UNIVERSITY OF SOUTHERN DENMARK

Port3 - ADA5 - E14

Mathias (manee12), Keerthikan (kerat12) & Anders (anbae12)

November 2014

1 Intro

”You are going to develop a travel-planning system in which you will need to implement a method for computing the cheapest route between destinations.
Data about the destinations and possible routes between them are placed in a file (to be found on black board next to the assignment) where each line contains a destination followed by the cities to which you can travel and the associated cost.
Notice that even though there is a route from A to B, there might not be one from B to A.”

2 Solution

2.1 Question #1

A routine for loading in the file and appropriate data structures for representing the data is shown in appendices D and B.
We are using a Hash-map where from cities are associated with accesable cities and thier egde cost.

2.2 Question #2

As mentioned before the from cities is associate to cities with a giving cost. The apporach for printing the ”to cities” is showed by the psodocode and the listing below.

```
void Graph::printFrom(std::string from){
    if(vertices.find(from) == vertices.end()){
        std::cout <<  "City \"" + from +  "\" not found" << std::endl;
        return ;
    }
    for(auto it = vertices[from]->edge.get_container().begin() ; it != vertices[from]->edge.get_contain
        std::cout << "To: " << it->first->element << " Cost: " <<it->second  << std::endl;
    }
}
```

Accesable cities from ”Odense” is copyed from the console output and showed below:

```
To: Stubbekøbing Cost: 20
To: Værløse Cost: 22
To: Hjørring Cost: 33
To: København Cost: 29
To: Søllested Cost: 54
To: Gedved Cost: 62
To: Broby Cost: 67
To: Odder Cost: 48
To: Hørning Cost: 34
To: Spenstrup Cost: 144
To: Dronningmølle Cost: 73
To: Karup Cost: 204
To: Kalundborg Cost: 173
To: Kerteminde Cost: 193
To: Jerup Cost: 87
To: Hovborg Cost: 221
To: Vedbæk Cost: 163
To: Rønde Cost: 187
```

To: Mørkøv Cost: 47
To: Langebæk Cost: 234
To: Langeskov Cost: 191
To: Ålsgårde Cost: 177
To: Nysted Cost: 102

2.3 Question #3

We have chosen to use the Dijkstras algorithm for computing the quickest route between two destinations. The properties of Dijkstras algorithm producing a shortest path for a graph with non-negative edge path costs.
The Dijkstras creates the graph of vetrices and egdes from a priority queues which ensure the algorithm at alltime get next vertex with the lowest egde cost. It was nessaray to make an overloadoperator *Comp* (appendix B.1) to get the queue match our data structure and sorting correctly for the *cost*.

```
DijkResult Dijkstras::Run(std::string from, std::string to){
    if (mGraph->vertices.find(from) == mGraph->vertices.end()) {
        std::cout<<"Not found: "<<from<<std::endl;
        exit(0);
    }
    if (mGraph->vertices.find(to) == mGraph->vertices.end()) {
        std::cout<<"Not found: "<<to<<std::endl;
        exit(0);
    }
    std::string depTown = from;
    std::string arTown = to;

    mGraph->vertices[from]->dist=0;
    dijkstrasQueue.push(mGraph->vertices[from]);
    while (!dijkstrasQueue.empty()) {
        from = dijkstrasQueue.top()->element;
        dijkstrasQueue.pop();
        while (!mGraph->vertices[from]->edge.empty()) {
            std::string to = mGraph->vertices[from]->edge.top().first->element;
            int cost   = mGraph->vertices[from]->edge.top().second;

            int edgeplusnode = cost + mGraph->vertices[from]->dist;

            if ( edgeplusnode <  mGraph->vertices[to]->dist) {
                mGraph->vertices[to]->dist=edgeplusnode;
                mGraph->vertices[to]->from=mGraph->vertices[from];
            }
            dijkstrasQueue.push(mGraph->vertices[from]->edge.top().first );
            mGraph->vertices[from]->edge.pop();
        }
    }
    auto route = path(mGraph->vertices[depTown], mGraph->vertices[arTown]);
    return DijkResult(route.second,mGraph->vertices[arTown]->dist,0, route.first);
}
```

3 Examples and Benchmarks

3.1 Ten different from and to cities

```


CONSOLE OUTPUT
```

3.2 Odense to Aalborg

```


CONSOLE OUTPUT
```

3.3 Odense to Holstebro

```


CONSOLE OUTPUT
```

3.4 Odense to Humlebæk

```


CONSOLE OUTPUT
```

4 Conclusion

Appendices

A main

```cpp
//
//  main.cpp
//  Navigation
//
//  Created by Mathias, Keerthikan og Anders.
//

#include "Vertex.h"
#include "FileHandle.h"
#include "Graph.h"
#include "Dijkstras.h"

int main(int argc, const char * argv[]) {
    std::shared_ptr<Graph> graph(new Graph);
    clock_timer timerrecord;
    std::string fromTown;
    std::string toTown;



    std::cout<<"Departure town: ";
    std::cin>>fromTown;
    std::cout<<"Arrival town:   ";
    std::cin>>toTown;

    //////////// Question #1 ////////////
    //FileHandle filehandle("../../data.raw");
    FileHandle filehandle("/Users/anderslaunerbaek/Documents/data.raw");
    filehandle.doParse(graph);
    //////////// Question #2 ////////////
    graph->printFrom(fromTown);
    //////////// Question #3 ////////////
    Dijkstras di(graph);




    timerrecord.start_timer();
    DijkResult result =  di.Run(fromTown, toTown);
    timerrecord.stop_timer();


    std::cout <<"---------------------"<<std::endl;
    std::cout <<"Departure: "<< fromTown <<std::endl;
    std::cout <<"Arrival:   "<< toTown <<std::endl;
    std::cout <<"Shifts:    "<< result.Shifts <<": " << result.Path << std::endl;
    std::cout <<"Ticket:    "<< result.Ticket <<",- DKK"<<std::endl;
    std::cout <<"Duration:  "<< timerrecord.duration <<" [ms]"<<std::endl;
    std::cout <<"---------------------"<<std::endl;
    return 0;
}
```

B Vertex

B.1 Vertex.h

```cpp
#include <map>
#include <string>
#include <vector>
#include <iostream>
#include <queue>

#ifndef VERTEX_H_
```

```
#define VERTEX_H_

//Inherents from priority_queue and adds get_container which returns the underlying container
template <class Container>
class Adapter : public Container {
public:
    typedef typename Container::container_type container_type;
    container_type &get_container() { return this->c; }
};

class Vertex;

//Comp used to compare values in prority_queue
struct Comp {
    bool operator()(const std::pair<Vertex*, int> &a ,const std::pair<Vertex*, int> &b ) const {
        return b.second < a.second;
    }
};

class Vertex {
    typedef std::priority_queue<std::pair<Vertex*, int>, std::vector<std::pair<Vertex*, int> >, Comp> C
    typedef Adapter<C> Container;
public:
    Vertex(std::string value);
    std::string element;
    Container edge;
    int dist;
    Vertex* from;
};

#endif /* VERTEX_H_ */
```

B.2 Vertex.cpp

```
/*
 * Vertex.cpp
 *
 *  Created on: Oct 26, 2014
 *      Author: exchizz
 */

#include "Vertex.h"
#include <limits>

Vertex::Vertex(std::string value) {
    element = value;
    dist = std::numeric_limits<int>::max();
    from=NULL;  // Used in dijkstras
}
```

C Graph

C.1 Graph.h

```
#include <map>
#include <string>
#include <vector>
#include <iostream>
#include "Vertex.h"
#include "LateXGenerator.h"
#include <queue>
#ifndef GRAPH_H_
#define GRAPH_H_

class Graph {
    typedef std::map <std::string, Vertex*> Vertices;
    //typedef std::pair <Vertex, int> Edge;
public:
    std::map <std::string, Vertex*> vertices;
    void addVertex(std::string value);
    void addEdge(std::string From, std::string To, int cost);
    std::string printFromDot(std::string from);
    void printFrom(std::string from);
};

#endif /* GRAPH_H_ */
```

C.2 Graph.cpp

```
#include "Graph.h"

void Graph::addVertex(std::string value) {
    if(vertices.find(value) != vertices.end()){
        throw new std::string("Element \"" + value + "\" already exists!");
    }
    vertices[value] = new Vertex(value);
}

void Graph::addEdge(std::string From, std::string To, int Cost) {
    if(vertices.find(From) == vertices.end())
        throw new std::string("From \"" + From + "\" does not exist!");

    if(vertices.find(To) == vertices.end())
        throw new std::string("To \"" + To + "\" does not exist!");

    Vertex* from = vertices.find(From)->second;
    Vertex* to = vertices.find(To)->second;

    from->edge.push(std::make_pair(to,Cost));
}

std::string Graph::printFromDot(std::string from){
    LateXGenerator lateXGenerator;
    lateXGenerator.AddVertex(from);
        for(auto it = vertices[from]->edge.get_container().begin() ; it != vertices[from]->edge.get_con
            std::cout << "From \"" + from + "\" to: " << it->first->element << std::endl;
            lateXGenerator.AddEdge(from,it->first->element, it->second);
        }
    return lateXGenerator.getOutput();
}

void Graph::printFrom(std::string from){
    if(vertices.find(from) == vertices.end()){
        std::cout <<  "City \"" + from +  "\" not found" << std::endl;
        return ;
    }
    for(auto it = vertices[from]->edge.get_container().begin() ; it != vertices[from]->edge.get_contain
        std::cout << "To: " << it->first->element << " Cost: " <<it->second  << std::endl;
    }
}
```

D FileHandle

D.1 FileHandle.h

```
#ifndef FILEHANDLE_H_
#define FILEHANDLE_H_
#include <fstream>
#include <vector>
#include <iostream>
#include <sstream>
#include <algorithm>
#include "Vertex.h"
#include "Graph.h"
#include <memory>

```

```cpp

class FileHandle {
public:
    FileHandle(std::string filename);
    void doParse(std::shared_ptr<Graph> &graph);
    std::string rtrim(std::string s);
    std::string ltrim(std::string s);
    std::string getFrom(std::stringstream &stream);
    void trim(std::string &);
    std::string to, cost;
    bool printException;
private:
    std::string line;
    std::ifstream fin;
};

#endif /* FILEHANDLE_H_ */
```

D.2 FileHandle.cpp

```cpp
#include "FileHandle.h"

FileHandle::FileHandle(std::string filename) {
    printException = false;
    fin.open(filename);
    if (!fin.good()){
        std::cout << "Unable to open file";
        exit(0);
    }
}

void FileHandle::doParse(std::shared_ptr<Graph> &graph){
    while(fin.peek() != -1){
        // Ignore starting { in line
        fin.seekg (1, std::ios::cur);

        //Get next line
        getline(fin, line);

        std::stringstream lineStream(line);

        std::string from = getFrom(lineStream);
        //Add vertex, else catch exception
        try {
            graph->addVertex(from);
        } catch (std::string *e){
            if(printException){
                std::cout << "exception: " << *e << std::endl;
            }
        }

        while(std::getline(lineStream,to,',') && std::getline(lineStream,cost,',')){
            //Remove leading and trailing whitespaces.
            trim(to);

            //Convert to integer
            int iCost;
            std::istringstream ( cost ) >>  iCost;

            //Add vertex if not existing, else catch exception
            try {
                graph->addVertex(to);
            } catch (std::string *e){
                if(printException){
                    std::cout << "exception: " << *e << std::endl;
                }
            }
            //Add edge
            graph->addEdge(from, to, iCost);
        }
    }
}
```

```
//Trim left side of string
std::string FileHandle::ltrim(std::string s){
    s.erase(s.begin(),find_if_not(s.begin(),s.end(),[](int c){return isspace(c);}));
    return s;
}
//Trim right side of string
std::string FileHandle::rtrim(std::string s){
    s.erase(find_if_not(s.rbegin(),s.rend(),[](int c){return isspace(c);}).base(), s.end());
    return s;
}
//Trim right and left
void FileHandle::trim(std::string &s){
    s = ltrim(rtrim(s));
}
//Extracts "from", from the line
std::string FileHandle::getFrom(std::stringstream &stream){
    std::string from;
    std::getline(stream,from,',');
    return from;
}
```

E Dijkstras

E.1 dijkstras.h

```
#ifndef __Navigation__dijkstras__
#define __Navigation__dijkstras__

#include <stdio.h>
#include <string>
#include <fstream>
#include <deque>
#include "Graph.h"
#include "Vertex.h"
#include "clock_timer.h"
#include <ctime>
#include <memory>
#include <limits>
struct Comp1 {
    bool operator()(const Vertex* a ,const Vertex* b ) const {
        return b->dist < a->dist;
    }
};

class DijkResult{
public:
    int Shifts;
    int Ticket;
    float Duration;
    std::string Path;

    DijkResult(int shifts, int ticket, float duration, std::string path){
        this->Shifts = shifts;
        this->Ticket = ticket;
        this->Duration = duration;
        this->Path = path;
    }
};
class Dijkstras{
    typedef std::priority_queue<Vertex*, std::vector<Vertex* >, Comp1> diQueue;
public:
    Dijkstras(std::shared_ptr<Graph> graph);
    DijkResult Run(std::string from, std::string to);
    std::pair<std::string, int> path(Vertex*, Vertex*);
    diQueue dijkstrasQueue;
private:
    std::shared_ptr<Graph> mGraph;
};

#endif /* defined(__Navigation__dijkstras__) */
```

E.2 dijkstras.cpp

```
#include "Dijkstras.h"

std::pair<std::string, int> Dijkstras::path(Vertex* from, Vertex* arrival){
    if (arrival->element == from->element) {
        return std::make_pair(arrival->element, 0);
    }
    auto val = path(from, arrival->from);
    return std::make_pair(val.first + " -> " + arrival->element, val.second+1 );
}
DijkResult Dijkstras::Run(std::string from, std::string to){
    if (mGraph->vertices.find(from) == mGraph->vertices.end()) {
        std::cout<<"Not found: "<<from<<std::endl;
        exit(0);
    }
    if (mGraph->vertices.find(to) == mGraph->vertices.end()) {
        std::cout<<"Not found: "<<to<<std::endl;
        exit(0);
    }
    std::string depTown = from;
    std::string arTown = to;

    mGraph->vertices[from]->dist=0;
    dijkstrasQueue.push(mGraph->vertices[from]);
    while (!dijkstrasQueue.empty()) {
        from = dijkstrasQueue.top()->element;
        dijkstrasQueue.pop();
        while (!mGraph->vertices[from]->edge.empty()) {
            std::string to = mGraph->vertices[from]->edge.top().first->element;
            int cost  = mGraph->vertices[from]->edge.top().second;

            int edgeplusnode = cost + mGraph->vertices[from]->dist;

            if ( edgeplusnode <  mGraph->vertices[to]->dist) {
                mGraph->vertices[to]->dist=edgeplusnode;
                mGraph->vertices[to]->from=mGraph->vertices[from];
            }
            dijkstrasQueue.push(mGraph->vertices[from]->edge.top().first );
            mGraph->vertices[from]->edge.pop();
        }
    }
    auto route = path(mGraph->vertices[depTown], mGraph->vertices[arTown]);
    return DijkResult(route.second,mGraph->vertices[arTown]->dist,0, route.first);
}

Dijkstras::Dijkstras(std::shared_ptr<Graph> graph){

    this->mGraph = graph;

    for(auto it = mGraph->vertices.begin(); it != mGraph->vertices.end(); ++it){
        it->second->dist = std::numeric_limits<int>::max();
        it->second->from = NULL;
        for(auto itwo = it->second->edge.get_container().begin(); itwo != it->second->edge.get_container
            itwo->first->dist = std::numeric_limits<int>::max();
            itwo->first->from = NULL;
        }
    }
};
```

F LateXGenerator

F.1 LateXGenerator.h

```
#include <string>
#ifndef LATEXGENERATOR_H_
#define LATEXGENERATOR_H_

class LateXGenerator {
public:
    LateXGenerator();
    std::string output;
    void AddVertex(std::string name);
    void AddEdge(std::string, std::string, int);
    std::string getOutput();

};

#endif /* LATEXGENERATOR_H_ */
```

F.2 LateXGenerator.cpp

```
/*
 * LateXGenerator.cpp
 *
 *  Created on: Oct 26, 2014
 *      Author: exchizz
 */

#include "LateXGenerator.h"
#include<sstream>

LateXGenerator::LateXGenerator() {
    output = "  digraph G{";
}

void LateXGenerator::AddVertex(std::string name){
    /*
    digraph G{
    a [label="Node A"];
    b [label="Node B"];
    a->b[label=" An edge"];
    }
    */
    output += name + " [label=\"" + name + "\"];";
}

void LateXGenerator::AddEdge(std::string From, std::string To, int Cost){
    std::ostringstream os;
    os << Cost;
    output += From + "->" + To + "[label=\"" + os.str() +  "\"];";
}

std::string LateXGenerator::getOutput(){
    return output + "}";
}
```

G clock_timer

G.1 clock_timer.h

```
//
//  clock_timer.h
//
//  Created by Anders Launer Baek on 12/09/14.
//  Copyright (c) 2014 Anders Launer Baek. All rights reserved.
//

/*
 remember to include header:
 #include "clock_timer.h"

 Useage:

 clock_timer timerrecord;
 timerrecord.start_timer();
 timerrecord.stop_timer();
 */
```

```

#ifndef __timer_clock__clock_timer__
#define __timer_clock__clock_timer__
#include <ctime>
#include <iostream>
class clock_timer{
public:
    void start_timer();
    void stop_timer();
    std::clock_t time;
    std::clock_t start;
    double duration;
};

#endif /* defined(__timer_clock__clock_timer__) */
```

G.2 clock_timer.cpp

```
//
//  clock_timer.cpp
//  timer_clock
//
//  Created by Anders Launer Baek on 12/09/14.
//  Copyright (c) 2014 Anders Launer Baek. All rights reserved.
//

#include "clock_timer.h"


void clock_timer::start_timer(){
    start = std::clock();
}

void clock_timer::stop_timer(){
    duration=( std::clock() - start ) / (double) CLOCKS_PER_SEC*1000;
    //std::cout<< "Time: "<<time<<"[ms]"<< std::endl;
}
```