EECS 3101 A: Design and Analysis of Algorithms

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/3101A`
Also on Moodle

Definitions - 1

- $G = (V, E)$, $V =$ set of nodes/vertices, $E =$ set of edges
- Edges incident on a vertex
- Adjacent vertices
- degree of a node
- neighborhood of a node
- Self-loop

Definitions - 2

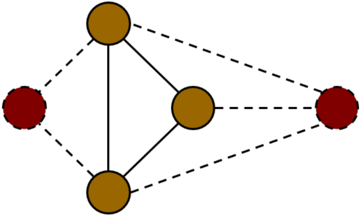
- Edge Types:
  - Directed edge: ordered pair of vertices (u, v)
    - u : origin, v : destination
  - Undirected edge: unordered pair of vertices (u, v)
- Graph Types:
  - Directed graph: all the edges are directed
  - Undirected graph: all the edges are undirected
- Paths:
  - Simple Paths
  - Cycles
  - Simple cycles: no vertex repeated

Elementary Properties

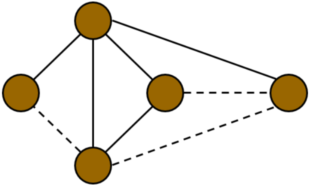
- The sum of degrees is even (equals twice the number of edges in an undirected graph)
- The sum of indegrees equals sum of outdegrees in a directed graph
- In an undirected graph $m \leq \frac{n(n-1)}{2}$
  What is the bound for directed graphs?

Subgraphs

- A subgraph S of a graph G is a graph such that
    - The vertices of S are a subset of the vertices of G
    - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



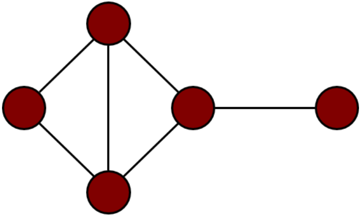
Subgraph



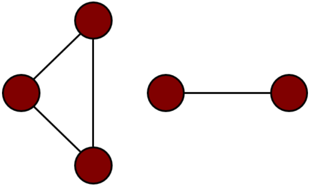
Spanning subgraph

Connected graphs

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



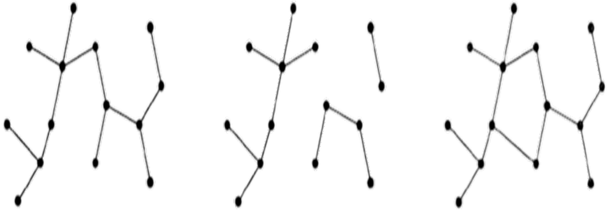
Connected graph



Disconnected graph with two connected components

Trees

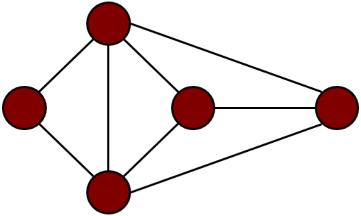
- A tree is a connected, acyclic, undirected graph
- A forest is a set of trees (not necessarily connected)



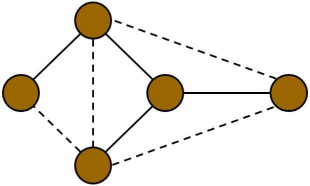
Tree, forest, a cyclic graph

Spanning Trees

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



graph



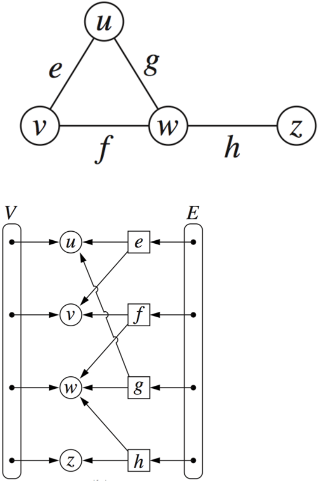
Spanning tree

Graph Representations

- Edge list
- Adjacency list
- Adjacency matrix

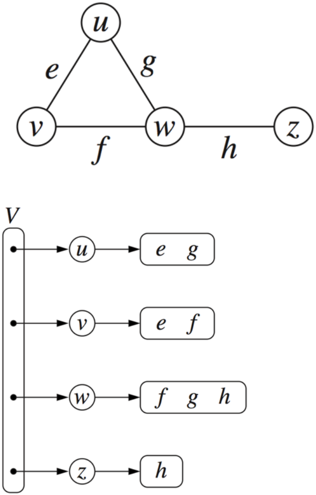
Edge Lists

- Vertex object: reference to position in vertex sequence
- Edge object: origin vertex object, destination vertex object, reference to position in edge sequence
- Vertex sequence: sequence of vertex objects
- Edge sequence: sequence of edge objects



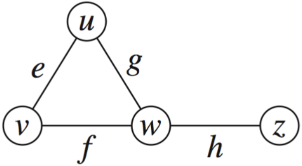
Adjacency Lists

- Incidence sequence for each vertex: sequence of references to edge objects of incident edges
- Augmented edge objects: references to associated positions in incidence sequences of end vertices



Adjacency Matrix

- Edge list structure
- Augmented vertex objects: Integer key (index) associated with vertex
- 2D-array adjacency array: Reference to edge object for adjacent vertices, null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



	0	1	2	3
$u \longrightarrow 0$		e	g	
$v \longrightarrow 1$	e		f	
$w \longrightarrow 2$	g	f		h
$z \longrightarrow 3$			h	

Graph Problems

- Connectivity: Are all vertices reachable from each other?
- Reachability: Is a node v reachable from a node u?
- Shortest Paths
- (Sub)graph Isomorphism
- Graph Coloring
- And many others

Coloring graphs

Basic idea:

- Assign colors to nodes
- Each edge should connect nodes of **different** colors
- Want to minimize the number of colors used
- The minimum number of colors is the property of a graph, called **chromatic number**

Bipartite graphs

- The set of vertices V can be partitioned into disjoint sets V_1, V_2 such that all edges go between V_1, V_2
- A graph is bipartite **if and only if** it is 2-colorable
- How do we know if a graph is 2-colorable?

Greedy Bipartite Graph Coloring - idea

Assumes a connected undirected graph

- start at any node and color it red; label it “finished”
- color its neighbours blue and label the nodes “started”
- consider any node labeled “started”.
- if it has a neighbour with the same color, exit with the message “not bipartite”
- else color its uncolored neighbours with the opposite color and label them “started”; label the current node “finished”

Greedy Bipartite Graph Coloring - Correctness

Part 1: If the algorithm fails the graph is not 2-colorable

- if the graph contains an odd cycle, it cannot be 2-colorable

- if the algorithm fails, the graph contains an odd cycle
  Why did the algorithm fail to 2-color? 2 nodes joined by the edge had the same color. So the distances from the least common ancestor of the 2 nodes to the nodes are both even or both odd. Adding the edge between them creates an odd cycle

Part 2: If the algorithm succeeds the graph is 2-colorable

More on Graph Coloring

- Determining if a graph has chromatic number of 1 or 2 is easy
- Determining if a graph has chromatic number 3 is NP-complete (believed to be intractable)
- For special classes of graphs, the chromatic number is known.
- For planar graphs the chromatic number is 4

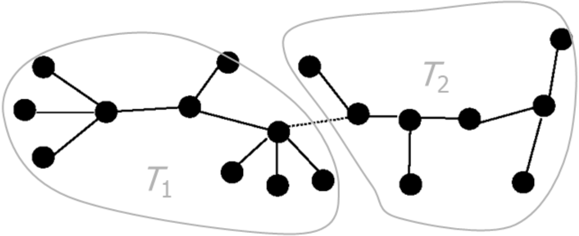
Minimum Spanning Trees (MST)

- Undirected, connected graph $G = (V, E)$
- Weight function $w : E \rightarrow \mathbb{R}$ (assigning cost or length or other values to edges)
- Spanning tree: tree that connects all vertices
- Minimum spanning tree: tree T that connects all the vertices and minimizes $w(T) = \sum_{(u,v)\in T} w(u, v)$

Minimum Spanning Trees: Questions

- Is DP applicable?
- Is a greedy strategy applicable?

MST: Optimal Substructure



- Removing the edge (u, v) partitions T into T_1 and T_2: $w(T) = w(T_1) + w(T_2) + w(u, v)$
- We claim that T_1 is the MST of $G_1 = (V_1, E_1)$, the subgraph of G induced by vertices in T_1.
- Similarly, T_2 is the MST of G_2

MST: Greedy Choice Property

Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution

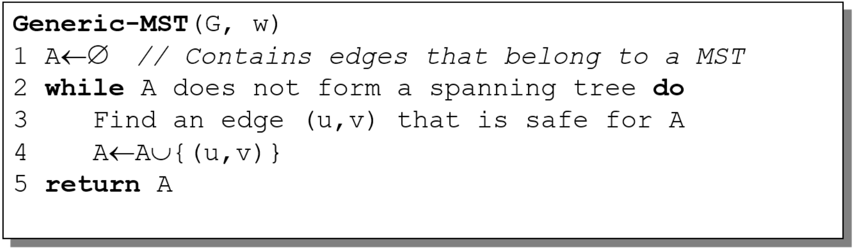
Theorem:

- Let $G = (V, E)$, and let $S \subseteq V$ and
- Let (u, v) be min-weight edge in G connecting S to $V - S$
- Then $(u, v) \in T$ for some MST T of G

MST: Proof of Greedy Choice Property

- Let (u, v) be min-weight edge in G connecting S to $V - S$; suppose $(u, v) \notin T$
- look at path from u to v in T
- swap (x, y), the first edge on path from u to v in T that crosses from S to $V - S$, with (u, v)
- this decreases the cost of T - contradiction (T supposed to be MST)

Generic MST Algorithm



```
Generic-MST(G, w)
A ← ∅   // Contains edges that belong to a MST
while A does not form a spanning tree do
    Find an edge (u,v) that is safe for A
    A ← A ∪ {(u,v)}
return A
```

- Loop invariant: before each iteration, *A* is a subset of some MST
- Safe edge : edge that preserves the loop invariant

Generic MST Algorithm - 2

```
MoreSpecific-MST(G, w)
  A←∅  // Contains edges that belong to a MST
  while A does not form a spanning tree do
      Make a cut (S, V-S) of G that respects A
      Take the min-weight edge (u,v) connecting S to V-S
      A←A∪{(u,v)}
  return A
```

- A cut respects *A* if no edge of *A* crosses the cut
- Same LI: before each iteration, *A* is a subset of an MST
- Correctness proof in Theorem 23.1 in the text
- Many ways to choose cuts

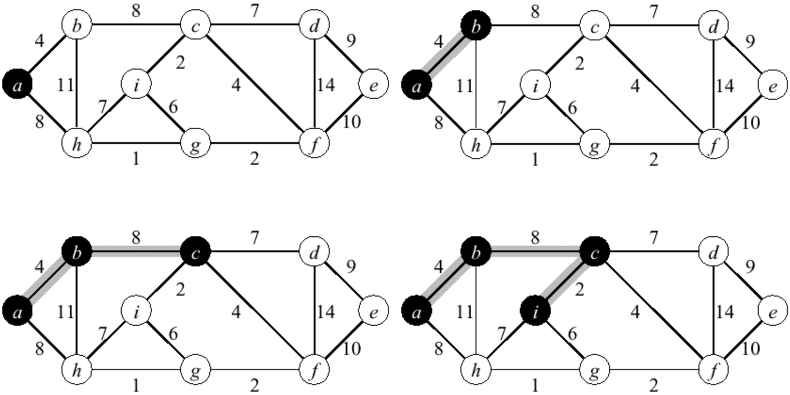
Prim's Algorithm

- Vertex based algorithm
- Grows one tree T, one vertex at a time
- Imagine a "blob" covering the portion of T already computed
- Label the vertices v outside the blob with $key[v] =$ the minimum weight of an edge connecting v to a vertex in the blob, $key[v] = \infty$, if no such edge exists
- At each iteration, add the minimum weight vertex to T

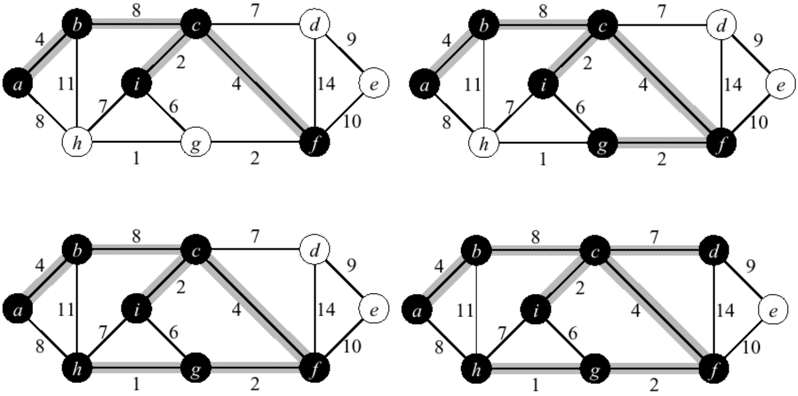
Prim's Algorithm: Steps

- Pseudocode on pg 634
- Put all vertices in a priority queue Q with labels ∞
- Remove the start vertex and set its label to 0
- While Q is not empty, remove the vertex u with the minimum label and add it to the tree;
  For each neighbour v of u in Q, if $w(u, v) < label[v]$, set $label[v] = w(u, v)$

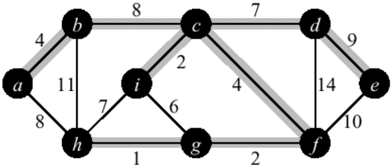
Prim's Algorithm: Illustration



Prim's Algorithm: Illustration



Prim's Algorithm: Illustration



Prim's Algorithm: Analysis

- Proof of correctness on page 636
- Time $= O(|V|T(\mathit{ExtractMin})) + O(|E|T(\mathit{ModifyKey}))$
- Times depend on PQ implementation
- Heap based PQ:
  $\mathit{BuildPQ} : O(n)$, $\mathit{ExtractMin}$ and $\mathit{ModifyKey}$: $O(\lg n)$
  So running time:
  $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$
- With Fibonacci heaps: $O(|V| \log |V| + |E|)$

Kruskal's Algorithm

- Edge based algorithm
- Add the edges one at a time, in increasing weight order
- The algorithm maintains A: a **forest** of trees. An edge is accepted it if connects vertices of distinct trees

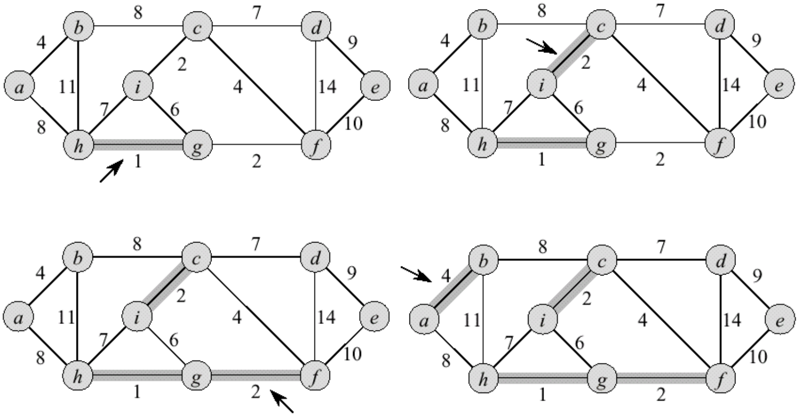
Kruskal's Algorithm: Requirements

We need an ADT that maintains a partition, i.e.,a collection of disjoint sets
Operations:

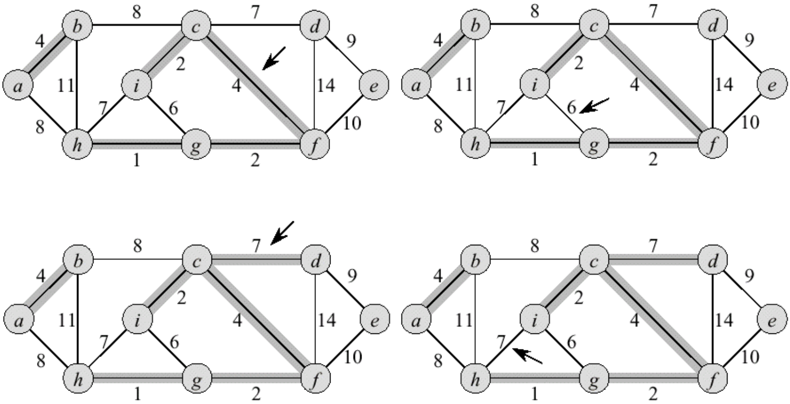
- *MakeSet*(S, x): $S \leftarrow S \cup \{\{x\}\}$
- *Union*(S_i, S_j): $S \leftarrow (S - \{S_i, S_j\}) \cup (S_i \cup S_j)$
- *FindSet*(S, x): returns unique $S_i \in S$, where $x \in S_i$

Good ADT's for maintaining collections of disjoint sets are covered in EECS 4101

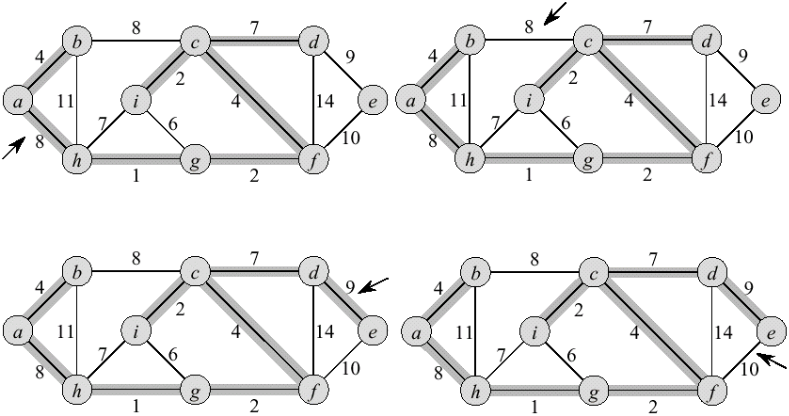
Kruskal's Algorithm: Illustration



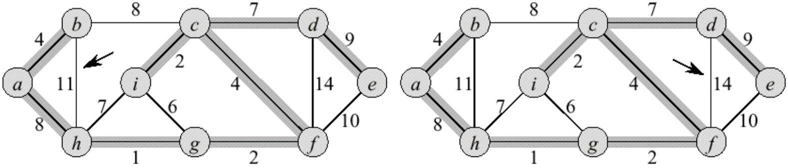
Kruskal's Algorithm: Illustration



Kruskal's Algorithm: Illustration



Kruskal's Algorithm: Illustration



Kruskal's Algorithm: Analysis

- Proof of correctness: easy since minimum weight edge has to be a safe edge
- Sorting the edges $O(|E| \lg |E|) = O(|E| \lg |V|)$
- $O(|E|)$ calls to FindSet, Union
- With advanced data structures, the running time is $O(|E| \lg |V|)$

Graphs: Exploration and Searching

Method to explore many key properties of a graph

- Nodes that are reachable from a specific node v
- Detection of cycles
- Extraction of strongly connected components
- Topological sorts
- Find a path with the minimum number of edges between two given vertices

Note: Some slides in this presentation have been adapted from the author's and Prof Elder's slides.

Graph Search Algorithms

- Depth-first Search (DFS)
- Breadth-first Search (BFS)

Breadth First Search

A general technique for traversing a graph

- A BFS traversal of a graph G
    - Visits all the vertices and edges of G
    - Determines whether G is connected
    - Computes the connected components of G
    - Computes a spanning forest of G
- BFS on a graph with $|V|$ vertices and $|E|$ edges takes $\Theta(|V| + |E|)$ time
- BFS can be further extended to solve other graph problems
    - Find and report a path with the minimum number of edges between two given vertices
    - Cycle detection

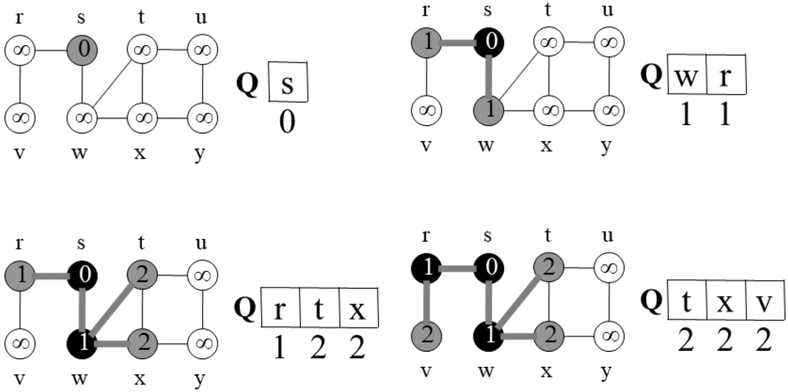
Breadth First Search - 2

- In BFS exploration takes place on a level or wavefront consisting of nodes that are all the same distance from the source s
- We can label these successive wavefronts by their distance: $L_0, L_1, \ldots$

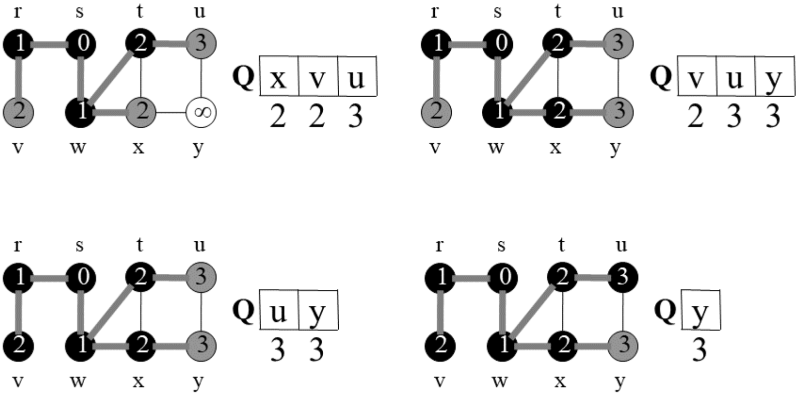
Breadth First Search - 3

- Input: directed or undirected graph $G = (V, E)$, source vertex $s \in V$
- Output: for all $v \in V$
  - $d[v]$, the shortest distance from s to v
  - $\pi[v] = u$, such that (u, v) is the last edge on the shortest distance from s to v
- Idea: send out search 'wave' from s
- Keep track of progress by colouring vertices:
  - Undiscovered vertices are coloured **white**
  - Just discovered vertices (on the wavefront) are coloured **grey**
  - Previously discovered vertices (behind wavefront) are coloured **black**

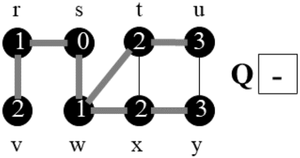
Breadth First Search - Example



Breadth First Search - Example



Breadth First Search - Example



Breadth First Search - Algorithm

```
BFS(G,s)
for each vertex u ∈ V[G]-{s}
    color[u] ← white
    d[u] ← ∞
    π[u] ← NIL
color[s] ← gray
d[s] ← 0
π[u] ← NIL
Q ← {s}
while Q ≠ ∅ do
    u ← head[Q]
    for each v ∈ Adj[u] do
        if color[v] = white then
            color[v] ← gray
            d[v] ← d[u] + 1
            π[v] ← u
            Enqueue(Q,v)
    Dequeue(Q)
    color[u] ← black
```

BFS: Properties

Notation: G_s: connected component containing s

- Property 1: BFS(G, s) visits all the vertices and edges of G_s
- Property 2: The discovery edges labeled by BFS(G, s) form a spanning tree T_s of G_s
- Property 3: For any vertex v reachable from s, the path in the breadth first tree from s to v corresponds to a shortest path in G

BFS: Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Vertices are enqueued if there color is white
- Assuming that en- and dequeuing takes $O(1)$ time the total cost of this operation is $O(|V|)$
- Adjacency list of a vertex is scanned when the vertex is dequeued (and only then ...)
- The sum of the lengths of all lists is $O(|E|)$. Consequently, $O(|E|)$ time is spent on scanning them
- Initializing the algorithm takes $O(|V|)$
- Thus BFS runs in $\Theta(|V| + |E|)$ time provided the graph is represented by an adjacency list structure

BFS Application: Shortest Unweighted Paths

- Goal: To recover the shortest paths from a source node s to all other reachable nodes v in a graph
  - The length of each path and the paths themselves are returned
- Notes:
  - There are an exponential number of possible paths
  - Analogous to level order traversal for trees
  - This problem is harder for general graphs than trees because of cycles!

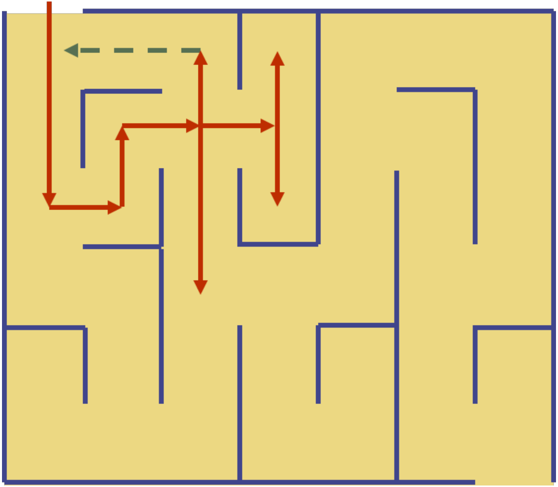
Depth-first Search

A DFS traversal of a graph G

- Visits all the vertices and edges of G
- Determines whether G is connected
- Computes the connected components of G
- Computes a spanning forest of G
- Find a cycle in the graph

Depth-first Search - 2

DFS: similar to a classic strategy for exploring a maze



Depth-first Search - Steps

- We start at vertex s, tying the end of our string to the point and painting s "visited (discovered)". Next we label s as our current vertex called u
- Now, we travel along an arbitrary edge (u, v)
- If edge (u, v) leads us to an already visited vertex v we return to u
- If vertex v is unvisited, we unroll our string, move to v, paint v "visited", set v as our current vertex, and repeat the previous steps

Depth-first Search - Steps

- Eventually, we will get to a point where *all incident edges on u lead to visited vertices*
- We then backtrack by unrolling our string to a previously visited vertex v. Then v becomes our current vertex and we repeat the previous steps
- Then, if all incident edges on v lead to visited vertices, we backtrack as we did before. We *continue to backtrack along the path we have traveled*, finding and exploring unexplored edges, and repeating the procedure

Depth-first Search - Algorithm

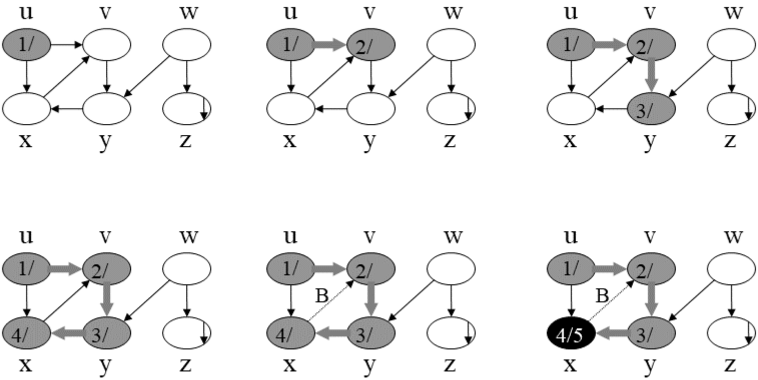
- Initialize: color all vertices white
- Visit each and every white vertex using $DFS - Visit$
- Each call to $DFS - Visit(u)$ roots a new tree of the depth-first forest at vertex u
- A vertex is **white** if it is undiscovered
- A vertex is **gray** if it has been discovered but not all of its edges have been discovered
- A vertex is **black** after all of its adjacent vertices have been discovered (the adj. list was examined completely)
- In addition to, or instead of labeling vertices with colours, they can be labeled with discovery and finishing times.

Depth-first Search - Algorithm

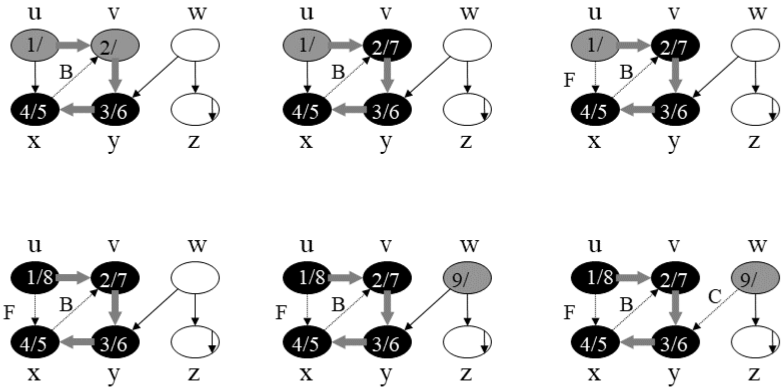
- Time is an integer that is incremented whenever a vertex changes state
    - from unexplored to discovered
    - from discovered to finished
- These discovery and finishing times can then be used to solve other graph problems (e.g., computing strongly-connected components)

- Two timestamps put on every vertex:
    - discovery time $d(v) \geq 1$
    - finish time $1 < f(v) \leq 2n$

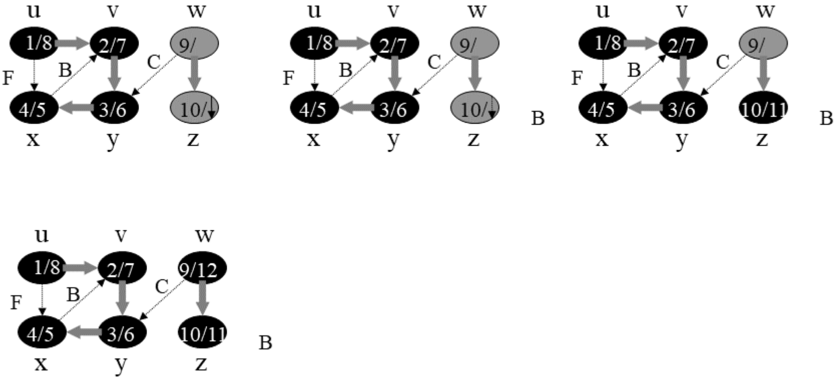
DFS - Example


u
v
w
1/
x
y
z
u
v
w
1/
2/
x
y
z
u
v
w
1/
2/
3/
x
y
z
u
v
w
1/
2/
4/
3/
x
y
z
u
v
w
1/
2/
B
4/
3/
x
y
z
u
v
w
1/
2/
B
4/5
3/
x
y
z

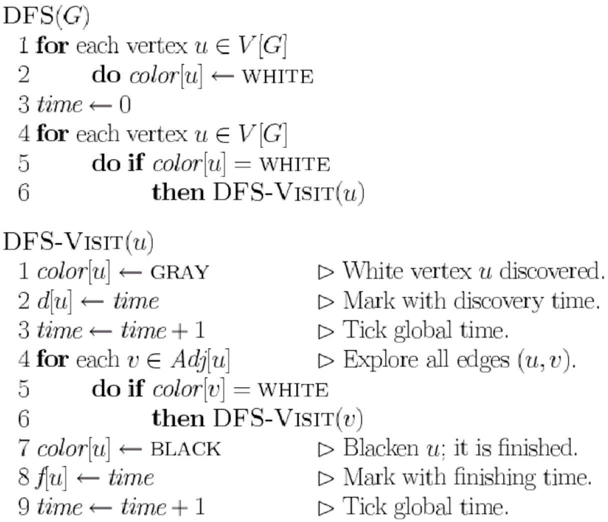
DFS - Example


u
v
w
1/
2/
B
4/5
3/6
x
y
z
u
v
w
1/
2/7
B
4/5
3/6
x
y
z
u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z
u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z
u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z
u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
x
y
z

DFS - Example


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
x
y
z
u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
B
x
y
z
u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/11
B
x
y
z
u
v
w
1/8
2/7
9/12
F
B
C
4/5
3/6
10/11
B
x
y
z

DFS - Algorithm



```
DFS($G$)
for each vertex $u \in V[G]$
    do $color[u] \leftarrow$ WHITE
$time \leftarrow 0$
for each vertex $u \in V[G]$
    do if $color[u] =$ WHITE
        then DFS-VISIT($u$)

DFS-VISIT($u$)
$color[u] \leftarrow$ GRAY                ▷ White vertex $u$ discovered.
$d[u] \leftarrow time$                    ▷ Mark with discovery time.
$time \leftarrow time + 1$                ▷ Tick global time.
for each $v \in Adj[u]$                   ▷ Explore all edges $(u, v)$.
    do if $color[v] =$ WHITE
        then DFS-VISIT($v$)
$color[u] \leftarrow$ BLACK               ▷ Blacken $u$; it is finished.
$f[u] \leftarrow time$                    ▷ Mark with finishing time.
$time \leftarrow time + 1$                ▷ Tick global time.
```

DFS-Visit - Algorithm

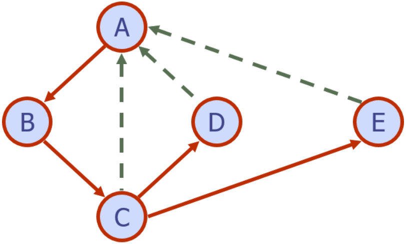
Q: How are the edges classified?

Q: What do back edges signify?

Notice the implicit stack in the code.

DFS: Properties

- Property 1:
  DFS-Visit(v) visits all the vertices and edges in the connected component of v
- Property 2:
  The discovery edges labeled by DFS(v) form a spanning tree of the connected component of v



DFS: Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
    - once as UNEXPLORED
    - once as VISITED
- Each edge is labeled twice
    - once as UNEXPLORED once as DISCOVERY or BACK
- Method DFS-Visit is called once for each vertex
- DFS runs in $\theta(n+m)$ time provided the graph is represented by the adjacency list structure:
  Recall that $\sum_v deg(v) = 2m$

DFS on Directed Graphs

- Tree edges are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v)
- Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree
- Forward edges are non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree
- Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other.
- Classifying edges can help to identify properties of the graph, e.g., a graph is acyclic iff DFS yields no back edges

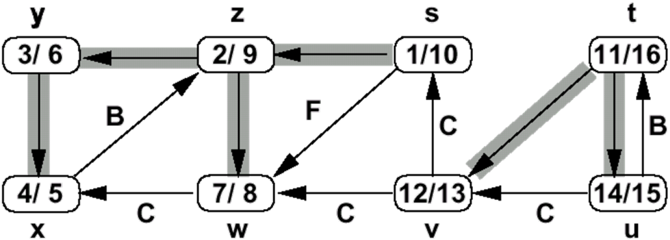
DFS on Undirected Graphs

- In a depth-first search of a connected undirected graph, every edge is either a tree edge or a back edge

DFS: Timestamps

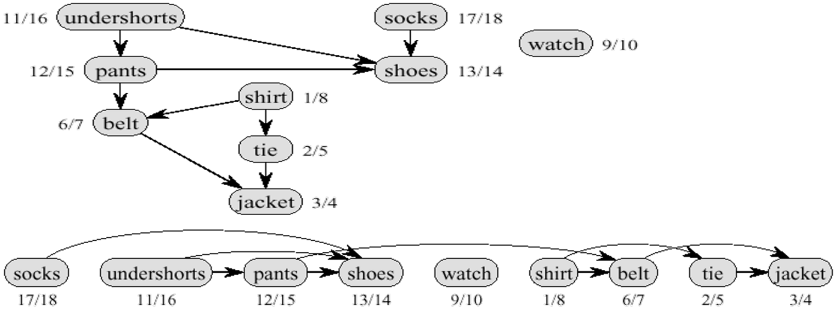
- In addition to labeling vertices with colours, they are labeled with discovery and finishing times.
- Time is an integer that is incremented whenever a vertex changes state
  - from unexplored to discovered
  - from discovered to finished
- These discovery and finishing times can then be used to solve other graph problems (e.g., computing strongly-connected components)
- Two timestamps put on every vertex:
  - discovery time $d(v) \geq 1$
  - finish time $1 < f(v) \leq 2n$

DFS Colors - Advantages



- Time stamps are useful for many purposes
- E.g., Topological Sort – sorting vertices of a directed acyclic graph

DFS Application: Topological Sort



- call DFS(G) to compute finishing times $f[v]$ for each vertex v
- return the list of vertices sorted in decreasing order of $f[v]$

DFS Application: Path Finding

- We can adapt the DFS algorithm to find a path between vertices u and z
- We call DFS(G, u) with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack
- Q: What is the color of the nodes on the path?

DFS Application: Cycle Finding

- We can adapt the DFS algorithm to find a simple cycle
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w