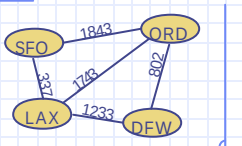
Graphs

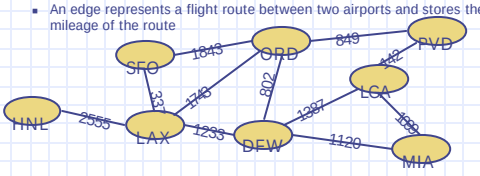


What are graphs?

- Graphs are collections of nodes in which various pairs of nodes are connected by line segments.
- Basic Concepts
  - Definition
  - Applications
  - Terminology
  - Properties
  - ADT
- Data structures for graphs
  - Adjacency list structure
  - Adjacency matrix structure

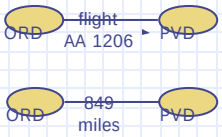
Graph

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are nodes and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



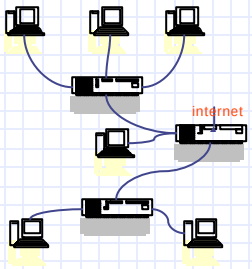
Edge Types

- 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



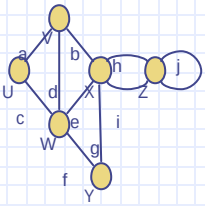
Applications

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



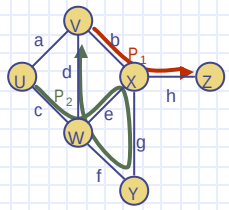
Terminology

- 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
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



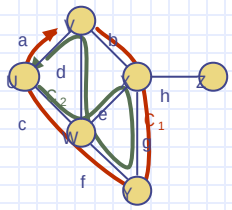
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- 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



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- 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,V) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,V) is a cycle that is not simple



Properties

Property 1

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

Proof: each endpoint is counted twice

Property 2

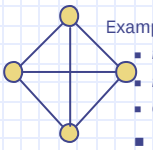
In an undirected graph with no self-loops and no multiple edges

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

Proof: each vertex has degree at most $(n-1)$

Notation

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



Example:

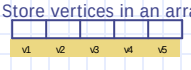
- $n = 4$
- $m = 6$
- $\deg(v) = 3$
- $\sum_v \deg(v) = 12$

Main Methods of the Graph ADT

- Vertices and edges
  - are nodes
  - store elements
- Accessor functions
  - incidentEdges(v)
  - endVertices(e)
  - isDirected(e)
  - origin(e)
  - destination(e)
  - opposite(v, e)
  - areAdjacent(v, w)
- Update functions
  - insertVertex(o)
  - insertEdge(v, w)
  - insertDirectedEdge(v, w)
  - removeVertex(v)
  - removeEdge(e)
- Generic functions
  - numVertices()
  - numEdges()
  - vertices()
  - edges()

Graphs - Data Structures

- Vertices
  - Map to consecutive integers
  - Store vertices in an array

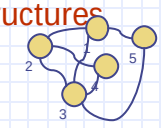


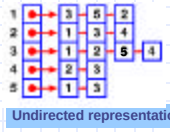
- Edges
  - Adjacency Matrix
    - Booleans -
      1 - edge exists
      0 - no edge
    - $O(|V|^2)$ space (where $|V|$ refers to the number of vertices)

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

Graphs - Data Structures

- Edges
  - Adjacency Lists
    - For each vertex
      - List of vertices "attached" to it
    - $O(|E|)$ space
      - ∴ Better for sparse graphs





Undirected representation

Performance

- n vertices
- m edges
- no parallel edges
- no self-loops

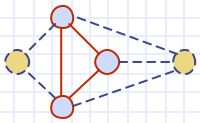
	Adjacency List	Adjacency Matrix
Space	$n + m$	n^2
incidentEdges(v)	deg(v)	n
areAdjacent (v, w)	min(deg(v), deg(w))	1
insertVertex(o)	1	n^2
insertEdge(v, w)	1	1
removeVertex(v)	deg(v)	n^2
removeEdge(e)	1	1

Graphs - Traversing

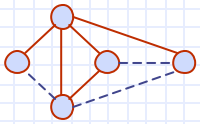
- Choices
  - Depth-First / Breadth-first
- Depth First Search (DFS)
  - Use an array of flags to mark "visited" nodes

Subgraphs

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



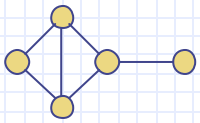
Subgraph



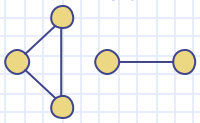
Spanning subgraph

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

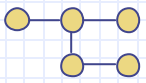


Non connected graph with two connected components

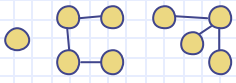
Trees and Forests

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

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



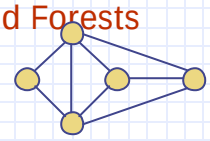
Tree



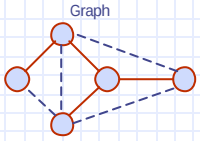
Forest

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



Graph



Spanning tree

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- 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
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph

Preliminary Graph Searching Strategy

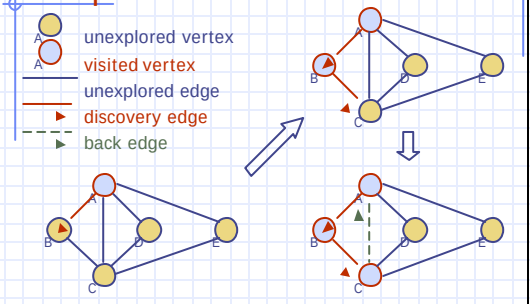
```
void GraphSearch(G, v)
/* Search graph G starting at vertex v */
{
   (let G = (V,E) and let v in V be a vertex of G.)

   for (each vertex w in V that is accessible from v) {
        Visit(w);
   }
}
```

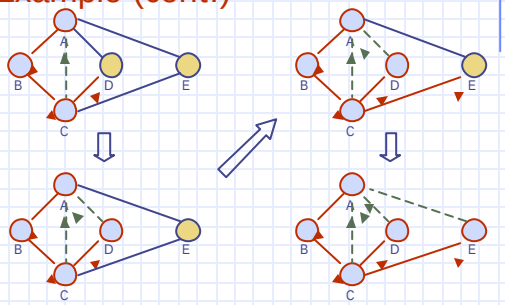

Program Strategy: First Refinement

```
typedef enum {false, true} Boolean;
void GraphSearch(G,v)                    /* Search graph G beginning at vertex v */
{
     (Let G = (V,E) be a graph.)
     (Let C be an empty container.)
     for (each vertex x in V) {
          x.Visited = false;             /* mark each vertex x in V as being unvisited */
     }
     /* Use vertex v in V as a starting point, and put v in container C */
     (Put v into C);
     while (C is non-empty) {
          (Remove a vertex x from container C);
          if (!(x.Visited)) {            /* if vertex hasn't been visited already */
               Visit(x);                 /* visit x, and then */
               x.Visited = true;         /* mark x as having been visited */
               for (each vertex w in Vx) {   /* Enter all unvisited vertices of Vx into C */
                    if (!(w.Visited))  (Put w into C);
               }
          }
     }
}
```


Example

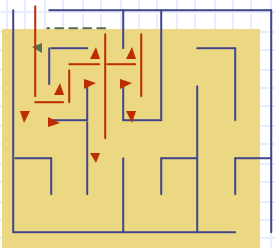


Example (cont.)



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)



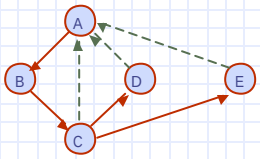
Properties of DFS

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*



Graphs - Depth-First

```
typedef struct t_graph {
  int n_nodes;
  graph_node *nodes;
  int *visited;
  AdjMatrix am;
} graph;
static int search_index = 0;

void search( graph *g ) {
  int k;
  for(k=0;k<g->n_nodes;k++) g->visited[k] = 0;
  search_index = 0;
  for(k=0;k<g->n_nodes;k++) {
    if ( !g->visited[k] ) visit( g, k );
  }
}
```

Graph data structure

Adjacency Matrix ADT

Mark all nodes "not visited"

Visit all the nodes attached to node 0, then ..

Graphs - Depth-First

```
void visit( graph *g, int k ) {
  int j;
  g->visited[k] = ++search_index;
  for(j=0;j<g->n_nodes;j++) {
    if ( adjacent( g->am, k, j ) ) {
      if ( !g->visited[j] ) visit( g, j );
  }
}
```

Mark the order in which this node was visited

Visit all the nodes adjacent to this one

Graphs - Depth-First

```
void visit( graph *g, int k ) {
  int j;
  g->visited[k] = ++search_index;
  for(j=0;j<g->n_nodes;j++) {
    if ( adjacent( g->am, k, j ) ) {
      if ( !g->visited[j] ) visit( g, j );
  }
}
```

Mark the order in which this node was visited

Visit all the nodes adjacent to this one

C hack ...
Should be `g->visited[j] != 0`

`Search_index == 0` means not visited yet!

Analysis of DFS: adjacency matrix

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice (n vertices)
  - once as UNEXPLORED
  - once as VISITED
- Each edge is visited twice (m edges)
  - one for each end-vertex
- Method adjacent is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency matrix structure

Graphs - Depth-First

Adjacency List version of visit

```
void visit( graph *g, int k ) {
  AdjListNode al_node;
  g->visited[k] = ++search_index;
  al_node = ListHead( g->adj_list[k] );
  while( n != NULL ) {
    j = ANodeIndex( ListItem( al_node ) );
    if ( !g->visited[j] ) visit( g, j );
    al_node = ListNext( al_node );
  }
}
```

Assumes a `List` ADT with methods

`ListHead`
`ANodeIndex`
`ListItem`
`ListNext`

Analysis of DFS: adjacency list

- Adjacency List
  - Time complexity
    - Visited set for each node
    - Each edge visited twice
      - Once in each adjacency list
    - $O(|V| + |E|)$ *i.e.,* $O(n + m)$
    - ← $O(|V|^2)$ for dense $|E| \sim |V|^2$ graphs
    - *but* $O(|V|)$ for sparse $|E| \sim |V|$ graphs
- Adjacency Lists perform better for sparse graphs

Graph - Breadth-first Traversal

Breadth-first requires a FIFO queue

```
static queue q;
void search( graph *g ) {
  q = ConsQueue( g->n_nodes );
  for(k=0;k<g->n_nodes;k++) g->visited[k] = 0;
  search_index = 0;
  for(k=0;k<g->n_nodes;k++) {
    if ( !g->visited[k] ) visit( g, k );
  }

void visit( graph *g, int k ) {
  al_node al_node;
  int j;
  AddIntToQueue( q, k );
  while( !Empty( q ) ) {
    k = QueueHead( q );
    g->visited[k] = ++search_index;
    ......
```


Graph - Breadth-first Traversal

```
void visit( graph *g, int k ) {
  al_node al_node;
  int j;
  AddIntToQueue( q, k );
  while( !Empty( q ) ) {
    k = QueueHead( q );
    g->visited[k] = ++search_index;
    al_node = ListHead( g->adj_list[k]);
    while( al_node != NULL ) {
      j = ANodeIndex(al_node);
      if ( !g->visited[j] ) {
        AddIntToQueue( g, j );
        g->visited[j] = -1; /* C hack, 0 = false! */
        al_node = ListNext( al_node );
        }
      }
    }
  }
```

Put this node on the queue