```
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//
// Graph.h: implement graph in adjacency list representation
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>
#include <limits>
#include <string>
#include <map>

#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

template <typename Object, typename Weight>
class graph {
public:

    class Vertex;

    class Edge {
    public:
        Edge(Vertex* v, Vertex* w, Weight setweight) {
            start = v;
            end = w;
            v->edge.push_back(this);
            w->inedge.push_back(this);
            weight = setweight;
            explored = false;
        }
        Edge() {
            explored = false;
        }
        Weight weight;
        Vertex* start;
        Vertex* end;
        bool explored;
    };

    class Vertex {
    public:
        Vertex(Object setelement) {
            level = 0;
            connectedcomponent = 0;
            element = setelement;
            back = NULL;
            explored = false;
        }
        Vertex() {
            level = 0;
            connectedcomponent = 0;
            back = NULL;
            explored = false;
        }
        Object element;
        vector<Edge*> edge;
        vector<Edge*> inedge;

        double value;
        size_t starttime, finishtime;
        size_t level;
        size_t connectedcomponent;
        float rank;
        Vertex* back;
        int color;
        bool explored;
```

```
    };

    //////////////////////////////////////////////////////////////////////////
private:

    vector<Edge*> edge;

    vector<Vertex*> vertex;

    size_t counter;

public:
    //////////////////////////////////////////////////////////////////////////
    graph();

    graph(graph& G);

    ~graph();

    void reset();
    void resetBack();
    void resetValues();
    void resetLevels();
    void resetExplored();
    void resetConnectedComponents();

    vector<Vertex*> incidentVertices(Vertex* v);
    vector<Edge*> incidentEdges(Vertex* v);
    vector<Edge*> outgoingEdges(Vertex* v);
    vector<Vertex*> adjacentVertices(Vertex* v);
    size_t indegree(Vertex* v);
    size_t outdegree(Vertex* v);
    size_t degree(Vertex* v);
    Vertex* startVertex(Edge* e);
    Vertex* endVertex(Edge* e);
    // is there an edge from v to w ?
    bool isAdjacent(Vertex* v, Vertex* w);

    Vertex* insertVertex(Object o);
    void insertEdge(Vertex* v, Vertex* w, Weight t);
    void insertUndirectedEdge(Vertex* v, Vertex* w, Weight t);
    void removeVertex(Vertex* v);
    void removeEdge(Edge* e);

    size_t numVertices();
    size_t numEdges();
    vector<Vertex*> vertices();
    vector<Edge*> edges();

    void print();
    void read_file(std::string filename);
    //////////////////////////////////////////////////////////////////////////

};


template <typename Object, typename Weight>
graph<Object, Weight>::graph() {
    ////////////////////////////////////////////
    // Implement default constructor here
    ////////////////////////////////////////////
    counter = 0;
}

template <typename Object, typename Weight>
graph<Object, Weight>::graph(graph<Object, Weight>& G) {
    ////////////////////////////////////////////
    // Implement copy constructor here
    ////////////////////////////////////////////

    //delete old elements
    for (typename vector<typename graph<Object, Weight>::Edge*>::iterator it = this->edge.begin();
          it != this->edge.end(); ++it)
```

```
            delete *it;
        for (typename vector<typename graph<Object, Weight>::Vertex*>::iterator it = this->vertex.begin
              (); it != this->vertex.end(); ++it)
            delete *it;

        //map from old to new pointers
        map<Vertex*, Vertex*> pointers;

        //copy vertices
        for (size_t i=0; i<G.vertex.size(); ++i)
            pointers[G.vertex[i]] = insertVertex(G.vertex[i]->element);

        //copy edges
        for (size_t i=0; i<G.edge.size(); ++i)
            insertEdge(pointers[G.edge[i]->start], pointers[G.edge[i]->end], G.edge[i]->weight);
    }

    template <typename Object, typename Weight>
    graph<Object, Weight>::~graph() {
        ///////////////////////////////////////////
        // Implement destructor here
        ///////////////////////////////////////////

        //delete edges
        for (typename vector<typename graph<Object, Weight>::Edge*>::iterator it = this->edge.begin();
              it != this->edge.end(); ++it)
            delete *it;

        //delete vertices
        for (typename vector<typename graph<Object, Weight>::Vertex*>::iterator it = this->vertex.begin
              (); it != this->vertex.end(); ++it)
            delete *it;
    }

    template <typename Object, typename Weight>
    void graph<Object, Weight>::reset() {
        counter = 0;
        resetBack();
        resetValues();
        resetLevels();
        resetExplored();
        resetConnectedComponents();
    }

    template <typename Object, typename Weight>
    void graph<Object, Weight>::resetBack() {
        for(size_t i=0; i<vertex.size(); i++)
            vertex[i]->back = NULL;
    }

    template <typename Object, typename Weight>
    void graph<Object, Weight>::resetValues() {
        for(size_t i=0; i<vertex.size(); i++)
            vertex[i]->value = numeric_limits<int>::max();
    }

    template <typename Object, typename Weight>
    void graph<Object, Weight>::resetLevels() {
        for(size_t i=0; i<vertex.size(); i++)
            vertex[i]->level = 0;
    }

    template <typename Object, typename Weight>
    void graph<Object, Weight>::resetExplored() {
        for(size_t i=0; i<vertex.size(); i++)
            vertex[i]->explored = false;
    }

    template <typename Object, typename Weight>
    void graph<Object, Weight>::resetConnectedComponents() {
        for(size_t i=0; i<vertex.size(); i++)
            vertex[i]->connectedcomponent = 0;
    }
```

```cpp
template <typename Object, typename Weight>
vector<typename graph<Object, Weight>::Edge*> graph<Object, Weight>::incidentEdges(Vertex* v) {
    vector<Edge*> result;
    ///////////////////////////////////////////////
    // Implement collecting incident edges here
    ///////////////////////////////////////////////
    result = v->inedge;
    return result;
}

template <typename Object, typename Weight>
vector<typename graph<Object, Weight>::Edge*> graph<Object, Weight>::outgoingEdges(Vertex* v) {
    vector<Edge*> result;
    ///////////////////////////////////////////////
    // Implement collecting outgoing edges here
    ///////////////////////////////////////////////
    result = v->edge;
    return result;
}

template <typename Object, typename Weight>
vector<typename graph<Object, Weight>::Vertex*> graph<Object, Weight>::incidentVertices(Vertex* v)
     {
    vector<Vertex*> result;
    ///////////////////////////////////////////////
    // Implement filling result vector here
    ///////////////////////////////////////////////

    //collect start vertices
    for (int i=0; i < v->inedge.size(); ++i)
        result.push_back(v->inedge[i]->start);

    return result;
}

template <typename Object, typename Weight>
vector<typename graph<Object, Weight>::Vertex*> graph<Object, Weight>::adjacentVertices(Vertex* v)
     {
    vector<Vertex*> result;
    ///////////////////////////////////////////////
    // Implement filling result vector here
    ///////////////////////////////////////////////

    //collect end vertices
    for (int i=0; i < v->edge.size(); ++i)
        result.push_back(v->edge[i]->end);

    return result;
}

template <typename Object, typename Weight>
size_t graph<Object, Weight>::outdegree(Vertex* v) {
    ///////////////////////////////////////////////
    // Implement calculating outdegree here
    ///////////////////////////////////////////////
    return v->edge.size();
}

template <typename Object, typename Weight>
size_t graph<Object, Weight>::indegree(Vertex* v) {
    ///////////////////////////////////////////////
    // Implement calculating indegree here
    ///////////////////////////////////////////////
    return v->inedge.size();
}

template <typename Object, typename Weight>
size_t graph<Object, Weight>::degree(Vertex* v) {
    ///////////////////////////////////////////////
    // Implement calculating degree here
    ///////////////////////////////////////////////
    return indegree(v) + outdegree(v);
```

```
}

template <typename Object, typename Weight>
typename graph<Object, Weight>::Vertex* graph<Object, Weight>::startVertex(Edge* e) {
    return e->start;
}

template <typename Object, typename Weight>
typename graph<Object, Weight>::Vertex* graph<Object, Weight>::endVertex(Edge* e) {
    return e->end;
}

// is there an edge from v to w ?
template <typename Object, typename Weight>
bool graph<Object, Weight>::isAdjacent(Vertex* v, Vertex* w) {
    ///////////////////////////////////////////////
    // Implement the adjacency checking here
    ///////////////////////////////////////////////

    //use vertex with less edges to check
    if (degree(v) < degree(w)) {
        //check all opposite vertices
        for (int i=0; i < v->edge.size(); ++i)
            if (v->edge[i]->end == w)
                return true;
    } else {
        //check all opposite vertices
        for (int i=0; i < w->edge.size(); ++i)
            if (w->edge[i]->end == v)
                return true;
    }

    return false;
}

template <typename Object, typename Weight>
typename graph<Object, Weight>::Vertex* graph<Object, Weight>::insertVertex(Object o) {
    ///////////////////////////////////////////////
    // Implement vertex insertion here
    ///////////////////////////////////////////////
    Vertex* vert = new Vertex(o);
    vertex.push_back(vert);
    return vert;
}

template <typename Object, typename Weight>
void graph<Object, Weight>::insertEdge(Vertex* v, Vertex* w, Weight t) {
    ///////////////////////////////////////////////
    // Implement edge insertion here
    ///////////////////////////////////////////////
    Edge* e = new Edge(v, w, t);
    edge.push_back(e);
}

template <typename Object, typename Weight>
void graph<Object, Weight>::removeEdge(Edge* e) {
    ///////////////////////////////////////////////
    // Implement removing edge here
    ///////////////////////////////////////////////

    //remove start vertex's pointer
    e->start->edge.erase(std::remove(e->start->edge.begin(), e->start->edge.end(), e), e->start->
          edge.end());

    //remove end vertex's pointer
    e->end->inedge.erase(std::remove(e->end->inedge.begin(), e->end->inedge.end(), e), e->end->
          inedge.end());

    //remove edge
    edge.erase(std::remove(edge.begin(), edge.end(), e), edge.end());
    delete *e;
}
```

```
template <typename Object, typename Weight>
void graph<Object, Weight>::insertUndirectedEdge(Vertex* v, Vertex* w, Weight t) {
    ///////////////////////////////////////////////
    // Implement inserting undirected edge here
    ///////////////////////////////////////////////
    insertEdge(v, w, t);
    insertEdge(w, v, t);
}

template <typename Object, typename Weight>
void graph<Object, Weight>::removeVertex(Vertex* v) {
    ///////////////////////////////////////////////
    // Implement removing vertex here
    ///////////////////////////////////////////////

    //remove all edges
    typename vector<typename graph<Object, Weight>::Edge*>::iterator it;
    for (it = v->edge.end(); it != v->edge.begin(); --it)
        removeEdge(*it);

    delete *v;
}

template <typename Object, typename Weight>
size_t graph<Object, Weight>::numVertices() {
    return vertex.size();
}

template <typename Object, typename Weight>
size_t graph<Object, Weight>::numEdges() {
    return edge.size();
}

template <typename Object, typename Weight>
vector<typename graph<Object, Weight>::Vertex*> graph<Object, Weight>::vertices() {
    return vertex;
}

template <typename Object, typename Weight>
vector<typename graph<Object, Weight>::Edge*> graph<Object, Weight>::edges() {
    return edge;
}

template <typename Object, typename Weight>
void graph<Object, Weight>::print() {
    cout << "vertices:" << endl;
    for(size_t i=0; i<vertex.size(); i++)
        cout << vertex[i]->element << endl;
    cout << "edges:" << endl;
    for(size_t i=0; i<edge.size(); i++)
        cout << "(" << edge[i]->start->element << "," << edge[i]->end->element << ")" << endl;
}

template <typename Object, typename Weight>
void graph<Object, Weight>::read_file(std::string filename) {
    ///////////////////////////////////////////////
    // Implement reading file here
    ///////////////////////////////////////////////

    ifstream stream;
    stream.open(filename.c_str());

    //initialize graph with number of vertices equal to first value
    int num_vertices;
    stream >> num_vertices;
    for (int i=0; i<num_vertices; ++i)
        insertVertex(i);

    //insert edges until file end
    int x, y, w;
    while (stream >> x >> y >> w)
        insertEdge(vertex[x], vertex[y], w);
}
```

```
#endif
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
//
// sssp.cpp: single source shortest path implementation
//
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

#include <iostream>
#include <stdlib.h>
#include <map>      //for output map
#include <limits>   //to get maximum to define infinity for a templated type

//#include "AdjMatGraph.h"
#include "Graph.h"

#include "timing.h" //timing

//return a map from vertices to the length of the shortest path from the source
template <typename Object, typename Weight>
std::map<typename graph<Object, Weight>::Vertex*, Weight>
single_source_shortest_path(graph<Object, Weight>& g,
                            typename graph<Object, Weight>::Vertex* source) {

    //type definitions
    typedef typename graph<Object, Weight>::Vertex* VertexPtrType;
    typedef typename graph<Object, Weight>::Edge* EdgePtrType;
    typedef std::map<VertexPtrType, Weight> ReturnType;

    Weight INF = numeric_limits<Weight>::max();

    ReturnType paths; //output map from vertices to shortest path length
    vector<VertexPtrType> vertices = g.vertices(); //graph vertices

    //initialize vertex properties
    for (size_t i = 0; i < vertices.size(); ++i) {

        //set all vertices to unexplored
        vertices[i]->explored = false;

        if (vertices[i] == source)
            paths[vertices[i]] = 0; //set source distance to 0
        else
            paths[vertices[i]] = INF; //set other vertices to infinity
    }

    //main algorithm loop
    for (int i = 0; i < vertices.size()-1; ++i) {

        //find next closest vertex
        Weight min = INF;
        VertexPtrType x = NULL;
        for (int k = 0; k < vertices.size(); ++k) {

            VertexPtrType nearest = vertices[k];
            Weight nearest_dist = paths[nearest];

            //set vertex to nearest if closer to the root than nearest
            if (!nearest->explored && nearest_dist <= min) {
                x = nearest;
                min = nearest_dist;
            }
        }

        //break the algorithm if no vertex found
        if (x == NULL)
            break;

        x->explored = true;
        vector<EdgePtrType> x_edges = g.outgoingEdges(x);
        Weight x_dist = paths[x];

        //iterate through outgoing edges
```

```
        for (int k = 0; k < x_edges.size(); ++k) {

            //get edge
            EdgePtrType e = x_edges[k];

            //find next adjacent vertex
            VertexPtrType y = e->end;

            //get current distance from map
            Weight y_dist = paths[y];

            //calculate total distance from root through x to adjacent vertex y
            Weight new_dist = x_dist + e->weight;

            //if shorter path is found update path for adjacent vertex y
            if (!y->explored && x_dist != INF && new_dist < y_dist)
                paths[y] = new_dist;
        }
    }

    return paths;
}

template <typename G>
void populate_mesh(G& g, size_t n) {

    //add vertices
    for (size_t i = 0; i < n*n; ++i)
        g.insertVertex('a' + i);

    typedef std::vector<typename G::Vertex*> VerticesType;
    VerticesType vert = g.vertices();

    //add edges
    for (size_t i=0; i<n*n; i++) {
        size_t row_start = (i/n)*n;
        size_t x = i+1;
        size_t y = i+n;

        if (x < row_start+n)
            g.insertUndirectedEdge(vert[i], vert[x], i+x);
        if (y < n*n)
            g.insertUndirectedEdge(vert[i], vert[y], i+y);
    }
}

template <typename Vector>
void print_edges(Vector const& v) {
    for (typename Vector::const_iterator it = v.begin(); it != v.end(); ++it)
        std::cout << "(" << (*it)->start->element << ", "
        << (*it)->end->element << ") weight = "
        << (*it)->weight << std::endl;
}

int main(int argc, char* argv[]) {
    typedef char VertexProperty;
    typedef int EdgeProperty;
    typedef graph<VertexProperty, EdgeProperty> GraphType;

    //-------------------------------------
    //perform time experiments with timing.h

    cout << "\n------------------------" << endl;
    cout << "TEST TIMING\n" << endl;
    time_algorithm<GraphType>();

    //-------------------------------------
    //test correctness of algorithm on simple graph

    cout << "\n------------------------" << endl;
    cout << "TEST CORRECTNESS\n" << endl;
    const size_t n = 3;
```

```
    //GraphType g(n*n);
    GraphType g;

    // add vertices and edges
    //g.read_file("power.g");
    populate_mesh(g, n);

    // print out graph
    std::cout << "Graph:" << std::endl;
    print_edges(g.edges());

    GraphType::Vertex* src = g.vertices()[0];

    // compute SSSP
    std::map<GraphType::Vertex*, EdgeProperty> shortest_paths = single_source_shortest_path(g, src)
        ;

    // print out SSSP
    std::cout << "\nSSSP:" << std::endl;

    std::map<GraphType::Vertex*, EdgeProperty>::iterator it = shortest_paths.begin();
    std::map<GraphType::Vertex*, EdgeProperty>::iterator eit = shortest_paths.end();

    for(; it != eit; ++it)
        std::cout << it->first->element << " is " << it->second << " hops away from " << src->
            element << std::endl;

    cout << setprecision(10);

    return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
//
// sssp.cpp: single source shortest path implementation
//
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

#include <ctime>    //for timing
#include <iomanip>   //for setprecision
#include <math.h>

//#include "AdjMatGraph.h"
#include "Graph.h"

//-------------------------------------------------------------
//functions for generating graphs

template <typename G>
void populate_real(G& g, size_t n) {

    //add vertices
    for (size_t i = 0; i < n*n; ++i)
        g.insertVertex('a' + i);

    typedef std::vector<typename G::Vertex*> VerticesType;
    VerticesType vert = g.vertices();

    //add edges
    for (size_t i=0; i<n*n; i++) {
        size_t row_start = (i/n)*n;
        size_t x = i+1;
        size_t y = i+n;

        if (x < row_start+n)
            g.insertUndirectedEdge(vert[i], vert[x], 1+rand()%n);
        if (y < n*n)
            g.insertUndirectedEdge(vert[i], vert[y], 1+rand()%n);
    }
}

template <typename G>
void populate_dense(G& g, size_t n) {

    //add vertices
    for (size_t i = 0; i < n*n; ++i)
        g.insertVertex('a' + i);

    typedef std::vector<typename G::Vertex*> VerticesType;
    VerticesType vert = g.vertices();

    //add edges
    for (size_t i=0; i<n*n-1; i++) {
        for (size_t k=i+1; k<n*n; ++k) {
            g.insertUndirectedEdge(vert[i], vert[k], 1+rand()%n);
        }
    }
}

template <typename G>
void populate_sparse(G& g, size_t n) {

    //add vertices
    for (size_t i = 0; i < n*n; ++i)
        g.insertVertex('a' + i);

    typedef std::vector<typename G::Vertex*> VerticesType;
    VerticesType vert = g.vertices();

    //add edges
    for (size_t i=0; i<n*n-1; i++) {
        g.insertUndirectedEdge(vert[i], vert[i+1], 1+rand()%n);
    }
```

```
}

//-----------------------------------------------------------------
//functions for timing

//averages the running time over _num_itr iterations and returns the average time.
template <typename G>
double run_test(G &g, size_t _num_itr) {

    typedef int EdgeProperty;
    typedef char VertexProperty;
    typedef graph<VertexProperty, EdgeProperty> GraphType;

    //determine source vertex
    GraphType::Vertex* src = g.vertices()[0];

    //initialize output map
    std::map<GraphType::Vertex*, EdgeProperty> shortest_paths;

    //start timing at new tick
    clock_t k = clock();
    clock_t start;
    do start = clock();
    while (start == k);

    //compute SSSP
    for (int itr=0; itr<_num_itr; ++itr)
         shortest_paths = single_source_shortest_path(g, src);

    //end timing
    clock_t end = clock();

    //show size and iterations
    //cout << shortest_paths.size() << ", ";
    //cout << _num_itr << ", ";

    //calculate average time per iteration
    double elapsed_time = double(end - start) / double(CLOCKS_PER_SEC);
    //cout << elapsed_time << ", "; //shows total time
    return elapsed_time / double(_num_itr);
}

//calls run_test over each size
template <typename G>
void initiate_test(string which_graph) {

    //definitions
    typedef char VertexProperty;
    typedef int EdgeProperty;
    typedef graph<VertexProperty, EdgeProperty> GraphType;

    //set size range
    size_t lower = 2;
    size_t upper = (size_t)pow(2, 13);

    //iterate from lower size bound to upper size bound
    for (size_t n = lower; n*n < upper; n*=1.5) {
        size_t iterations = size_t(pow(upper, 1) / (1 + pow(n, 2) - pow(lower, 2)));
        if (iterations < 1) iterations = 1;

        //initialize graph and fill
        typedef graph<VertexProperty, EdgeProperty> GraphType;

        //GraphType g(n*n);
        GraphType g;

        //choose graph type
        if (which_graph == "real")
            populate_real(g, n);
        else if (which_graph == "sparse")
            populate_sparse(g, n);
        else if (which_graph == "dense")
            populate_dense(g, n);
```

```
        //run test over iterations
        cout << run_test(g, iterations) << endl;
    }
}

//calls initiate_test for each graph type
template <typename G>
void time_algorithm() {

    //determines which tests are performed
    vector<string> test_types;
    test_types.push_back("real");
    test_types.push_back("sparse");
    test_types.push_back("dense");

    for (int i=0; i<test_types.size(); ++i) {
        //display which graph type is being tested
        cout << "-- Testing " << test_types[i] << "() graph:" << endl;

        cout << "Size, Iter, Time(sec)" << endl;
        initiate_test<G>(test_types[i]);

        cout << endl;
    }
}
```

CSCE 221H
Assignment Coverpage

Please list below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students	Other People	Printed Material	Web Material (give url)	Other Sources
1.	1.	1.	1. www.cplusplus.com	1.
2.	2.	2.	2. www.wikipedia.com	2.
3.	3.	3.	3. see URL's below	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

http://stackoverflow.com/questions/4148428/template-way-of-finding-maximum-allowable-value
http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. *Disciplinary actions range from grade penalty to expulsion*. Please consult the Aggie Honor System Office for additional information regarding academic misconduct - it is your responsibity to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received or given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Assignment (circle one):	CSE Culture Assn	Other Assn
Non-Culture Assignment: assignment type:	Program 4 report	
CSE Culture Seminar: Speaker/Seminar/Date:		
CSE Culture Biography: Person's Name:		
Today's Date:	April 29, 2014	
Printed Name:	Carsten Hood	
Signature:		

Carsten Hood
CSCE 221H-200
Spring 2014

Program 4 Report
Graph Data Type:
Single Source Shortest Path Algorithm

Introduction

This report outlines experiments intended to demonstrate the complexity of a single source shortest path, or SSSP, algorithm, also known as Dijkstra's algorithm. The algorithm maps each vertex in an input graph to its least distance from a designated root vertex. Adjacency matrix and adjacency list structures are used for testing. Both implementations are analyzed and compared using the maximum spread of input sizes and graphs of varying vertex degrees: sparse, real world, and dense.

Theoretical Analysis

The complexity of Dijkstra's algorithm depends on the temporary data structures involved and the underlying graph structure. To start the operation, all vertices are marked as unexplored. The distance of the root vertex is set to zero, representing the distance to self. The distances of all other vertices are set to infinity, represented in my implementation by the maximum value for the given data type of the edge weight using the C++ limits library. Overall this takes O(n) time for both matrix and list graph implementations with n representing the number of vertices, as vertices are accessed instantly in a sequence and each is examined once. Then the main algorithm loop iterates over the vertices.

The first task within the loop is to identify an unexamined vertex closest to the growing "cloud" of examined vertices. Using a heap-based priority queue, finding the minimum theoretically takes O(logn) time. However, the distances used as keys must be updatable in an SSSP algorithm. This is impossible to do efficiently in most standard heap implementations, as the key to be updated must be located within the structure, and then the heap's order, possibly upset by the new value, must be maintained. Despite my attempts, I could not implement any priority queue structure that was more efficient over the course of the algorithm than an unordered sequence, which allowed for instant access when updating the key values. I believe this is because finding the vertex with the minimum distance occurs only once in each step, at the beginning; updating key values, however, may occur up to n - 1 times at each step, once for each adjacent vertex. While certain libraries offer Fibonacci heaps, which allow for updating key values and theoretically offer logarithmic operations, I could not implement them on my machine. My research indicated that Fibonacci heaps offered slower running times in practice for small to medium range inputs, to which I am limited given the strength of my test machine's processor. I resolved to test the SSSP algorithm using a linear search implemented as efficiently as possible. In each iteration in the extract-minimum loop a distance is retrieved from the output sequence, implemented as a binary search tree with O(logn) lookup. This results in O(nlogn) running time overall for the extract-minimum operation within the main loop.

The next operation in the main loop is retrieving the new vertex's adjacent edges. In an adjacency list this takes time directly relative to the degree of the vertex. With a sparse graph this may take nearly constant time. With a dense graph, when vertices are connected to most other vertices, the time approaches O(n). In the adjacency matrix structure the complexity is always O(n), as possible connections with every other vertex must be examined.

Lastly the algorithm iterates over the outgoing edges to update. There are up to n – 1 edges for each vertex, but explored edges are eliminated from further operations. However, in my implementation, by the exploration status of an edge is checked it has already required some operations, so this loop can be determined to take O(n). In each iteration within this loop, distances are twice retrieved from the output map. Each retrieval takes (O(logn)). Therefore this operation takes O(nlogn).

Overall, after the O(n) time for the algorithm's initialization, there are 2 O(nlogn) and an O(n) operation within a loop that iterates over n - 1 vertices. Therefore the complexity of the SSSP implementation is O(n) + (n -1) * (2*O(nlogn) + O(n)) = O((n^2)logn). Using a linear extract-minimum operation avoids the cost of locating and updating keys and then maintaining a more complex priority queue structure. While this may be inefficient for large sizes, for the inputs tested it is likely more efficient, and the resulting data bears a more distinct complexity as a result.

Experimental Setup

The experiments described in this report were performed on a Mac computer running OS X 10.9.1 with a 2.7 GHz Intel Core i7 processor and 8GB RAM memory. C++ code was compiled using Apple LLVM 5.0.

The standard library ctime was used for timing. To avoid clock inaccuracy for small input sizes executions were repeated so that the measured time was between a hundredth of a second and ten seconds. This value was divided by the number of iterations to produce the average running time per iteration. In each iteration the output sequence is simply reset to the results of the SSSP function; this operation takes linear time and can be discounted in analysis over a range of sizes because of the greater complexity of the SSSP algorithm.

Integers were used to represent edge weights for the simplicity of their random generation and numerical operations. All edge weights were randomly generated with values ranging from 1 to the square root of the given graph's input size. No negative weights were used: this would compromise the SSSP algorithm.

Three functions generated input graphs given a size n for different densities dictated by a string parameter. The graph types used were dense graphs with the maximum number of edges, n(n-1), sparse graphs with the minimum, n-1, and a mesh-type real world graph intended to represent an in-between data set. Graph sizes ranged from 4 to 3969 vertices; beyond this value the algorithm's running time would exceed the capability of the test machine. Sizes were determined by squaring a counter variable that increased by larger increments each step; the resulting distribution scales at the same rate as using successive multiples of two. The input set is {4, 9, 16, 36, 81, 169, 361, 784, 1764, 3969}.

Experimental Results

Figures 1 and 2 plot the running times of the single source shortest path algorithm on adjacency list and adjacency matrix structures respectively. Real, sparse, and dense graphs are tested. In both plots the sparse and real graphs appear to have identical running times. At first I suspected that my testing code had mistakenly used the same input data for both tests, but inspection of the output data reveals that the algorithm executes slightly faster on the sparse graph than the real-world graph for all input sizes, as expected in analysis. There resemblance owes itself to the algorithm used to generate the real-world graph, which results in a distribution closer to a sparse graph than a dense graph. The plot charting the performance of the SSSP operation on a dense graph, which contains the maximum number of edges, is clearly less efficient for both implementations, and increasingly so as the input size increases. This is because the maximum number of edges must be examined for each vertex in the main algorithm loop.

Figure 3 compares the effects of using an adjacency list structure and an adjacency matrix structure on the SSSP algorithm. Consistent with analysis, the list-based structure was more efficient for all input sizes and graph types. On the plots this is only noticeable at small input sizes, which make the variation more apparent due to the nature of the log-log plot scale. A vertex in the adjacency matrix implementation must check with all n vertices to identify its edges, while the adjacency list accesses adjacent edges directly through a vertex's member variable. Since this is the primary difference between the graph structures' effects on the SSSP algorithm, it follows that the list implementation is faster. However, the difference is slight. This demonstrates experimentally that the cost of the other operations in the main algorithm loop—finding the next minimum vertex and looping through its outgoing edges—largely outweigh the cost of retrieving the outgoing edges.

Figure 4 allows for analysis of the SSSP operation's performance in relation to its theoretically expected complexity by dividing output times by the expected time, O(n^2logn). This results in two nearly horizontal plots, suggesting that results concur with analysis. Both graphs begin with a higher relative cost for small input sizes; this is a result of the cost of operations that a take time independent of input size. Their effect on total running time is later overshadowed by the cost of operations repeated for larger sizes. The data in Figure 4 can be used to determine the Big-O constants for both experiments. After small input sizes the plot of the adjacency list levels out soon after n = 36, where the average running time is 1.39E-08. This value bounds all subsequent time values. Therefore, the constants are k0 = 34 and C = 1.39E-08. Similarly, Big-O constants of the form (k0, C) are determined to be (36, 1.41E-08) for the adjacency matrix representation.

Summary

This report analyzes Dijkstra's single source shortest path algorithm across three input graphs and two underlying graph data structures. An implementation of the algorithm using a linear priority queue search provides an assortment of data and plots to compare to theoretical analysis. The experimental results confirmed the expected behavior of Dijkstra's algorithm with no inconsistencies. However, the close similarity between all test results was unexpected. The adjacency list graph implementation proved only slightly faster than the matrix-based structure, though the gap between the resulting sets of data widens as input sizes increase. Similarly, the differences between the running times of sparse, real, and dense graphs, while existent and consistent with theory, are barely noticeable. This general lack of distinct deviation can likely be attributed to the limited range of input sizes. For massive data sets of the type that cannot be tested with instruments to which I have access, I suspect the varying input densities would result in conspicuously disparate results. Even with the sizes tested, the gap between the data widens, although this is not noticeable in the plots. This<> confirms that input graph structure and implementation play an increasingly important role when dealing with large data sets.

Plots

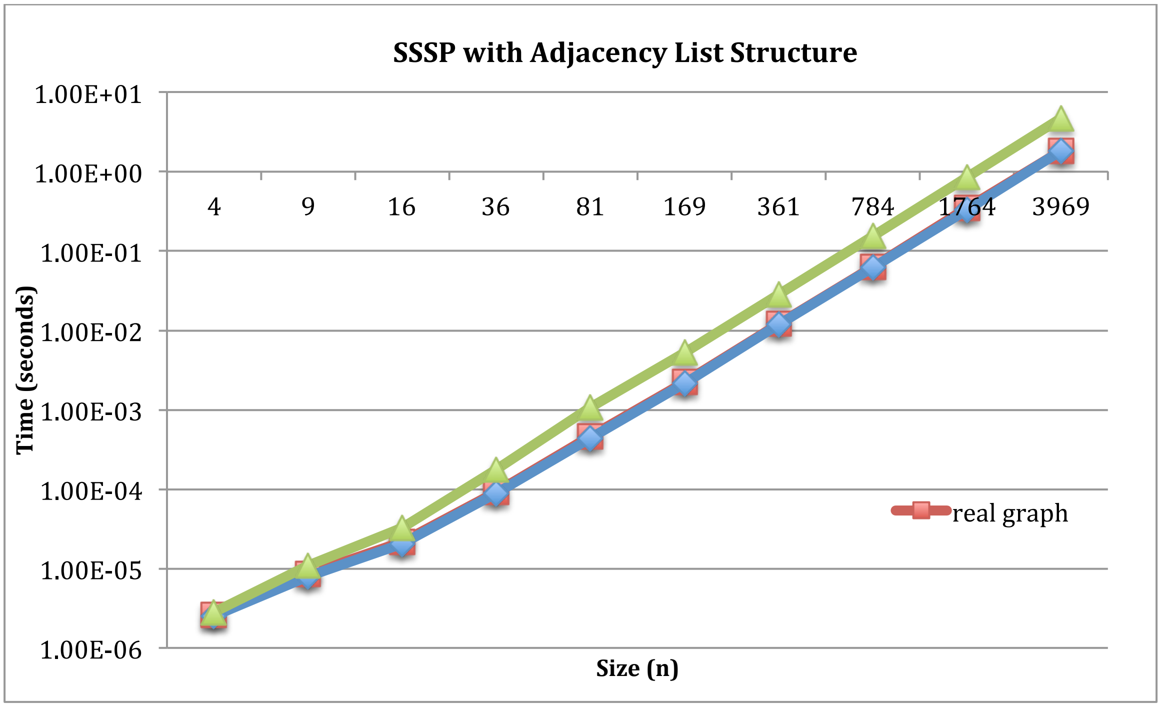

Figure 1: Execution time in seconds vs. input size for an adjacency list structure tested with real, sparse, and dense graphs. Input sizes resemble a base-2 logarithmic distribution.

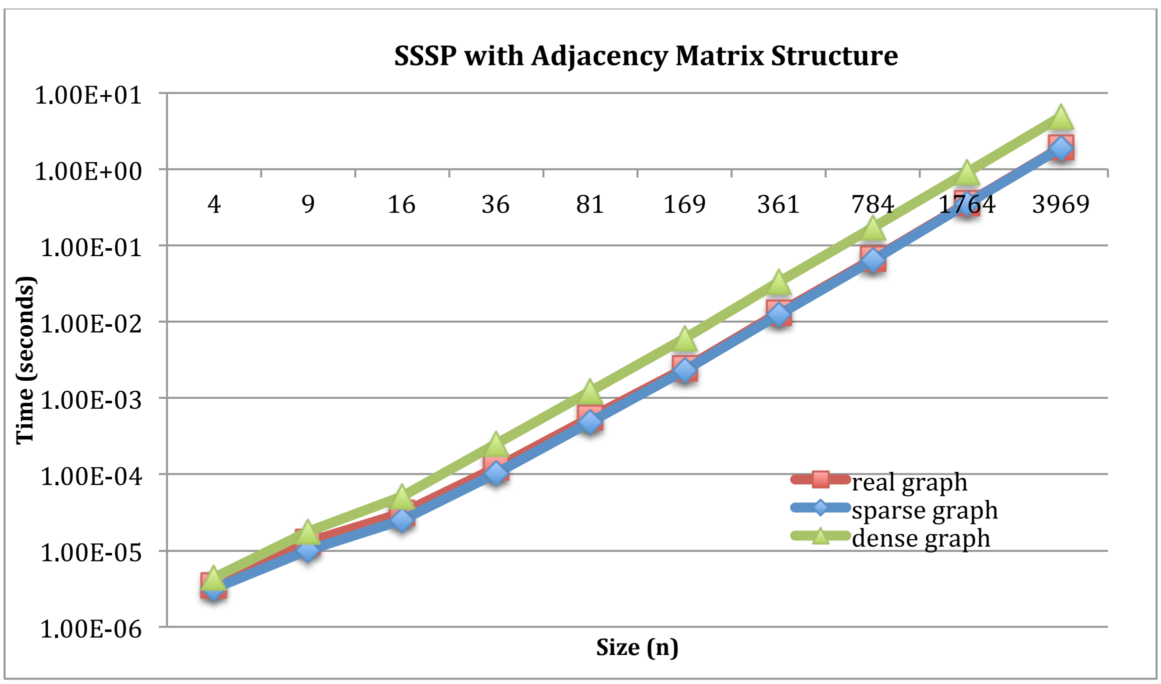

Figure 2: Execution time in seconds vs. input size for an adjacency matrix structure tested with real, sparse, and dense graphs. Input sizes resemble a base-2 logarithmic distribution.

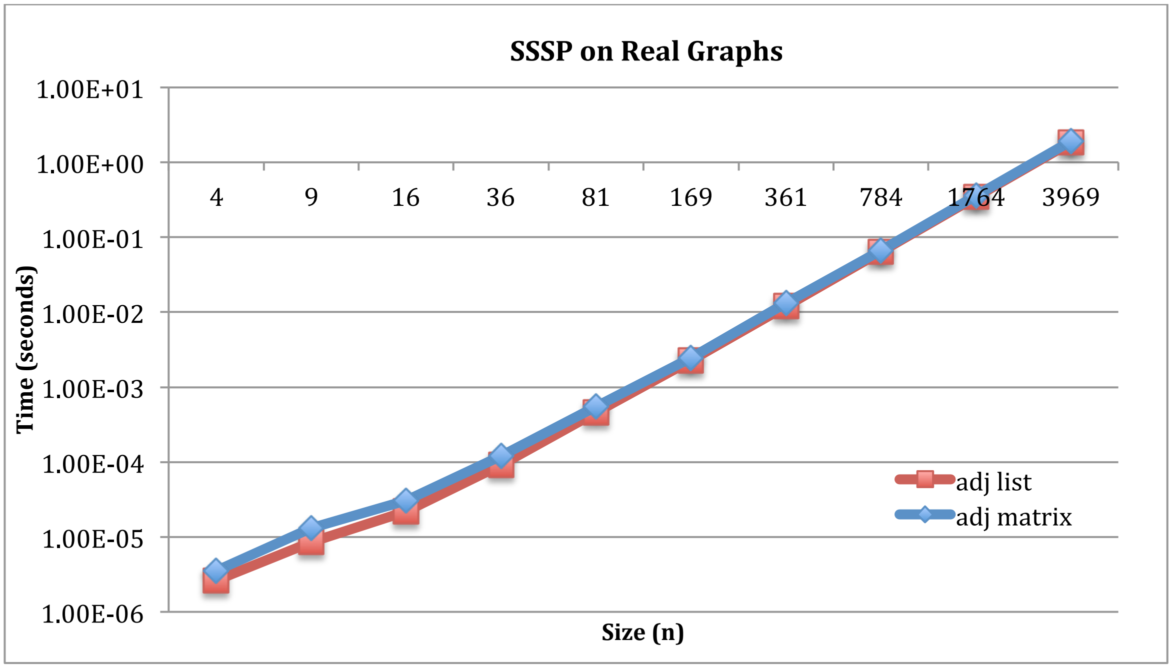

Figure 3: Execution time in seconds vs. input size for real graphs generated through a hash formula. Implementations using an adjacency matrix structure and an adjacency list structure are shown. Input sizes resemble a base-2 logarithmic distribution.

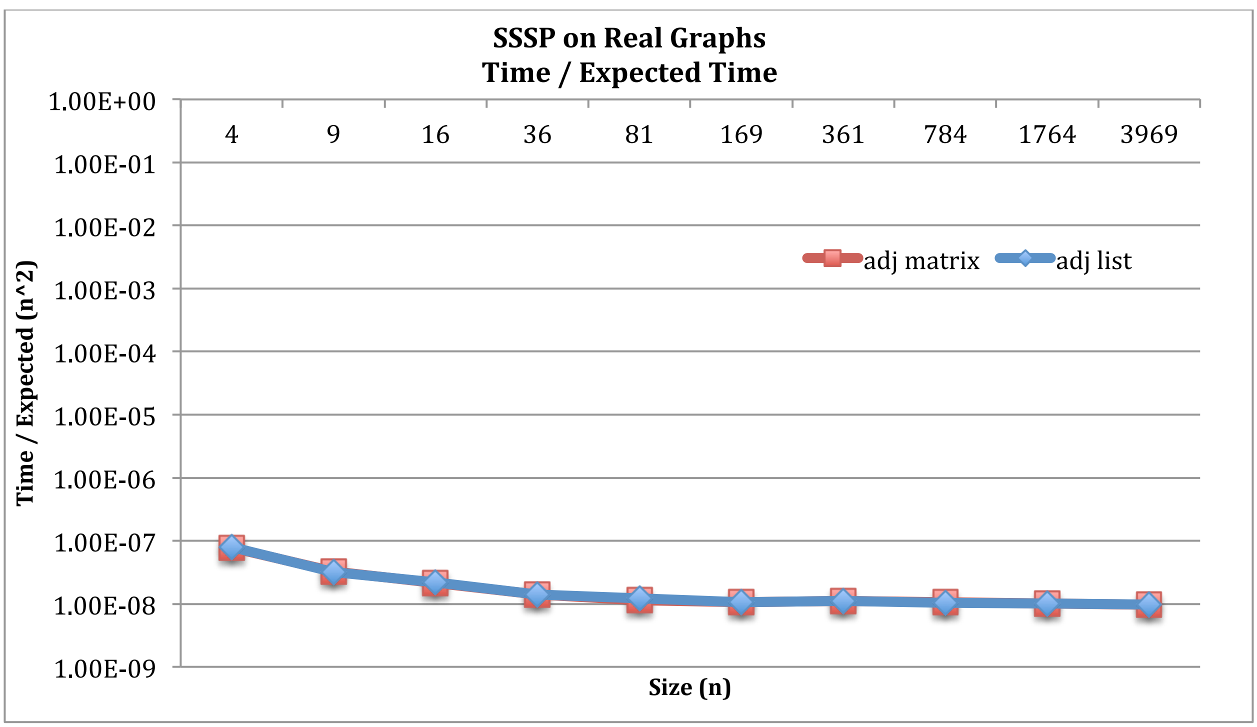

Figure 4: Execution time in seconds vs. time divide by expected time, O(n^2logn) using an adjacency matrix structure and an adjacency list structure. Real graphs were used as input, and input sizes resemble a base-2 logarithmic distribution.

CSCE 221H
Assignment Coverpage

Please list below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students	Other People	Printed Material	Web Material (give url)	Other Sources
1.	1.	1.	1. www.cplusplus.com	1.
2.	2.	2.	2. www.wikipedia.com	2.
3.	3.	3.	3. see URL's below	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

http://stackoverflow.com/questions/4148428/template-way-of-finding-maximum-allowable-value
http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. *Disciplinary actions range from grade penalty to expulsion*. Please consult the Aggie Honor System Office for additional information regarding academic misconduct - it is your responsibity to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received or given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Assignment (circle one):	CSE Culture Assn	Other Assn
Non-Culture Assignment: assignment type:	Program 4 code	
CSE Culture Seminar: Speaker/Seminar/Date:		
CSE Culture Biography: Person's Name:		
Today's Date:	April 29, 2014	
Printed Name:	Carsten Hood	
Signature:		