LECTURE 37: GRAPHS

Today

- Reading
  - Bailey Chapter 16 (Graphs)
- Objectives
  - Iterators
  - Graph algorithms

C++ Standard Template Library

- Contains familiar collections
  - vector
  - deque
  - list
  - map
  - priority_queue
  - pair

Iterators

- Iterator type depends on container and const-ness

- Map from name to mailing address:

```
unordered_map<string, string>  address_book;
```

- Possible iterator declarations:

```
unordered_map<string, string>::iterator itr;
unordered_map<string, string>::const_iterator itr;
```

Iterators

- Operators on iterators
  - itr++ and ++itr
  - *itr returns reference to element being pointed to
- Operators on collection
  - begin() – returns iterator to first element
  - end() – returns iterator pointing just past last element

Iterators

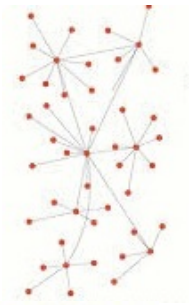
```
vector<int> vec;

// add some integers to vec

// itr is an iterator over vec
vector<int>::iterator itr = vec.begin();

// use itr in a for-loop to loop over vec
for(; itr != vec.end(); ++itr) {
   cout << *itr << endl;
}
```

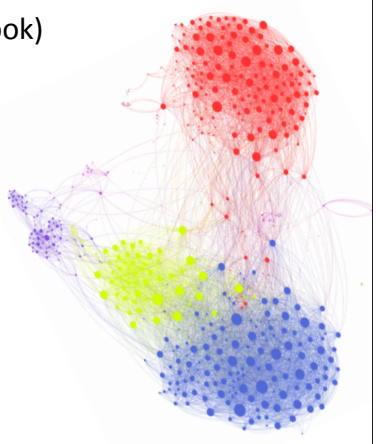
Graphs in Real Life



- Transportation networks
  - Airline flight paths
  - Roads, interstates, etc. (Google maps)
  - Finding shortest route, cheapest route
  - Find route that minimizes fuel costs
- Communication networks
  - Electrical grid, phone networks, computer networks
  - Minimize cost for building infrastructure
  - Minimize losses, route packets faster

More Graphs

- Social networks
  - People and relationships (e.g. Facebook)
  - Does this person know this person?
  - Can this person introduce me to that person (e.g. job opportunities)?



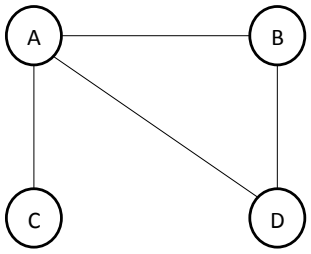
http://griffsgraphs.com/2012/07/02/a-facebook-network/

Definitions

- A graph is a generalization of a tree
- A graph G is a pair (V,E)
  - V is a finite, non-empty, set of vertices
  - E is the set of edges that connect pairs of vertices
  - Called “vertices” or “nodes”

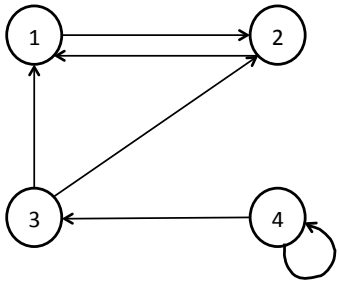
Example: Undirected Graph

- G = (V,E) where
- V = {A, B, C, D}
- E = {(A, C), (A,B), (A,D), (B,D)}



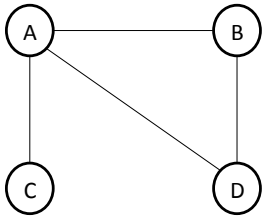
Example: Directed Graph

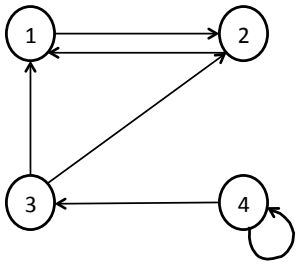
- G = (V,E) where
- V = {1, 2, 3, 4}
- E = {(1,2), (2,1), (3,1), (4,3), (4,4), (3,2)}



Definitions

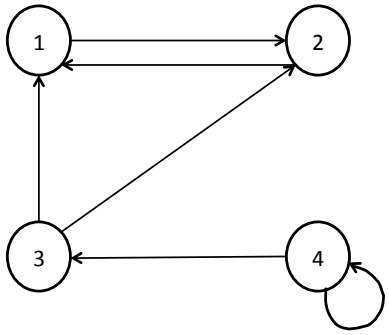
- path
- simple path
- path length
- cycle
- simple cycle





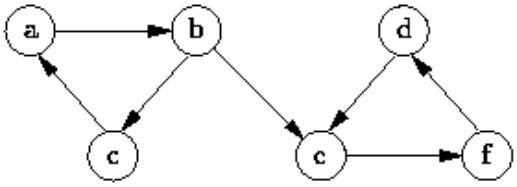
Definitions

- self loop
- incident
- adjacent
- degree
- simple graph
- acyclic graph

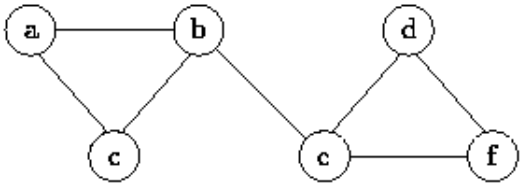


Connected Components

weakly connected



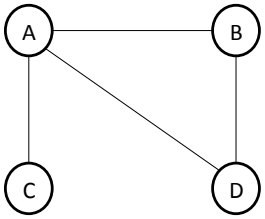
connected



Adjacency Matrix

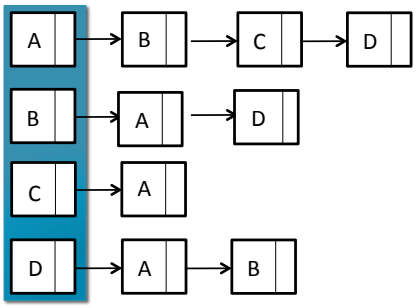
- Store a |V|-by-|V| boolean matrix
  - Entry (i,j) is 1 if there is an edge from vertex i to vertex j
  - Symmetric if undirected
  - Space? Time to lookup edge? Time to iterate over incident edges?

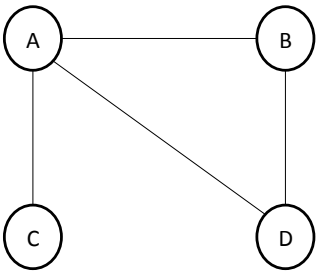
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0



Adjacency List

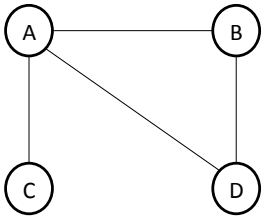
- Store a list of linked lists
  - Use map from vertex labels to lists
  - Space? Time to lookup edge? Time to iterate over incident edges?





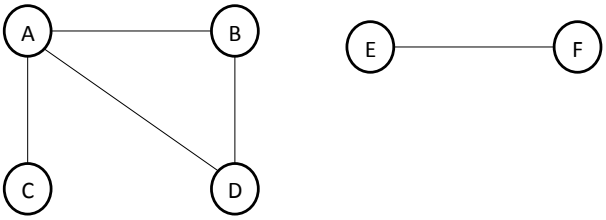
Breadth-first Search

- 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
- Use same algorithm as BFS but replace queue with stack/ recursion