Graph: representation and traversal

CISC5835, Computer Algorithms
CIS, Fordham Univ.

Instructor: X. Zhang

Acknowledgement

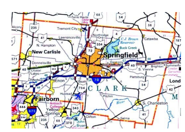
- The set of slides have use materials from the following resources
  - Slides for textbook by Dr. Y. Chen from Shanghai Jiaotong Univ.
  - Slides from Dr. M. Nicolescu from UNR
  - Slides sets by Dr. K. Wayne from Princeton
    - which in turn have borrowed materials from other resources

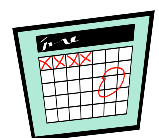
Outline

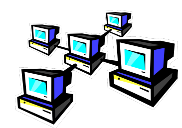
- **Graph Definition**
- **Graph Representation**
- **Path, Cycle, Tree, Connectivity**
- **Graph Traversal Algorithms**
  - **Breath first search/traversal**
  - **Depth first search/traversal**
  - …
- **Minimal Spaning Tree algorithms**
- **Dijkstra algorithm: shortest path a**

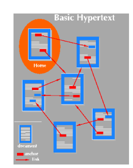
Graphs

- Applications that involve not only a set of items, but also the connections between them


Maps

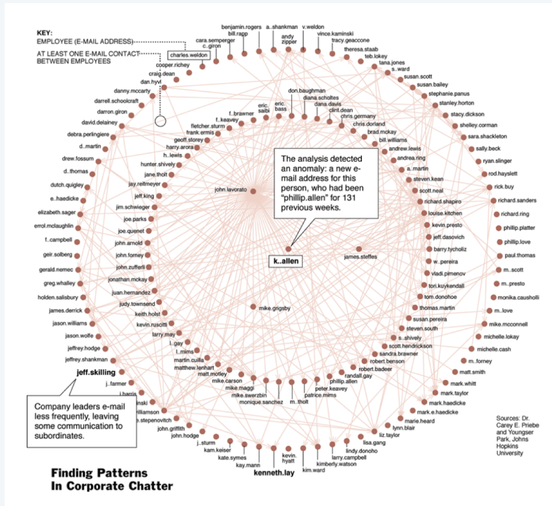

Schedules


Computer networks


Hypertext


Circuits

One week of Enron emails



Some graph applications

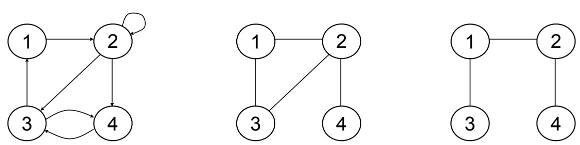
graph	node	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

Graphs - Background

Graphs = a set of nodes (vertices) with edges (links) between them.

Notations:

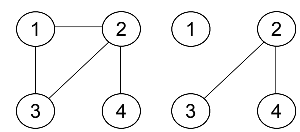
- G = (V, E) - graph
- V = set of vertices (size of V = n)
- E = set of edges (size of E = m)



Directed graph | Undirected graph | Acyclic graph

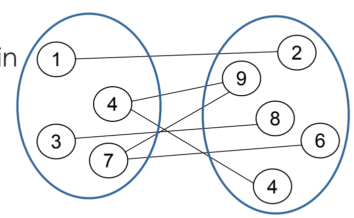
Other Types of Graphs

- A graph is **connected** if there is a path between every two vertices



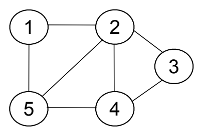
Connected | Not connected

- A **bipartite graph** is an undirected graph G = (V, E) in which $V = V_1 + V_2$ and there are edges only between vertices in V_1 and V_2

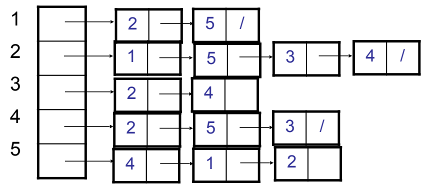


Graph Representation

- **Adjacency list representation** of G = (V, E)
  - An array of n lists, one for each vertex in V
  - Each list `Adj[u]` contains all the vertices v such that there is an edge between **u** and **v**
    - `Adj[u]` contains the vertices adjacent to **u** (in arbitrary order)
  - Can be used for both directed and undirected graphs

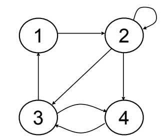


Undirected graph

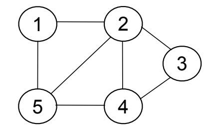


Properties of Adjacency List Representation

- Sum of the lengths of all the adjacency lists
  - Directed graph: size of E (m)
    - Edge (**u**, **v**) appears only once in u's list
  - Undirected graph: 2* size of E (2m)
    - **u** and **v** appear in each other's adjacency lists: edge (**u**, **v**) appears twice



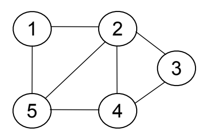
Directed graph



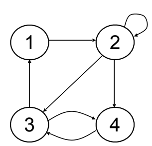
Undirected graph

Properties of Adjacency List Representation

- Memory required
  - Θ(m+n)
- Preferred when
  - the graph is sparse: $m << n^2$
- Disadvantage
  - no quick way to determine whether there is an edge between node u and v
  - Time to determine if (u, v) exists: O(degree(u))
- Time to list all vertices adjacent to u:
  - Θ(degree(u))



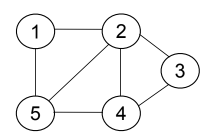
Undirected graph



Directed graph

Graph Representation

- **Adjacency matrix representation** of G = (V, E)
  - Assume vertices are numbered 1, 2, … n
  - The representation consists of a matrix A_{nxn}
  - $a_{ij} = \begin{cases} 1 & \text{if (i, j) belongs to E, if there is edge (i,j)} \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

For undirected graphs matrix A is symmetric:

$a_{ij} = a_{ji}$

$A = A^T$

Properties of Adjacency Matrix Representation

- Memory required
  - $\Theta(n^2)$, independent on the number of edges in G
- Preferred when
  - The graph is dense: m is close to n^2
  - need to quickly determine if there is an edge between two vertices
- Time to list all vertices adjacent to u:
  - $\Theta(n)$
- Time to determine if (u, v) belongs to E:
  - $\Theta(1)$

Weighted Graphs

- **Weighted graphs** = graphs for which each edge has an associated weight **w(u, v)**

  w: *E* -> R, weight function

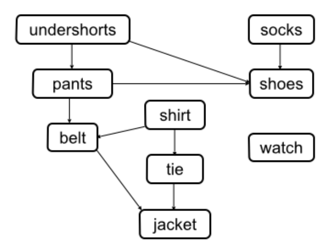
- Storing the weights of a graph
  - Adjacency list:
    - Store **w(u,v)** along with vertex **v** in **u**'s adjacency list
  - Adjacency matrix:
    - Store **w(u, v)** at location **(u, v)** in the matrix

NetworkX: a Python graph library

- http://networkx.github.io/
- **Node:** any hashable object as a node. Hashable objects include strings, tuples, integers, and more.
- **Arbitrary edge attributes:** weights and labels can be associated with an edge.
- internal data structures: based on an adjacency list representation and uses Python dictionary.
  - **adjacency structure:** implemented as *a dictionary of dictionaries*
    - *top-level* (outer) dictionary: keyed by nodes to values that are themselves dictionaries keyed by neighboring node to edge attributes associated with that edge.
  - Support: fast addition, deletion, and lookup of nodes and neighbors in large graphs.
- underlying datastructure is accessed directly by methods

Graphs Everywhere

- Prerequisite graph for CIS undergrad courses
- Three jugs of capacity 8, 5 and 3 liters, initially filled with 8, 0 and 0 liters respectively. How to pour water between them so that in the end we have 4, 4 and 0 liters in the three jugs?
- what's this graph representing?



Outline

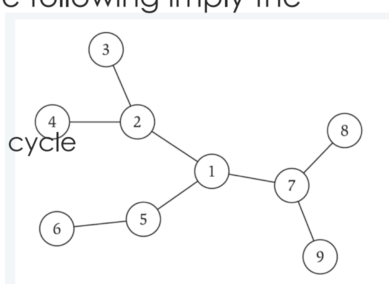
- **Graph Definition**
- **Graph Representation**
- **Path, Cycle, Tree, Connectivity**
- **Graph Traversal Algorithms**
  - **basis of other algorithms**

Paths

- A **Path** in an undirected graph G=(V,E) is a sequence of nodes $v_1,v_2,\ldots,v_k$ with the property that each consecutive pair v_{i-1}, v_i is joined by an edge in E.
- A path is **simple** if all nodes in the path are distinct.
- A cycle is a path $v_1,v_2,\ldots,v_k$ where $v_1=v_k$, $k>2$, and the first k-1 nodes are all distinct
- An undirected graph is **connected** if for every pair of nodes u and v, there is a path between u and v

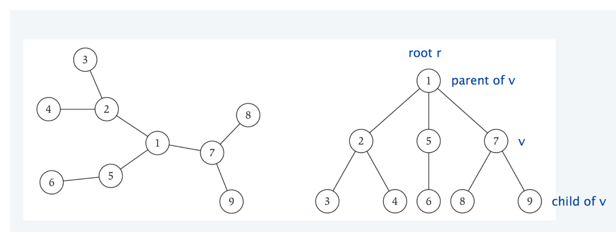
Trees

- A **undirected graph is a tree if it is connected and does not contain a cycle.**
- **Theorem**: Let G be an undirected graph on n nodes. Any two of the following imply the third.
  - G is connected
  - G does not contain a cycle
  - G has n-1 edges



Rooted Trees

- Given a tree T, choose a root node r and orient each edge away from r.
- Importance: models hierarchy structure



a tree

the same tree, rooted at 1

Outline

- **Graph Definition**
- **Graph Representation**
- **Path, Cycle, Tree, Connectivity**
- **Graph Traversal Algorithms**
  - **basis of other algorithms**

Searching in a Graph

- **Graph searching** = systematically follow the edges of the graph to visit all vertices of the graph
  - Graph algorithms are typically elaborations of the basic graph-searching algorithms
  - e.g. puzzle solving, maze walking…
- Two basic graph searching algorithms:
  – Breadth-first search
  – Depth-first search
- Difference: the order in which they explore unvisited edges of the graph

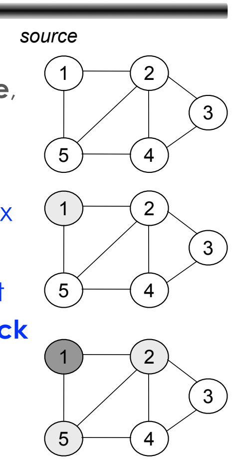
Breadth-First Search (BFS)

- **Input**:
  – A graph *G* = (V, E) (directed or undirected)
  – A **source** vertex **s** from V
- **Goal**:
  – Explore the edges of *G* to "discover" every vertex reachable from **s**, taking the ones closest to **s** first
- **Output**:
  - d[v] = distance (smallest # of edges) from **s** to **v**, for all **v** from V
  – A "breadth-first tree" rooted at **s** that contains all reachable vertices

Breadth-First Search (cont.)

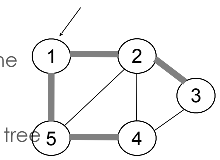
- Keeping track of progress:
  – Color each vertex in either **white**, **gray** or **black**
  – Initially, all vertices are **white**
  – When being discovered a vertex becomes **gray**
  – After discovering all its adjacent vertices the node becomes **black**

– Use FIFO queue Q to maintain the set of gray vertices



Breadth-First Tree

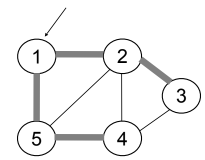
- BFS constructs **a breadth-first tree**
  - Initially contains root (source vertex **s**)
  - When vertex **v** is discovered while scanning adjacency list of a vertex **u** ⇒ vertex **v** and edge **(u, v)** are added to the tree
    - A vertex is discovered only once ⇒ it has only one parent
  - **u** is the **predecessor** (**parent**) of **v** in the breadth-first tree



- **Breath-first tree** contains nodes that **are reachable from source node, and all edges from each node's predecessor to the node**

BFS Application

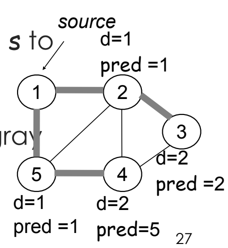
- BFS constructs **a breadth-first tree**
- BFS finds shortest (hop-count) path from src node to all other reachable nodes
- **E.g., What's shortest path from 1 to 3?**
  - **perform BFS using node 1 as source node**
  - **Node 2 is discovered while exploring 1's adjacent nodes => pred. of node 2 is node 1**
  - **Node 3 is discovered while exploring node 2's adjacent nodes => pred. of node 3 is node 2**
  - **so shortest hop count path is: 1, 2, 3**



- **Useful when we want to find minimal steps to reach a state**

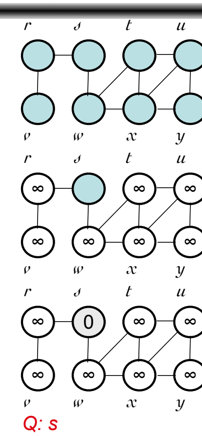
BFS: Implementation Detail

- G = (V, E) represented using adjacency lists
- **color[u]** – color of vertex **u** in **V**
- **pred[u]** – predecessor of **u**
  - If **u** = **s** (root) or node **u** has not yet been discovered then pred[u] = NIL
- **d[u]** – distance (hop count) from source **s** to vertex **u**
- Use a FIFO queue Q to maintain set of gray vertices



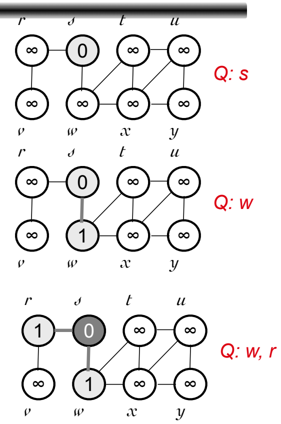
BFS(V, E, s)

for each **u** in **V - {s}**
  do color[u] = WHITE
    d[u] ← ∞
    pred[u] = NIL
color[s] = GRAY
d[s] ← 0
pred[s] = NIL
Q = empty
Q ← ENQUEUE(Q, s)

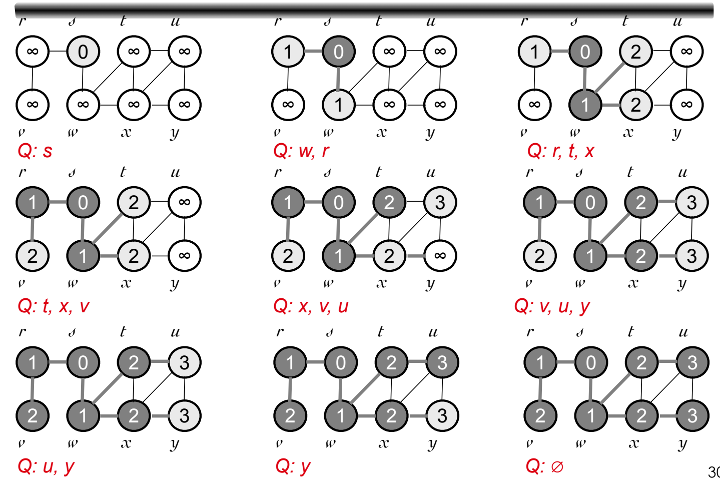


BFS(V, E, s)

```
while Q not empty
    do u ← DEQUEUE(Q)
        for each v in Adj[u]
            do if color[v] = WHITE
                then color[v] = GRAY
                    d[v] ← d[u] + 1
                    pred[v] = u
                    ENQUEUE(Q, v)
    color[u] = BLACK
```



Example



Analysis of BFS

```
1. for each u ∈ V - {s}
2.     do color[u] ← WHITE
3.         d[u] ← ∞
4.         pred[u] = NIL
```

$O(|V|)$

```
5. color[s] ← GRAY
6. d[s] ← 0
7. pred[s] = NIL
8. Q ← ∅
9. Q ← ENQUEUE(Q, s)
```

$\Theta(1)$

Analysis of BFS

```
while Q not empty
    do u ← DEQUEUE(Q)                  Θ(1)
        for each v in Adj[u]
            do if color[v] = WHITE
                then color[v] = GRAY
                    d[v] ← d[u] + 1
                    pred[v] = u         Θ(1)
                    ENQUEUE(Q, v)
    color[u] = BLACK
```

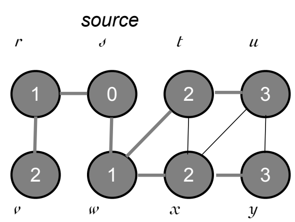
Scan Adj[u] for all vertices u in the graph

- Each vertex u is processed only once, when the vertex is dequeued
  - Sum of lengths of all adjacency lists = Θ(|E|)
  - Scanning operations: O(|E|)

- Total running time for BFS = O(|V| + |E|)

Shortest Paths Property

- BFS finds the shortest-path distance from the source vertex **s** ∊ **V** to each node in the graph
- Shortest-path distance = **d(s, u)**
  - Minimum number of edges in any path from **s** to **u**

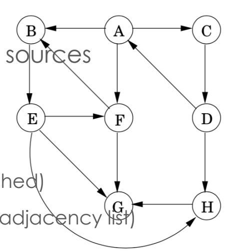


Outline

- **Graph Definition**
- **Graph Representation**
- **Path, Cycle, Tree, Connectivity**
- **Graph Traversal Algorithms**
  - **Breath first search/traversal**
  - **Depth first search/traversal**
  - …

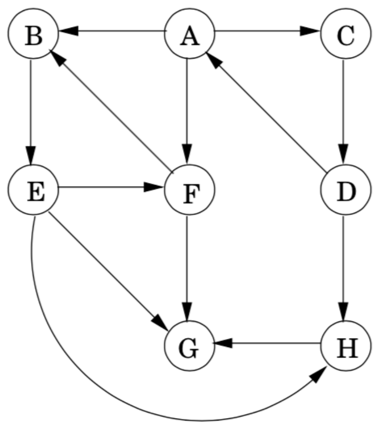
Depth-First Search

- **Input:**
  - G = (V, E) (No source vertex given!)
- **Goal**:
  - Explore edges of G to "discover" every vertex in V starting at most current visited node
  - Search may be repeated from multiple sources
- **Output:**
  - 2 **timestamps** on each vertex:
    - d[v] = discovery time (time when v is first reached)
    - f[v] = finishing time (done with examining v's adjacency list)
  - Depth-first forest



Depth-First Search: idea

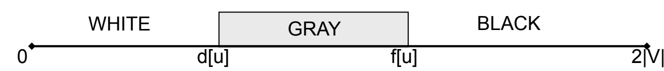
- Search "deeper" in graph whenever possible
  - explore edges of most recently discovered vertex **v** (that still has unexplored edges)
  - After all edges of v have been explored, "backtracks" to parent of v
- Continue until all vertices reachable from original source have been discovered
- If undiscovered vertices remain, choose one of them as a new source and repeat search from that vertex
  - different from BFS!!!
- DFS creates a **"depth-first forest"**



DFS Additional Data Structures

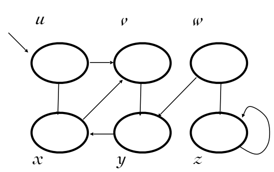
- Global variable: time-step
  - Incremented when nodes are discovered/finished
- color[u] – color of node u
  - **White** not discovered, **gray** discovered and being processing and **black** when finished processing
- pred[u] – predecessor of u (from which node we discover u)
- d[u]– discovery (time when u turns gray)
- f[u] – finish time (time when u turns black)

1 ≤ d[u] < f [u] ≤ 2 |V|



DFS(V, E): top level

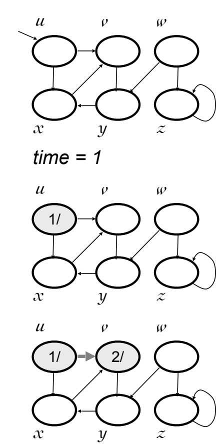
```
for each u ∈ V
    do color[u] ← WHITE
        pred[u] ← NIL
time ← 0
for each u ∈ V
    do if color[u] = WHITE
        then DFS-VISIT(u)
```



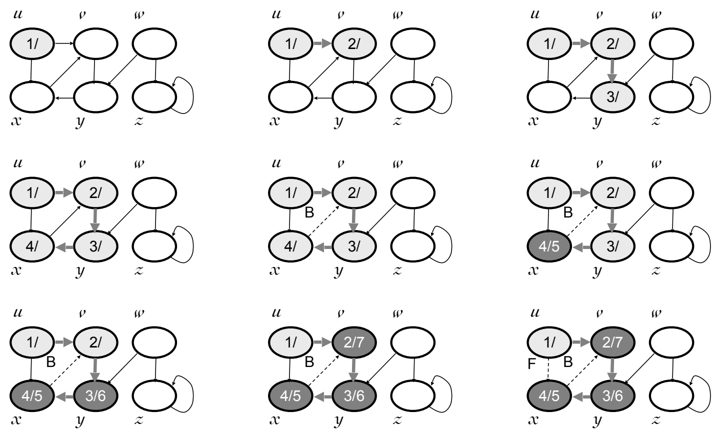
- Every time **DFS-VISIT(u)** is called, **u** becomes the root of a new tree in the depth-first forest

DFS-VISIT(u): DFS exploration from u

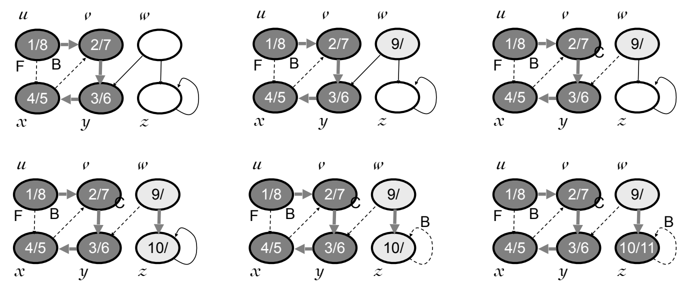
```
color[u] ← GRAY
time ← time+1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
        then pred[v] ← u
            DFS-VISIT(v)
color[u] ← BLACK //done with u
time ← time + 1
f[u] ← time  //finish time
```

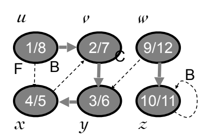


Example



Example (cont.)

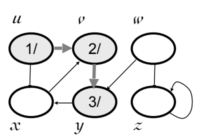




The results of DFS may depend on:
- The order in which nodes are explored in procedure DFS
- The order in which the neighbors of a vertex are visited in DFS-VISIT

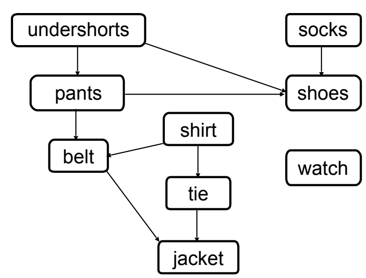
Properties of DFS

- **u** = pred[**v**] ⟺ DFS-VISIT(**v**) was called during a search of **u**'s adjacency list
  - u is the predecessor (parent) of v
- More generally, vertex v is a descendant of vertex **u** in depth first forest ⟺ v is discovered while **u** is gray



DAG

- Directed acyclic graphs (DAGs)
  - Used to represent precedence of events or processes that have a **partial order**

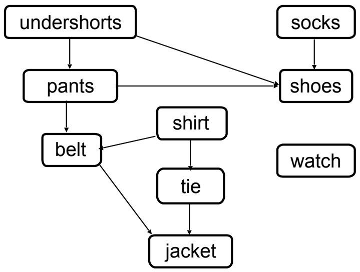


Put on socks before put on shoes

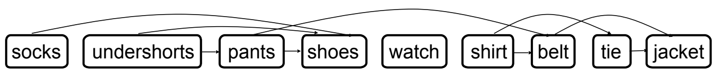
No precedence between belts and shoes

Topological sort helps us establish a total order/ linear order. Useful for task scheduling.

Topological Sort

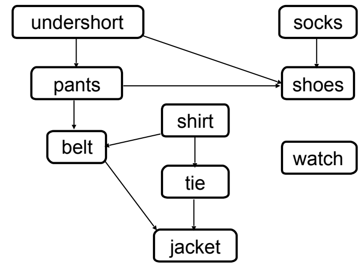


Topological sort of a directed acyclic graph G = (V, E): **a linear order** of vertices such that if there exists an edge **(u, v)**, then **u** appears before **v** in the ordering.



Topological sort:
an ordering of vertices so that
all directed edges go from left to right.

Topological Sort via DFS



TS requires that we put u before v **if there is a path from u to v**

e.g., socks before shoes

undershorts before jacket

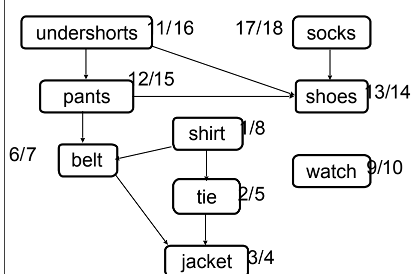
Observation: If we perform DFS on a DAG, if there is a path from u to v, then f[u]>f[v]

So arrange nodes in reverse order of their finish time

Consider when DFS_visit(undershorts) is called, jacket is either

* white: then jacket will be discovered in DFS_visit(undershorts), turn black, before eventually undershorts finishes. f[jacket] < f[undershorts]
* black (if DFS_visit(jacket) was called): then f[jacket] < f[undershorts]
* node jacket cannot be gray (which would mean that DFS_visit(jacket) is ongoing …)

Topological Sort



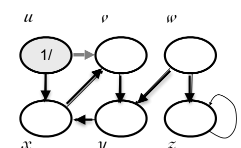
TOPOLOGICAL-SORT(V, E)

1. Call DFS(V, E) (to compute finishing times **f[v]** for each vertex **v**): when a node is finished, push it on to a stack
2. pop nodes in stack and arrange them in a list

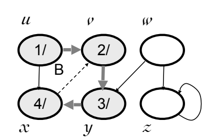
Running time: $\Theta(|V| + |E|)$

Edge Classification*

- **Via DFS traversal, graph edges can be classified into four types.**
- **When in DFS_visit (u), we follow edge (u,v) and find node, if v is:**
- WHITE vertex: then (u,v) is a **tree edge**
  - v was first discovered by exploring edge (u, v)
- GRAY node: then (u,v) is a **Back edge**
  - (u, v) connects u to an ancestor v in a depth first tree
  - Self loops (in directed graphs) are also back edges



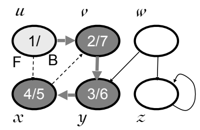
(u,v) is a tree edge



(x,v) is a back edge

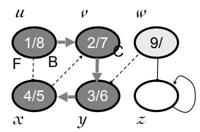
Edge Classification*

- if v is black vertex, and d[u]<d[v], (u,v) is a **Forward edge** (u,v):
  - Non-tree edge **(u, v)** that connects a vertex **u** to a descendant **v** in a depth first tree



(u,x) is a forward edge

- if v is black vertex, and d[u] > d[v], (u,v) is a **Cross edge** (u,v):
  - go between vertices in same depth-first tree (as long as there is no ancestor / descendant relation) or between different depth-first trees



(w,y) is a cross edge

Analysis of DFS(V, E)

```
for each u ∈ V                          } Θ(|V|)
    do color[u] ← WHITE
       pred[u] ← NIL
time ← 0
for each u ∈ V                          } Θ(|V|) – without counting the time for DFS-VISIT
    do if color[u] = WHITE
         then DFS-VISIT(u)
```


Analysis of DFS-VISIT(u)

```
color[u] ← GRAY
time ← time+1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
         then pred[v] ← u
              DFS-VISIT(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

DFS-VISIT is called exactly once for each vertex

Each loop takes |Adj[u]|

Total: $\underbrace{\Sigma_{u \in V} |Adj[u]|}_{\Theta(|E|)} + \Theta(|V|) =$

$= \Theta(|V| + |E|)$

DFS without recursion*

Data Structure: use stack (Last In First Out!)
to store all gray nodes

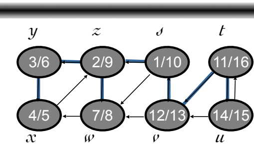
Pseudocode:
1. Start by push source node to stack
2. Explore node at stack top, i.e.,
   * push its next white adj. node to stack)
   * if all its adj nodes are black, the node turns black, pop it from stack
3. Continue (go back 2) until stack is empty

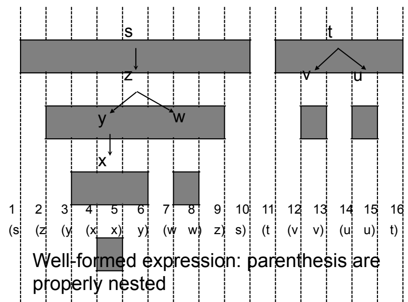
4. If there are white nodes remaining, go back to 1 using another white node as source node

Parenthesis Theorem*

In any DFS of a graph G, for all **u**, **v**, exactly one of the following holds:

1. [d[u], f[u]] and [d[v], f[v]] are disjoint, and neither of **u** and **v** is a descendant of the other
2. [d[v], f[v]] is entirely within [d[u], f[u]] and **v** is a descendant of **u**
3. [d[u], f[u]] is entirely within [d[v], f[v]] and **u** is a descendant of **v**





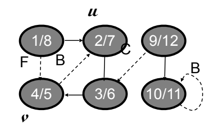
Well-formed expression: parenthesis are properly nested

Other Properties of DFS*

Corollary

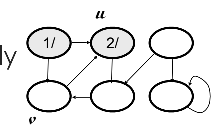
Vertex **v** is a proper descendant of **u**

$\Longleftrightarrow$ d[u] < d[v] < f[v] < f[u]



Theorem (White-path Theorem)

In a depth-first forest of a graph G, vertex **v** is a descendant of **u** if and only if at time **d[u]**, there is a path **u** ⇨ **v** consisting of only white vertices.



Cycle detection via DFS

A directed graph is **acyclic** $\Longleftrightarrow$ a DFS on G yields no back edges.

Proof:

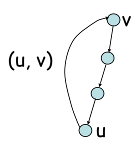
"⇒": acyclic ⇒ no back edge

- Assume **back edge** ⇒ prove **cycle**
- Assume there is a back edge **(u, v)**

⇒ **v** is an ancestor of **u**

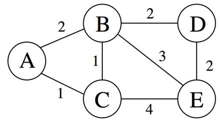
⇒ there is a path from **v** to **u** in G (**v** ⇨ **u**)

⇒ **v** ⇨ **u** + the back edge **(u, v)** yield a cycle



three graph algorithms

Shortest Distance Paths



- Distance/Cost of a path in weighted graph
  - sum of weights of all edges on the path
  - path A,B,E, cost is 2+3=5
  - path A, B, C, E, cost is 2+1+4=7
- How to find shortest distance path from a node, A, to all another node?
  - assuming: all weights are positive
  - This implies no cycle in the shortest distance path
    - Why? Prove by contradiction.
    - If A->B->C->..->B->D is shortest path, then A->B->D is a shorter!
  - d[u]: the distance of the shortest-distance path from A to u

d[A] = 0

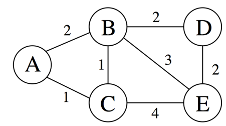
d[D] = min {d[B]+2, d[E]+2}

because B, E are the two only possible previous node in path to D

Dijkstra Algorithm

- Input: positive weighted graph G, source node s
- Output: shortest distance path from s to all other nodes that is reachable from s

s



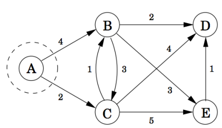
Expanding frontier (one hop a time)
1). Starting from A:
We can go to B with cost B, go to C with cost 1

going to all other nodes (here D, E) has to pass B or C
are there cheaper paths to go to C?
are there cheaper paths to B?

2). Where can we go from C? B, E
Two new paths: (A,C,B), (A,C,E)
Better paths than before? => update current optimal path
Are there cheaper paths to B?
3). Where can we go from B?
...

for each node u, keep track pred[u] (previous node in the path leading to u),
d[u] current shortest distance

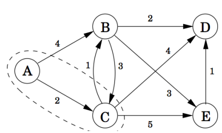
dist **pred**



A: 0	D: ∞
B: 4	E: ∞
C: 2	

Q: C(2), B(4), D, E

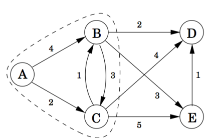
A: null
B: A
C: A
D: null,
E: null



A: 0	D: 6
B: 3	E: 7
C: 2	

Q: B(3), D(6), E(7)

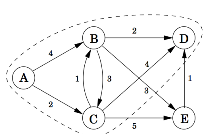
A: null
B: C
C: A
D: C,
E: C



A: 0	D: 5
B: 3	E: 6
C: 2	

Q: D(5), E(6)

A: null
B: C
C: A
D: B,
E: B



A: 0	D: 5
B: 3	E: 6
C: 2	

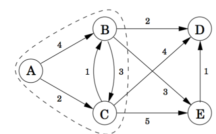
Q: E(6)

A: null
B: C
C: A
D: B,
E: B

best paths to each node via nodes circled & associated distance

Dijkstra Alg Demo

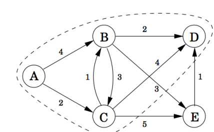
dist **pred**



A: 0	D: 5
B: 3	E: 6
C: 2	

Q: D(5), E(6)

A: null
B: C
C: A
D: B,
E: B

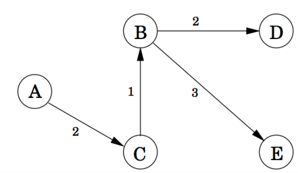


A: 0	D: 5
B: 3	E: 6
C: 2	

Q: E(6)

A: null
B: C
C: A
D: B,
E: B

best paths to each node via nodes circled & associated distance



Dijkstra Alg Demo

Dijkstra's algorithm & snapshot

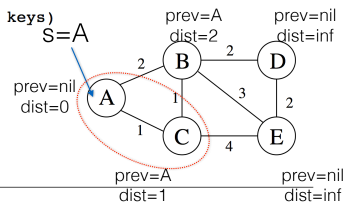
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0

H = makequeue(V)  (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
      if dist(v) > dist(u) + l(u, v):
         dist(v) = dist(u) + l(u, v)
         prev(v) = u
         decreasekey(H, v)
```

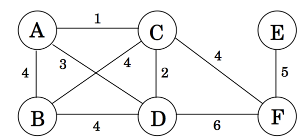
H: priority queue (min-heap in this case)

C(dist=1), B(dist=2), D(dist=inf), E (dist=inf)



Minimum Spanning Trees

- Minimum Spanning Tree Problem: Given a weighted graph, choose a subset of edges so that resulting subgraph is connected, and the total weights of edges is minimized
  - to minimize total weights, it never pays to have cycles, so resulting connection graph is connected, undirected, and acyclic, i.e., a *tree*.
- Applications:
  - Communication networks
  - Circuit design
  - Layout of highway systems



Formal Definition of MST

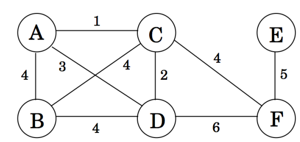
- Given a connected, undirected, weighted graph $G = (V, E)$, a *spanning tree* is an *acyclic* subset of edges $T \subseteq E$ that connects all vertices together.
- *cost* of a spanning tree T: the sum of edge weights in the spanning tree

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

- A *minimum spanning tree (MST)* is a spanning tree of minimum weight.

Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, *G*
- **Find:** Minimum - weight spanning tree, *T*



Acyclic subset of edges(*E*) that connects all vertices of *G*.

Notice: there are many spanning trees for a graph
We want to find the one with the minimum cost

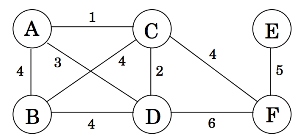
Such problems are ***optimization problems****: there are multiple viable solutions, we want to find best (lowest cost, best perf) one.*

Greedy Algorithms

- A problem solving strategy (like divide-and-conquer)
- Idea: build up a solution piece by piece, in each step always choose the option that offers best immediate benefits (a myopic approach)
  - Local optimization: choose what seems best right now
  - not worrying about long term benefits/global benefits
- Sometimes yield optimal solution, sometimes yield suboptimal (i.e., not optimal)
- Sometimes we can bound difference from optimal…

Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, *G*
- **Find:** Minimum - weight spanning tree, *T*



How to greedily build a spanning tree?

~~ Always choose lightest edge? Might lead to cycle.~~*

** Repeat for n-1 times:*

find next lightest edge <u>that does not introduce cycle,</u>

add the edge into tree

=> Kruskal's algorithm

Kruskal's Algorithm

Figure 5.4 Kruskal's minimum spanning tree algorithm.

```
procedure kruskal(G,w)
Input:    A connected undirected graph G = (V,E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u,v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u,v} to X
      union(u,v)
```

Implementation detail:

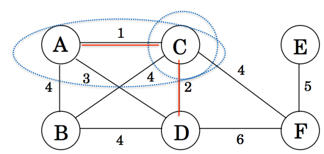
* Maintain sets of nodes that are connected by tree edges

* find(u): return the set that u belongs to

* find(u)=find(v) means u, v belongs to same group (i.e., u and v are already connected)

Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, *G*
- **Find:** Minimum - weight spanning tree, *T*



<u>Example:</u>

Suppose we start grow tree from C,

step 1. A has lightest edge to tree, add A and the edge (A-C) to tree

// tree is now A-C

step 2: D has lightest edge to tree add D and the edge (C-D) to tree

….

How to greedily build a spanning tree?

** Grow the tree from a node (any node),*

** Repeat for n-1 times:*

** connect one node to the tree by choosing <u>node with lightest edge connecting to tree nodes</u>*

This is Prim algorithm.

Prim's Algorithm

```
procedure prim(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the array prev

for all u ∈ V:
   cost(u) = ∞
   prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0

H = makequeue(V)    (priority queue, using cost-values as keys)
while H is not empty:
   v = deletemin(H)
   for each {v, z} ∈ E:
      if cost(z) > w(v, z):
         cost(z) = w(v, z)
         prev(z) = v
         decreasekey(H, z)
```

cost[u]: stores weight of lightest edge connecting u to current tree

It will be updated as the tree grows

deletemin() takes node v with lowest cost out
 * this means node v is done(added to tree) // v, and edge v - prev(v) added to tree

H is a priority queue (usually implemented as heap, here it's min-heap: node with lostest cost at root)

Summary

- Graph everywhere: represent binary relation
- Graph Representation
  - Adjacent lists, Adjacent matrix
- Path, Cycle, Tree, Connectivity
- Graph Traversal Algorithm: systematic way to explore graph (nodes)
  - BFS yields a fat and short tree
    - App: find shortest hop path from a node to other nodes
  - DFS yields forest made up of lean and tall tree
    - App: detect cycles and topological sorting (for DAG)