DFS and BFS

Class 32

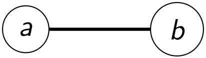
Graph

- a graph $G = (V, E)$ consists of vertices (aka nodes) often denoted v or w
- and edges $e = (v, w)$ which are 2-tuples of vertices
- the number of vertices $n = |V|$
- the number of edges $m = |E|$

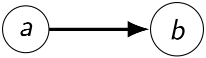
- graphs are represented graphically (duh)
- vertices are drawn as circles, usually with labels inside
- edges as lines

Graph Variations

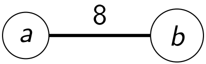
graphs may be



undirected



directed



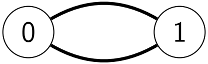
weighted

Graph Terminology

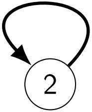
- a graph with directed edges is a digraph
- a path is a sequence of edges
- a cycle is a path which begins and ends on the same vertex, has at least one additional vertex, and has no repeated edges
- an acyclic graph has no cycles
- a digraph without cycles is termed a DAG
- a vertex in an undirected graph has degree: the number of edges touching it
- a vertex in a digraph has indegree and outdegree

Unusual Edges

- in general, we do not allow parallel edges in undirected graphs (but they're ok in digraphs if they go opposite directions)
- in general, we do not allow self-loops in any graph



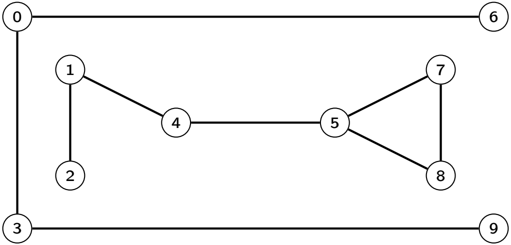
parallel



loop

Connectivity

- an undirected graph is either connected or not
- the connected components are perfect islands



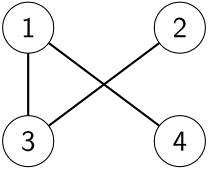
- this graph has two connected components

Graph Implementation

- a graph ADT is a pretty picture
- how do we implement one in a program?
- there are two implementations that are typically used for graphs in programs

  1. adjacency matrix
  2. adjacency lists (lists is plural)

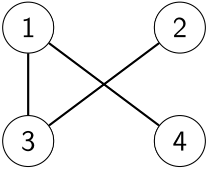
Adjacency Matrix

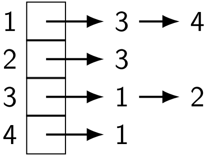


		To			
		1	2	3	4
From	1			T	T
	2			T	
	3	T	T		
	4	T			

- undigraph is symmetric; digraph not necessarily
- undigraph has redundant information; digraph not
- space used is $\mid V \mid \times \mid V \mid$; sparse graph has much wasted space
- very easy to understand and work with
- weighted graph uses weight instead of T/F

Adjacency Lists





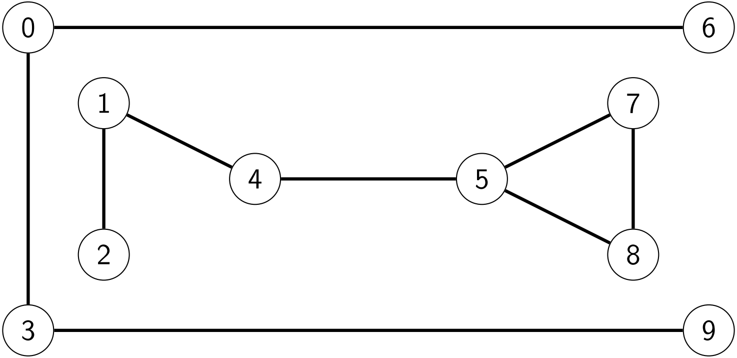
- redundant information in undigraph, not in digraph
- space used is $|E| + |V|$ for digraph, $2|E| + |V|$ for undigraph
- weighted graphs require structs of information
- note: we will always show our lists ordered, so that we will get the same answer when there's a tie

DFS in Undirected Graphs

- given an arbitrary undirected graph
- we wish to access every vertex
- how do we do this?

- there are two fundamental approaches to “iterating” over the vertices of a graph
  - depth-first search: DFS
  - breadth-first search: BFS
- these are called “search” because they “find” every vertex
- note: if there is a choice, we will always choose the next vertex in numerical or alphabetical order (not necessary in real code)

DFS Example



DFS Tree Edges

- when performing DFS, every edge used to reach a previously unvisited vertex is a tree edge because these edges form a tree
- what is a tree?

Tree

An empty graph (0 vertices and 0 edges) is a tree.
A non-empty graph is a tree if it:

$$\left.\begin{array}{l}\text{has } n \text{ vertices and } n-1 \text{ edges}\\ \text{is acyclic}\\ \text{is connected}\end{array}\right\}\ \text{any 2 are sufficient}$$

- a tree may be unrooted with no distinguished vertices
- or rooted with a distinguished root vertex

DFS Back Edges

after dfs:

- DFS produces a rooted DFS search tree
- every edge in the original G not a tree edge is a back edge
- a back edge always connects an indirect ancestor-descendant pair in the DFS tree

Uses of DFS

- DFS shows graph connectivity
- DFS shows whether a graph is cyclic or acyclic
- DFS finds paths in graphs

DFS Implementation

- uses adjacency lists
- see code

Previsit and Postvisit Orderings

- we can keep track of the order in which vertices are arrived at and are left
- for any two vertices, these orderings are either disjoint

$$\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \quad [\,]\,[\,]$$

- or one is contained within the other

$$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u) \quad [\,[\,]\,]$$

- a mixed ordering is impossible — why?

$$\text{pre}(u) < \text{pre}(v) < \text{post}(u) < \text{post}(v) \quad [\,\{\,]\,\}$$

DFS on Directed Graphs

- the same algorithm works for directed graphs as for undirected graphs
- however, the situation with edges is somewhat more complicated
- DFS on a digraph still generates a DFS tree
- there are three categories of non-tree edges

forward	edges lead from a vertex to a non-child descendant in the DFS tree
back	edges lead to an ancestor in the DFS tree
cross	edges connect two vertices that have no ancestor-descendant relationship

Previsit and Postvisit Ordering

- just as with undirected graphs, edge types can be read directly off the relationship of pre and post numbers
- u is an ancestor of v iff u is discovered first and v is discovered during `explore(u)`

$$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u) \quad [\,[\,]\,]$$

- tree edges connect parent to child; forward edges connect ancestor to descendant more distant than that
- back edges connect descendant to ancestor
- cross edges have disjoint numberings

$$\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \quad [\,]\,[\,]$$

Iterating Over Vertices

- DFS iterates over vertices, but has no loop structure
- how does DFS iterate?

Iterating Over Vertices

- DFS iterates over vertices, but has no loop structure
- how does DFS iterate?

- using recursion
- remember, recursion and iteration are interchangeable

- what is the fundamental data structure of recursion?

Iterating Over Vertices

- DFS iterates over vertices, but has no loop structure
- how does DFS iterate?
- using recursion
- remember, recursion and iteration are interchangeable
- what is the fundamental data structure of recursion?
- the runtime stack

DFS and BFS

- bfs is very similar to dfs, but
- bfs uses a queue to implement iterations
- bfs is optimal for finding shortest paths in graphs

BFS Pseudocode

```
void explore(graph)
{
  vector<size_t> distance(graph.size(), SIZE_MAX)
  for (size_t vertex = 0; vertex < graph.size(); vertex++)
  {
    if (distance.at(vertex) == SIZE_MAX)
    {
      distance.at(vertex) = 0
      queue.push(vertex)
      bfs(graph, queue, distance);
    }
  }
}
```

BFS Pseudocode

```
void bfs(graph, queue, distance)
{
  while (!queue.empty())
  {
    vertex = queue.top();
    queue.pop()
    for each vertex w adjacent to vertex
    {
      if (distance.at(w) == SIZE_MAX)
      {
        distance.at(w) = distance.at(vertex) + 1
        queue.push(w)
      }
    }
  }
}
```

BFS

typically do not record pre- and post-visit numberings

on undirected graph

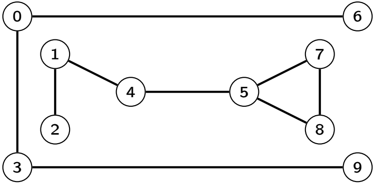
- like dfs, forms a search tree
- a vertex visited for the first time is reached via a tree edge
- all other edges are cross edges
- all cross edges are between vertices at the same level or one level different (why?)
- there are no back edges (why?)

on directed graph

- like dfs, there are forward, back, and cross edges

Connectivity

- an undirected graph is either connected or not
- the connected components are perfect islands

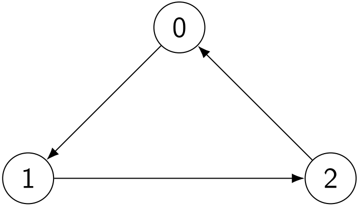


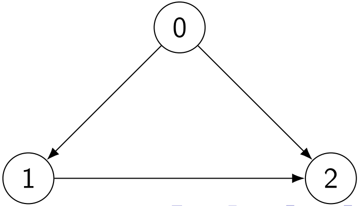
- for digraphs, connectivity is more complicated

Digraphs

- a digraph is strongly connected if for every pair v, w of vertices there is a path from v to w
- a digraph is weakly connected if it is not strongly connected, and for every pair v, w of vertices, there is either a path from v to w or a path from w to v

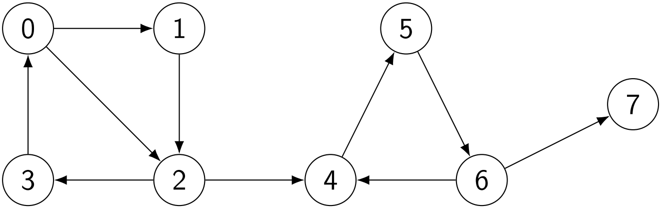
- another way of defining weak connectivity is to pretend that the graph is undirected — if it is connected when considered as an undirected graph, but not strongly connected, then it is weakly connected





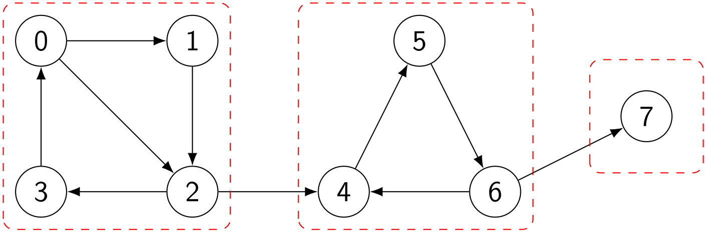
Strongly Connected Components

- we can use DFS for another graph algorithm
- find the strongly connected components of an arbitrary digraph
- the vertices of a digraph can be partitioned into disjoint maximal sets of vertices reachable via a directed path
- each set is a strongly connected component



Strongly Connected Components

- we can use DFS for another graph algorithm
- find the strongly connected components of an arbitrary digraph
- the vertices of a digraph can be partitioned into disjoint maximal sets of vertices reachable via a directed path
- each set is a strongly connected component



Strongly Connected Components

an algorithm for finding strongly connected components

1. perform DFS on the entire graph, generating pre- and post-visit numbers (but ignore the pre-visit numbers)
2. reverse the direction of every edge in the graph
3. perform DFS on the entire reversed graph, but considering the vertices in descending order of postvisit numbering (from step 1)
4. the connected components determined by the DFS in step 3 are strongly connected components of the original graph