COMP.CS.300 Data structures and algorithms 1, autumn 2022

# Programming assignment 2: Railroads

Last modified on 11/21/2022

## Changelog

Below is a list of substantial changes to this document after its initial publication:

- 11/17/2022: Added a note to next_stations_from saying that the returned stations can be in arbitrary order.

## Contents

## Topic of the assignment

In the second programming assignment the first programming assignment will be extended to also include whole train connections and route searches related to them. Some operations in the assignment are compulsory, others are not A compulsory operation is one that the student must do to pass assignment 2. Non-compulsory operations will still affect the student's grade..

This assignment 2 document only describes new assignment 2 features. The idea is to copy the assignment 1 implementation as the starting point of assignment 2, and continue from there. All assignment 1 features and commands of the main program are also available in assignment 2, even if they are not repeated in this document. However, operations graded in assignment 1 are not graded again in assignment 2, and they don't affect the grade from assignment.

## Terminology

The most important new terms:

- **Train.** (Describes the path of a single train through stations) Every train has a unique *string id*, and a list of stations through which the train runs, as well as departure times from each station (and the arrival time to the final station).
- **Route.** A route is a sequence of stations from the origin station to the goal station using train connections. The *length* of a route is calculated (to minimize rounding errors) in the following way: the distance from a station to the next station ( $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ ) **is *first* rounded down to an integer**, and then these added together.
- **Cycle.** A route has a cycle, if the route arrives again to a station through which it has already passed.

In addition to the specifications given in programming assignment 1, the following must be taken into account:

- *In this assignment you cannot necessarily have much choice in the asymptotic performance of the new operations, because that's dictated by the typical algorithms. For this reason the implementation of the algorithms and correct behaviour are a more important grading criteria than asymptotic performance alone.*
- Implementing operations `route_least_stations`, `route_shortest_distance`, `route_with_cycle`, and `route_earliest_arrival` are not compulsory to pass assignment 2. **If you only implement the compulsory parts, the maximum grade for assignment 2 is 2.**
- If the implementation is bad enough, the assignment can be rejected.

## Structure and functionality of the program.

Part of the program code is provided by the course, part has to be implemented by students.

### Parts provided by the course

The following files are provided to the students: *mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui.* You are **NOT ALLOWED TO MAKE ANY CHANGES TO THESE FILES.**

File *datastructures.hh*

- `class Datastructures`: The implementation of this class is where students write their code. The public interface of the class is provided by the course. You are **NOT ALLOWED TO CHANGE THE PUBLIC INTERFACE.** That is, you are now allowed to change the names, return types or parameters of the given public member functions, etc. Of course you are still allowed to add new methods and data members to the private side.

- Type definition `TrainID` (consists of characters A-Z, a-z, 0-9, and dash -), which used as a unique identifier for each train.
- Constant `NO_TRAIN`, which is used as a return value in some operations, if the operation fails.

File *datastructures.cc*

- Here you write the code for the your operations.

## On using the graphical user interface

When compiled with QtCreator, a graphical user interface is provided. It allows running operations and test scripts, and visualizing the data.

***Note!*** The graphical representation gets all its information from student code! **It's not a depiction of what the "right" result is, but what output a student's code provides.** The UI uses operation `all_stations`() to get a list of stations, and asks their information with `get_...`() operations. If the drawing feature of regions is on, they are obtained with operation `all_regions`(), and the coordinates of each region with `get_region_coords`(). If the drawing feature of trains is on, they are obtained by calling operation `next_stations_from`() for each station.

## Commands recognized by the program and the public interface of the Datastructures class

When the program is run, it waits for the user to input one or more of the commands given in the table below. The commands, whose explanation mentions a member function, call the respective member function of the Datastructure class (implemented by students). Some commands are completely implemented by the code provided by the course.

If the program is given a file as a command line parameter, the program executes commands from that file and then quits.

The operations below are listed in the order in which we recommend them to be implemented (of course you should first design the class taking into account all operations).

| `Command`<br>`Public member function`<br>`(In commands optional parameters are in []-brackets and alternatives separated by \|)` | **Explanation** |
|---|---|
| (All programming assignment 1 commands and operations are also available.) | (And they do the same thing as in programming assignment 1.) |
| `clear_trains`<br>`void clear_trains()` | Clears out all trains, but doesn’t touch stations or regions. *This operation is not included in the performance tests.* |
| *`add_train ID Station1:Time1 CStation2:Time2... StationN:TimeN`*<br>*`bool add_train(TrainID trainid, std::vector<std::pair<StationID, Time> stationtimes)`* | Adds a train to the data structure with given unique id. The train runs through the given stations and departs from them at given times. The time of the last station is the arrival time to the final station (only needed for the route_earliest_arrival command). The departure times are also added to the station's info so that the assignment 1 operation `station_departures_after` shows them. If there already is a train with the given id or some station id is not found, nothing is done and `false` is returned, otherwise `true` is returned. |
| `next_stations_from` *`ID`*<br>`std::vector<StationID> next_stations_from(StationID id)` | Returns the stations that are immediate next stations on trains running through the given station. If no trains leave from the station, an empty vector is returned. If a station with given id doesn't exist, a vector with single element NO_STATION is returned. The main program sorts the result based on the id, so the stations can be returned in arbitrary order. (Main program calls this in various places.) |
| *`train_stations_from StationID TrainID`*<br>`std::vector<StationID> train_stations_from(StationID stationid, TrainID trainid)` | Returns a list of stations, through which the given train runs after leaving the given station. If there is no station or train with the given id, or the train does not leave from the given station, a vector with single element NO_STATION is returned. |
| **(The operations below should probably be implemented only after the ones above have been implemented.)** | |

| **Command** <br> **Public member function** <br> (In commands optional parameters are in []-brackets and alternatives separated by \|) | **Explanation** |
|---|---|
| `route_any StationID1 StationID2` <br> `std::vector<std::pair<StationID, Distance>> route_any(StationID fromid, StationID toid)` | Returns any (arbitrary) route between the given stations. The vector returned contains pairs StationID & Distance. The first pair contains the departure station and a distance of 0. Subsequent pairs come in order along the route, where the distance is always the total distance from the departure station. The final pair contains the destination station and the total distance of the trip. If no route can be found between the stations, an empty vector is returned. If either of the station ids is not found, a vector with one element {NO_STATION, NO_DISTANCE} is returned. |
| **(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)** | |
| `route_least_stations StationID1 StationID2` <br> `std::vector<std::pair<StationID, Distance>> route_least_stations(StationID fromid, StationID toid)` | Returns a route between the given stations so that it contains as few stations as possible. If several routes exist with equally few stations, any of them can be returned. The vector returned contains pairs StationID & Distance, whose contents are described in operation route_any. If no route can be found between the stations, an empty vector is returned. If either of the station ids is not found, a vector with one element {NO_STATION, NO_DISTANCE} is returned. |
| `route_with_cycle StationID` <br> `std::vector<StationID> route_with_cycle(StationID fromid)` | Returns a route starting from the given station such that the route has a single cycle, i.e. the route returns to a station it has already passed through. If several routes with a cycle exist, any of them can be returned. The returned vector first has the starting station, and then the rest of the stations along the route. The vector's final element should be the repeated station which causes the cycle. If no cyclic route can be found, an empty vector is returned. If the station id is not found, a vector with one element NO_STATION is returned. |

| **Command**<br>**Public member function**<br>(In commands optional parameters are in []-brackets and alternatives separated by \|) | **Explanation** |
|---|---|
| ***route_shortest_distance StationID1 StationID2***<br>***std::vector<std::pair<StationID, Distance>>***<br>***route_shortest_distance(StationID fromid, StationID toid)*** | Returns a route between the given stations so that its length is as small as possible. If several equally short routes exist, any of them can be returned. The vector returned contains pairs StationID & Distance, whose contents are described in operation route_any. If no route can be found between the stations, an empty vector is returned. If either of the station ids is not found, a vector with one element {NO_STATION, NO_DISTANCE} is returned. |
| ***route_earliest_arrival StationID1 StationID2 StartTime***<br>***std::vector<std::pair<StationID, Time>>***<br>***route_earliest_arrival(StationID fromid, StationID toid, Time starttime)*** | **Note! This is the most challenging non-compulsory operation!** *(Tip: try to come up with a suitable cost function.)* Returns a route between the given stations that arrives to the destination as early as possible. If several routes with the same arrival time exist, any of them can be returned. The vector returned contains pairs StationID & Time. The first pair contains the departure station and the departure time from it. Subsequent pairs contain the stations along the route, in order and the departure times from each station. The final pair contains the destination station and the train's arrival time to it. If no route can be found between the stations, an empty vector is returned. If either of the station ids is not found, a vector with one element {NO_STATION, NO_TIME} is returned. *Note! The operation doesn't have to find routes where the day changes during the route.* |
| **(The following operations are already implemented by the main program.)** | **(Here only changes to phase 1 are mentioned.)** |
| **random_trains *n***<br>(implemented by main program) | Add *n* new trains running between random stations. Note! The values really are random, so they can be different for each run, and they don't in any way form a sensible "map". |

| **Command**<br>**Public member function**<br>(In commands optional parameters are in []-brackets and alternatives separated by \|) | **Explanation** |
|---|---|
| **perftest all\|compulsory\|*cmd1*[;*cmd2*...] *timeout repeat n1*[;*n2*...]**<br>(implemented by main program) | Run performance tests. Clears out the data structure and add *n1* random stations, regions, and trains. Then a random command is performed *repeat* times. The time for adding elements and running commands is measured and printed out. Then the same is repeated for *n2* elements, etc. If any test round takes more than *timeout* seconds, the test is interrupted (this is not necessarily a failure, just arbitrary time limit). If the first parameter of the command is *all*, commands are selected from all commands. If it is *compulsory*, random commands are selected only from operations that have to be implemented. If the parameter is a list of commands, commands are selected from that list (in this case it's a good idea to include also random_add so that elements are also added during the test loop). If the program is run with a graphical user interface, "stop test" button can be used to interrupt the performance test (it may take a while for the program to react to the button). |

## "Data files"

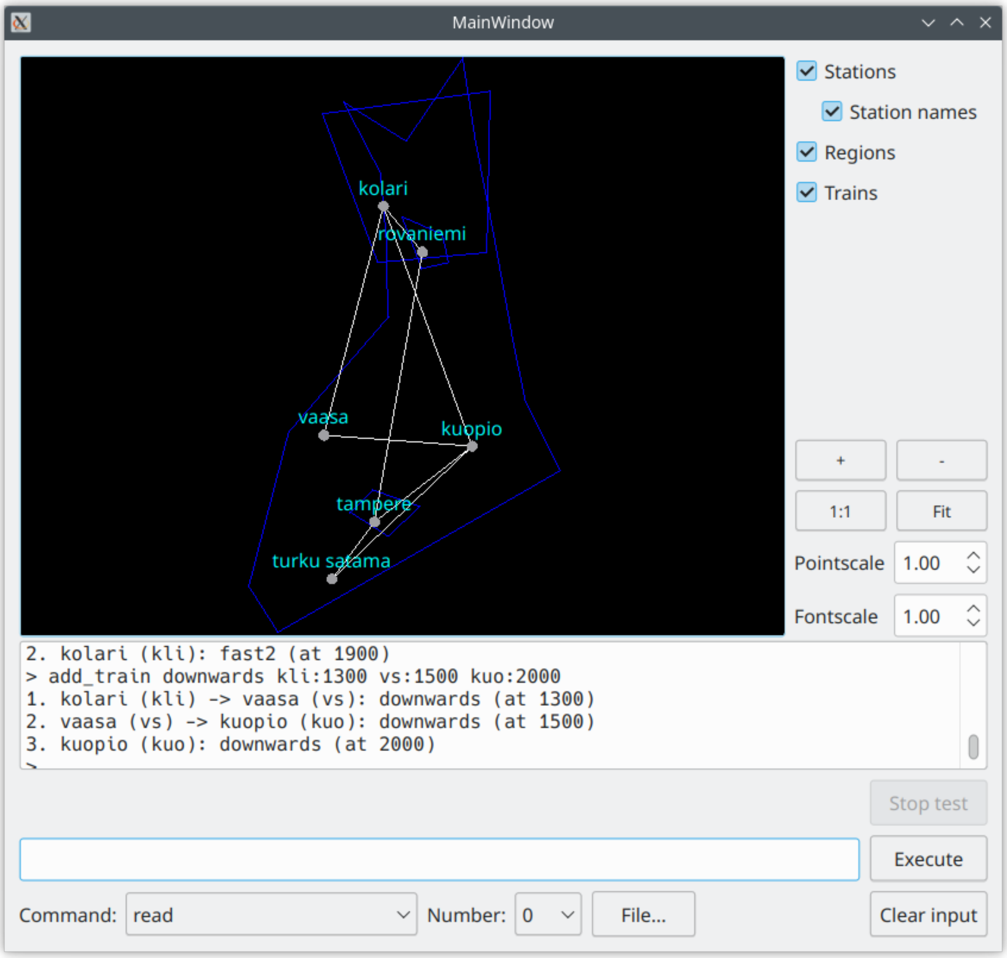
The easiest way to test the program is to create "data files", which can add a bunch of places, areas, and trains. Those files can then be read in using the "read" command, after which other commands can be tested without having to enter the information every time by hand.

Below is an examples of a data file, which adds trains to the application:

- *example-trains.txt*

```
# Some imaginary example trains
add_train tukutuku tus:0800 tpe:0900 kuo:1000
add_train upwards tpe:0930 roi:1600 kli:2000
add_train fast1 tus:1000 kuo:1200
add_train fast2 kuo:1100 kli:1900
add_train downwards kli:1300 vs:1500 kuo:2000
```

## Screenshot of user interface



```
2. kolari (kli): fast2 (at 1900)
> add_train downwards kli:1300 vs:1500 kuo:2000
1. kolari (kli) -> vaasa (vs): downwards (at 1300)
2. vaasa (vs) -> kuopio (kuo): downwards (at 1500)
3. kuopio (kuo): downwards (at 2000)
>
```

Above is a screenshot of the graphical user interface after *example-stations.txt*, *example-regions.txt*, and *example-trains.txt* have been read in.

## Example run

Below are example outputs from the program. The example's commands can be found in files *example-compulsory-in.txt* and *example-all-in.txt*, and the outputs in files *example-compulsory-out.txt* and *example-all-out.txt*. If you wish to perform a small test on all the compulsory commands, you can do this by running the following:

```
testread "example-compulsory-in.txt" "example-compulsory-out.txt"
```

### example-compulsory

```
> clear_all
Cleared all stations
> clear_trains
All trains removed.
> all_stations
```

```
No stations!
> read "example-stations.txt" silent
** Commands from 'example-stations.txt'
...(output discarded in silent mode)...
** End of commands from 'example-stations.txt'
> read "example-trains.txt"
** Commands from 'example-trains.txt'
> # Some imaginary example trains
> add_train tukutuku tus:0800 tpe:0900 kuo:1000
1. turku satama (tus) -> tampere (tpe): tukutuku (at 0800)
2. tampere (tpe) -> kuopio (kuo): tukutuku (at 0900)
3. kuopio (kuo): tukutuku (at 1000)
> add_train upwards tpe:0930 roi:1600 kli:2000
1. tampere (tpe) -> rovaniemi (roi): upwards (at 0930)
2. rovaniemi (roi) -> kolari (kli): upwards (at 1600)
3. kolari (kli): upwards (at 2000)
> add_train fast1 tus:1000 kuo:1200
1. turku satama (tus) -> kuopio (kuo): fast1 (at 1000)
2. kuopio (kuo): fast1 (at 1200)
> add_train fast2 kuo:1100 kli:1900
1. kuopio (kuo) -> kolari (kli): fast2 (at 1100)
2. kolari (kli): fast2 (at 1900)
> add_train downwards kli:1300 vs:1500 kuo:2000
1. kolari (kli) -> vaasa (vs): downwards (at 1300)
2. vaasa (vs) -> kuopio (kuo): downwards (at 1500)
3. kuopio (kuo): downwards (at 2000)
>
** End of commands from 'example-trains.txt'
> next_stations_from tpe
1. tampere (tpe) -> kuopio (kuo)
2. tampere (tpe) -> rovaniemi (roi)
> train_stations_from tpe upwards
1. tampere (tpe) -> rovaniemi (roi)
2. rovaniemi (roi) -> kolari (kli)
> route_any tus roi
1. turku satama (tus) -> tampere (tpe) (distance 0)
2. tampere (tpe) -> rovaniemi (roi) (distance 294)
3. rovaniemi (roi) (distance 1425)
```

## example-all

```
> # First read in compulsory example
> read "example-compulsory-in.txt"
** Commands from 'example-compulsory-in.txt'
...
** End of commands from 'example-compulsory-in.txt'
> route_least_stations tus kli
1. turku satama (tus) -> kuopio (kuo) (distance 0)
2. kuopio (kuo) -> kolari (kli) (distance 797)
3. kolari (kli) (distance 1853)
> route_with_cycle kuo
1. kuopio (kuo) -> kolari (kli)
2. kolari (kli) -> vaasa (vs)
3. vaasa (vs) -> kuopio (kuo)
```

```
4. kuopio (kuo)
> route_shortest_distance tus kli
1. turku satama (tus) -> tampere (tpe) (distance 0)
2. tampere (tpe) -> rovaniemi (roi) (distance 294)
3. rovaniemi (roi) -> kolari (kli) (distance 1425)
4. kolari (kli) (distance 1673)
> route_earliest_arrival tus kli 0700
1. turku satama (tus) -> tampere (tpe) (at 0800)
2. tampere (tpe) -> kuopio (kuo) (at 0900)
3. kuopio (kuo) -> kolari (kli) (at 1100)
4. kolari (kli) (at 1900)
```