ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Basic Graph Algorithms in C++ & STL

Florian Jug, Christoph Krautz

September 16, 2009

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Introduction

1.1 What you'll find here

- How to store a graph
  1. Adjacency Matrix
  2. Adjacency List
- Basic ideas + code snippets for...
  1. Graph traversal
  2. Shortest Path
  3. Minimum Spanning Trees

1.2 Literature - will be available in the computer room

- C++ Tutorial: http://www.cplusplus.com/doc/tutorial/
- C++ Reference: http://www.cppreference.com/wiki/start
- STL Reference: http://www.sgi.com/tech/stl/table_of_contents.html
- Bjarne Stroustrup, The C++ Programming Language
- Meyers, More Effective C++
- Mayers, Effective STL
- Sedgewick, Algorithmen in C++

- Sedgewick, Algorithms in C++ (Part 5)
- Cormen, Introductions to Algorithms

Chapter 2

How to Store a Graph

2.1 Adjacency Matrix

Idea: save a n^2 matrix of `bools` or other data types (e.g., if the edges have weights).

- Good: speed of adjacency lookup
- Bad: space, speed of (out)edge iteration

```
vector<char> adj(n*n,0);
```

You can also use `bool` or `int` for various speed/space tradeoffs.

2.2 Adjacency List

Idea: save a list of lists. One list for each node, containing all the adjacent nodes.

- Good: space, speed of (out)edge iteration
- Bad: speed of adjacency lookup

```
vector<vector<int> > adj_list (n);
vector<vector<pair<int, double> > > weighted_adj_list (n);
...
const vector<int>& neighborhood = adj_list[node_number];
int first_neighbor = neighborhood[0];
...
const vector<pair<int, double> >& neighborhood = weighted_adj_list[node_number];
```

```
int first_neighbor = neighborhood[0].first;
double weight_of_n = neighborhood[0].second;
```

Chapter 3

Basic Ideas + Code Snippets for Elementary Graph Algorithms

3.1 Graph Traversal

3.1.1 BFS - Breadth First Search

```
vector<bool> visited(n, false);
queue<int> bfs_queue;
int next_num = 0;
vector<int> bfs_num(n);
for (int u = 0; u < n; u++) {
    if (visited[u])
        continue;
    bfs_queue.push(u);
    visited[u] = true;
    while (!bfs_queue.empty()) {
        int v = bfs_queue.front();
        bfs_queue.pop();
        bfs_num[v] = next_num++;
        vector<int>& neighbors = edges[v];
        for (vector<int>::iterator it = neighbors.begin();
             it != neighbors.end(); it++) {
            if (!visited[*it]) {
                visited[*it] = true;
                bfs_queue.push(*it);
            }
        }
    }
}
```

3.1.2 DFS - Depth First Search

```
void dfs(int v, const vector<vector<int> > &edges,
         vector<bool> &visited, int &next_num,
         vector<int> &dfs_num)
{
    visited[v] = true;
    dfs_num[v] = next_num++;
    cerr << "visit " << dfs_num[v] << ": " << v << "\n";
    const vector<int> &neighbors = edges[v];
    for (vector<int>::const_iterator it = neighbors.begin();
         it != neighbors.end(); it++) {
        if (!visited[*it])
            dfs(*it, edges, visited, next_num, dfs_num);
    }
}
```

and then later

```
vector<bool> visited(n, false);
int next_num = 0;
vector<int> dfs_num(n);
dfs(0, edges, visited, next_num, dfs_num);
```

Writing a DFS recursively is comparatively easy, but sometimes not possible because of stack depth restrictions. At the time when this tutorial was written, the stack size on the judge was limited to 16MB. With optimizations, the above function requires a stack frame of 72 bytes, limiting you to graphs of fewer than 240000 vertices. This will decrease further as soon as you start adding variables for a modified DFS.

So it may sometimes be necessary to do it with manual stacks. This is surprisingly hard to get right:

```
vector<bool> visited(n, false);
vector<int> dfs_stack;
vector<int> dfs_neighbor_pos;
int next_num = 0;
vector<int> dfs_num(n);
for (int u = 0; u < n; u++) {
    if (visited[u])
        continue;
    dfs_stack.push_back(u);
    dfs_neighbor_pos.push_back(0);
    visited[u] = true;
    dfs_num[u] = next_num++;
```

```
        cerr << "visit␣" << dfs_num[u] << ":␣" << u << "\n";
        while (!dfs_stack.empty()) {
            int v = dfs_stack.back();
            int i = dfs_neighbor_pos.back();
            dfs_stack.pop_back();
            dfs_neighbor_pos.pop_back();
            vector<int>& neighbors = edges[v];
            for (; i < (int)neighbors.size(); i++) {
                int w = neighbors[i];
                if (!visited[w]) {
                    /* defer looking at v */
                    dfs_stack.push_back(v);
                    dfs_neighbor_pos.push_back(i+1);
                    /* look at w next */
                    visited[w] = true;
                    dfs_num[w] = next_num++;
                    dfs_stack.push_back(w);
                    dfs_neighbor_pos.push_back(0);
                    cerr << "visit␣" << dfs_num[w] << ":␣" << w << "\n";
                    break;
                }
                /* if we fall off this loop (as opposed to break), we
                 * are done looking at v */
            }
        }
    }
```

3.2 Shortest Path

We distinguish:

APSP (All Pair Shortest Path) After you've run an APSP algorithm you know all shortest paths within the graph. (n^2 many!)
Example: Floyd-Warshall

SSSP (Single Source Shortest Path) After you've run a SSSP algorithm you know all shortest paths from a given node s to all other nodes. ($n-1$ many!)
Examples: Dijkstra, Bellmann-Ford

- Snippet 1: Floyd-Warshall

```
double sp[n][n];      //here we want to store the length
                      //of the shortest paths form i to
                      //j (sp[i*n+j]).
```

```
// initialize correctly here...
for (int mid = 0; mid < n; mid++) {
  for (int i = 0; i < n; i++) {
    if (i == mid)
      continue;
    for (int j = 0; j < n; j++) {
      if (j == mid || i == j)
        continue;
      if (sp[i*n+mid] != -1 && sp[mid*n+j] != -1 &&
         (sp[i*n+j] == -1 ||
          sp[i*n+mid] + sp[mid*n+j] < sp[i*n+j])) {
        sp[i*n+j] = sp[i*n+mid] + sp[mid*n+j];
      }
    }
  }
}
return sp;
```

- Snippet 2: (incomplete) Bellman-Ford

```
vector<bool> queued (n, false);
vector<int> parent (n,-1);
queue<int> q;
q.push(start_node);
 while (!q.empty()) {
  int curr = q.front(); q.pop();
  // for every neighbour update dist and parent
  // if dist updated and !queued q.push(neighb)
}
// !!! detection for negative cycles is needed !!!
```

- Snippet 3: Dijkstra (init)

```
int d[n];         //saved distances from start_node to node i
bool visit[n];    //flags to set nodes state to 'visited'
priority_queue<pair<int,int> > queue;
...
for (int i=0; i<n; i++) {
  d[i] = numeric_limits<int>::max();
  visit[i] = false;
}
d[start_node] = 0;
queue.push(pair<int,int>(d[start], start_node));
```

- Snippet 4: Dijkstra (main-loop)

```
while(!queue.empty()) {
  cur = queue.top(); queue.pop();
  i = cur.second;
  if (!visit[i]) {
    visit[i] = true;
    for (j=0; j<graph[i].size(); j++) {
      to = graph[i][j].first;
      if (!visit[to] &&
          d[i] + graph[i][j].second < d[to]) {
        d[to] = d[i] + graph[i][j].second;
        queue.push(pair <int,int>(-d[to], to));
      }
    }
  }
}
```

3.3 Minimum Spanning Trees

Kruskal

The idea is the following: insert edges (ordered by their weight) into the spanning tree as long as they don't close a circle.

```
Input: G = (V,E,l)
Output: T = (V,F)

sort(E)
F := {}          // empty set

for i:=1 to size(E)
  if (V, union (F, {e[i]})) contains no cycle
  then F = union (F, {e[i]}))

return (V,F)
```

In order to implement the pseudocode from above, you need a special data structure, the *Union Find Structure (UFS)*:

- Two operations
  1. `find` - gives representative for a set
  2. `union` - unites two sets

- Tree structure
- `find`
  1. walk up in the tree till the root is found
  2. return root as representative
- `union`
  1. find both representatives
  2. link representative of smaller height to representative of greater height.
- Use path compression in find operations (link all elements on the path directly to the root)

3.3.1 Prim

Select an arbitrary start vertex and insert into a priority queue (PQ) with weight 0, all other vertices are inserted into the PQ with weight infinity. The following snipped gives you an idea about the main loop:

```
while (!pq.empty()) {
  pq_pair curr = pq.top(); pq.pop();

  mst.add(curr, pred[curr]);

  for each edge e at curr {
    if (pq.find(e.dest) > e.weight) {
      pq.decrease_key(e.dest, e.weight);
      pred[e.dest] = curr;
    }
  }
}
```

The STL provides a priority queue, but their implementation does not contain a `decrease_key` implementation. An easy workaround is to just add a new entry instead of decreasing the key of an existing one. Additionally you might need to save which node was already read out of the queue and ignore cases where you read out the old entries.