INSERT TITLE

Michael Morikawa

May 12, 2020

Lab Questions

Question 1 Outline an approach to represent an undirected graph using Edge List Structure. Try it out with a simple undirected graph with about 4 vertices and a few edges.
In an Edge List Structure there are vertex objects stored in a collection; the vertices simply store the element and the position in the collection. The edges contain an element and the positions of the vertices that are the endpoints of the edge. So if we have a graph with vertices A, B, C, D and edges 1, 2, 3. If edge 1 connects vertices A and B then in the edge object it will store 1 as the element, and pointers to vertices A and B. Edge 2 connects B and C will be similar but the endpoints will be pointing to B and C.

Question 2 Discuss advantages of Adjacency List over Edge List Structure for an undirected graph.
Adjacency List Structure runs faster than Edge List because it the list structure we store in the vertices references to the edges incident to the vertex, and in the edges we store the endpoint vertices for the edge. This allows for faster calculation of the incident edges of a vertex, and determining if a vertex is adjecnt to another. Instead of being proportional to the number of edges like in the edge list it is proportional to the degree of the vertex.

Source Code

AdjacencyListGraph.hpp

```
#pragma once
#include <iostream>
#include <list>
#include <vector>
#include "Graph.h"

using namespace std;

// Created by T. Vo for CSCI 230
// Based on Java version of Goodrich book w/o template
// and minimal exception handling
// Some operations are incomplete and there are provisions
// to change from map to a list/vector for adjacency list
//Modified by Michael Morikawa to use a vector for adj list

class AdjacencyListGraph : public Graph
{
private:
    bool isDirected;
    list<Vertex *> vertices;
    list<Edge *> edges;
```

```
/* A vertex of an adjacency list graph representation. */
class InnerVertex : public Vertex
{
private:
    string element;
    //Position<Vertex<V>> pos;d
    Vertex *pos;

    vector<pair<Vertex *, Edge *>> *outgoing;
    vector<pair<Vertex *, Edge *>> *incoming;

public:
    /* Constructs a new InnerVertex instance storing the given element. */
    InnerVertex(string elem, bool graphIsDirected = false)
    {
        element = elem;
        outgoing = new vector<pair<Vertex *, Edge *>>();
        if (graphIsDirected)
            incoming = new vector<pair<Vertex *, Edge *>>();
        else
            incoming = outgoing; // if undirected, alias outgoing map
    }

    /* Returns the element associated with the vertex. */
    string getElement() { return element; }

    /* Stores the position of this vertex within the graph's vertex list. */
    void setPosition(Vertex *p) { pos = p; }

    /* Returns the position of this vertex within the graph's vertex list. */
    Vertex *getPosition() { return pos; }

    /* Returns reference to the underlying map of outgoing edges. */
    vector<pair<Vertex *, Edge *>> *getOutgoing() { return outgoing; }

    /* Returns reference to the underlying map of incoming edges. */
    vector<pair<Vertex *, Edge *>> *getIncoming() { return incoming; }
}; //------------ end of InnerVertex class ------------

//---------------- nested InnerEdge class ----------------
/* An edge between two vertices. */
class InnerEdge : public Edge
{
private:
    int element;
    Edge *pos;
    vector<Vertex *> endpoints;

public:
    /* Constructs InnerEdge instance from u to v, storing the given element. */
    InnerEdge(Vertex *u, Vertex *v, int elem)
    {
        element = elem;
```

```
            endpoints.push_back(u);
            endpoints.push_back(v);
        }

        /* Returns the element associated with the edge. */
        int getElement() { return element; }

        /* Returns reference to the endpoint array. */
        vector<Vertex *> getEndpoints() { return endpoints; }

        /* Stores the position of this edge within the graph's vertex list. */
        void setPosition(Edge *p) { pos = p; }

        /* Returns the position of this edge within the graph's vertex list. */
        Edge *getPosition() { return pos; }
    }; //------------ end of InnerEdge class ------------

public:
    /*
        * Constructs an empty graph.
        * The parameter determines whether this is an undirected or directed graph.
        */
    AdjacencyListGraph(bool directed = false)
    {
        isDirected = directed;
    }

    ~AdjacencyListGraph()
    {
        // should deallocate memory here
    }

    /* Returns the number of vertices of the graph */
    int numVertices()
    {
        return static_cast<int>(vertices.size());
    }

    /* Returns the number of edges of the graph */
    int numEdges()
    {
        return static_cast<int>(edges.size());
    }

    /* Returns the vertices of the graph as an iterable collection */
    list<Vertex *> getVertices()
    {
        return vertices;
    }

    /* Returns the edges of the graph as an iterable collection */
    list<Edge *> getEdges()
    {
        return edges;
```

```
}

/*
    * Returns the number of edges leaving vertex v.
    * Note that for an undirected graph, this is the same result
    * returned by inDegree
    * throws IllegalArgumentException if v is not a valid vertex?
    */
int outDegree(Vertex *v) // throws IllegalArgumentException;
{
    InnerVertex *vert = static_cast<InnerVertex *>(v);
    return static_cast<int>(vert->getOutgoing()->size());
}

/**
    * Returns the number of edges for which vertex v is the destination.
    * Note that for an undirected graph, this is the same result
    * returned by outDegree
    * throws IllegalArgumentException if v is not a valid vertex
    */
int inDegree(Vertex *v) // throws IllegalArgumentException;
{
    InnerVertex *vert = static_cast<InnerVertex *>(v);
    return static_cast<int>(vert->getIncoming()->size());
}

/*
    * Returns an iterable collection of edges for which vertex v is the origin.
    * Note that for an undirected graph, this is the same result
    * returned by incomingEdges.
    * throws IllegalArgumentException if v is not a valid vertex
    */
vector<Edge *> outgoingEdges(Vertex *v) // throws IllegalArgumentException;
{
    vector<Edge *> temp;
    vector<pair<Vertex *, Edge *>> *vecPtr = static_cast<InnerVertex *>(v)->getOutgoing();
    for (auto it = vecPtr->begin(); it != vecPtr->end(); ++it)
    {
        temp.push_back(it->second);
    }
    return temp;
}

/*
    * Returns an iterable collection of edges for which vertex v is the destination.
    * Note that for an undirected graph, this is the same result
    * returned by outgoingEdges.
    * throws IllegalArgumentException if v is not a valid vertex
    */
vector<Edge *> incomingEdges(Vertex *v) // throws IllegalArgumentException;
{
    vector<Edge *> temp;
    vector<pair<Vertex *, Edge *>> *vecPtr = static_cast<InnerVertex *>(v)->getIncoming();
    for (auto it = vecPtr->begin(); it != vecPtr->end(); ++it)
    {
        temp.push_back(it->second);
    }
    return temp;
}

/* Returns the edge from u to v, or null if they are not adjacent. */
Edge *getEdge(Vertex *u, Vertex *v) // throws IllegalArgumentException;
{
    Edge *temp = nullptr;
    vector<pair<Vertex *, Edge *>> *vecPtr = static_cast<InnerVertex *>(u)->getOutgoing();
    auto it = vecPtr->begin();
    for (it, it = vecPtr->begin(); it != vecPtr->end(); it++)
    {
        if (it->first == v)
        {
            break;
        }
    }
    if (it != vecPtr->end())
        temp = it->second;
    return temp; // origin.getOutgoing().get(v);    // will be null if no edge from u to v
}

/*
    * Returns the vertices of edge e as an array of length two.
    * If the graph is directed, the first vertex is the origin, and
    * the second is the destination.  If the graph is undirected, the
    * order is arbitrary.
    */
vector<Vertex *> endVertices(Edge *e) // throws IllegalArgumentException;
{
    vector<Vertex *> endpoints = static_cast<InnerEdge *>(e)->getEndpoints();
    return endpoints;
}

/* Returns the vertex that is opposite vertex v on edge e. */
Vertex *opposite(Vertex *v, Edge *e) // throws IllegalArgumentException;
{
    vector<Vertex *> endpoints = static_cast<InnerEdge *>(e)->getEndpoints();

    if (endpoints[0] == v)
        return endpoints[1];
    else
        return endpoints[0];
}

/* Inserts and returns a new vertex with the given element. */
Vertex *insertVertex(string element)
{
    Vertex *v = new InnerVertex(element, isDirected);
    vertices.push_back(v);
    static_cast<InnerVertex *>(v)->setPosition(vertices.back());
    return v;
```

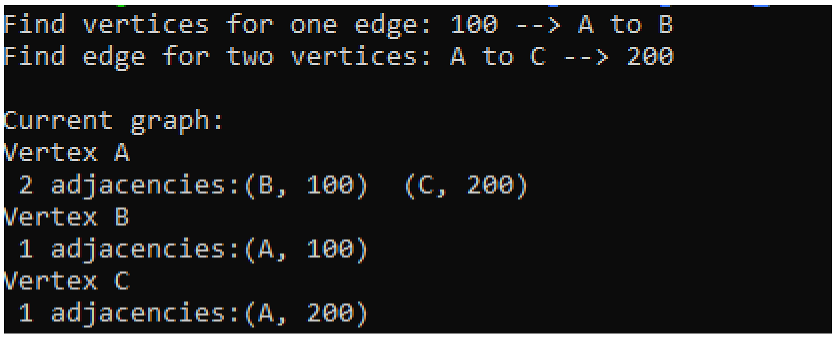
```
}

/*
    * Inserts and returns a new edge between vertices u and v, storing given element.
    *
    * throws IllegalArgumentException if u or v are invalid vertices, or if an edge already exists
    */
Edge *insertEdge(Vertex *u, Vertex *v, int element) // throws IllegalArgumentException;
{
    Edge *e = new InnerEdge(u, v, element);
    edges.push_back(e);
    static_cast<InnerEdge *>(e)->setPosition(edges.back());
    InnerVertex *origin = static_cast<InnerVertex *>(u);
    InnerVertex *dest = static_cast<InnerVertex *>(v);
    (origin->getOutgoing())->push_back(pair<Vertex *, Edge *>(v, e));
    (dest->getIncoming())->push_back(pair<Vertex *, Edge *>(u, e));

    return e;
}

/* Removes a vertex and all its incident edges from the graph. */
void removeVertex(Vertex *v) // throws IllegalArgumentException;
{
    //for (Edge<E> e : vert.getOutgoing().values())
    //        removeEdge(e);
    //for (Edge<E> e : vert.getIncoming().values())
    //        removeEdge(e);
    //// remove this vertex from the list of vertices
    //vertices.remove(vert.getPosition());
}

/* Removes an edge from the graph. */
void removeEdge(Edge *e) // throws IllegalArgumentException;
{
    // remove this edge from vertices' adjacencies
    //InnerVertex<V>[] verts = (InnerVertex<V>[]) edge.getEndpoints();
    //verts[0].getOutgoing().remove(verts[1]);
    //verts[1].getIncoming().remove(verts[0]);
    //// remove this edge from the list of edges
    //edges.remove(edge.getPosition());
}

void print()
{
    for (auto itr = vertices.begin(); itr != vertices.end(); itr++)
    {
        cout << "Vertex " << (*itr)->getElement() << endl;
        if (isDirected)
            cout << " [outgoing]";
        cout << " " << outDegree(*itr) << " adjacencies:";
        for (auto e : outgoingEdges(*itr))
            cout << "(" << opposite(*itr, e)->getElement() << ", " << e->getElement() << ")"
                 << "  ";
        cout << endl;
```

```
            if (isDirected)
            {
                cout << " [incoming]";
                cout << " " << inDegree(*itr) << " adjacencies:";
                for (auto e : incomingEdges(*itr))
                    cout << "(" << opposite(*itr, e)->getElement() << ", " << e->getElement() << ")"
                         << "  ";
                cout << endl;
            }
        }
    }
};
```

testGraph.cpp

```
#include <iostream>
#include <string>
#include "AdjacencyListGraph.hpp"

using namespace std;

// A simple driver to test AdjacencyMapGraph class
// Created by T. Vo for CSCI 230

int main()
{
        AdjacencyListGraph g1; // use g1(true) for digraph
        Vertex *v1 = g1.insertVertex("A");
        Vertex *v2 = g1.insertVertex("B");
        Vertex *v3 = g1.insertVertex("C");
        Edge *e1 = g1.insertEdge(v1, v2, 100);
        Edge *e2 = g1.insertEdge(v1, v3, 200);

        cout << "Find vertices for one edge: ";
        vector<Vertex *> endPoints = g1.endVertices(e1);
        Vertex *origin = endPoints[0];
        Vertex *dest = endPoints[1];
        cout << e1->getElement() << " --> " << origin->getElement() << " to " << dest->getElement() << 

        cout << "Find edge for two vertices: ";
        Edge *myEdge = g1.getEdge(v1, v3);
        cout << v1->getElement() << " to " << v3->getElement() << " --> " << myEdge->getElement() << en

        cout << "\nCurrent graph:" << endl;
        g1.print();
        cout << endl;

        return 0;
}
```

Output



```
Find vertices for one edge: 100 --> A to B
Find edge for two vertices: A to C --> 200

Current graph:
Vertex A
 2 adjacencies:(B, 100)  (C, 200)
Vertex B
 1 adjacencies:(A, 100)
Vertex C
 1 adjacencies:(A, 200)
```