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

Programming assignment 2: Connecting Towns

Last modified on 11/26/2021

Changelog

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

- 26.11. Fixed the example output (all_roads was printing out incorrect road distances).

Contents

Topic of the assignment

In the second phase the program of phase 1 will be extended to also include road connections between towns, and performing 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.*

In practice, implementing phase 2 is probably easiest to do by copying the Datastructures class's private section (+ other possible additions) in datastructures.hh to phase 2 datastructures.hh. And similarly to copy the phase 1 implementations in datastructures.cc to phase 2 datastructures.cc (and remove empty default implementations there). Phase 1 performance comments in datastructures.hh do no have to be repeated, unless the asymptotic performance of some phase 1 operations changes in phase 2.

Terminology

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

- Road. A road always goes directly between two towns, and it can be travelled in either direction. The *length* of a road is the same is the distance between the towns, i.e. $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ **rounded down to an integer**.
- **Route.** A route is a series of roads, which are “connected” so that the next road starts from the town where the previous one ends. The *length* of a route is the sum of the lengths of the roads it contains. A route cannot directly return using the same road that was used to arrive to a town.
- **Cycle.** A route has a cycle, if while travelling the route you arrive again to a town 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_road`, `least_towns_route`, `shortest_route`, `road_cycle_route`, and `trim_road_network` 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.

On using the graphical user interface

Note! The graphical representation gets all its information from student's code! **It's not a depiction of what the "right" result is, but what information student's code gives out.** The UI uses the operation `all_towns`() to get a list of towns, and asks their information with the `get_...`() operations. If the option to draw vassal relationships is on, they are obtained with the operation `get_town_vassals`(). If drawing of roads is on, they are obtained with operation `get_roads_from`().

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_all` `void clear_all();`	Clears out the data structure, i.e. removes all towns **and roads** (after this `town_count` returns 0 **and all_roads returns an empty list**).
`clear_roads` `void clear_roads()`	Clears out all roads, but doesn't touch towns or vassalships. *This operation is not included in the default performance tests.*
`all_roads` `std::vector<std::pair<TownID, TownID>> all_roads()`	Returns a list (vector) of the ways in any (arbitrary) order (the main routine sorts them based on their ID). Each road is in the list only once and expressed as a pair of town ids, so that the smaller valued id is first in the pair. *This operation is not included in the default performance tests.*
`add_road` ***ID1 ID2*** `bool add_road(TownID town1, TownID town2)`	Adds a road between given towns. If either town doesn't exist, or a road already exists between towns, nothing is done and `false` is returned, otherwise `true`.
`roads_from` ***ID*** `std::vector<TownID> get_roads_from(TownID id)`	Returns in arbitrary order a list of towns to which a direct road goes from the given town. If there's no town with the given ID, a single element NO_TOWNID is returned.

Command **Public member function** (In commands optional parameters are in []-brackets and alternatives separated by |)	**Explanation**
(The operations below should probably be implemented only after the ones above have been implemented.)	
`any_route ID1 ID2` `std::vector<TownID> any_route(TownID fromid, TownID toid)`	Returns any (arbitrary) route between the given towns. The returned vector first has the starting town. Then come the rest of the towns along the route and finally the destination town. If no route can be found between the towns, an empty vector is returned. If either of the ids does not correspond to a town, one element {NO_TOWNID} is returned.
(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	
`remove_road ID1 ID2` `bool remove_road(TownID town1, TownID town2)`	Removes a road between given towns. If either town doesn't exist, or a road does not exists between towns, nothing is done and `false` is returned, otherwise `true`.
`least_towns_route ID1 ID2` `std::vector<TownID> least_towns_route(TownID fromid, TownID toid)`	Returns a route between the given crossroads so that it contains the minimum number of towns. If several routes exist with as few towns, any of them can be returned. The returned vector first has the starting town. Then come the rest of the towns along the route and finally the destination town. If no route can be found between the towns, an empty vector is returned. If either of the ids does not correspond to a town, one element {NO_TOWNID} is returned.
`road_cycle_route ID` `std::vector<TownID> road_cycle_route(TownID startid)`	Returns a route starting from the given town so that has a cycle, i.e. the route returns to a town already passed on the route. The returned vector first has the starting town. Then come the rest of the towns along the route and finally the town reached again (causing the cycle). If no cyclic route can be found from the point, an empty vector is returned. If the id does not correspond to a town, one element {NO_TOWNID} is returned. Note 1! A route that immediately travels back the same road it used to arrive to a town doesn't count as a cycle. Note 2! To keep the printout unambiguous, the *main program* reverses the cycle, if necessary, so that it begins with the smaller town id.

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

`shortest_route ID1 ID2`
`std::vector<TownID>`
`shortest_route(TownID fromid,`
`TownID toid)`
	Returns a route between the given towns 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 town. Then come the rest of the towns along the route, and finally the destination town. If no route can be found between the towns, an empty vector is returned. If either of the ids does not correspond to a town, one element {NO_TOWNID} is returned.

`trim_road_network` `Distance trim_road_network()`	Trims the roads (= removes them) so that the total length of the remaining roads is as small as possible, but along these roads a route can still be found between any towns that originally had a route between them. The rest of the roads are removed. The return value is the total distance of the remaining road network. If there are several possible road networks with the same total distance, any one of them may be returned. **Note! This is a challenging bonus operation, whose algorithm is not found on the course videos, although the algorithm is mentioned elsewhere in course materials.**
(The following operations are already implemented by the main program.)	**(Here only changes to phase 1 are mentioned.)**
random_roads n (implemented by main program)	Add at most n random roads between the towns so that the added towns do not cross each other (to keep the UI clear). It is possible that less than n roads is added. *In perftest this command always adds at most 10 roads.*
random_road_network (implemented by main program)	Adds a road network between the towns so that each town can be reached from any other town, and the roads do not intersect with each other.

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 adds *n1* random towns **and roads** (see random_add). Then a random command is performed *n* 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 an 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, the "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 towns and roads. 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-roadss.txt*

```
# Adding towns
add_town Hki Helsinki (3,0) 3
add_town Tpe Tampere (2,2) 4
add_town Ol Oulu (3,7) 10
add_town Kuo Kuopio (6,3) 9
add_town Tku Turku (1,1) 2
# Adding crossroads as extra towns
add_town x1 xx (3,3) 6
add_town x2 xy (4,4) 8
# Adding roads
add_road Tpe x1
# add_road x1 x2
add_road x2 Ol
add_road Ol Kuo
```

```
add_road Tpe Kuo
add_road Hki Tpe
add_road Tpe Tku
```

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_all
Cleared all towns
> clear_roads
All roads removed.
> read "example-data.txt"
** Commands from 'example-data.txt'
> # Adding towns
> add_town Hki Helsinki (3,0) 3
Helsinki: tax=3, pos=(3,0), id=Hki
> add_town Tpe Tampere (2,2) 4
Tampere: tax=4, pos=(2,2), id=Tpe
> add_town Ol Oulu (3,7) 10
Oulu: tax=10, pos=(3,7), id=Ol
> add_town Kuo Kuopio (6,3) 9
Kuopio: tax=9, pos=(6,3), id=Kuo
> add_town Tku Turku (1,1) 2
Turku: tax=2, pos=(1,1), id=Tku
> # Adding crossroads as extra towns
> add_town x1 xx (3,3) 6
xx: tax=6, pos=(3,3), id=x1
> add_town x2 xy (4,4) 8
xy: tax=8, pos=(4,4), id=x2
> # Adding roads
> add_road Tpe x1
Added road: Tampere <-> xx
> # add_road x1 x2
> add_road x2 Ol
Added road: xy <-> Oulu
> add_road Ol Kuo
Added road: Oulu <-> Kuopio
> add_road Tpe Kuo
Added road: Tampere <-> Kuopio
> add_road Hki Tpe
Added road: Helsinki <-> Tampere
> add_road Tpe Tku
Added road: Tampere <-> Turku
>
** End of commands from 'example-data.txt'
> all_roads
1: Hki <-> Tpe (2)
```

```
2: Kuo <-> Ol (5)
3: Kuo <-> Tpe (4)
4: Ol <-> x2 (3)
5: Tku <-> Tpe (1)
6: Tpe <-> x1 (1)
> roads_from Tpe
1. Helsinki: tax=3, pos=(3,0), id=Hki
2. Kuopio: tax=9, pos=(6,3), id=Kuo
3. Turku: tax=2, pos=(1,1), id=Tku
4. xx: tax=6, pos=(3,3), id=x1
> any_route Tku Hki
1. Turku
2. Tampere (distance 1)
3. Helsinki (distance 3)
> # Non-compulsory operations
> # First add a road to create more routes
> add_road x1 x2
Added road: xx <-> xy
> least_towns_route Hki Ol
1. Helsinki
2. Tampere (distance 2)
3. Kuopio (distance 6)
4. Oulu (distance 11)
> road_cycle_route Hki
1. Helsinki
2. Tampere (distance 2)
3. Kuopio (distance 6)
4. Oulu (distance 11)
5. xy (distance 14)
6. xx (distance 15)
7. Tampere (distance 16)
> shortest_route Hki Ol
1. Helsinki
2. Tampere (distance 2)
3. xx (distance 3)
4. xy (distance 4)
5. Oulu (distance 7)
> remove_road Tpe Hki
Removed road: Tampere <-> Helsinki
> roads_from Tpe
1. Kuopio: tax=9, pos=(6,3), id=Kuo
2. Turku: tax=2, pos=(1,1), id=Tku
3. xx: tax=6, pos=(3,3), id=x1
> all_roads
1: Kuo <-> Ol (5)
2: Kuo <-> Tpe (4)
3: Ol <-> x2 (3)
4: Tku <-> Tpe (1)
5: Tpe <-> x1 (1)
6: x1 <-> x2 (1)
> add_road Hki Tpe
Added road: Helsinki <-> Tampere
> trim_road_network
The remaining road network has total distance of 12
```

```
> all_roads
1: Hki <-> Tpe (2)
2: Kuo <-> Tpe (4)
3: Ol <-> x2 (3)
4: Tku <-> Tpe (1)
5: Tpe <-> x1 (1)
6: x1 <-> x2 (1)
```