DFS and BFS

Class 14
Graph

• a graph is an ADT $G = (V, E)$
• consists of vertices (aka nodes), e.g., $v$ or $w$
• and edges $e = (v, w)$
• we take note of the number of vertices $n = |V|$
• and the number of edges $m = |E|$
• a graph is an ADT $G = (V, E)$
• consists of vertices (aka nodes), e.g., $v$ or $w$
• and edges $e = (v, w)$
• we take note of the number of vertices $n = |V|$
• and 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 list of vertices \( v_1, v_2, \ldots, v_k \).
- Path length is \( k - 1 \).
- Simple path: unique interior vertices.
- Cycle \( v_1 = v_k \), length \( > 0 \), at least one intervening vertex.
- Acyclic graph has no cycles.
- Acyclic digraph is termed a DAG.
- Complete graph \( K_N \) undirected, has every possible edge \( u, v : u \neq v \).
- Vertex degree, indegree, outdegree.
Unusual Cases

- 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 loops in any graph

![Diagram](parallel-loop.png)
Graph Implementation

• a graph ADT is a pretty picture
• but how do we implement one in a program?
• what data structure do we use?
Graph Implementation

- a graph ADT is a pretty picture
- but how do we implement one in a program?
- what data structure do we use?

- two implementations are typically used for graphs in programs
  1. adjacency matrix
  2. adjacency lists (lists is plural)
Adjacency Matrix

- very easy to understand and work with
- undigraph matrix is symmetric; digraph not necessarily
- undigraph has redundant information; digraph does not
- space used is in Θ(|V|^2); sparse graph has much wasted space
- weighted graph uses weight instead of T/F
- C++ does not have a built-in matrix class, so we either have to use an unsafe old-fashioned 2D array, or write our own class
Adjacency Lists

- a vector of linked 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 vertices in the same order as we walk down the list
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

Diagram of a graph with nodes labeled from 0 to 9. The edges connect the nodes in a network.
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?
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:

- has $n$ vertices and $n - 1$ edges
- is acyclic
- is connected

\[
\begin{cases}
\text{any 2 are sufficient}
\end{cases}
\]
after dfs:

• 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
Uses of DFS

- DFS shows graph connectivity
- DFS shows whether a graph is cyclic or acyclic
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
  \[ \text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \]  
• or one is contained within the other
  \[ \text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u) \]  
• a mixed ordering is impossible — why?
  \[ \text{pre}(u) < \text{pre}(v) < \text{post}(u) < \text{post}(v) \]
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 $\text{explore}(u)$

\[
\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u) \quad \left[ \right] \quad \left[ \right]
\]

• 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

\[
\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v) \quad \left[ \right] \quad \left[ \right]
\]
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
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

```c
void bfs(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)
            explore(graph, queue, distance);
        }
    }
}
```
BFS Pseudocode

```java
void explore(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

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