Lecture 17: Representing network of entities
About These Slides

- Based on Chapter 23 of the book
  
  *An Introduction to Programming Through C++*
  
  by Abhiram Ranade (Tata McGraw Hill, 2014)

- Original slides by Abhiram Ranade
  
  - First update by Sunita Sarawagi
Outline

• We would like to represent any object of interest on a computer
  – Road map of India
  – Electrical circuit
  – Mathematical expressions
  – ...

• All of these are examples of “graphs”

• How to represent graphs on a computer
Graph

- Graph $G = (V,E)$, where
  - $V =$ set of “vertices”
  - $E =$ set of “edges” = sets of pairs of vertices
- Example: Road map of India
  - $V =$ set of cities
  - $E =$ pairs of cities connected directly by a road
- Edges may be ordered or unordered
  - Unordered: $(u,v)$ and $(v,u)$ both refer to the same edge
  - Ordered: $(u,v)$ and $(v,u)$ refer to distinct edges
- Roadmap: edges are usually unordered
  - However, we may choose ordered edges to indicate one-way roads.
- Vertices/Edges may be associated with attributes
  - Vertices in road map may have names, e.g. city names
  - Edges in road map may have names and lengths
A graph of friends

• Vertices = persons, Edge: connect friends
• Unordered: friendship is mutual
• Example:
  - V =\{Harry, Hermione, Ron, Draco, Crabbe\}
  - E =\{(Harry, Hermione), (Ron, Hermione), (Harry, Ron), (Draco, Crabbe)\}
Representing a person

• “For every entity you should have a class”
  – What information would you put in each object of the class?

```cpp
struct Person{
    string name;
    string address;
    vector<Person*> friends;
};
```

• Person* or Person?
Representing the 5 persons

Person persons[5];
persons[0].name = “Harry”;
persons[1].name = “Hermione”;
persons[2].name = “Ron”;
persons[3].name = “Draco”;
persons[4].name = “Crabbe”;

• Now we have created the vertices
Adding the edges

- Harry, Hermione are friends, so we should do...
  persons[0].friends.push_back(&persons[1]);
  persons[1].friends.push_back(&persons[0]);
- We need to make entries for both.
- So we could instead have a function
  void MF(Person &p, Person &q) {
    p.friends.push_back(&q);
    q.friends.push_back(&p);
  }
- So now we just call it for each friendship:
  MF(persons[0], persons[1]);
  MF(persons[1], persons[2]);
  MF(persons[2], persons[0]);
  MF(persons[3], persons[4]);
Exercise

• Read in the name of a person and print that person's friends.

```cpp
    cin >> name;
    for(int i=0; i<5; i++)
        if(name == persons[i].name){
            for(size_t j=0;
                j<persons[i].friends.size();
                j++)
                cout << persons[i].friends[j]->name<<endl;
        }
```
A C++11 Enhancement

• Read in the name of a person and print that persons friends.

```cpp
    cin >> name;
    for(int i=0; i<5; i++)
        if(name == persons[i].name){
            // fp is of type (Person * &)= auto
            for(auto fp : persons[i].friends)
                cout << fp->name << endl;
        }
```

• “Range based loop”: for( type id : container){..}

• Block executed for all elements id of the container

• vectors, maps, are containers
Another enhancement: can we avoid the search completely?

```cpp
map<string, vector<string>> friends;
friends["Harry"].push_back("Hermione");
friends["Hermione"].push_back("Harry");
...

// Print friends of all persons
for(auto p : friends){
    cout << p.first <<": ";
    for(auto f : p.second) cout << f <<' ';
    cout << endl;
}
```
What if edges have attributes?

- Suppose friendships have "intensity"
- Solution 1:

```cpp
struct Person{
    string name;
    vector<Person*> friends;
    vector<double> intensity;
};
```
struct EdgeData{
    double intensity;
    double duration;
};

struct Person{
    string name;
    vector<Person*> friends;
    vector<EdgeData*> edgedata;
};

void makefriends(Person &p, Person &q, EdgeData *e){
    p.friends.push_back(&q);
    p.edgedata.push_back(e);
    q.friends.push_back(&p);
    q.edgedata.push_back(e);
}
Remarks

• Solution 1 stores two copies of intensity – in each of the two Person objects

• Solution 2 stores one copy; each Person object has a pointer to it.
  – Will require less memory if there is a lot of edge data
  – If there are multiple copies of the same information we are always worried about updating both copies consistently – source of bugs.

• Vertex data and edge data can both be on the heap if needed.
Adjacency Matrix Representation

struct VertexData{ string name;};
struct EdgeData{
    bool valid;
    double intensity, duration;
};
VertexData v[nVertices];
EdgeData e[nVertices][nVertices];
• If there is an edge from vertex i to vertex j, then set
  – e[i][j].valid = true, e[i][j].intensity = ...
• If no edge then set
  – e[i][j].valid = false;
• Many variations possible. See book.
Remarks

• For graph with V vertices and E edges
  – Adjacency list uses: $O(V+E)$ memory
  – Adjacency matrix uses: $O(V^2)$ memory
  – Adjacency list is better if graph has few edges.
Announcements

• Thursday graded lab.
• Cribs: empty, negative marks for needless cribs.
• Help session.
Example Graph Queries

• Check if x and y are direct friends.

```cpp
map<string, vector<string>> friends;
 cin >> x >> y;
 bool xy_friends = false;
 for (string f : friends[x]) {
   if (f == y) {
     xy_friends = true;
     break;
   }
 }
```
Example Graph Queries

- Find friend of friends, or the set of nodes reachable by one hop on a graph.

```cpp
map<string, vector<string>> friends;
cin >> query;
map<string, int> friendsOfFriends;
for (string f : friends[query]) {
    for (string g : friends[f]) {
        friendOfFriends[g]++;
    }
}
for (auto s : friendOfFriends)
    cout << s.first << " ";
```
Example: Is there a path between two nodes?

```cpp
bool check_friends(string x, map<string,bool>& visited, string y) {
    if (x == y) return true;
    visited[x]=true;
    for (string f : friends[x]) {
        if (visited.find(f) == visited.end()) {
            if (check_friends(f, visited, y)) return true;
        }
    }
    return false;
}
main() {cin >> x >> y; map<string,bool> visited;
cout << check_friends(x,visited, y);}
```
Graph between different types of nodes

- **Web:**
  - Given pages, each with a url and a sequence of words in it.
  - Given a query word, find all page-urls that contain it.
- **View this as a graph with**
  - two types of nodes
    - Page nodes
    - Word nodes.
  - Directed edges from pages to word that contain it.
void loadPages(Web &web) {
    map<string, vector<string>> pages;
    map<string, vector<string>> words;
    for (int i = 0; i < num_pages; i++) {
        string url; int num_words;
        cin >> url >> num_words;
        while (!num_words) {
            string word; cin >> word;
            pages[url].push_back(word);
            words[word].push_back(url);
        }
    }
    while (true) {
        cin >> query;
        for (auto u : words[query]) {
            cout << u << " ";
        }
    }
}