COMP.CS.300 Data structures and algorithms 1, spring 2021

Programming assignment 2: Retkikartta (hiking map)

Last modified on 05/05/2021

Changelog

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

- 6.4. Changed ways_from() so that it returns an empty vector, if there are no ways from the given coordinate (i.e., it is not a crossroad).
- 6.4. Added the missing description of remove_way operation.
- 6.4. Added a note to route_with_cycle that to ensure unambiguous printout, if needed the main program reverses the cycle so that it starts with the smaller wayid. The return value from route_with_cycle() no longer has distances.
- 26.4. Added command random_ways (implemented by the main program).
- 27.4. The example run now uses clear_ways in the beginning instead of clear_all.
- 3.5. Added a note that route_any functionality doesn't have to produce information about the IDs of taken ways.
- 4.5. Updated the example output

Contents

Topic of the assignment

In the second phase the program of phase 1 will be extended to also include hiking ways (paths, tracks, etc.) and hiking route searches. Some operations in the assignment are compulsory, others are not (compulsory = required to pass the assignment, not compulsory = can pass without it, but it is still part of grading).

This phase 2 document only describes new phase 2 features. The idea is to copy the phase 1 implementation as the starting point of phase 2, and continue from there. All features and commands of the main program are also available in phase 2, even though they are not repeated in this document.

Terminology

Below is explanation for the most important terms in phase 2:

- **Way.** (Describes path, tracks, roads, etc, which you can use for hiking.) Every way has a unique *string id* (which consists of characters A-Z, a-z, and 0-9), and a list of coordinates that describe how the way progresses on the map. A way can be hiked in either direction. The *length* of a way is calculated (to minimize rounding errors) in the following way: the distance from a way coordinate to the next coordinate ($\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$) **is *first* rounded down to an integer**, and then these added together.
- **Crossroad.** A hiker can only move from one way to another in crossroads, and in this assignment only the endpoints of ways count as crossroads. In other words, even if two ways intersected “in the middle”, you cannot move from one way to the other in that point (this is to make the assignment easier).
- **Route.** A route is a series of ways, which are “connected” so that the next way starts from the crossroad where the previous one ends. The *length* of a route is the sum of the lengths of the ways it contains.
- **Cycle.** A route has a cycle, if while hiking the route you arrive again to a crossroad through which the route has already passed.

In the assignment one goal is to practice how to efficiently use ready-made data structures and algorithms (STL), but it also involves writing one’s own efficient algorithms and estimating their performance (of course it’s a good idea to favour STL’s ready-made algorithms/data structures when their can be justified by performance). In other words, in grading the assignment, asymptotic performance is taken into account, and also the real-life performance (= sensible and efficient implementation aspects). “Micro optimizations” (like do I write ”a = a+b;” or ”a += b;”, or how do I tweak compiler’s optimization flags) do not give extra points.

The goal is to code an as efficient as possible implementation, under the assumption that all operations are executed equally often (unless specified otherwise on the command table). In many cases you'll probably have to make compromises about the performance. In those cases it helps grading when you document your choices and their justification in the **document file** that is submitted as part of the assignment. (Remember to write the document in addition to the actual code!)

Especially note the following (all phase 1 notes also apply):

- *In this assignment you cannot necessarily have much choice in the asymptotic performance of the new operations, because that's dictated by the algorithms. For this reason the implementation of the algorithms and correct behaviour are a more important grading criteria than asymptotic performance alone.*
- **As part of the assignment, file datastructures.hh contains a comment next to each operation. Into that comment you should write your estimate of the asymptotic performance of the operation, and a short rationale for you estimate.**
- **As part of the assignment submission, a document should be added to git (in the same directory/folder as the code). This document should contain reasons for choosing the data structures used in the assignment (especially performance reasons). Acceptable formats are plain text (readme.txt), markdown (readme.md), and Pdf (readme.pdf).**
- Implementing operations `remove_way`, `route_least_crossroads`, `route_shortest_distance`, `route_with_cycle`, and `trim_ways` are not compulsory to pass the assignment. **If you only implement the compulsory parts, the maximum grade for the assignment 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

Files *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** (change names, return type or parameters of the given public member functions, etc., of course you are allowed to add new methods and data members to the private side).

- Type definition `Coord`, which is used in the public interface to represent (x,y) coordinates. As an example, some comparison operations (==, !=, <) and a hash function have been implemented for this type.
- Type definition `WayID`, which used as a unique identifier for each way.
- Type definition `DISTANCE`, which is used for distances (= lengths).
- Constants `NO_WAY`, `NO_COORD`, and `NO_DISTANCE`, which are used as return values, if information is requested for a thing that doesn't exist.

File *datastructures.cc*

- Here you write the code for the your operations.
- Function `random_in_range`: Like in the first assignment, returns a random value in given range (start and end of the range are included in the range). You can use this function if your implementation requires random numbers.

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.

The UI has a command line interface, which accepts commands described later in this document. In addition to this, the UI shows graphically created places, areas and ways (*if you have implemented necessary operations, see below*). The graphical view can be scrolled and zoomed. Clicking on a place name (or area border, which requires precision) prints out its information, and also inserts the ID on the command line (a handy way to give commands ID parameters). Clicking on a crossroad prints out its coordinates and copies them on the command line. The user interface has selections for what to show graphically.

Note! The graphical representation gets all its information from student code! **It's not a depiction of what the "right" result is, but what information students' code gives out.** The UI uses operation `all_places`() to get a list of places, and asks their information with `get_...`() operations. If drawing of areas is on, they are obtained with operation `all_areas`(), and the coordinates of each area with `get_area_coords`(). If drawing of ways is on, they are obtained with operation `all_ways`() and `get_way_coords()`, and those results are also used for drawing crossroads.

Parts of the program to be implemented as the assignment

Files *datastructure.hpp* and *datastructure.cpp*

- `class Datastructures`: The given public member functions of the class have to be implemented. You can add your own stuff into the class (new data members, new member functions, etc.)

- **In file *datastructures.hh*, for each member function you implement, write an estimation of the asymptotic performance of the operation (with a short rationale for your estimate) as a comment above the member function declaration.**

Additionally the readme.pdf mentioned before is written as a part of the assignment.

Note! The code implemented by students does not print out any output related to the expected functionality, the main program does that. If you want to do debug-output while testing, use the `cerr` stream instead of `cout` (or `qDebug`, if you you use Qt), so that debug output does not interfere with the tests.

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

When the program is run, it waits for commands explained 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 **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
(All phase 1 commands and operations are also available.)	(And they do the same thing as in phase 1.)
`clear_ways` `void clear_ways()`	Clears out all ways (and thus also crossroads), but doesn't touch places or areas. *This operation is not included in the default performance tests.*
`all_ways` `std::vector<WayID> all_ways()`	Returns a list (vector) of the ways in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*
`add_way ID Coord1 Coord2...` `bool add_way(WayID id, std::vector<Coord> coords)`	Adds a way to the data structure with given unique id and coordinates. If there already is a way with the given id, nothing is done and `false` is returned, otherwise `true` is returned.

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**

`way_coords ID`
`std::vector<Coord>`
`get_way_coords(WayID id)`
	Returns the coordinate vector of the way with given ID, or a vector with single item NO_COORD, if such way doesn't exist. (Main program calls this in various places.)

`ways_from Coord` `std::vector<std::pair<WayID, Coord>> ways_from(Coord xy)`	Returns a list of ways starting from the given coordinate. The second element in the pair is the crossroad to which each way leads to. If there is no crossroad in that coordinate, an empty vector is returned.
(The operations below should probably be implemented only after the ones above have been implemented.)	
`route_any Coord Coord` `std::vector<std::tuple<Coord, WayID, Distance>> route_any(Coord fromxy, Coord toxy)`	Returns any (arbitrary) route between the given crossroads. The returned vector first has the starting point and starting distance 0 (**the value of WayID doesn't matter in this operation, the main program ignores it**). Then come the rest of the crossroads along the route and the total distance up to each crossroad. Finally comes the destination crossroad with NO_WAY as way, and the total distance. If no route can be found between the crossroads, an empty vector is returned. If either of the coordinates is not a crossroad (no way leads to it), one element {NO_COORD, NO_WAY, NO_DISTANCE} is returned.
(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	
`remove_way ID` `bool remove_way(WayID id)`	Removes a way with the given id. If a way with given id does not exist, does nothing and returns `false`, otherwise returns `true`. (If there are no more ways leading to the removed way's endpoints, they are of course no longer counted as crossroads.)

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**

`route_least_crossroads Coord Coord`
`std::vector<std::tuple<Coord, WayID, Distance>>`
`route_least_crossroads(Coord fromxy, Coord toxy)`
	Returns a route between the given crossroads so that it contains the minimum number of crossroads. If several routes exist with as few crossroads, any of them can be returned. The returned vector first has the starting point, what way is taken from there, and starting distance 0. Then come the rest of the crossroads along the route, what way is used to continue the route, and the total distance up to each crossroad. Finally comes the destination crossroad with NO_WAY as way, and the total distance. If no route can be found between the crossroads, an empty vector is returned. If either of the coordinates is not a crossroad (no way leads to it), one element {NO_COORD, NO_WAY, NO_DISTANCE} is returned.

`route_with_cycle Coord` `std::vector<std::tuple<Coord, WayID>> route_with_cycle(Coord fromxy)`	Returns a route starting from the given crossroad so that has a cycle, i.e. the route returns to a crossroad already passed on the route. The returned vector first has the starting point and what way is taken from there. Then come the rest of the crossroads along the route and what way is used to continue the route. Finally comes the crossroad reached again (causing the cycle) with NO_WAY as way. If no cyclic route can be found from the point, an empty vector is returned. If the coordinate is not a crossroad (no way leads to it), one element {NO_COORD, NO_WAY} is returned. Note 1! A route that immediately travels back the same way it used to arrive to a node doesn't count as a cycle. Note 2! To keep the printout umambiguous, the *main program* reverses the cycle (if necessary) so that it begins with the smaller way id.

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**

`route_shortest_distance Coord Coord`
`std::vector<std::tuple<Coord, WayID, Distance>>`
`route_shortest_distance(Coord fromxy, Coord toxy)`
	Returns a route between the given crossroads so that its length is as small as possible. If several equally short routes exist, any of them can be returned. The returned vector first has the starting point, what way is taken from there, and starting distance 0. Then come the rest of the crossroads along the route, what way is used to continue the route, and the total distance up to each crossroad. Finally comes the destination crossroad with NO_WAY as way, and the total distance. If no route can be found between the crossroads, an empty vector is returned. If either of the coordinates is not a crossroad (no way leads to it), one element {NO_COORD, NO_WAY, NO_DISTANCE} is returned.

`trim_ways` `Distance trim_ways()`	Trims the ways (= removes them) so that the total length of the remaining ways is as small as possible, but along these ways a route can still be found between any crossroads that originally had a route between them. The rest of the ways are removed. The return value is the total distance of the remaining way network. If there are several possible way networks with the same total distance, any one of them may be returned.
(The following operations are already implemented by the main program.)	**(Here only changes to phase 1 are mentioned.)**
random_ways *n* (implemented by main program)	Add *n* new ways to random coordinates (for testing). 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 **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
perftest all|compulsory|*cmd1*[;*cmd2*...] ***timeout repeat n1*[;*n2*...]** (implemented by main program)	!!! Run performance tests. Clears out the data structure and add *n1* random places, areas, and ways. 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 ways. 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.

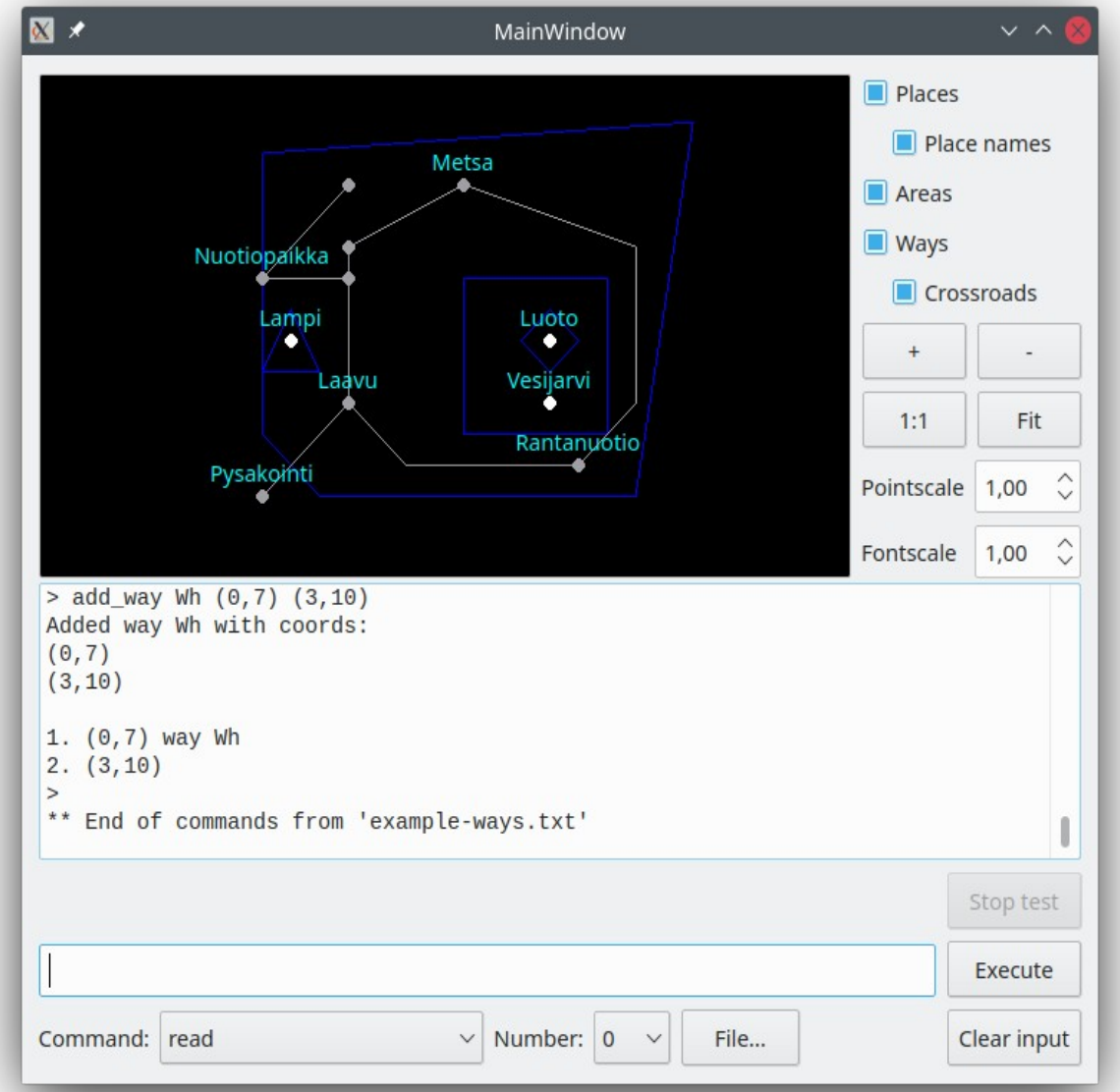
Below is an examples of a data file, which adds ways to the application:

- *example-ways.txt*

```
# Ways
add_way Wa (0,0) (3,3)
add_way Wb (3,3) (5,1) (11,1)
add_way Wc (3,3) (3,7)
add_way Wd (0,7) (3,7)
add_way We (7,10) (3,8)
add_way Wf (3,7) (3,8)
add_way Wg (11,1) (13,3) (13,8) (7,10)
add_way Wh (0,7) (3,10)
```

Screenshot of user interface

Below is a screenshot of the graphical user interface after *example-places.txt*, *example-areas.txt*, and *example-ways.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*. I.e., you can use the example as a small test of compulsory behaviour by running command
`testread "example-compulsory-in.txt" "example-compulsory-out.txt"`

```
> clear_ways
All routes removed.
> all_ways
No ways!
> read "example-places.txt" silent
** Commands from 'example-places.txt'
...(output discarded in silent mode)...
** End of commands from 'example-places.txt'
> read "example-areas.txt" silent
** Commands from 'example-areas.txt'
...(output discarded in silent mode)...
** End of commands from 'example-areas.txt'
> read "example-ways.txt"
** Commands from 'example-ways.txt'
> # Ways
> add_way Wa (0,0) (3,3)
Added way Wa with coords: (0,0) (3,3)
1. (0,0) way Wa
2. (3,3)
> add_way Wb (3,3) (5,1) (11,1)
Added way Wb with coords: (3,3) (5,1) (11,1)
1. (3,3) way Wb
2. (11,1)
> add_way Wc (3,3) (3,7)
Added way Wc with coords: (3,3) (3,7)
1. (3,3) way Wc
2. (3,7)
> add_way Wd (0,7) (3,7)
Added way Wd with coords: (0,7) (3,7)
1. (0,7) way Wd
2. (3,7)
> add_way We (7,10) (3,8)
Added way We with coords: (7,10) (3,8)
1. (7,10) way We
2. (3,8)
> add_way Wf (3,7) (3,8)
Added way Wf with coords: (3,7) (3,8)
1. (3,7) way Wf
2. (3,8)
> add_way Wg (11,1) (13,3) (13,8) (7,10)
Added way Wg with coords: (11,1) (13,3) (13,8) (7,10)
1. (11,1) way Wg
2. (7,10)
> add_way Wh (0,7) (3,10)
Added way Wh with coords: (0,7) (3,10)
1. (0,7) way Wh
2. (3,10)
>
** End of commands from 'example-ways.txt'
> all_ways
1. Wa
2. Wb
3. Wc
4. Wd
```

```
5. We
6. Wf
7. Wg
8. Wh
> way_coords Wb
Way Way id Wb has coords:
(3,3)
(5,1)
(11,1)

> ways_from (3,3)
1. (0,0) way Wa
2. (11,1) way Wb
3. (3,7) way Wc
> route_any (3,7) (3,10)
1. (3,7) distance 0
2. (0,7) distance 3
3. (3,10) distance 7>
(… the remaining output is from example-all-out.txt)
> route_least_crossroads (0,0) (7,10)
1. (0,0) way Wa distance 0
2. (3,3) way Wb distance 4
3. (11,1) way Wg distance 12
4. (7,10) distance 25
> route_with_cycle (0,0)
1. (0,0) way Wa
2. (3,3) way Wb
3. (11,1) way Wg
4. (7,10) way We
5. (3,8) way Wf
6. (3,7) way Wc
7. (3,3)
> route_shortest_distance (0,0) (7,10)
1. (0,0) way Wa distance 0
2. (3,3) way Wc distance 4
3. (3,7) way Wf distance 8
4. (3,8) way We distance 9
5. (7,10) distance 13
> trim_ways
The remaining ways have a total length of 28
> all_ways
1. Wa
2. Wb
3. Wc
4. Wd
5. We
6. Wf
7. Wh
>
```