```
 //
//  main.cpp
//  Dijkstra
//
//  Created by Anthony Cormican on 1/7/17.
//  Copyright (c) 2017 Anthony Cormican. All rights reserved.
//

#include <iostream>
#include <string>
#include <cassert>
#include <chrono>
#include <random>
#include <iomanip>

using namespace std;
const unsigned int maxNodes = 50;
const bool bidirectional = false;

// the data contained in a graph node
class Node
{
public:
    Node() : desc(""), id(0), weight(0.0) {}
    Node(string i, float w) : desc(i), id(0), weight(w) {}

    void   setName (string& name)   { desc = name; }
    string getName ()               { return desc; }
    void   setWeight(const float w) { weight = w; }
    float  getWeight()              { return weight; }
    void   print()                  { cout << "(" << setprecision(2) << desc <<
        ", " << weight << ")"; }
    void   printName()              { cout << "(" << setprecision(2) << desc <<
        ")"; }
    void   printWeight()            { cout << "(" << setprecision(2) << weight <<
        ")"; }
    Node*  get_ptr()                { return this; }
    int    getId()                  { return id; }
    void   setId (const int i)      { id = i; }

    bool visited;
private:
    string desc;        // name of this node
    float weight;       // edge weighting
    int id;             // id of this node
};

// a container class to hold the data
class Element
{
public:
    Element() : next(nullptr) { data = new Node(); data = nullptr; }
    Element(Node& n) : next(nullptr) { data = new Node(); data = &n; }

    Node* data;
    Element* next;
```

```
};

// the class that allows the manipulation of a linked list to
// hold pointers to each node
template<class T>
class List
{
public:
    List() : head(nullptr) {}
    ~List() {}

    void add(Node& n);
    void addAllowDuplicate(Node& n);
    T*   first() const;
    T*   get_head() const;
    T*   get_element(const int n) const;
    void del();
    void del(const int i);
    void print(const int& i) const;
    void printRecursive(T* curr_ptr, const int& i) const;
    void printAddresses() const;
    void release();
    T*   contains(string name) const;
    T*   contains(const int e) const;
    void insert(string name, Element& e);
    void insert(const int i, Element& e);
    int size() const;
private:
    T* head;
};

// insert a new element at the end of the list only if does not
// already exist
template<class T> void List<T>::add(Node &n)
{
    T* temp = new T();
    assert(temp != nullptr);

    // avoid duplicates
    T* already_here = this->contains(n.getName());
    if (already_here == nullptr)
    {
        temp->data = &n;
        temp->data->visited = false;
        temp->next = head;
        head = temp;
    }
}

// insert an element, even if it already exists in the list
template<class T> void List<T>::addAllowDuplicate(Node &n)
{
    T* temp = new T();
    assert(temp != nullptr);

    temp->data = &n;
```

```cpp
        temp->data->visited = false;
        temp->next = head;
        head = temp;
    }

    // return a pointer to the first element in a list
    template<class T> T* List<T>::first() const
    {
        T* temp = head;
        while (temp->next != nullptr)
            temp = temp->next;
        return temp;
    }

    // return a pointer to the first element in a list
    template<class T> T* List<T>::get_head() const
    {
        return head;
    }

    // return a pointer to the nth element in a list
    template<class T> T* List<T>::get_element(const int e) const
    {
        T* temp = head;
        while (temp != nullptr)
            if (temp->data->getId() != e) temp = temp->next; else break;
        return temp;
    }

    // delete an element from a list
    template<class T> void List<T>::del()
    {
        T* temp = head;
        head = head->next;
        //cout << "Deleting node: " << temp->data->getName() << endl;
        delete temp;
    }

    // delete an element from a list
    template<class T> void List<T>::del(const int i)
    {
        T* temp = head;
        T* prev = temp;

        // check if it's the head
        if (temp->data->getId() == i)
        {
            temp = head;
            head = temp->next;
            delete temp;
            return;
        }

        while (temp != nullptr)
        {
            if (temp->data->getId() == i)
            {
                prev->next = temp->next;
                //cout << "Deleting node: " << temp->data->getName() << endl;
                delete temp;
                break;
            }
            else
            {
                prev = temp;
                temp = temp->next;
            }
        }
    }

    // used to enable printing from the tail to the head of the list
    template<class T> void List<T>::printRecursive(T* curr_ptr, const int &i) const
    {
        if (curr_ptr == nullptr)
            return;
        printRecursive(curr_ptr->next, i);
        cout << "->";
        switch (i)
        {
            case 0: curr_ptr->data->print(); break;
            case 1: curr_ptr->data->printName(); break;
            case 2: curr_ptr->data->printWeight(); break;
            default: break;
        }
    }

    // display the elements in a list
    // using recursion to print from end to start
    template<class T> void List<T>::print(const int& i) const
    {
        T* temp = head;
        printRecursive(temp, i);
        cout << endl;
    }

    // output memory addresses for all nodes
    template<class T> void List<T>::printAddresses() const
    {
        T* temp = head;
        while(temp != nullptr)
        {
            cout << "Address of node ID: " << temp->data->getName() << " is " << temp
                ->data->get_ptr() << endl;
            temp = temp->next;
        }
        cout << endl << endl;
    }

    // reclaims memory used by the list
    template<class T> void List<T>::release()
    {
        while (head != nullptr)
```

```
        del();
}

// returns a pointer to the node specified by name if it exists
// otherwise returns a nullptr
template<class T> T* List<T>::contains(const string name) const
{
    T* temp = head;
    while (temp != nullptr)
    {
        if (temp->data->getName() != name)
            temp = temp->next;
        else
            break;
    }
    return temp;
}

// returns a pointer to the node specified by id, e,  if it exists
// otherwise returns a nullptr
template<class T> T* List<T>::contains(const int e) const
{
    T* temp = head;
    while (temp != nullptr)
        if (temp->data->getId() != e)
            temp = temp->next;
        else
            break;
    return temp;
}

// insert the element e after the node identified by name
template<class T> void List<T>::insert(string name, Element& e)
{
    T* temp = new T();
    temp = this->contains(name);

    if (temp !=nullptr)
    {
        e.next = temp->next;
        temp->next = &e;
    }
}

// insert the element e after the node identified by i
template<class T> void List<T>::insert(const int i, Element& e)
{
    T* temp = new T();
    temp = this->contains(i);

    if (temp !=nullptr)
    {
        e.next = temp->next;
        temp->next = &e;
    }
}
```

```cpp
// return the size of the list
template<class T> int List<T>::size() const
{
    int count = 0;
    T* temp = head;
    while (temp != nullptr)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

// class for containing a graph of nodes of type T
template<class T>
class Graph
{
public:
    typedef vector<List<T>*> pathList;

    Graph();
    void print(string title) const;
    T* get_ptr(const int id);
    T* get_head(const int id);
    int numVertices() const { return size; }
    int numEdges() const;
    bool adjacent(const int x, const int y);
    bool findPaths(const int x, const int y, pathList& path);
    static void releasePaths(pathList& path);
    bool paths(const int x, const int y, pathList& path);
    bool nbr(T* l, List<T>& visited, const int& y, typename pathList::iterator&
        pc, pathList& path);
    static float getShortestDistance(pathList& path);
    Graph<T>* addEdge(const int x, const int y);
    Graph<T>* deleteEdge(const int x, const int y);
    int get_node_value(const int x) { return getVertex(x)->data->getId(); };
    void set_node_value(string s, const int x) { getVertex(x)->data->setId(s); }
    float get_edge_value(const int x, const int y) const;
    void set_edge_value(const float e, const int i) { getVertex(i)->data->
        setWeight(e); }
    Graph<T>* biDirect();
    Graph<T>* removeDuplicates();

    T* getVertex(const int i) const
    {
        if (i < 0 || i >= size)
            return nullptr;
        else
            return edges[i].first();
    }
    int getVertexID(const int i) const { return edges[i].first()->data->getId();
        }
    T* getFirstEdge(const int i) const { return edges[i].get_head(); }

    int size;                           // number of nodes
    List<T> edges[maxNodes];            // array of linked lists holding all the
        edges for a node
private:
    bool checkExist(const int x, const int y) const;
    bool stoprec = false;               // used for breaking recursion
};

template<class T> Graph<T>::Graph()
{
    // initialise the first nodes in the edge list
    for (int i=0; i<maxNodes; ++i)
        edges[i] = *new List<T>();
}

// display the graph as an edge list abstraction
template<class T> void Graph<T>::print(string title) const
{
    cout << title << endl;
    cout << "ID:\t| Edges" << endl;
    cout << "---------------------------------------" << endl;
    for (int i=0; i<size; ++i)
    {
        cout << getVertexID(i) << ":\t| [";
        T* f = getFirstEdge(i);
        while(f->next != nullptr)
        {
            cout << f->data->getId();
            f = f->next;
            if (f->next != nullptr)
                cout << ", ";
        }
        cout << "]" << endl;
    }
    cout << endl;
}

// return a pointer to a graph element
template<class T> T* Graph<T>::get_ptr(const int id)
{
    assert(id < size);
        //cout << "First item "<<id<<" is "<<edges[id].first()->data->desc
            <<endl;
    return edges[id].first();
}

// return a pointer to the head element
template<class T> T* Graph<T>::get_head(const int id)
{
    assert(id < size);
    //cout << "First item "<<id<<" is "<<edges[id].first()->data->desc <<endl;
    return edges[id].get_head();
}

// returns the total number of edges in the graph
template<class T> int Graph<T>::numEdges() const
{
    int count = 0;
    for (int i=0; i<size; ++i)
        count += edges[i].size() - 1;
    return count;
}

// returns true if x has an edge to y
template<class T> bool Graph<T>::adjacent(const int x, const int y)
{
    assert(x < y);

    Element* temp = get_head(x);        // get the head of x's edge list
    while (temp != nullptr)
    {
        if (temp->data->getId() == y)
            return true;
        temp=temp->next;
    }
    return false;
}

// interface to paths() - this just does the display
template<class T> bool Graph<T>::findPaths(const int x, const int y, pathList&
    path)
{
    if (!checkExist(x, y))
        return false;
    cout << "-------------------------------------" << endl;
    cout << "Paths from node " << x << " to node " << y << ":" << endl;

    path.push_back(new List<Element>());
    bool found = false;
    found = paths(x, y, path);

    for (int i=0;i<path.size();++i)
    {
        path[i]->print(1);

    }
    cout << "shortest distance is " << setprecision(5) << getShortestDistance
        (path) << endl;
    cout << endl;
    return found;
}

// reclaim the memory by this pathList
template<class T> void Graph<T>::releasePaths(pathList& path)
{
    // clean up
    for (int i=0; i<path.size(); ++i)
    {
        path[i]->release();
        path.pop_back();
    }
    path.clear();
}
```

```
// if any paths exist between x and y this function will return true
// and return a vector of vertex lists for each path from x to y in allPaths
template<class T> bool Graph<T>::paths(const int x, const int y, pathList& path)
{
    typename vector<List<T>*>::iterator pc;
    pc = path.begin();

    float shortest_distance = 0;
    bool found = false;
    List<T> temp = edges[x];
    if (temp.contains(y) != nullptr)
    {
        //cout << "(" << x << ")->(" << y << ")" << endl << endl;
        shortest_distance = get_edge_value(x, y);

        (*pc)->add(*getVertex(x)->data);
        (*pc)->add(*getVertex(y)->data);
        return true;
    }

    T* w = getFirstEdge(x);
    while (w->next != nullptr)
    {
        stoprec = false;
        //cout << "(" << x << ")->";

        (*pc)->add(*getVertex(x)->data);
        List<T>* visited = new List<T>();
        found = nbr(w, *visited, y, pc, path);
        visited->release();
        w = w->next;
    }
    cout << endl;

    // remove any NULL elements from the path lists created by the
    // recursion
    int i = 0;
    while(i<path.size())
    {
        if (path[i]->size() == 0)
        {
            path[i]->release();
            path.erase(path.begin()+i);
        }
        else
            ++i;
    }

    return found;
}

// the recursive routine to parse every possible node in the graph and return all
    possible
// paths from node x to node y as defined in the previous method
template<class T> bool Graph<T>::nbr(T* l, List<T>& visited, const int& y,
    typename pathList::iterator& pc,
                 pathList& path)
{
    bool found = false;
    int data_id = 0;

    while (l != nullptr)
    {
        data_id = l->data->getId();

        if (edges[data_id].contains(y) == nullptr)
        {
            if (visited.contains(data_id) == nullptr)
            {
                visited.add(*l->data->get_ptr());
                //cout << "(" << data_id << ")->";
                (*pc)->add(*getVertex(data_id)->data);
                nbr(edges[data_id].get_head(), visited, y, pc, path);
            }
            else
            {
                // move to the next node
                while (visited.contains(data_id) != nullptr && l->next!= nullptr)
                {
                    visited.add(*l->data->get_ptr());
                    l = l->next;
                    data_id = l->data->getId();
                }
                nbr(l, visited, y, pc, path);
            }
        }

        if (!stoprec)
        {
            (*pc)->add(*getVertex(data_id)->data);
            (*pc)->add(*getVertex(y)->data);
            //cout << "(" << data_id << ")->(" << y << ")" << endl;
            path.push_back(new List<T>());
            pc = path.end()-1;
        }

        stoprec = true;
        break;
    }
    return found;
}

// return the shortest distance from a set of paths
template<class T> float Graph<T>::getShortestDistance(pathList& path)
{
    float minimum = 0;
    float sum = 0;
    for (auto it : path)
    {
        sum = 0;
        T* temp = (*it).get_head()->next;   // don't include the last node as we
```

```
                have arrived there
            while(temp != nullptr)
            {
                sum += temp->data->getWeight();
                temp = temp->next;
            }
            if (sum < minimum)
                minimum = sum;
        }
        return sum;
    }

    // Add an edge between x and y if it does not already exist
    template<class T> Graph<T>* Graph<T>::addEdge(const int x, const int y)
    {
        assert(x != y);
        if (!checkExist(x, y))
            return false;

        Node* n = this->get_ptr(y)->data;
        this->edges[x].add(*n);
        return this;
    }

    // Delete the edge between x and y if it exists
    template<class T> Graph<T>* Graph<T>::deleteEdge(const int x, const int y)
    {
        assert(x != y);
        if (!checkExist(x, y))
            return false;

        this->edges[x].del(y);
        return this;
    }

    // return the edge value between to nodes
    template<class T> float Graph<T>::get_edge_value(const int x, const int y) const
    {
        assert(x != y);
        if (!checkExist(x, y))
            return false;

        // check that they are connected
        if (edges[x].contains(y) != nullptr)
            return edges[x].get_element(y)->data->getWeight();
        else
            return 0;
    }

    // Create two way edges (if not already there) for all edges
    template<class T> Graph<T>* Graph<T>::biDirect()
    {
        // to ensure edge bi-directionality, go through each edge list and add the
            reverse-direction edge to the
        // corresponding node; i.e. map (2,3) to (3,2)
        for (int i=0; i<size; ++i)
        {
            Element* temp = this->get_head(i);        // get the head of this edge
                list
            while (temp != nullptr)
            {
                int thisNode = temp->data->getId();       // get the id we're looking
                    for
                if (thisNode != i)
                {
                    Node* n = this->get_ptr(i)->data;
                    this->edges[thisNode].add(*n);
                }
                temp = temp->next;
            }
        }
        return this;
    }

    // Removes any two-way edges forcing a one way graph
    template<class T> Graph<T>* Graph<T>::removeDuplicates()
    {
        for (int i=0; i<size; ++i)
        {
            //cout << endl; g->print();
            Element* temp = this->get_head(i);        // get the head of this edge
                list
            while (temp != nullptr)
            {
                int thisNode;
                thisNode = temp->data->getId();       // get the id we're looking for
                if (thisNode != i)
                {
                    this->edges[thisNode].del(i);
                }
                temp = temp->next;
            }
        }
        return this;
    }

    // check that the nodes x and y exist in the graph
    template<class T> bool Graph<T>::checkExist(const int x, const int y) const
    {
        // check that both nodes exist
        if (getVertex(x) == nullptr)
        {
        cout << "Node " << x << " does not exist in this graph" << endl;
        return false;
        }

        if (getVertex(y) == nullptr)
        {
            cout << "Node " << y << " does not exist in this graph" << endl;
            return false;
        }
        return true;
```

```
}

// Generates a random graph with:
//      number of nodes: size
//      maximum distance/weighting for aan edge: maxDistance
//      number of edges from each node: edgeDensity (this is a probability, not a
    discrete value of edges)
// returns a Graph object containing 'size' nodes with edge lists containing no
    self referencing nodes and
// ensuring that bi-directionality is implemented
Graph<Element>* makeRandomGraph(float edgeDensity, float maxDistance, int size)
{
    Graph<Element>* g = new Graph<Element>;
    assert(size <= maxNodes);

    srand(static_cast <unsigned> (time(0)));    // seed the random number
        generation

    //create 'size' nodes first
    for (int i=0; i<size; ++i)
    {
        Node* t = new Node();
        string s = to_string(i);
        t->setName(s);
        t->setId(i);
        // generate a random edge distance within the range
        const float r2 = static_cast <float> (rand()) / (static_cast <float>
            (RAND_MAX/maxDistance-1)) + 1;
        t->setWeight(r2);
        g->edges[i].add(*t);
    }
    g->size = size;

    //traverse each node and add edges according to probability density
    for (int i=0; i<size;++i)
    {
        for (int j=0; j<size; ++j)
        {
            if (j != i)
            {
                const float r = static_cast <float> (rand()) / static_cast <float
                    > (RAND_MAX);
                if (r < edgeDensity)
                {
                    Element* e = g->get_ptr(j);
                    if (e != nullptr)
                    {
                        Node* temp = g->get_ptr(j)->data;   // get pointer to
                            first element in this array
                        // add if not already there
                        if (g->edges[i].contains(temp->getName()) ==
                            nullptr)    // do not create any self referential
                            loops
                            g->edges[i].add(*temp);
                    }
                }
            }
        }
    }

    if (bidirectional)
        g->biDirect();
    else
        g->removeDuplicates();
    return g;
}

int main(int argc, const char * argv[])
{
    typedef vector<List<Element>*> pathList;    // to hold the list of paths
    const int size = 50;

    // Compute averages for graph 1
    Graph<Element>* g = makeRandomGraph(0.2, 10, size);
    g->print("Graph 1: Edge Density 20%, Distance Range = 10, Size = 50");
    cout << "Graph 1:" << endl;
    cout << "Number of Vertices: " << g->numVertices() << endl;
    cout << "Number of Edges: " << g->numEdges() << endl;

    float distance = 0;
    for (int i=1; i<size; ++i)
    {
        auto thisPath = *new pathList();
        g->findPaths(0, i, thisPath);
        distance += Graph<Element>::getShortestDistance(thisPath);
        Graph<Element>::releasePaths(thisPath);
    }
    cout << "Average distance of Graph 1 = " << distance / size << endl;
    cout <<
        "#-------------------------------------------------------------------------
        ----#" << endl;
    cout << endl << endl;

    // Compute averages for graph 2
    Graph<Element>* g2 = makeRandomGraph(0.4, 10, size);
    g->print("Graph 2: Edge Density 40%, Distance Range = 10, Size = 50");
    cout << "Graph 2:" << endl;
    cout << "Number of Vertices: " << g2->numVertices() << endl;
    cout << "Number of Edges: " << g2->numEdges() << endl;

    distance = 0;
    for (int i=1; i<size; ++i)
    {
        auto thisPath = *new pathList();
        g2->findPaths(0, i, thisPath);
        distance += Graph<Element>::getShortestDistance(thisPath);
        Graph<Element>::releasePaths(thisPath);
    }

    cout << "Average distance of Graph 2 = " << distance / size << endl;
    return 0;
}
```