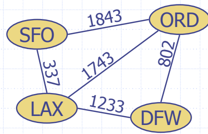
Graphs

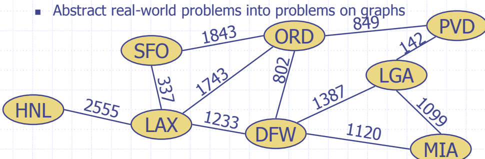


Outline and Reading

- Graphs (§6.1)
  - Definitions
  - Applications
  - Terminology
  - Properties
  - ADT
- Data structures for graphs (§6.2)
  - Edge list structure
  - Adjacency list structure
  - Adjacency matrix structure

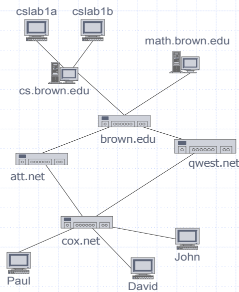
Graph

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of edges (pairs of vertices)
  - Vertices and edges are positions and store elements
- Graphs useful for representing real-world relationships:
  - vertex = airport
  - edge = flight route, storing mileage
  - Abstract real-world problems into problems on graphs



Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



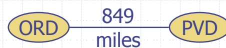
Sample problems

- What is cheapest way to fly from X to Y?
- If airport X closes from bad weather, can I still fly between every other pair of cities?
- Many classes have prereqs; in what order can I take the classes for my major?
- How much traffic can flow between intersection X and intersection Y
- How can I minimize the amount of wiring needed to connect some outlets together?

Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight

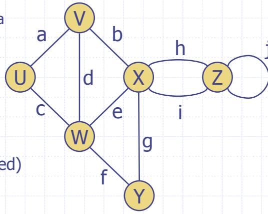
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route



- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

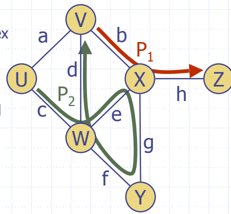
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges (typically not used)
  - h and i are parallel edges
- Self-loop (typically not used)
  - j is a self-loop



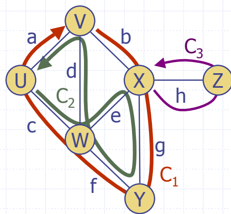
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins and ends with some vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Reachable
  - path exists
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple
  - Z is reachable from U



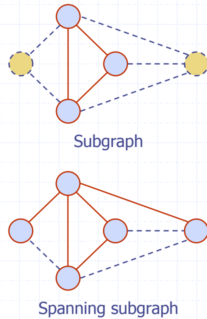
Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
  - edges traversed only in one direction
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple
  - C_3=(X,h,Z,h,X) is not a cycle



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

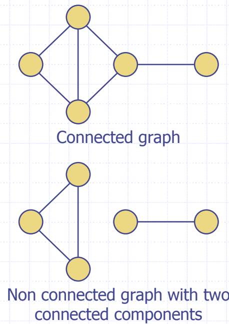


Subgraph

Spanning subgraph

Connectivity

- 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



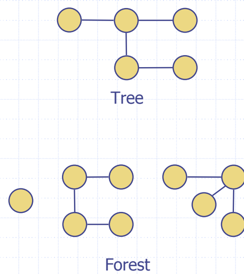
Connected graph

Non connected graph with two connected components

Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  Different definition than a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

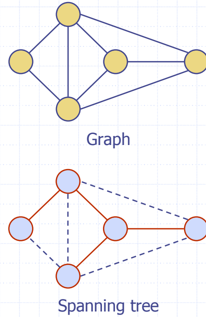


Tree

Forest

Spanning Trees and Forests

- 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



Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

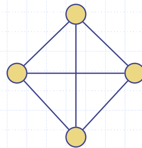
In an undirected graph (with no self-loops or parallel edges)

$m \leq n\,(n-1)/2$

Proof: at most one edge for every unique combination of 2 vertices

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$ for all vertices

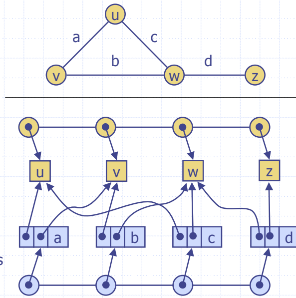
What is the bound on m for a directed graph?

Main Methods of the Graph ADT

- Vertices and Edges accessor method:
  - Object element()
- Update methods
  - Vertex insertVertex(o)
  - Edge insertEdge(v, w, o)
  - void removeVertex(v)
  - void removeEdge(e)
- Accessor methods
  - int numVertices()
  - int numEdges()
  - Vertex aVertex()
- Accessor methods
  - Iterator vertices()
  - Iterator edges()
  - Iterator incidentEdges(v)
  - Vertex[2] endVertices(e)
  - Vertex opposite(v, e)
  - boolean areAdjacent(v, w)
- Methods for directed edges
  - Vertex origin(e)
  - Vertex destination(e)
  - boolean isDirected(e)
  - Edge insertDirectedEdge(v, w, o)

Edge List Structure

- Vertex object
  - element
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - directed boolean flag
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects

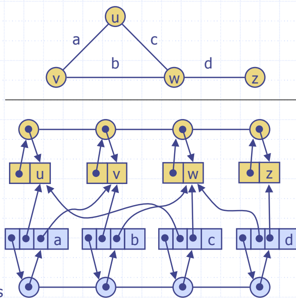


Graph ADT with Positions

- Recall
  - Position = place where item is stored in a sequence
- In Goodrich's book:
  - A Vertex is a Position
  - An Edge is a Position
- Features of Positions
  - Enables faster removal
  - Implementation slightly more complex
  - Unnecessary when removeVertex and removeEdge are not used

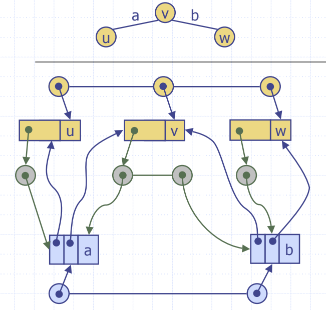
Edge List Structure (w/ Positions)

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



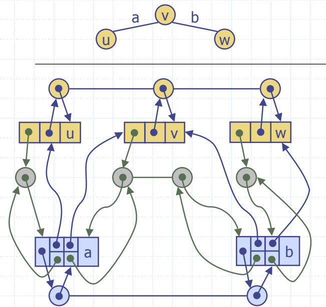
Adjacency List Structure

- Edge list structure
- Each Vertex now stores incidence sequence
  - sequence of references to edge objects of incident edges



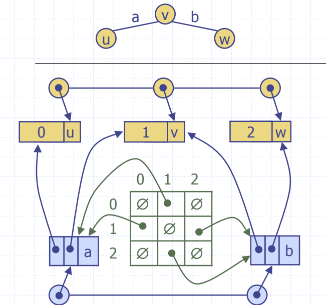
Adjacency List Structure (w/ Positions)

- Edge list structure (w/ Positions)
- 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 Structure

- 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

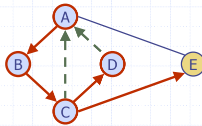


Asymptotic Performance

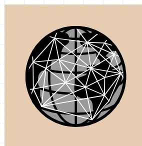
n vertices, m edges	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
Iterating through incidentEdges(v)	$O(m)$	$O(\deg(v))$	$O(n)$
areAdjacent (v, w)	$O(m)$	$O(\min(\deg(v), \deg(w)))$	$O(1)$
insertVertex(o)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(v, w, o)	$O(1)$	$O(1)$	$O(1)$
removeVertex(v)	$O(m)$	$O(\deg(v))$	$O(n^2)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$

Notes: Assuming no parallel edges or self-loops
Using Positions (for removeVertex and removeEdge)

Depth-First Search

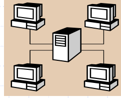


Outline and Reading



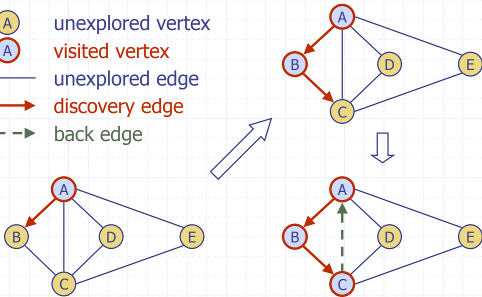
- Depth-first search (§6.3.1)
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS (§6.5)
  - Path finding
  - Cycle finding

Depth-First Search

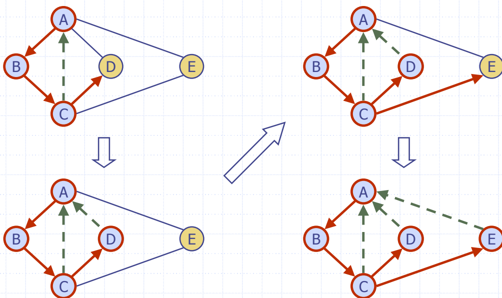


- Depth-first search (DFS) is
  - general graph traversal technique
  - visits all the vertices and edges of G
  - with n vertices and m edges takes $O(n+m)$ time
  - a recursive traversal like Euler tour for binary trees
- A DFS traversal of a graph G can be used to
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
  - Find and report a path between two given vertices
  - Find a cycle in the graph

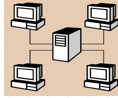
Example



Example (cont.)



DFS Algorithm

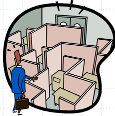


- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

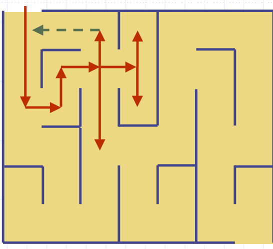
```
Algorithm DFS_Sweep(G)
   Input graph G
   Output labeling of the edges of G
      as discovery edges and
      back edges
 for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
 for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
 for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
   Input graph G and a start vertex v of G
   Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
   setLabel(v, VISITED)
   for all e ∈ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
         w ← G.opposite(v,e)
         if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
         else
            setLabel(e, BACK)
```

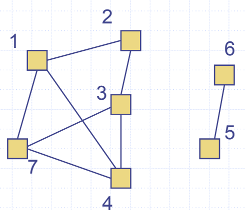

DFS and Maze Traversal



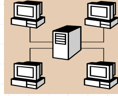
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Another Example of Depth First Search



DFS Algorithm

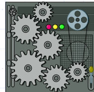


- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm DFS_Sweep(G)
   Input graph G
   Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
   Input graph G and a start vertex v of G
   Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
   setLabel(v, VISITED)
   for all e ∈ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
         w ← G.opposite(v,e)
         if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            DFS(G, w)
         else
            setLabel(e, BACK)
```


Analysis of DFS



- 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
- DFS(G, v) called once for each vertex v
- Inner loop in DFS(G, v) runs in O($\deg(v)$) time
  - Not counting time inside recursive calls
  - Assuming adjacency list implementation
- DFS runs in $O(n + m)$ time
  - Recall that $\Sigma_v \deg(v) = 2m$

Properties of DFS

Property 1

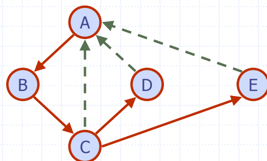
$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

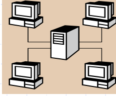
The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v called DFS tree, rooted at v

Property 3

$DFS_Sweep(G)$ visits all vertices and edges of G



Connected Components & DFS Spanning Forest



```
Algorithm ccDFS_Sweep(G)
   Input graph G
   Output labeling of the vertices
      and edges of G based on
      component number.
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  comp_num ← 1
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      perform DFS(G, v) search,
         labeling vertices and edges
         found in the search with
         comp_num.
      comp_num ← comp_num +1
```

- Use ***DFS*(*G, v*)** to label all edges and vertices in one connected component (property 1)
- ***DFS_Sweep*** can label all connected components (property 3)
- In a ***DFS_Sweep*** call, consider subgraph of
  - all vertices
  - all DISCOVERY Edges
- By DFS property 2 and 3, this subgraph is a Spanning Forest of G.

More Properties of DFS

Classifying Edges by DFS

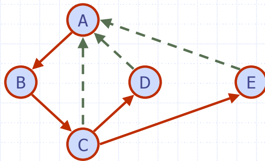
Edge (v,w) type:

- tree = in the DFS tree
- back = w is ancestor of v in DFS tree
- forward = w is descendent of v in DFS tree
- cross = w is neither ancestor nor descendant of v in DFS tree

(Assuming edge first explored from v to w).

Property 4: edges labeled BACK are in fact back edges

Property 5: back edges form a cycle



Path Finding

- Specialize DFS to find a path between two given vertices v and z
- Call ***DFS*(*G, v, z*)** where
  - G is the graph
  - v is the start vertex
  - z is the destination vertex
- 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

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

Cycle Finding

- Specialize DFS 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

```
Algorithm cycleDFS(G, v)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        cycleDFS(G, w)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```


abstract DFS Template

- Use Template Method Design Pattern to implement DFS
- Extend template to implement any algorithm that uses DFS.
- Extensions need to define the following:
  - *startVisit()*
  - *traverseDiscovery()*
  - *traverseBack()*
  - *isDone()*
  - *finishVisit()*
  - a method that returns results.

```
Algorithm DFS(G, v)
  setLabel(v, VISITED)
  startVisit(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        traverseDiscovery(e)
        if (not isDone())
          DFS(G, w)
      else
        setLabel(e, BACK)
        traverseBack(e)
  finishVisit(v)
```