TIE-20106 Data structures and algorithms, spring 2019

Programming assignment 2: Game of FIBRES

Last modified on 04/08/2019

Previously on Data structures and Algorithms...

This programming assignment extends Programming assignment 1 (Beacons of RGB). Everything written in the previous assignment description still applies, and this assignment description only mentions new things to be implemented in this assignment, or things that have changed since assignment 1.

Topic of the assignment

As time goes on, the civilization of Cmyk has progressed, and its engineers have realized that the constant fog in the atmosphere dims down light of the beacons, so the light can more effectively be transmitted from one beacon to the next using underground fibre optic cables. The fibres create a network with crossroads (called "xpoints"), and in each xpoint the incoming light can be transmitted into another fibre. If a beacon is located on an xpoint (or at the end of a single fibre), it can receive and transmit light using the fibres. In this assignment the class implemented in the first assignment is extended to handle the fibre network and find light routes in the network.

The idea of this assignment is still to practice the use of ready-made data structures and algorithms (STL), but it also practicing writing own algorithms and analyzing their performance.

Every fibre has two endpoints with given coordinates. The light can pass through the fibre in both directions. Additionally, every fibre slows down the light by a given amount (from physics courses we remember that the speed of light in different mediums differs), so each fibre is given a "cost" which tells how long the light takes to travel through the fibre. In the program you can add and remove fibres, ask for fibres starting from a given point, search routes for light with different criteria and as a non-compulsory part optimize the fibre network.

Since this is a Data structures and algorithms assignment, performance of the program is still a grading criteria. 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). Getting better performance helps to increase the grade. 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 performance alone.

Especially note the following (some of these are new, some are repeated because of their importance):

- **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, which 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 formast are plain test (readme.txt), markdown (readme.md), and Pdf (readme.pdf).**
- Implementing operations `route_least_xpoints()`, `route_fastest()`, `route_fibre_cycle()`, ja `trim_fibre_network()` is not compulsory to pass the assignment. However they affect the grade of the assignment. Some of the non-compulsory operations may require algorithms that are not discussed during the lectures (but possibly elsewhere on the course).
- If the implementation is bad enough, the assignment can be rejected.

On sorting

If the return value of an operation is a list of sorted coordinates, the sorting should be done using the "<" operation that has already been defined for the coordinate type.

On printing light routes

Many operations in this assignment return a route for a lightbeam to travel in the fibres. In the code this happens by returning a list of pairs, where the pair contains a coordinate on the route and the cost of the route up to that coordinate. The first item on the returned list is the starting point (with cost 0), then the next xpoint in the route, etc., and the destination points as the last (with the cost of the whole route).

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

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

New in this assignment is that the graphical representation is able to show the fibres, and the user can choose whether mouse click produces a beacon ID (like in assignment 1) or a coordinate of an xpoint (you can only click on xpoints for this).

Additionally the UI has selection "Beam routes". Selecting this make the UI try to route beacon lightbeams through the fibre network (using a compulsory operation `route_any()`. If no route is found using this operation, the lightbeam is drawn directly from beacon to beacon (i.e. through air). (If several ligthbeams are passing through the same fibre, only one of them is shown.)

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_beacons`() to get a list of beacons, and asks their information with `get_...`() operations. If drawing lightbeams is on, they are get with operation `get_lightsources`(), and if lightbeam coloring is on, the colors are get with operation `total_color`() (non-compulsory). Jos drawing fibres is on, hey are get with operation `all_xpoints`() and `get_fibres_from()`. (And as mentioned earlier, routing of lightbeams is done with `route_any()`).

Parts of the program to be implemented as the assignment

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

- `class Datastructure`: 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 should not do 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.

Below only new operations of this assignment are listed, also all operations from assignment 1 are available.

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 they to be implemented (of course you should first design the class taking into account all operations).

Command Public member function	Explanation
`all_xpoints` `std::vector<Coord> all_xpoints()`	Returns a list (vector) of all xpoints of the added fibres sorted according to their coordinates (see On sorting earlier). **Each xpoint is in the list only once.** *This operation is not included in the default performance tests.*
`add_fibre (x1,y1) (x2,y2) cost` `bool add_fibre(Coord xpoint1, Coord xpoint2, Cost cost)`	Adds a fibre to the data structure between the given coordinates and with the given cost. If there already is a fibre between given points or the points are the same point, nothing is done and `false` is returned, otherwise `true` is returned.
`fibres (x,y)` `std::vector<std::pair<Coord, Cost>> get_fibres_from(Coord xy)`	Returns a list of coordinates that have a fibre directly conneting them to the given point, and the cost of that fibre. The return value must be sorted in increasing order of coordinates. If no fibres are connected to the given point, an empty list is returned.
`all_fibres` `std::vector<std::pair<Coord, Coord>> all_fibres()`	Returns a list of all fibres. In the list every fibre is represented as a pair of coordinates (which are the fibre's end points). The list is sorted primarily based on the first coordinate of the pair, secondarily based on the second coordinate. Each fibre is in the list only once, so that its 1. coordinate is smaller than 2. coordinate.
`remove_fibre (x1,y1) (x2,y2)` `bool remove_fibre(Coord xpoint1, Coord xpoint2)`	Removes a fibre between the given coordinates. If no such fibre exists, does nothing and returns `false`, otherwise returns `true`.
`clear_fibres` `void clear_fibres()`	Clears out the data fibre network, i.e. removes all fibres. Note! All beacons and their lightbeams remain.
(The operations below should probably be implemented only after the ones above have been implemented.)	
`route_any (x1,y1) (x2,y2)` `std::vector<std::pair<Coord, Cost>> route_any(Coord fromxy, Coord toxy)`	Returns any (arbitrary) route along the fibres between the given points (see "On printing light routes"). The returned vector first has the starting point with cost 0, then the rest of the xpoints along the route and a cost up to that xpoint, and the ending point with the total cost as the last element. If no route can be found between the points, an empty vector is returned.
(Implementing the following operations is not compulsory, but they improve the grade of the assignment.)	

Command Public member function	Explanation
`route_least_xpoints (x1,y1) (x2,y2)` `std::vector<std::pair<Coord, Cost>> route_least_xpoints(Coord fromxy, Coord toxy)`	Returns a route along the fibres between the given points with the least amount of xpoints (see "On printing light routes"). The returned vector first has the starting point with cost 0, then the rest of the xpoints along the route and a cost up to that xpoint, and the ending point with the total cost as the last element. If no route can be found between the points, an empty vector is returned.
`route_fastest (x1,y1) (x2,y2)` `std::vector<std::pair<Coord, Cost>> route_fastest(Coord fromxy, Coord toxy)`	Returns the fastest route along the fibres between the given points (see "On printing light routes"), i.e. that route with the smallest cost. The returned vector first has the starting point with cost 0, then the rest of the xpoints along the route and a cost up to that xpoint, and the ending point with the total cost as the last element. If no route can be found between the points, an empty vector is returned.
`route_fibre_cycle (x1,y1)` `std::vector<Coord> route_fibre_cycle(Coord startxy)`	Checks whether it is possible to travel along the fibre network from the given point so that the route forms a cycle (route returns to an xpoint within the route along a different fibre than before). The return value is a route that ends with the cycle (the last xpoint in the return value is the xpoint that is reached twice). If a route is not found, an empty vector is returned. *Note, in order to make the output unambiguous, the main program prints out only the cycle, and so that the cycle is printed from the forking point in the direction of the smaller coordinate. The graphical UI shows the whole route.*
`trim_fibre_network` `Cost trim_fibre_network()`	Leaves fibres, whose combined cost is as small as possible, but along these fibres a route can still be found between points that originally had a route between them. The rest of the fibres are removed. The return value is the total cost of the remaining fibre network. If there are several possible fibre networks with the same cost, any one of them may be returned.
(The following operations are already implemented by the main program.)	
random_fibres n (implemented by main program)	Adds at most n random fibres between beacons so that the added fibres 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 fibres.*
random_labyrinth xsize ysize extra_routes (implemented by main program)	Adds a random fibre labyrinth, with ysize rows of xsize xpoints. Extra_route tells how many "extra" fibres are added to the labyrinth. If extra_routes is 0, every xpoint in the labyrinth has exactly one route to any other xpoint.

Command Public member function	Explanation
perftest all/compulsory/cmd1;cmd2... timeout n n1;n2;n3... (pääohjelman toteuttama)	Run performance tests. Clears out the data structure and add *n1* random beacons (see random_add) **add between them a random number of fibres**. 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* employees, 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). **Note! In assignment 2 perftest does not by default test sorting and lightbeam related operations of assignment 1, only adding beacons, often performed operations and operations related to fibres (while adding beacons lightbeams are still also added).**

"Data files"

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

Below are examples of a data files, one of which adds beacons, the other lightbeams:

- *example-beacons.txt (updated to match the file 8. April)*

```
# Add beacons
add_beacon G1 Lime (0,0) (0,255,0)
add_beacon M1 Fuchsia (6,0) (255,0,255)
add_beacon R1 Crimson (1,6) (220,20,60)
add_beacon B2 Teal (11, 10) (0,128,128)
add_beacon M2 Indigo (10, 4) (75,0,130)
```

- *example-lightbeams.txt*

```
# Add light sources
add_lightbeam G1 M1
add_lightbeam R1 M2
add_lightbeam M1 M2
add_lightbeam M2 B2
```

- *example-fibres.txt*

```
add_fibre (0,0) (6,0) 1
add_fibre (6,0) (10,4) 1
add_fibre (10,4) (6,6) 1
add_fibre (6,6) (11,10) 1
add_fibre (0,0) (1,6) 2
add_fibre (1,6) (6,6) 3
```

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"`*

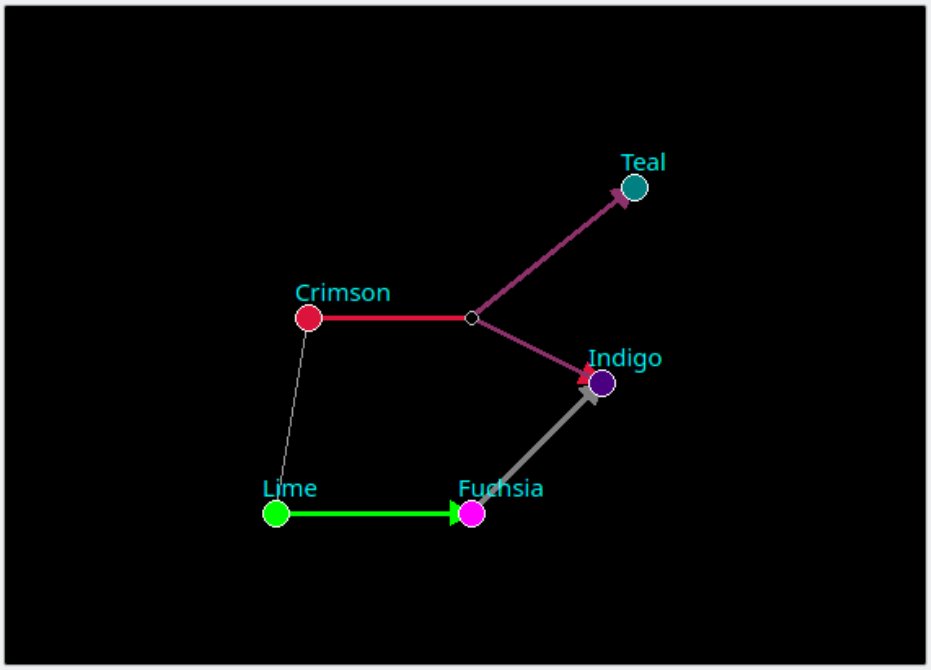
```
> read "example-compulsory-in.txt"
** Commands from 'example-compulsory-in.txt'
> clear_beacons
Cleared all beacons
> clear_fibres
All fibres removed.
> read "example-beacons.txt"
** Commands from 'example-beacons.txt'
> # Add beacons
> add_beacon G1 Lime (0,0) (0,255,0)
Lime: pos=(0,0), color=(0,255,0):1530, id=G1
> add_beacon M1 Fuchsia (6,0) (255,0,255)
Fuchsia: pos=(6,0), color=(255,0,255):1020, id=M1
> add_beacon R1 Crimson (1,6) (220,20,60)
Crimson: pos=(1,6), color=(220,20,60):840, id=R1
> add_beacon B2 Teal (11, 10) (0,128,128)
Teal: pos=(11,10), color=(0,128,128):896, id=B2
> add_beacon M2 Indigo (10, 4) (75,0,130)
Indigo: pos=(10,4), color=(75,0,130):355, id=M2
>
** End of commands from 'example-beacons.txt'
> read "example-lightbeams.txt"
** Commands from 'example-lightbeams.txt'
> # Add light sources
> add_lightbeam G1 M1
Added lightbeam: Lime -> Fuchsia
> add_lightbeam R1 M2
Added lightbeam: Crimson -> Indigo
> add_lightbeam M1 M2
Added lightbeam: Fuchsia -> Indigo
> add_lightbeam M2 B2
Added lightbeam: Indigo -> Teal
>
** End of commands from 'example-lightbeams.txt'
```

```
> read "example-fibres.txt"
** Commands from 'example-fibres.txt'
> add_fibre (0,0) (6,0) 1
Added fibre: (0,0) <-> (6,0), cost 1
> add_fibre (6,0) (10,4) 1
Added fibre: (6,0) <-> (10,4), cost 1
> add_fibre (10,4) (6,6) 1
Added fibre: (10,4) <-> (6,6), cost 1
> add_fibre (6,6) (11,10) 1
Added fibre: (6,6) <-> (11,10), cost 1
> add_fibre (0,0) (1,6) 2
Added fibre: (0,0) <-> (1,6), cost 2
> add_fibre (1,6) (6,6) 3
Added fibre: (1,6) <-> (6,6), cost 3
>
** End of commands from 'example-fibres.txt'
> all_xpoints
1. (0,0)
2. (6,0)
3. (10,4)
4. (1,6)
5. (6,6)
6. (11,10)
> fibres (6,6)
1. (10,4) : 1
2. (1,6) : 3
3. (11,10) : 1
> remove_fibre (6,6) (1,6)
Removed fibre: (6,6) <-> (1,6)
> fibres (6,6)
1. (10,4) : 1
2. (11,10) : 1
> route_any (0,0) (11,10)
0.    (0,0) : 0
1. -> (6,0) : 1
2. -> (10,4) : 2
3. -> (6,6) : 3
4. -> (11,10) : 4
> clear_fibres
All fibres removed.
> all_xpoints
>
** End of commands from 'example-compulsory-in.txt'
> read "example-non-compulsory-in.txt"
** Commands from 'example-non-compulsory-in.txt'
> clear_beacons
Cleared all beacons
> clear_fibres
All fibres removed.
> read "example-beacons.txt"
** Commands from 'example-beacons.txt'
⋮
** End of commands from 'example-beacons.txt'
> read "example-lightbeams.txt"
** Commands from 'example-lightbeams.txt'
⋮
** End of commands from 'example-lightbeams.txt'
> read "example-fibres.txt"
```

```
** Commands from 'example-fibres.txt'
⋮
** End of commands from 'example-fibres.txt'
> route_least_xpoints (0,0) (11,10)
0.    (0,0) : 0
1. -> (1,6) : 2
2. -> (6,6) : 5
3. -> (11,10) : 6
> route_fastest (0,0) (11,10)
0.    (0,0) : 0
1. -> (6,0) : 1
2. -> (10,4) : 2
3. -> (6,6) : 3
4. -> (11,10) : 4
> route_fibre_cycle (11,10)
0.    (6,6)
1. -> (10,4)
2. -> (6,0)
3. -> (0,0)
4. -> (1,6)
5. -> (6,6)
> all_fibres
(0,0) -> (6,0)
(0,0) -> (1,6)
(6,0) -> (10,4)
(10,4) -> (6,6)
(1,6) -> (6,6)
(6,6) -> (11,10)
> trim_fibre_network
The remaining fibre network has total cost of 6
> all_fibres
(0,0) -> (6,0)
(0,0) -> (1,6)
(6,0) -> (10,4)
(10,4) -> (6,6)
(6,6) -> (11,10)
>
** End of commands from 'example-non-compulsory-in.txt'
> quit
```

Screenshot of user interface

Below is a screenshot of the graphical user interface after *example-beacons.txt* and *example-lightbeams.txt*, and *example-fibres.txt* have been read in.



Beacons
Beacon names
Beams
Beam color
Beam routes
Fibres
Pick Beacon
Pick Xpoint
+
-
1:1
Fit
Fontscale
1.00

```
0.    (1,6) : 0
1. -> (6,6) : 2
>
** End of commands from 'example-fibres.txt'
```

Stop test
Execute
Command: read
Number: 0
File...
Clear input