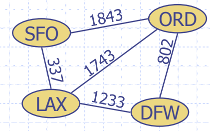
10. Graphs

- Graphs: definition, terminology, properties, and ADT.
- Subgraph, tree, forest, spanning tree, connectivity.
- Data structures for graphs.
- Traversing: depth first search (DFS) and breadth first search (BFS).
- Finding shortest paths.

Graphs

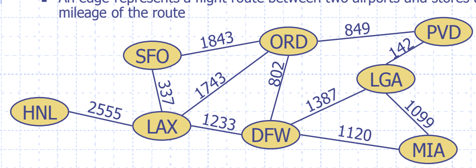


What are graphs?

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

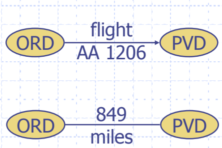
Graph

- 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 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



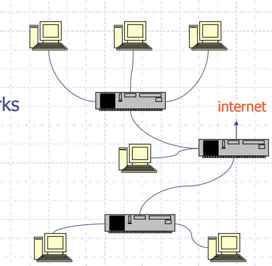
Edge Types

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



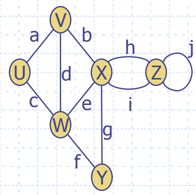
Applications

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



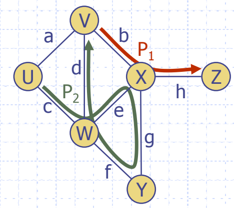
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of edge 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



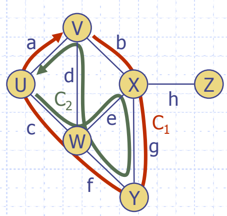
Terminology (cont.)

- 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,↵) 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



Properties

Property 1

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

Proof: each endpoint is counted twice

Property 2

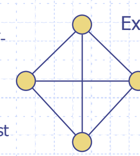
In an undirected graph with no self-loops and no multiple edges

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

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

Notation

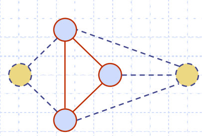
n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



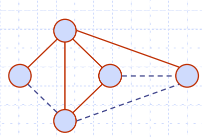
Example:
- $n = 4$
- $m = 6$
- $\deg(v) = 3$
- $\Sigma_v \deg(v) = 12$

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



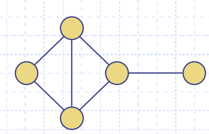
Subgraph



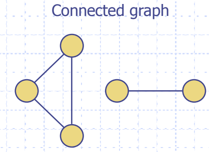
Spanning subgraph

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

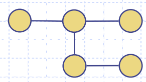


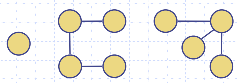
Non connected graph with two connected components

Trees and Forests

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

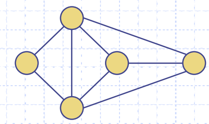
  This definition of tree is different from the one of a rooted tree
- A forest is a disjoint union of trees
- The connected components of a forest are trees

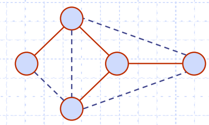

Tree


Forest

Spanning Trees and Forests

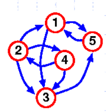
- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest


Graph


Spanning tree

Graphs - Data Structures

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



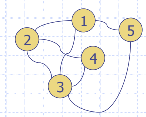
v1	v2	v3	v4	v5

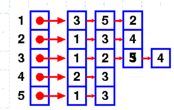
- Edges
  - Adjacency Matrix
    - Booleans -
      1 - edge exists
      0 - no edge
    - $O(|V|^2)$ space (where $|V|$ refers to the number of vertices)

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

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
  - vertices()
  - edges()

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

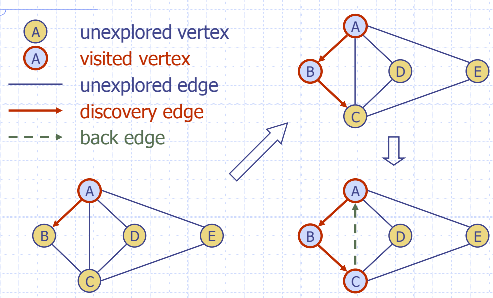
Graphs - Traversing

- Graph traversing (searching) is to visit all the vertices in a graph.
- Choices
  - Depth-First / Breadth-first
- Depth First Search (DFS)
  - Use an array of flags to mark "visited" nodes
- Breadth First Search (BFS)
  - Use an FIFO queue to contain the frontier of "visited" nodes for further search

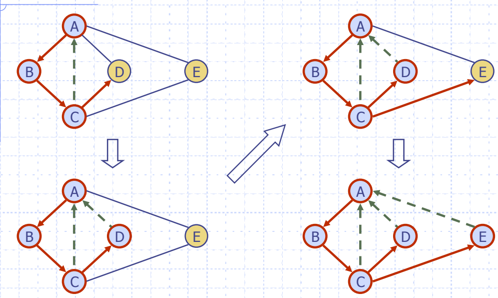
Depth-First Searching

- Depth-first search (DFS) is a general technique for traversing a graph.
- In a DFS, one starts at a node, and explores as far as possible along a path before backtracking.
- When backtracking happens, one goes back to a visited node and explores a path that has not been visited.
- A DFS algorithm can be implemented by using a stack.

Example

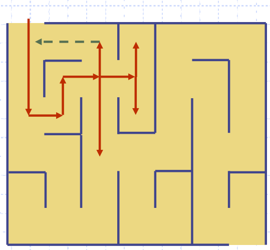


Example (cont.)



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)



Depth-First Searching

- A DFS traversal of a graph G can be used to
  - Visit all the vertices and edges of G
  - Determine whether G is connected
  - Compute the connected components of G
  - Compute 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

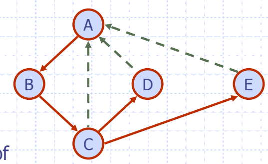
Properties of DFS

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 k, i.e. visit node 0, node 1, ...

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(al_node != 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

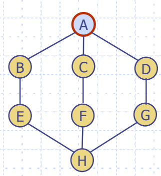
- Adjacency List
  - Time complexity
    - Visited set for each node
    - Each edge visited twice
      - Once in each adjacency list
    - $O(|V| + |E|)$ *i.e.* $O(n + m)$
    - ← $O(|V|^2)$ for dense $|E| \sim |V|^2$ graphs
    - *but* $O(|V|)$ for sparse $|E| \sim |V|$ graphs
- Adjacency Lists perform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

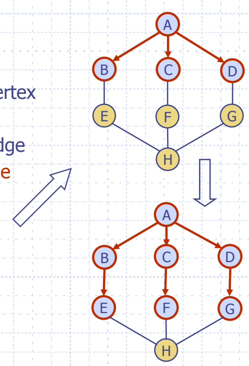
Breadth-First Searching

- Breadth-first search (BFS) is also a general technique for traversing a graph.
- In a BFS, one starts at a node, and explores the neighbor nodes first, and then explores the next level neighbor nodes.
- A BFS algorithm can be implemented by using a queue.

Example

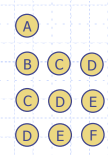
A unexplored vertex
A visited vertex
unexplored edge
discovery edge

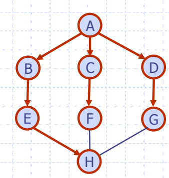




Example

A unexplored vertex
A visited vertex
unexplored edge
discovery edge





Graph - Breadth-first Traversal

Breadth-first requires a FIFO queue

```
static queue q;
void search( graph *g ) {
  q = CreateQueue();
  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 ); /* Add the g node to the queue */
  while( !Empty( q ) ) {
    k = QueueHead( q );    /* Get the queue head */
    g->visited[k] = ++search_index; /* Mark kth g node visited */
    al_node = ListHead( g->adj_list[k]); /* Get the 1st on A list */
    while( al_node != NULL ) {
      j = ANodeIndex(al_node); /* Find the index of the g node */
      if ( !g->visited[j] ) { /* If the g node not visited */
        AddIntToQueue( g, j ); /* Add it to the queue */
        al_node = ListNext( al_node ); /* Get the next neighbor */
        }
      }
    }
  }
```