Graphs – Implementation Tips

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder
Graphs – Implementation Tips

Standard reminder to set phones to silent/vibrate mode, please!
Graphs – Implementation Tips

• Today's lesson:
  • Finish off the topic of Graphs with some tips on the various techniques to implement them
    - Should be helpful for your lab work this week!
  • Look into the C++ Standard Library `vector` class
    - Generic array class that handles memory management “behind the scenes”
  • Take a quick look a `list` class, also from the Standard Library.
Graphs – Implementation Tips

• We'll be using my own introductory tutorial on C++ vectors:
  http://www.mochima.com/tutorials/vectors.html
Graphs – Implementation Tips

• Copyright / academic integrity statement:
  • The code samples are only for illustration purposes.
  • You are NOT ALLOWED to directly copy fragments of code into your lab project.
  • You are of course allowed to use the ideas; but directly copying from these slides to your project would constitute an academic offence.
Graphs – Implementation Tips

- We recall our two typical implementation strategies — adjacency lists and adjacency matrix
- We briefly discussed this when talking about topological sort.
Graphs – Implementation Tips

- With *adjacency lists*, vertices are associated with a number between 0 and $|V| - 1$, or between 1 and $|V|$, disregarding element 0.

- An array of adjacencies is defined — each element of the array is a list (either a dynamic array or a linked list) of the vertices adjacent to the vertex corresponding to that subscript.
Graphs – Implementation Tips

- **Ajacency list — example:**

  \[
  V = \{1, 2, 3, 4, 5, 6\} \\
  E = \{(1,2), (1,4), (2,3), (2,4), (2,5), (3,4), (5,3), (6,3), (6,5)\}
  \]
Graphs – Implementation Tips

• Adjacency list — example:

\[ V = \{ 1, 2, 3, 4, 5, 6 \} \]
\[ E = \{ (1, 2), (1, 4), (2, 3), (2, 4), (2, 5), (3, 4), (5, 3), (6, 3), (6, 5) \} \]

Representation:
Graphs – Implementation Tips

- Adjacency list — example:

\[
V = \{1, 2, 3, 4, 5, 6\} \\
E = \{(1,2), (1,4), (2,3), (2,4), (2,5), (3,4), (5,3), (6,3), (6,5)\}
\]

Could also be a “jagged array” (an array of arrays):
Graphs – Implementation Tips

- The former could be implemented as a vector of lists:

```cpp
std::vector<std::list<int>>
```
Graphs – Implementation Tips

• The former could be implemented as a vector of lists:

  ```cpp
  std::vector< std::list<int> >
  ```

• Notice a little “paper cut” feature in C++ (fixed in the new C++11 standard), the space between the two `>` at the end is not optional!! (if you omit it, the compiler would parse the `>>` as the bitshift operator, and would result in a syntax error!)
Graphs – Implementation Tips

• The latter is similar — a vector of vectors:

```cpp
std::vector< std::vector<int> >
```
Graphs – Implementation Tips

- Adding edges works similarly with both vector of vectors and vector of lists — the following example inserts edge \((u, v)\):

  ```cpp
  vector< vector<int> > adjacencies(n+1);
  // ...
  adjacencies[u].push_back (v);
  ```

  (the sample assumes a `using std::vector;` directive, or `using namespace std;` — the latter generally not recommended on a header file)
Graphs – Implementation Tips

• This trick works for unweighted graphs, since we only need to denote the presence of an edge.
Graphs – Implementation Tips

• This trick works for unweighted graphs, since we only need to denote the presence of an edge.

• The fact that the value is in the “row” array means that that vertex is adjacent to the vertex corresponding to the subscript of the given row.
Graphs – Implementation Tips

- This trick works for unweighted graphs, since we only need to denote the presence of an edge.
  - The fact that the value is in the “row” array means that that vertex is adjacent to the vertex corresponding to the subscript of the given row.
- If we need to indicate a weight (e.g., a double value), we could use an additional trick.
Graphs – Implementation Tips

• Create a simple class Edge that has two data members — target vertex, and weight:

```cpp
class Edge
{
    int d_vertex;
    double d_weight;

public:
    Edge (int vertex, double weight)
        : d_vertex(vertex), d_weight(weight)
    {}

    int vertex() const { return d_vertex; }
    double weight() const { return d_weight; }
};
```
Graphs – Implementation Tips

• With this, you would declare a vector of vectors of Edges:

```cpp
vector< vector<Edge> > adjacencies;
```

• Adding edges is similar — the following example inserts edge \((u, v)\) with weight \(w\):

```cpp
vector< vector<int> > adjacencies(n+1);
// ...
adjacencies[u].push_back (Edge(v,w));
```
Graphs – Implementation Tips

As an example, if we wanted to compute the weight of the graph (the sum of the weights of all edges):

```cpp
for (vector< vector<Edge> >::size_type v = 1; v < adjacencies.size(); ++v)
{
    for (vector<Edge>::size_type e = 0; e < adjacencies[v].size(); ++e)
    {
        sum += adjacencies[v][e].weight();
    }
}
```
Graphs – Implementation Tips

- If we're representing a directed graph, we notice that this representation provides a very efficient (constant time) way to determine the out-degree of a vertex — for example, the out-degree of vertex $v$ is given by:

  \[
  \text{adjacencies}[v].\text{size()}
  \]
Graphs – Implementation Tips

• If we're representing a directed graph, we notice that this representation provides a very efficient (constant time) way to determine the out-degree of a vertex — for example, the out-degree of vertex \( v \) is given by:

\[
\text{adjacencies}[v].size()
\]

• (BTW... Why only for directed graphs?)
Graphs – Implementation Tips

• Removing an edge is also simple (plus/minus complications with first locating the edge). To remove elements from a vector (inefficient, but simple in terms of syntax):

http://www.mochima.com/tutorials/vectors.html#insert_remove
Graphs – Implementation Tips

- If you want to use a linked list (std::list) for the list of edges, removing is more efficient, but the code as a whole gets slightly more complicated — see my STL tutorial for details: http://www.mochima.com/tutorials/STL.html

- The code to remove edge \((u,v)\) goes more or less like:
Graphs – Implementation Tips

```cpp
for (list<Edge>::iterator e = adjacencies[u].begin();
e != adjacencies[u].end();
++e)
{
    if (e->vertex() == v)
    {
        adjacencies.erase (e);
        break;
    }
}
```
Graphs – Implementation Tips

• Adjacency lists have the advantage of being more storage-efficient when $|E|$ is much less than $|V|^2$
Graphs – Implementation Tips

- Adjacency lists have the advantage of being more storage-efficient when $|E|$ is much less than $|V|^2$
- Additionally, they have the advantage of more efficient access for things like operations on each of the adjacent vertices to a given vertex (such as Prim's and Dijkstra's algorithms)
  - We just need to iterate over the elements of the linked list or array of edges (the “row”)
Graphs – Implementation Tips

- Adjacency lists have the advantage of being more storage-efficient when \(|E|\) is much less than \(|V|^2\)
- Additionally, they have the advantage of more efficient access for things like operations on each of the adjacent vertices to a given vertex (such as Prim's and Dijkstra's algorithms)
  - We just need to iterate over the elements of the linked list or array of edges (the “row”)
  - So this is also an advantage only when we have few edges.
Graphs – Implementation Tips

- Adjacency lists are not particularly efficient, for example, to test whether a vertex is adjacent to another vertex — the sample below checks whether vertex $v$ is adjacent to vertex $u$:

```cpp
for (vector<Edge>::size_type e = 0; 
     e < adjacencies[u].size(); 
     ++e)
{
    if (adjacencies[u][e].vertex() == v)
    {
        return true;
        // assuming a function/method
    }
}
return false;
```
Graphs – Implementation Tips

• With a linked list, the loop would go more or less like:

```cpp
for (list<Edge>::iterator e = adjacencies[u].begin();
    e != adjacencies[u].end();
    ++e)
{
    if (e->vertex() == v)
    {
        return true;
    }
}
return false;
```
Graphs – Implementation Tips

• Let's take a look at the implementation using the Adjacency matrix approach...
Graphs – Implementation Tips

- Two-dimensional dynamic arrays in C++ can be conveniently implemented as an array of arrays (that is, a vector of vectors).
Graphs – Implementation Tips

- Two-dimensional dynamic arrays in C++ can be conveniently implemented as an array of arrays (that is, a vector of vectors).
- The main difference is that we want to have the allocated full-size for all rows right from the start.
  - So, in the constructor we would do something like:
Graphs – Implementation Tips

```cpp
Graph::Graph (int n)
    : adjacencies(n+1)
{
    for (vector<···>::size_type i = 1; i <= n; ++i)
    {
        adjacencies[i].resize(n+1);
    }
}
```

See [http://www.mochima.com/tutorials/vectors.html#resize](http://www.mochima.com/tutorials/vectors.html#resize) for more details.
Graphs – Implementation Tips

• The data type of `adjacencies` in this case would be `vector< vector<bool> >` if an unweighted graph (we just store `true` in `adjacencies[u][v]` to indicate that there is an edge from `u` to `v`), or `vector< vector<double> >` if weighted.
Graphs – Implementation Tips

- The data type of `adjacencies` in this case would be `vector< vector<bool> >` if an unweighted graph (we just store `true` in `adjacencies[u][v]` to indicate that there is an edge from `u` to `v`), or `vector< vector<double> >` if weighted.

- In this sense, the implementation for a weighted graph is a little bit simpler (really, just a liiitle bit)
Graphs – Implementation Tips

• So, what types of operations are efficient with an adjacency matrix?
Graphs – Implementation Tips

- So, what types of operations are efficient with an adjacency matrix?
  - Checking if two given vertices are adjacent is quite trivial — example to check if vertex v is adjacent to vertex u:

```cpp
bool Graph::adjacent (int u, int v) const
{
    return adjacencies[u][v];
}
```
Graphs – Implementation Tips

- That was for an unweighted graph (adjacencies stores bool values). For a weighted graph, assuming non-negative weights:

```cpp
bool Graph::adjacent (int u, int v) const
{
    return adjacencies[u][v] >= 0;
}
```
Graphs – Implementation Tips

• Things that require going over each adjacent vertex tend to be less efficient:

```cpp
    for (vector<double>::size_type i = 1; i < adjacencies[u].size(); ++i)
        {
            if (adjacencies[u][v] >= 0)
                {
                    // do whatever is required
                }
        }
```
Graphs – Implementation Tips

- With either approach (adjacency list or adjacency matrix), one important advantage is that a big portion of the memory management is taken care of for you — classes vector and list encapsulate all the memory management aspects.
  - Their constructor, destructor, copy-constructor, and assignment operators handle all the details.
Graphs – Implementation Tips

• But this is really no different than coding it yourself by properly breaking down the design into pieces:
Graphs – Implementation Tips

• But this is really no different than coding it yourself by properly breaking down the design into pieces:

  • If you create your linked list class, you'd provide a constructor, destructor, copy-constructor, etc.
Graphs – Implementation Tips

• But this is really no different than coding it yourself by properly breaking down the design into pieces:
  • If you create your linked list class, you'd provide a constructor, destructor, copy-constructor, etc.
  • So, if you use that linked list as a data member in your class Graph, you wouldn't need to provide a destructor for Graph (since the data member encapsulates all the functionality required).
Summary

• During today's class, we:
  • Finished off the topic of graphs.
  • Discussed some implementation details and tips.
  • Looked into standard library facilities vector and list
    - Vector useful for both adjacency lists and adjacency matrix
    - List requires iterators for accessing elements.