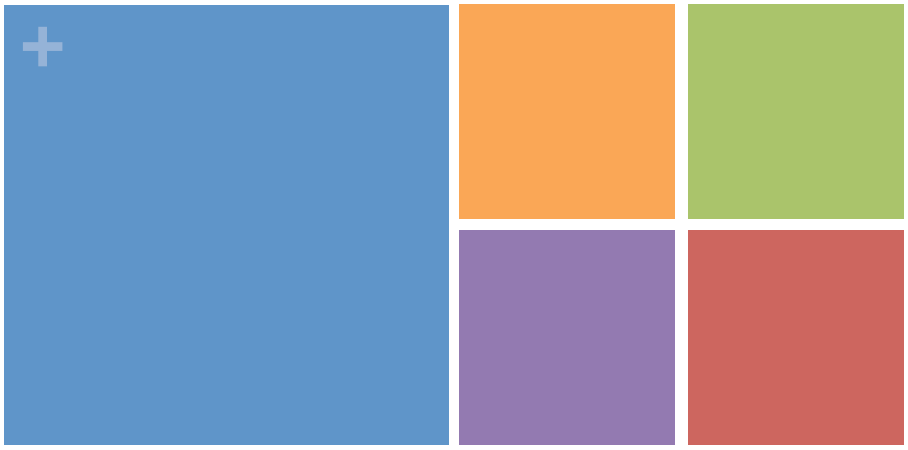


Lecture 39: Graphs

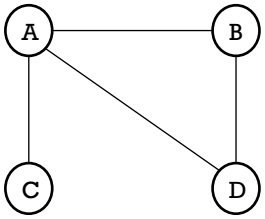
Today

- Reading
  - Correction: JS Chapter 16
  - Not "Weiss Chapter 16"
- Objectives
  - Breadth-first search
  - Depth-first search
  - Dijkstra's Algorithm

Recap: Adjacency Matrix

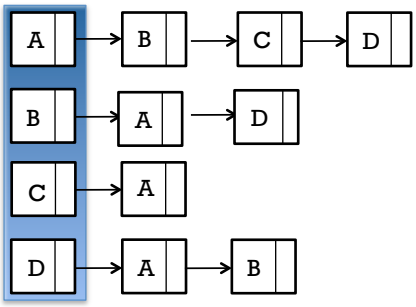
- Store a |V|-by-|V| boolean matrix (two-dimensional array)
  - Entry (i,j) is 1 if there is an edge from vertex i to vertex j
  - Symmetric if undirected
  - Space? Time to lookup edge?

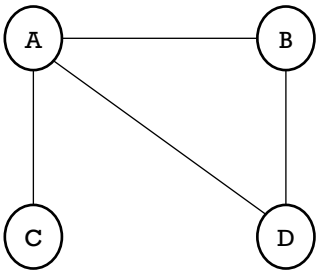
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0



Recap: Adjacency List

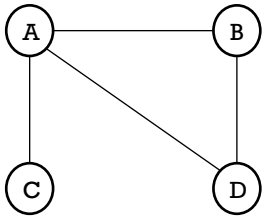
- Store a list of linked lists
  - Use map from vertex labels to lists
  - Space? Time to lookup edge?





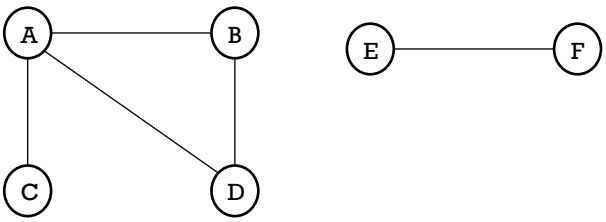
Breadth-first Search

- Equivalent to a level-order traversal of a tree
  - Search all nodes 1 away, 2 away, 3 away, etc
- Uses a queue data structure
- Basic algorithm:
  - Enqueue the start node
  - While the queue is not empty:
    - Dequeue a node
    - Check if node previously visited
    - If not, mark as visited and enqueue all children



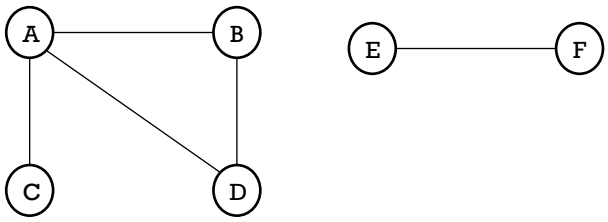
Breadth-first Search

- If graph has multiple connected components
  - Wrap BFS inside a for-loop that iterates through all nodes
- See *bfs_dfs_demo.cpp*
  - Uses a `typedef` (allows you to rename a type)
  - Better to use `map<string, vector<string>>` instead of `pair`



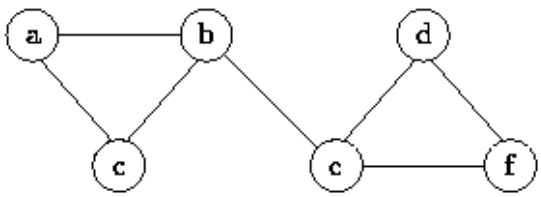
Depth-first search

- Equivalent to a pre-order traversal of a tree
  - except may get stuck in cycles
- Can use the same algorithm as BFS
  - Either use a stack or use recursion



Detecting Cycles

- Can use depth-first search to see if we loop back
- How can we detect a loop?
  - A node in our adjacency list has already been visited but it is not the node that added us (we call this node our parent)
  - Works for an undirected graph



Single Source Shortest Paths

- Starting at node s, find the "shortest" path to all other nodes
- If edges have no weight then can use BFS
  - "Shortest path" is defined to be the path with fewest edges
- If edges have (non-negative) weights, use Dijkstra's Algorithm
  - Dijkstra's Algorithm is BFS with a priority queue
  - The priority is the distance from the start node to current node
  - Keep track of parent node (i.e. preceding node in the path)

Single Source Shortest Path

```
map<int,int> shortest paths(int start,
                            const map<int,list<pair<int,int> > > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {

    int v = frontier.top_serialnumber();
    int p = frontier.top_priority();
    frontier.pop();

    for (the neighbors (n,w) of v)
      if (n == parents[v])
        ; // do nothing
      else if (n is not in the frontier and has not been visited) {
        parents[n] = v;
        frontier.push(n, p + w);
      }else if (p + w < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + w);
      }

    } // end while

  return parents;
}
```

priority queue to keep track of nodes to be visited

remove node with lowest priority, i.e. "closest" node

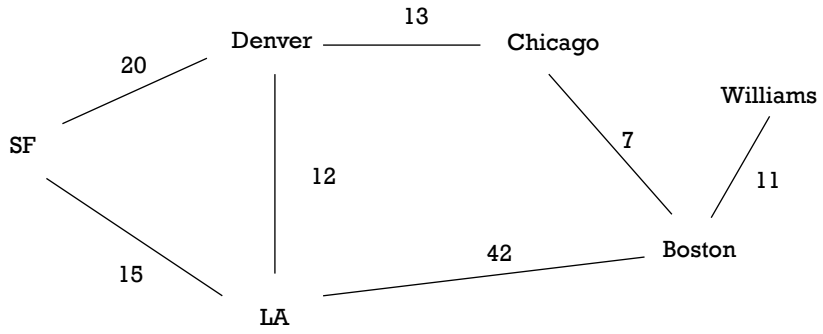
for each neighboring/ adjacent node

This is the node that put us on the queue! Nothing to do

First time we've seen this node

Found a shorter path to this node

Single Source Shortest Path



This Week's Assignment

- Write three graph algorithms:
  - Use DFS to find all connected components
  - Use DFS to return a cycle if one exists
  - Use Dijkstra's algorithm to find single source shortest paths
- The graph is stored as an adjacency list:

```
// maps node label to adjacency list
map<int, list<int>>
```

- Create a graph from Netflix data. Experiment with different ways of defining "adjacency"
- Run connected component function on Netflix graph