TIE-20106 Data structures and algorithms, spring 2020

Programming assignment 2: Nykse Meni? (Did’t just go?)

Last modified on 04/27/2020

Changelog

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

- 7.4. Clarified that add_route also returns false, if only one stop is given for the route.

Important about phase 2

The phase 2 of the programming assignment is built on top of phase 1. This document describes the new features of phase 2. Unless this document says otherwise, everything feature of phase 1 is the same in phase 2.

Implementing phase 2 doesn’t require implementing any non-compulsory part of phase 1, and implementing those non-compulsory phase 1 operations in phase 2 is not taken into account in grading phase 2. To be a little more specific, phase 2 main requires that adding stops and getting their information works in phase 1, other phase 1 functionality is mostly not needed. In other words, doing phase 2 shouldn’t depend on how “well” you managed to implement phase 1.

Topic of the assignment

The topic of the assignment this year is public transportation (to honor the near completion of the Tampere tram project). In the second phase the bus stop information of phase 1 is expanded by info on bus routes running through those stops (and as a non-compulsory feature, timetables of buses running on those routes). 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).

Terminology

Below is explanation for the most important terms in the assignment:

- **Route.** A bus route is a named sequence of bus stops, through which buses running on that route drive. Every route has a unique *ID* (which consists of characters A-Z, a-z, 0-9) and a list of stop IDs, through which the route goes. *Note! Routes are unidirectional, i.e. buses run through the stops in the given order, but do not come back on the same route (unless another route with a different id has been added for the return route).*

- **Trip.** A trip is a non-compulsory part of the assignment, which describes the timetable of an individual bus on a route. A trip contains the ID of the route (which has to be the ID of an already existing route) and a list of times, on which the bus leaves from each stop. Naturally there has to be the same number of times as there are bus stops on the route. This assignment assumes that every bus continues its journey immediately, i.e. the times of a bus arriving to a stop and leaving from that stop are the same.
- **Journey.** A journey describes how to travel from a starting stop through other stops to a destination stop. A journey consists of a list, where each element has a stop id, a route id (which route takes to the next stop, for the last stop the value is NO_ROUTE since the journey doesn't continue), and depending on the operation either distance (the total distance up to that stop) or time (departure time from the stop, or the arrival time to the destination for the last stop).
- (Phase 1 term "stop" is found on the phase 1 document.)

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 (some of these are new, some are repeated because of their importance):

- 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, 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 formats are plain test (readme.txt), markdown (readme.md), and Pdf (readme.pdf).**

- Implementing operations `journey_least_stops()`, `journey_shortest_distance()`, `journey_with_cycle()`, `add_trip()`, `route_times_from()`, and `journey_earliest_arrival()` is not compulsory to pass the assignment. However they affect the grade of the assignment. Some of the non-compulsory operations may require modifying the algorithms presented on this course.
- If the implementation is bad enough, the assignment can be rejected.

On distances between stops

Some operations require information on the distance between coordinates. The comparison is based on the "normal" euclidean distance $\sqrt{x^2+y^2}$. In the case of routes, the distance is rounded down to the nearest integer.

On printing journeys

When the course side code output journeys, it omits all values that are NO_… . For example, if some stop on the journey has NO_ROUTE as route id, the departure route is not printed at all (this mainly shows in the printout of the last stop of a journey, as well as if you have implemented walking connections).

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*

- Everything that was there in phase 1 (you can copy-paste the necessary parts of your phase 1 implementation into phase 2).
- Type definition `RouteID`, which used as a unique identifier for each route (every route has a different id).
- Type definition `Time`, which is used to represent a time of day. The type is an integer that tells the number of seconds since midnight.
- Type definition `Duration`, which is used to represent a duration of a trip. The type is an integer that tells the duration in seconds.
- Type definition `Distance`, which is used to represent a distance. The type is an integer that tells the distance in metres.
- Type definition `NAME`, which is used for names of stops and regions, for example. (The type is a string, and the main routine allows names to contain characters a-z, A-Z, 0-9, space, and dash -).

- Constants `NO_ROUTE`, `NO_TIME`, `NO_DURATION`, and `NO_DISTANCE`, which are used as return values, if information is requested doesn't exist or is not applicable.

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. The UI now has new phase 2 controls.

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_stops`() to get a list of stops, and asks their information with `get_...`() operations. If drawing regions is on, they are obtained with operation `all_regions`(), and the name of each region with `get_region_name`(). The location and size of the region is obtained by calling `region_bounding_box`(). **NOTE! Operation `region_bounding_box()` is a non-compulsory operation. If it has not been implemented, regions are not drawn graphically.** If drawing of routes is on, the UI gets all routes leaving from a stop using `routes_from()`. If there are several routes between two stops, their ids are shown separated by a comma. When the UI shows the result route of an operation (in red), it first shows the route id in the result, and then in parenthesis all possible routes between stops, if there are more routes. *(The non-compulsory walking routes are shown as dotted lines, and their id as a dash. In the return values walking routes are marked with id NO_ROUTE).*

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).

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

Command **Public member function**	**Explanation**
`all_routes` `std::vector<RouteID> all_routes()`	Returns a list (vector) of the routes in any (arbitrary) order (the main routine sorts them based on their ID). *This operation is not included in the default performance tests.*
`clear_all` `void clear_all()`	Clears out the data structures (after this `all_stops()`, `all_regions()`, and `all_routes()` return empty vectors). *This operation is not included in the default performance tests.*
`add_route` `bool add_route(RouteID id, std::vector<StopID> stops)`	Adds a new route running through the given stops. If there already exists a route with the given id, some stop id is not found, or only one stop is given, nothing is done and `false` is returned, otherwise `true` is returned.
`routes_from` `std::vector<std::pair<RouteID, StopID>> routes_from(StopID stopid)`	Returns a list of routes leaving from the given stop, and what stop is next on each route. If no routes leave from the stop, an empty list is returned. If there is no stop with the given id, pair {NO_ROUTE, NO_STOP} is returned.
`route_stops` `std::vector<StopID> route_stops(RouteID id)`	Returns a list of route's stops in the driving order (i.e. in the same order which stops were given when adding the route). If there's no route with the given id, a single element NO_STOP is returned. *This operation is not included in the default performance tests.*
`clear_routes` `void clear_routes()`	Clears out the routes and trips (after this `all_routes()` returns an empty vector), but *doesn't* delete stops or regions. *This operation is not included in the default performance tests.*
(The operations below should probably be implemented only after the ones above have been implemented.)	
`journey_any` `std::vector<std::tuple<StopID, RouteID, Distance>> journey_any(StopID fromstop, StopID tostop)`	Returns any (arbitrary) journey between the given stops (see "On printing journeys"). The returned vector first has the starting point with distance 0, then the rest of the stops along the journey and the total distance up to each stop, and finally the destination stop with the total distance as the last element. If no route can be found between the points, an empty vector is returned. If either of the stop ids is not found, one element {NO_STOP, NO_ROUTE, NO_DISTANCE} is returned. *Note! In this operation the RouteIDs in the return value don't matter, they can be anything, like NO_ROUTE or the route used to get from the stop to another.*

Command Public member function	Explanation
(Implementing the following operations is not compulsory, but they are part of the grading. The operations are listed in the order of estimated difficulty.)	

`journey_least_stops`
`std::vector<std::tuple<StopID, RouteID, Distance>>`
`journey_least_stops(StopID fromstop, StopID tostop)`
	Returns a journey between the given stops with the minimum number of stops (see "On printing journeys"). The returned vector first has the starting point with distance 0 and the route that takes to the next stop. Then the rest of the stops along the journey, the total distance up to that each stop, and again the route that takes to the next stop. And finally the destination stop with the total distance and NO_ROUTE (since the journey doesn't continue) as the last element. If no route can be found between the points, an empty vector is returned. If either of the stop ids is not found, one element {NO_STOP, NO_ROUTE, NO_DISTANCE} is returned. If there are several possible journeys with the same number of stops, any one of them can be returned.

`journey_with_cycle`
`std::vector<std::tuple<StopID, RouteID, Distance>>`
`journey_with_cycle(StopID fromstop)`
	Returns a journey between the given stops (see "On printing journeys") that has a cycle, i.e. the journey returns to a stop already on the journey. The returned vector first has the starting point with distance 0 and the route that takes to the next stop. Then the rest of the stops along the journey, the total distance up to that each stop, and again the route that takes to the next stop. And finally the recurring stop that causes the cycle with the total distance and NO_ROUTE (since the journey doesn't continue) as the last element. If no route can be found between the points, an empty vector is returned. If either of the stop ids is not found, one element {NO_STOP, NO_ROUTE, NO_DISTANCE} is returned. If there are several possible journeys with cycles, any one of them can be returned.

Command Public member function	Explanation

`journey_shortest_distance`
`std::vector<std::tuple<StopID, RouteID, Distance>>`
`journey_shortest_distance(StopID fromstop, StopID tostop)`
	Returns a journey between the given stops with the minimum total distance (see "On printing journeys"). The returned vector first has the starting point with distance 0 and the route that takes to the next stop. Then the rest of the stops along the journey, the total distance up to that each stop, and again the route that takes to the next stop. And finally the destination stop with the total distance and NO_ROUTE (since the journey doesn't continue) as the last element. If no route can be found between the points, an empty vector is returned. If either of the stop ids is not found, one element {NO_STOP, NO_ROUTE, NO_DISTANCE} is returned. If there are several possible journeys with the same total distance, any one of them can be returned.

`add_trip` `bool add_trip(RouteID routeid, const std::vector<Time> &stop_times)`	Adds a trip to the given route (see the definition of "trip" earlier in this document). The trips departs from the stops at the given times (except for the last stop, where it just arrives at the given time). If there's no route with the given id, nothing is done and `false` is returned, otherwise `true` is returned. The course side code checks that there's the same number of times as there are stops (by calling route_stops()).

`route_times_from`
`std::vector<std::pair<Time, Duration> >`
`route_times_from(RouteID routeid, StopID stopid)`
	Returns info on when buses on a given route depart from the given stop. For each bus, departure time from the stop and the duration of the trip to the next stop is returned. The times can be returned in arbitrary order (the main program sorts the result based on the time). If there's no route or stop with the given id, or if the route doesn't run through the given stop, a single element {NO_TIME, NO_DURATION} is returned. *This operation is not included in the default performance tests.*

Command **Public member function**	**Explanation**

`journey_earliest_arrival`
`std::vector<std::tuple<StopID, RouteID, Time>>`
`journey_earliest_arrival(StopID fromstop, StopID tostop, Time starttime)`
	Returns a journey between the given stops with the earliest possible arrival time (see "On printing journeys"). The operation only takes into account routes with trips added with `add_trip` (and walking routes, if `add_walking_connections` has been implemented). The returned vector first has the starting point, the route that takes to the next stop, and departure time from the stop. Then the rest of the stops along the journey, the route that takes to the next stop, and departure time from the stop. And finally the destination stop with the arrival time and NO_ROUTE (since the journey doesn't continue) as the last element. If no route can be found between the points, an empty vector is returned. If either of the stop ids is not found, one element {NO_STOP, NO_ROUTE, NO_DISTANCE} is returned. If there are several possible journeys with the same arrival time, any one of them can be returned. *Note! In this operation you can assume (if you wish) that arrival happens during the same day as departure, i.e. you don't have to take into account journeys that last over midnight.*

`add_walking_connections` `void add_walking_connections()`	Adds a walking connection between each stop and its 5 nearest stops (the route id of a walking connection is NO_ROUTE). If you implement operation journey_earliest_arrival(), the walking connections have no departure time (they are always possible) and you can choose their duration yourself based on your walking speed and the distance (for example 4 km/h). *Note! This operation is not included in the published correctness tests, and it's not by default included in the performance tests. It's meant as an extra challenge to journey_earliest_arrival() operation. (Note 2: If you want an additional challenge, in case of several possible trips with the same arrival time, try to choose the one where you have to change the bus as few times as possible.)*
(The following operations are already implemented by the main program.)	
random_add n (implemented by main program)	Add *n* new stops with random id, name, and coordinates (for testing). With a 80 % probability the stop is also added to a random region. Note! The values really are random, so they can be different for each run.
random_route_trips (implemented by main program)	Adds a random route between existing stops. The added route has 5 stops and 2 trips (if you don't implement add_trip(), you don't have to care about the trips).

Command **Public member function**	**Explanation**
random_seed n (implemented by main program)	Sets a new seed to the main program's random number generator. By default the generator is initialized to a different value each time the program is run, i.e. random data is different from one run to another. By setting the seed you can get the random data to stay same between runs (can be useful in debugging).
read ”filename” (implemented by main program)	Reads more commands from the given file (This can be used to read a list of stops from a file, run tests, etc.)
stopwatch on / off / next (implemented by main program)	Switch time measurement on or off. When program starts, measurement is "off". When it is turned "on", the time it takes to execute each command is printed after the command. Option "next" switches the measurement on only for the next command (handy with command "read" to measure the total time of a command file).
perftest all/compulsory/cmd1;cmd2... **timeout n n1;n2;n3...** (implemented by main program)	Run performance tests. Clears out the data structure and add *n1* random stops and some regions (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 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).
testread "in-filename" "out-filename" (implemented by main program)	Runs a correctness test and compares results. Reads command from file in-filename and shows the output of the commands next to the expected output in file out-filename. Each line with differences is marked with a question mark. Finally the last line tells whether there are any differences.
help (implemented by main program)	Prints out a list of known commands.
quit (implemented by main program)	Quit the program. (If this is read from a file, stops processing that file.)

"Data files"

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

Below are examples of a data files, one of which adds stops, the other regions:

- *example-stops.txt*

```
# Add stops
add_stop 1 One (1,1)
add_stop 2 Two (6,2)
add_stop 3 Three (0,6)
add_stop 4 Four (7,7)
add_stop 5 Five (4,4)
add_stop 6 Six (2,9)
```

- *example-routes.txt*

```
# Add routes
add_route A 1 3 4
add_route B 4 6 3
add_route C 1 2 5
```

- *example-trips.txt*

```
# Add trips
add_trip A 08:30:00 08:40:00 09:00:00
add_trip B 08:45:00 09:00:00 10:00:00
add_trip C 08:00:00 08:10:00 08:20:00
add_trip C 09:00:00 09:15:00 09:30:00
```

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-all-in.txt" "example-all-out.txt"`

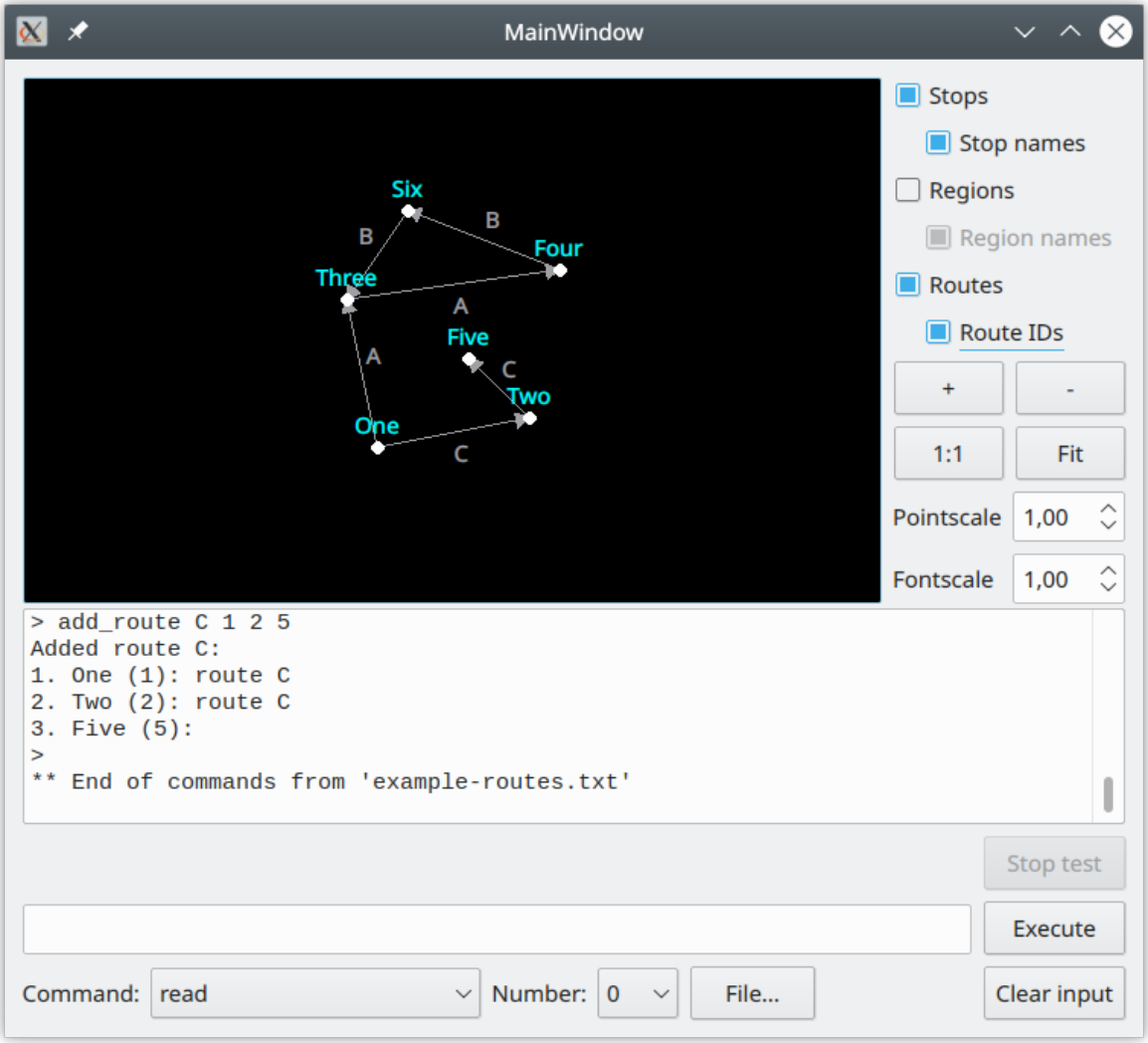
```
> read "example-compulsory-in.txt"
** Commands from 'example-compulsory-in.txt'
> read "example-stops.txt"
** Commands from 'example-stops.txt'
> # Add stops
> add_stop 1 One (1,1)
One: pos=(1,1), id=1
> add_stop 2 Two (6,2)
Two: pos=(6,2), id=2
> add_stop 3 Three (0,6)
Three: pos=(0,6), id=3
> add_stop 4 Four (7,7)
Four: pos=(7,7), id=4
> add_stop 5 Five (4,4)
Five: pos=(4,4), id=5
> add_stop 6 Six (2,9)
Six: pos=(2,9), id=6
>
** End of commands from 'example-stops.txt'
```

```
> read "example-routes.txt"
** Commands from 'example-routes.txt'
> # Add routes
> add_route A 1 3 4
Added route A:
1. One (1): route A
2. Three (3): route A
3. Four (4):
> add_route B 4 6 3
Added route B:
1. Four (4): route B
2. Six (6): route B
3. Three (3):
> add_route C 1 2 5
Added route C:
1. One (1): route C
2. Two (2): route C
3. Five (5):
>
** End of commands from 'example-routes.txt'
> all_routes
1. A
2. B
3. C
> routes_from 1
1. Three (3): route A
2. Two (2): route C
> route_stops A
1. One (1): route A
2. Three (3): route A
3. Four (4):
> journey_any 1 5
1. One (1): distance 0
2. Two (2): distance 5
3. Five (5): distance 7
>
** End of commands from 'example-compulsory-in.txt'
> journey_with_cycle 1
1. One (1): route A distance 0
2. Three (3): route A distance 5
3. Four (4): route B distance 12
4. Six (6): route B distance 17
5. Three (3): distance 20
> add_route D 5 4
Added route D:
1. Five (5): route D
2. Four (4):
> journey_least_stops 1 4
1. One (1): route A distance 0
2. Three (3): route A distance 5
3. Four (4): distance 12
> journey_shortest_distance 1 4
1. One (1): route C distance 0
2. Two (2): route C distance 5
3. Five (5): route D distance 7
4. Four (4): distance 11
> read "example-trips.txt"
** Commands from 'example-trips.txt'
```

```
> # Add trips
> add_trip A 08:30:00 08:40:00 09:00:00
Added trip to route A
1. One (1): route A at 08:30:00
2. Three (3): route A at 08:40:00
3. Four (4): at 09:00:00
> add_trip B 08:45:00 09:00:00 10:00:00
Added trip to route B
1. Four (4): route B at 08:45:00
2. Six (6): route B at 09:00:00
3. Three (3): at 10:00:00
> add_trip C 08:00:00 08:10:00 08:20:00
Added trip to route C
1. One (1): route C at 08:00:00
2. Two (2): route C at 08:10:00
3. Five (5): at 08:20:00
> add_trip C 09:00:00 09:15:00 09:30:00
Added trip to route C
1. One (1): route C at 09:00:00
2. Two (2): route C at 09:15:00
3. Five (5): at 09:30:00
>
** End of commands from 'example-trips.txt'
> route_times_from C 2
Route C leaves from stop Two: pos=(6,2), id=2
 at following times:
08:10:00 ( duration 00:10:00)
09:15:00 ( duration 00:15:00)
> journey_earliest_arrival 1 5 08:45:00
1. One (1): route C at 09:00:00
2. Two (2): route C at 09:15:00
3. Five (5): at 09:30:00
> journey_earliest_arrival 1 6 08:00:00
No journey found!
> add_trip D 08:25:00 08:30:00
Added trip to route D
1. Five (5): route D at 08:25:00
2. Four (4): at 08:30:00
> route_times_from A 1
Route A leaves from stop One: pos=(1,1), id=1
 at following times:
08:30:00 ( duration 00:10:00)
> journey_earliest_arrival 1 6 08:00:00
1. One (1): route C at 08:00:00
2. Two (2): route C at 08:10:00
3. Five (5): route D at 08:25:00
4. Four (4): route B at 08:45:00
5. Six (6): at 09:00:00
>
```

Screenshot of user interface

Below is a screenshot of the graphical user interface after *example-stops.txt* and *example-routes.txt* have been read in.


MainWindow
Six
B
B
Four
Three
A
Five
A
C
Two
One
C
Stops
Stop names
Regions
Region names
Routes
Route IDs
+
-
1:1
Fit
Pointscale
1,00
Fontscale
1,00
> add_route C 1 2 5
Added route C:
1. One (1): route C
2. Two (2): route C
3. Five (5):
>
** End of commands from 'example-routes.txt'
Stop test
Execute
Command:
read
Number:
0
File...
Clear input