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 a appropriate data structure for representing the data is shown in appendices B and D. We are using a Hash-map where from-cities are associated with accesable to-cities and their edge cost.
All the from-vertices are listed in a hash-map, the reason we using a hash-map is to make the lookup time constant.
Each vertex is represented by a class called Vertex. This includes a string with the name of the vertex, a priority queue including all of its edges. The reason why a priority queue has been chosen instead of an ordinary vector is to avoid using a sorting algorithm to sort the vector.
To parse the file, the following approach has been used:

```
foreach(line in file){
    from = getFrom(line)
    while(getline(line,to,',') && getline(line,cost,',')) {
        graph->addVertex(to)
        graph->addEdge(from, to, cost)
    }
}
```

For each line in the file, the from-city is extracted. For the rest of the line, the while loop will parse to and cost. If getline can not get a value, it will return false. For each to-city and associated cost, it will create a vertex and add an edge between the two cites. If the from-city already exist it will throw an exception. This has been omitted in the code above for simplicity.

2.2 Question #2

As mentioned before the from-cities are associate to-cities with a giving cost. The approach for printing the to-cities is showed by the pseudocode and the listing below.

```
printFrom(from-city){
    check all vertices
    if ( end of vertices ) return "from-city not found "
    while(! vertices (from-city) not end ){
        print associated cities and giving cost
        }
}
```

It first checks if the from-city is in the hash-map. If not, it returns and will not print any associated cities. If found, it will iterate through all the associated cities and their cost.
Accessible cities from "Odense" is copied from the console output and showed in table 2.1.

To	Cost
Stubbekøbing	20
Værløse	22
Hjørring	33
København	29
Søllested	54
Gedved	62
Broby	67
Odder	48
Hørning	34
Spenstrup	144
Dronningmølle	73
Karup	204
Kalundborg	173
Kerteminde	193
Jerup	87
Hovborg	221
Vedbæk	163
Rønde	187
Mørkøv	47
Langebæk	234
Langeskov	191
Ålsgårde	177
Nysted	102

Table 2.1: Associated arrival cities and cost from Odense.

2.3 Question #3

We have chosen to use the Dijkstra's algorithm for computing the distances to each vertex from a given vertex.

```
DijkResult Run(from, to){
    if (from from in list) {
        cout<<"Not found: " << to << endl;
        exit(0);
    }

    string depTown = from;
    string arTown = to;

    mGraph->verticies[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, route.first);
}
```

The algorithm works by starting at the from-vertex, visit each adjacent vertex (to-cities) and add them to the Dijkstra priority queue. Then add cost and "move" to the node with the lowest cost. The algorithm will again visit each adjacent vertex and update the cost if it's smaller than the exiting cost. By default all vertices cost are initialized to the maximum value an integer can represent, hereby an "infinity cost".

3 Examples and Benchmarks

3.1 Ten different from and to cities

Table 3.1 shows the planning duration from different from-cities and to-cities.

From-city	To-city	Duration
Odense	Aalborg	60,752 [ms]
Næstved	Odense	62,742 [ms]
Balle	Janderup	61,288 [ms]
Beder	Glumsø	63,465 [ms]
Blokhus	Glostrup	62,664 [ms]
Borre	Vadum	63,448 [ms]
Bredebro	Gistrup	63,218 [ms]
Bælum	Hornsyld	62,492 [ms]
Fakse	Bredebro	61,774 [ms]
Farum	Hadsten	61,869 [ms]
Average runtime		62,37 [ms]

Table 3.1: Duration for ten different from-cities and to-cities.

3.2 Test from from-city to to-cities

Table 3.2 the shows planning duration from one from-city to three different to-cities. As you see in the table 3.2 the first planning duration is fairly high according to the two next route plannings.

From-city	To-city	Duration
Odense	Næstved	62,802 [ms]
Odense	København	0,005 [ms]
Odense	Vadum	0,005 [ms]

Table 3.2: Duration for planning from Odense to three cities.

3.3 Planning, Shifts and Ticket price

Following three examples show the route planning from a giving city to a giving city. It return the path and cheapest price and the planning duration.

3.3.1 Odense to Næstved

Departure: Odense
Arrival: Næstved
Shifts: 6: Odense → Værløse → Rødvig Stevns → Humble → Skørping → Kerteminde → Næstved
Ticket: 64,- DKK
Duration: 62,306 [ms]

3.3.2 Odense to Sønderborg

Departure: Odense
Arrival: Sønderborg
Shifts: 5: Odense → Værløse → Hornsyld → Ebberup → Vig → Sønderborg
Ticket: 88,- DKK
Duration: 62,148 [ms]

3.3.3 Vadum to Vejle

Departure: Vadum
Arrival: Vejle
Shifts: 7: Vadum → Højbjerg → Glesborg → Gjern → Assels Øster → Ærøskøbing → Børkop → Vejle
Ticket: 62,- DKK
Duration: 60,99 [ms]

4 Conclusion

By using the Dijkstra's algorithm and having the data in a suitable data structure it was possible to get a nice cheapest route planning algorithm. To make a suitable algorithm which have an average duration of 62,37 [ms] (table 3.1) for planning the cheapest route between ten different from-cities and to-cities, we conclude the portfolio is successful.

Appendices

A main

```
//
//  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

```
#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 <queue>
#ifndef GRAPH_H_
#define GRAPH_H_

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


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>

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
```

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_containe
            itwo->first->dist = std::numeric_limits<int>::max();
            itwo->first->from = NULL;
        }
    }
};
```

F clock_timer

F.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.
//

#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__) */
```

F.2 clock_timer.cpp

```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;
}
```