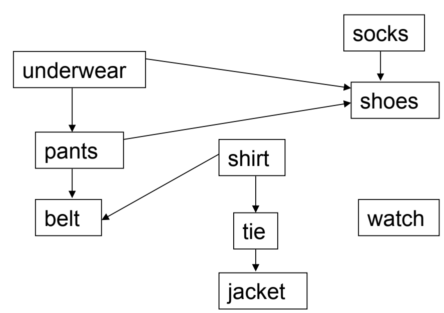
Can represent dependency graphs

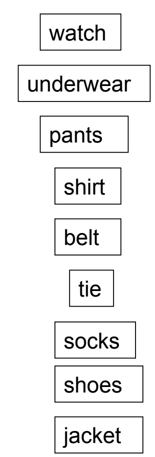


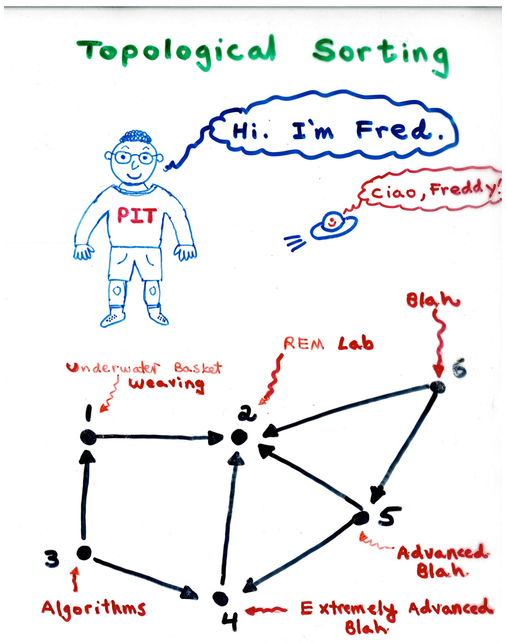
Topological sort

A linear ordering of all the vertices such that for all edges $(u,v) \in E$, u appears before v in the ordering

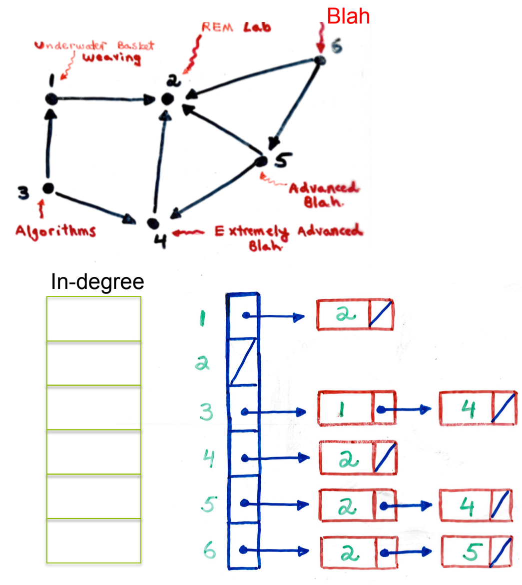
An ordering of the nodes that "obeys" the dependencies, i.e. an activity can't happen until it's dependent activities have happened







Worksheet: Describe an algorithm in clear English and give its running time as a function of n (number of vertices) and m (number of edges).

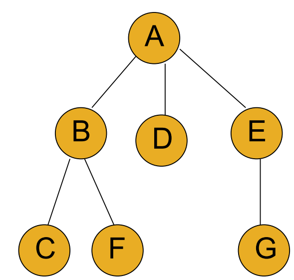


Breadth First Search (BFS) on Trees

```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
              Enqueue(Q, c)
```

Tree BFS

```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
              Enqueue(Q, c)
```



Q:

Running time of Tree BFS

Adjacency list

- How many times does it visit each vertex?
- How many times is each edge traversed?

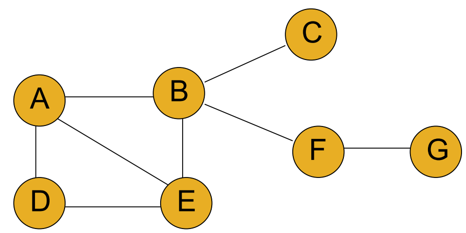
Adjacency matrix

- For each vertex visited, how much work is done?

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

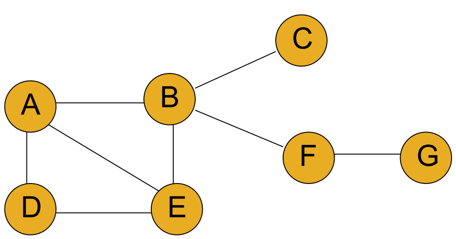
BFS for graphs

What needs to change for graphs?




```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

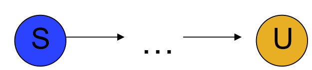


Is BFS correct?

Does it visit all nodes reachable from the starting node?

Can you prove it?

Assume we "miss" some node 'u', i.e. a path exists, but we don't visit 'u'



Runtime of BFS

Adjacency list:

Adjacency matrix:

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Depth First Search (DFS)

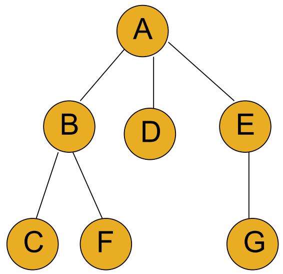
```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

Depth First Search (DFS)

```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

Tree DFS



DFS on graphs

DFS(G)

```
for all v ∈ V
        visited[u] ← false
for all v ∈ V
        if !visited[v]
                DFS-Visit(v)
```

DFS-Visit(u)

```
visited[u] ← true
PreVisit(u)
for all edges (u, v) ∈ E
        if !visited[v]
                DFS-Visit(v)
PostVisit(u)
```

What does DFS do?

Finds connected components

Each call to DFS-Visit from DFS starts exploring a new set of connected components

Helps us understand the structure/connectedness of a graph

Is DFS correct?

Does DFS visit all of the nodes in a graph?

DFS(G)

```
for all v ∈ V
        visited[u] ← false
for all v ∈ V
        if !visited[v]
                DFS-Visit(v)
```

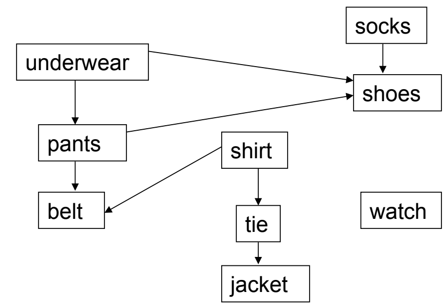
Running time?

Like BFS

- Visits each node exactly once
- Processes each edge exactly twice (for an undirected graph)

DAGs

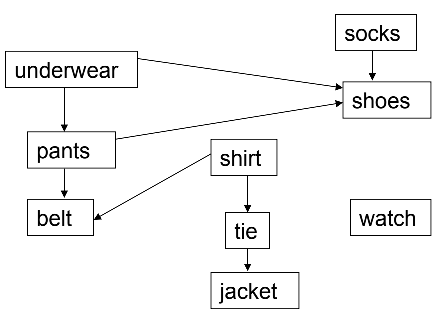
Can represent dependency graphs

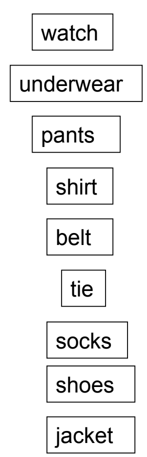


Topological sort

A linear ordering of all the vertices such that for all edges (u,v) ∈ E, u appears before v in the ordering

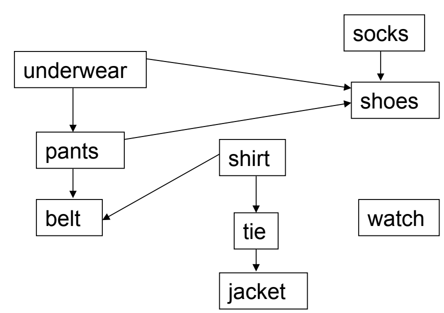
An ordering of the nodes that "obeys" the dependencies, i.e. an activity can't happen until it's dependent activities have happened





Topological sort

```
TOPOLOGICAL-SORT1(G)
  Find a node v with no incoming edges
  Delete v from G
  Add v to linked list
  TOPOLOGICAL-SORT1(G)
```



Running time?

```
TOPOLOGICAL-SORT1(G)
  Find a node v with no incoming edges
  Delete v from G
  Add v to linked list
  TOPOLOGICAL-SORT1(G)
```

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
  for all edges (u, v) ∈ E
          active[v] ← active[v] + 1
  for all v ∈ V
          if active[v] = 0
                  ENQUEUE(S, v)
  while !EMPTY(S)
          u ← DEQUEUE(S)
          add u to linked list
          for each edge (u, v) ∈ E
                  active[v] ← active[v] − 1
                  if active[v] = 0
                          ENQUEUE(S, v)
```

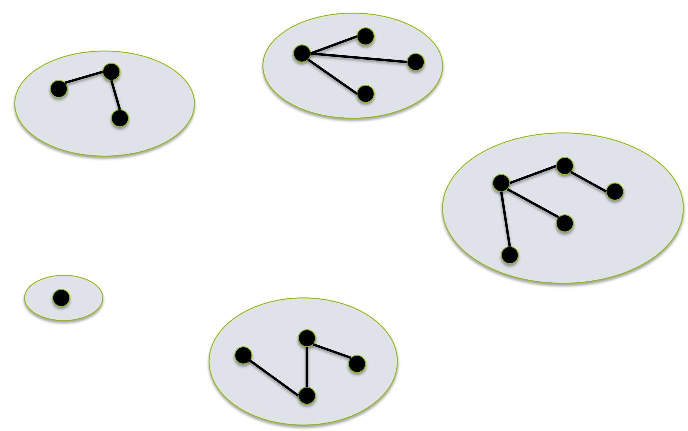
Running time?

How many times do we process each node?

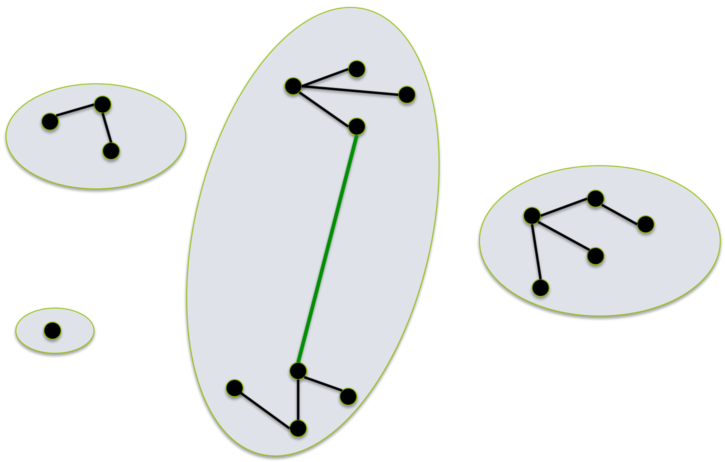
How many times do we process each edge?

```
TOPOLOGICAL-SORT2(G)
  for all edges (u, v) ∈ E
          active[v] ← active[v] + 1
  for all v ∈ V
          if active[v] = 0
                  ENQUEUE(S, v)
  while !EMPTY(S)
          u ← DEQUEUE(S)
          add u to linked list
          for each edge (u, v) ∈ E
                  active[v] ← active[v] − 1
                  if active[v] = 0
                          ENQUEUE(S, v)
```

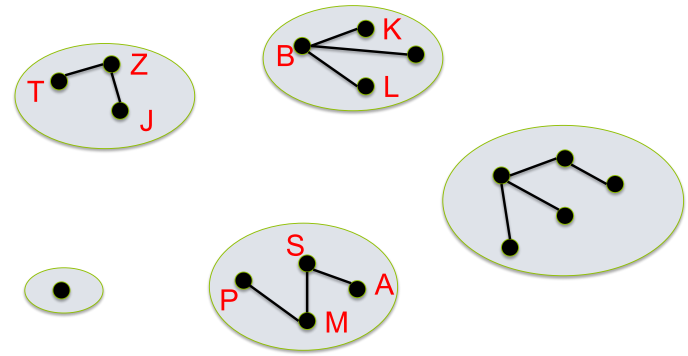
The Union-Find ADT!



The Union-Find ADT!



The Union-Find ADT!




```
makeSet(x)  # make a singleton set with element x
findSet(x)  # find the name of the set containing x
unionSets(x, y)     # merge the sets containing
                    # x and y into a single new set
```

Connectedness

Given an undirected graph, for every node u ∈ V, can we reach all other nodes in the graph?

Running time:

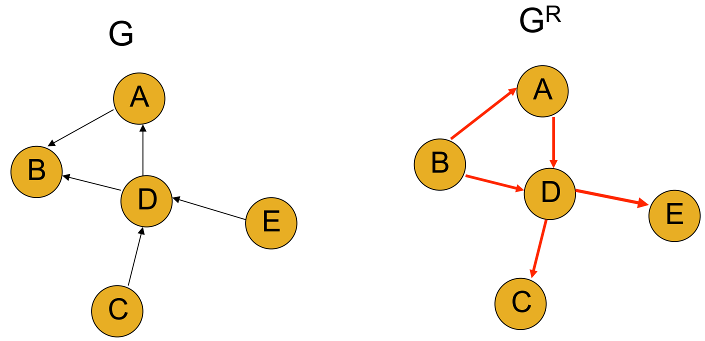
Strongly connected

Given a directed graph, can we reach any node v from any other node u?

Ideas?

Transpose of a graph

Given a graph G, we can calculate the transpose of a graph G^R by reversing the direction of all the edges



Running time to calculate G^R?

Strongly connected

STRONGLY-CONNECTED(G)

Run DFS or BFS from some node u
if not all nodes are visited
  return false
Create graph G^R by reversing all edge dirctions
Run DFS or BFS on G^R from node u
if not all nodes are visited
  return false
return true

Is it correct?

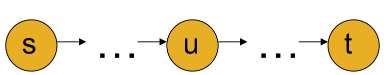
What do we know after the first pass?

– Starting at u, we can reach every node

What do we know after the second pass?

– All nodes can reach u. Why?
– We can get from u to every node in G^R, therefore, if we reverse the edges (i.e. G), then we have a path from every node to u

Which means that any node can reach any other node. Given any two nodes s and t we can create a path through u



Runtime?

STRONGLY-CONNECTED(G)

Run DFS or BFS from some node u
if not all nodes are visited
  return false
Create graph G^R by reversing all edge dirctions
Run DFS or BFS on G^R from node u
if not all nodes are visited
  return false
return true