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.

```
1 void Graph::printFrom(std::string from){
2     if(vertices.find(from) == vertices.end()){
3         std::cout <<  "City \"" + from +  "\" not found" << std::endl;
4         return ;
5     }
6     for(auto it = vertices[from]->edge.get_container().begin() ; it != vertices[from]->edge.get_containe
7         std::cout << "To: " << it->first->element << " Cost: " <<it->second  << std::endl;
8     }
9 }
```

Accesable cities from "Odense" is copyed from the console output and showed in table 2.1.

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*.

```
1 DijkResult Dijkstras::Run(std::string from, std::string to){
2     if (mGraph->vertices.find(from) == mGraph->vertices.end()) {
3         std::cout<<"Not found: "<<from<<std::endl;
4         exit(0);
5     }
6     if (mGraph->vertices.find(to) == mGraph->vertices.end()) {
7         std::cout<<"Not found: "<<to<<std::endl;
8         exit(0);
```

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.

```
9       }
10      std::string depTown = from;
11      std::string arTown = to;
12
13      mGraph->vertices[from]->dist=0;
14      dijkstrasQueue.push(mGraph->vertices[from]);
15      while (!dijkstrasQueue.empty()) {
16          from = dijkstrasQueue.top()->element;
17          dijkstrasQueue.pop();
18          while (!mGraph->vertices[from]->edge.empty()) {
19              std::string to = mGraph->vertices[from]->edge.top().first->element;
20              int cost  = mGraph->vertices[from]->edge.top().second;
21
22              int edgeplusnode = cost + mGraph->vertices[from]->dist;
23
24              if ( edgeplusnode <  mGraph->vertices[to]->dist) {
25                  mGraph->vertices[to]->dist=edgeplusnode;
26                  mGraph->vertices[to]->from=mGraph->vertices[from];
27              }
28              dijkstrasQueue.push(mGraph->vertices[from]->edge.top().first );
29              mGraph->vertices[from]->edge.pop();
30          }
31      }
32      auto route = path(mGraph->vertices[depTown], mGraph->vertices[arTown]);
33      return DijkResult(route.second,mGraph->vertices[arTown]->dist,0, route.first);
34  }
```

3 Examples and Benchmarks

3.1 Ten different from and to cities

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

3.2 Test from from-city to to-cities

Table 3.2 shows planning duration from one from-city to three differnent to-cities. And as you can see in the table, after the first planning the next planning durations are fairly low.

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 planning paths and cheapest price from different cities.

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

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_
```

```
8  #define VERTEX_H_
9
10 //Inherents from priority_queue and adds get_container which returns the underlying container
11 template <class Container>
12 class Adapter : public Container {
13 public:
14     typedef typename Container::container_type container_type;
15     container_type &get_container() { return this->c; }
16 };
17
18 class Vertex;
19
20 //Comp used to compare values in prority_queue
21 struct Comp {
22     bool operator()(const std::pair<Vertex*, int> &a ,const std::pair<Vertex*, int> &b ) const {
23         return b.second < a.second;
24     }
25 };
26
27 class Vertex {
28     typedef std::priority_queue<std::pair<Vertex*, int>, std::vector<std::pair<Vertex*, int> >, Comp> C
29     typedef Adapter<C> Container;
30 public:
31     Vertex(std::string value);
32     std::string element;
33     Container edge;
34     int dist;
35     Vertex* from;
36 };
37
38 #endif /* VERTEX_H_ */
```

B.2 Vertex.cpp

```
1  /*
2   * Vertex.cpp
3   *
4   *  Created on: Oct 26, 2014
5   *      Author: exchizz
6   */
7
8  #include "Vertex.h"
9  #include <limits>
10
11 Vertex::Vertex(std::string value) {
12     element = value;
13     dist = std::numeric_limits<int>::max();
14     from=NULL;  // Used in dijkstras
15 }
```

C Graph

C.1 Graph.h

```
1  #include <map>
2  #include <string>
3  #include <vector>
4  #include <iostream>
5  #include "Vertex.h"
6  #include "LateXGenerator.h"
7  #include <queue>
8  #ifndef GRAPH_H_
9  #define GRAPH_H_
10
11 class Graph {
12     typedef std::map <std::string, Vertex*> Vertices;
13     //typedef std::pair <Vertex, int> Edge;
14 public:
```

```
15      std::map <std::string, Vertex*> vertices;
16      void addVertex(std::string value);
17      void addEdge(std::string From, std::string To, int cost);
18      std::string printFromDot(std::string from);
19      void printFrom(std::string from);
20  };
21
22  #endif /* GRAPH_H_ */
```

C.2 Graph.cpp

```
1   #include "Graph.h"
2
3   void Graph::addVertex(std::string value) {
4       if(vertices.find(value) != vertices.end()){
5           throw new std::string("Element \"" + value + "\" already exists!");
6       }
7       vertices[value] = new Vertex(value);
8   }
9
10  void Graph::addEdge(std::string From, std::string To, int Cost) {
11      if(vertices.find(From) == vertices.end())
12          throw new std::string("From \"" + From + "\" does not exist!");
13
14      if(vertices.find(To) == vertices.end())
15          throw new std::string("To \"" + To + "\" does not exist!");
16
17      Vertex* from = vertices.find(From)->second;
18      Vertex* to = vertices.find(To)->second;
19
20      from->edge.push(std::make_pair(to,Cost));
21  }
22
23  std::string Graph::printFromDot(std::string from){
24      LateXGenerator lateXGenerator;
25      lateXGenerator.AddVertex(from);
26          for(auto it = vertices[from]->edge.get_container().begin() ; it != vertices[from]->edge.get_con
27              std::cout << "From \"" + from + "\" to: " << it->first->element << std::endl;
28              lateXGenerator.AddEdge(from,it->first->element, it->second);
29          }
30      return lateXGenerator.getOutput();
31  }
32
33  void Graph::printFrom(std::string from){
34      if(vertices.find(from) == vertices.end()){
35          std::cout <<  "City \"" + from +  "\" not found" << std::endl;
36          return ;
37      }
38      for(auto it = vertices[from]->edge.get_container().begin() ; it != vertices[from]->edge.get_contain
39          std::cout << "To: " << it->first->element << " Cost: " <<it->second  << std::endl;
40      }
41  }
```

D FileHandle

D.1 FileHandle.h

```
1   #ifndef FILEHANDLE_H_
2   #define FILEHANDLE_H_
3   #include <fstream>
4   #include <vector>
5   #include <iostream>
6   #include <sstream>
7   #include <algorithm>
8   #include "Vertex.h"
9   #include "Graph.h"
10  #include <memory>
11
```

```
12
13 class FileHandle {
14 public:
15     FileHandle(std::string filename);
16     void doParse(std::shared_ptr<Graph> &graph);
17     std::string rtrim(std::string s);
18     std::string ltrim(std::string s);
19     std::string getFrom(std::stringstream &stream);
20     void trim(std::string &);
21     std::string to, cost;
22     bool printException;
23 private:
24     std::string line;
25     std::ifstream fin;
26 };
27
28 #endif /* FILEHANDLE_H_ */
```

D.2 FileHandle.cpp

```
1  #include "FileHandle.h"
2
3  FileHandle::FileHandle(std::string filename) {
4      printException = false;
5      fin.open(filename);
6      if (!fin.good()){
7          std::cout << "Unable to open file";
8          exit(0);
9      }
10 }
11
12 void FileHandle::doParse(std::shared_ptr<Graph> &graph){
13     while(fin.peek() != -1){
14         // Ignore starting { in line
15         fin.seekg (1, std::ios::cur);
16
17         //Get next line
18         getline(fin, line);
19
20         std::stringstream lineStream(line);
21
22         std::string from = getFrom(lineStream);
23         //Add vertex, else catch exception
24         try {
25             graph->addVertex(from);
26         } catch (std::string *e){
27             if(printException){
28                 std::cout << "exception: " << *e << std::endl;
29             }
30         }
31
32         while(std::getline(lineStream,to,',') && std::getline(lineStream,cost,',')){
33             //Remove leading and trailing whitespaces.
34             trim(to);
35
36             //Convert to integer
37             int iCost;
38             std::istringstream ( cost ) >>  iCost;
39
40             //Add vertex if not existing, else catch exception
41             try {
42                 graph->addVertex(to);
43             } catch (std::string *e){
44                 if(printException){
45                     std::cout << "exception: " << *e << std::endl;
46                 }
47             }
48             //Add edge
49             graph->addEdge(from, to, iCost);
50         }
51     }
52 }
```

```
53 //Trim left side of string
54 std::string FileHandle::ltrim(std::string s){
55     s.erase(s.begin(),find_if_not(s.begin(),s.end(),[](int c){return isspace(c);}));
56     return s;
57 }
58 //Trim right side of string
59 std::string FileHandle::rtrim(std::string s){
60     s.erase(find_if_not(s.rbegin(),s.rend(),[](int c){return isspace(c);}).base(), s.end());
61     return s;
62 }
63 //Trim right and left
64 void FileHandle::trim(std::string &s){
65     s = ltrim(rtrim(s));
66 }
67 //Extracts "from", from the line
68 std::string FileHandle::getFrom(std::stringstream &stream){
69     std::string from;
70     std::getline(stream,from,',');
71     return from;
72 }
```

E Dijkstras

E.1 dijkstras.h

```
1  #ifndef __Navigation__dijkstras__
2  #define __Navigation__dijkstras__
3
4  #include <stdio.h>
5  #include <string>
6  #include <fstream>
7  #include <deque>
8  #include "Graph.h"
9  #include "Vertex.h"
10 #include "clock_timer.h"
11 #include <ctime>
12 #include <memory>
13 #include <limits>
14 struct Comp1 {
15     bool operator()(const Vertex* a ,const Vertex* b ) const {
16         return b->dist < a->dist;
17     }
18 };
19
20 class DijkResult{
21 public:
22     int Shifts;
23     int Ticket;
24     float Duration;
25     std::string Path;
26
27     DijkResult(int shifts, int ticket, float duration, std::string path){
28         this->Shifts = shifts;
29         this->Ticket = ticket;
30         this->Duration = duration;
31         this->Path = path;
32     }
33 };
34 class Dijkstras{
35     typedef std::priority_queue<Vertex*, std::vector<Vertex* >, Comp1> diQueue;
36 public:
37     Dijkstras(std::shared_ptr<Graph> graph);
38     DijkResult Run(std::string from, std::string to);
39     std::pair<std::string, int> path(Vertex*, Vertex*);
40     diQueue dijkstrasQueue;
41 private:
42     std::shared_ptr<Graph> mGraph;
43 };
44
45 #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_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.
//
```

```
7
8  /*
9   remember to include header:
10  #include "clock_timer.h"
11
12  Useage:
13
14  clock_timer timerrecord;
15  timerrecord.start_timer();
16  timerrecord.stop_timer();
17  */
18
19 #ifndef __timer_clock__clock_timer__
20 #define __timer_clock__clock_timer__
21 #include <ctime>
22 #include <iostream>
23 class clock_timer{
24 public:
25     void start_timer();
26     void stop_timer();
27     std::clock_t time;
28     std::clock_t start;
29     double duration;
30 };
31
32 #endif /* defined(__timer_clock__clock_timer__) */
```

F.2 clock_timer.cpp

```
1  //
2  //  clock_timer.cpp
3  //  timer_clock
4  //
5  //  Created by Anders Launer Baek on 12/09/14.
6  //  Copyright (c) 2014 Anders Launer Baek. All rights reserved.
7  //
8
9  #include "clock_timer.h"
10
11
12 void clock_timer::start_timer(){
13     start = std::clock();
14 }
15
16 void clock_timer::stop_timer(){
17     duration=( std::clock() - start ) / (double) CLOCKS_PER_SEC*1000;
18     //std::cout<< "Time: "<<time<<"[ms]"<< std::endl;
19 }
```