The data structure used by a graph application has a big impact on the efficiency and ease of coding.

Many designs start off with the nodes. I guess the nodes, in the problems that are being modelled, often have a physical reality while the links can be abstract relationships. So it is more natural to start writing a node class, and add on the links later.

However, when coding algorithms that solve problems in graph theory, it becomes clear that the links are the real focus. So, lets start with a link class.

    class cLink

    {

    public:

        cLink(int c = 1)

            : myCost(c)

        {

        }

        int myCost;         // a constraint e.g. distance of a road, max xapacity of a pipe

        int myValue;        // a calculated value, e.g. actual flow through a pipe

    };

If we store the out edges of node in a map keyed by the destination node index, then the map will be an attribute of the source node.

    class cNode

    {

    public:

        cNode(const std::string &name = "???")

            : myName(name)

        {

        }

        std::string myName;

        std::map<int, cLink> myLink; // out edges of node, keyed by destination

    };

We have links and nodes, so we are ready to construct a class to store the graph

class cGraph {

public:

std::map<int, cNode> myG; // the graph, keyed by internal node index

};

Where did the node index come from? Humans are terrible at counting, so better the computer generates the index when the node is added.

cGraph::createNode( const std::string& name )

{

    int n = myG.size();

    myG.insert(std::make\_pair(n, cNode(name)));

}

This has a snag - it can create two nodes with the same name. We need to be able to check if node with a specified name exists.

        int cGraph::find(const std::string &name)

        {

            for (auto n : myG)

            {

                if (n.second.myName == name)

                {

                    return n.first;

                }

            }

            return -1;

        }

This is inefficient. However, it only needs to be done once when the node is added. Then the algorithms that search through the graph can use fast lookup of nodes by index number.

Now we can prevent two nodes being created with the same name

        int cGraph::findoradd(const std::string &name)

        {

            // search among the existing nodes

            int n = find(name);

            if (n < 0)

            {

                // node does not exist, create a new one

                // with a new index and add it to the graph

                n = myG.size();

                myG.insert(std::make\_pair(n, cNode(name)));

            }

            return n;

        }

Humans, in addition to being terrible counters, are also over confident in their counting prowess. When they specify a graph like this

1 -> 2

1 -> 3

Let’s not be taken in. Let’s regard these numbers as names and continue to use our own node index system.

/\*\* Add costed link between two nodes

 \*

 \* If the nodes do not exist, they will be added.

 \*

 \*/

        void addLink(

            const std::string & srcname,

            const std::string & dstname,

            double cost = 1)

        {

int u = findoradd(srcname);

int v = findoradd(dstname);

            myG.find(u)->second.myLink.insert(

std::make\_pair(v, cLink(cost)));

            if (!myfDirected)

                myG.find(v)->second.myLink.insert(

std::make\_pair(u, cLink(cost)));

        }

With the addition of a few getters and setters, we are ready to start implementing graph algorithms!