COMP.CS.300 Data structures and algorithms 1, fall 2023

Programming assignment 1: ScholarNet

Last modified on 6. marrask. 2023

Change log

- get_referenced_by_chain explanation was missing a functionality, which was tested. This considers the case where the ID doesn't match any publications. An explanation has been added (25. lokak. 2023)
- optimization testing added (3. marrask. 2023)
- Fixed `get_publications_after` order according to the tests (6. marrask. 2023)
- A chapter about reporting AI tool usage added at the end of the document (20.11.2023)

Introduction to the assignment

In this programming assignment, you will practice implementing data structures and algorithms. This year's programming assignment is about ScholarNet, a social network for affiliations. The goal of the assignment is to create a program where you can enter and search for information about affiliations, and their publications. Publications refer to each other, which chains publications together. Some operations in these assignments are compulsory, the rest are optional. Compulsory operations are required to pass the assignment; optional parts are not required to pass but implementing them will affect the grade positively.

The assignment consists of implementing a given class, which will store the required information in its data structures, and whose methods implement the required functionality. The main program and the Qt-based graphical user interface are provided by the course, but running the program in text-only mode without Qt libraries is also possible.

The goal is to produce code that is as efficient as possible in terms of command counts. Each operation has its weight – the formula to define the performance estimate can be found in Perfestimate grader in Plussa. You are expected to have to make compromises about the performance: all functions cannot be perfectly fast and efficient at the same time. Micro-optimizations like deciding between 'a = a + b' or 'a += b', will not significantly affect your final grade.

Note the following (some are new, some are repeated because of their importance):

- The main program given by the course can be run either with a graphical user interface (when compiled with QtCreator/qmake), or as a textual command line program (when compiled just with g++ or some other C++ compiler). In both cases the basic functionality and students' implementation is the same.
- **Hint:** If the performance of any of your operations is worse than nlog(n) on average, the performance is not ok. Most operations can be implemented to be faster than that. Especially for operations you use often, even linear performance is not great.
- **As part of the assignment**, it is expected that you provide an estimate of the asymptotic performance for each operation you implement (with O-notation), and a brief rationale. Within the file datastructures.hh, you will find code comments preceding each operation to facilitate this. The comments look like the following code snippet:

```
// Estimate of performance: O(n)
// Short rationale for estimate: for
loop goes through
// all the elements
void print_all(std::vector<int>
numbers);
```

- In performance tests, the key aspect is how the command counts vary based on the data size (N). In the grader results, reference implementation results are available for comparison, the reference implementation is labeled as “teacher”. However, the comparison does not determine the grade. The pass criterion is to create a functional code that doesn’t crash. Final grading occurs after all results

personnel.

- The command count includes all work done during the operation.
- The better the performance, the better the grade.
- If the pass criteria are not fulfilled, the assignment will be rejected.
    - Perfestimate grader provides informative feedback for the reasons of rejection and multiple submissions are allowed – students can gradually improve their implementation based on the feedback

- **Note:**
    - Compiler optimization is not effective – a student can only change datastructures.hh and datastructures.cc files, thus the performance depends only on algorithmic choices and design of the program.
    - Parallel execution is not allowed

Terminology

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

- **Affiliation**: affiliations are organizations such as universities or companies. Every affiliation has at least
    - a unique string ID,
    - a name,
    - a location (x, y),
        - x and y are integers; x coordinates grow to the right, y grows up
        - two affiliations cannot have the same coordinates
- **Publication**: Publications are scholarly works produced by affiliations. Each

    - Publications reference other publications, thus forming a citation network.

- **References**: References link publications to each other. Each publication can be referencing an arbitrary number of other publications, but every publication can be referenced by at most one parent publication.
    - References are non-cyclic:
        - Publication 1 will not be referenced by publication 2 if publication 2 is already directly or indirectly referred to by publication 1.
        - The assignment does not need to consider circular references in any manner. This has been taken care of by filtering the data.

On sorting

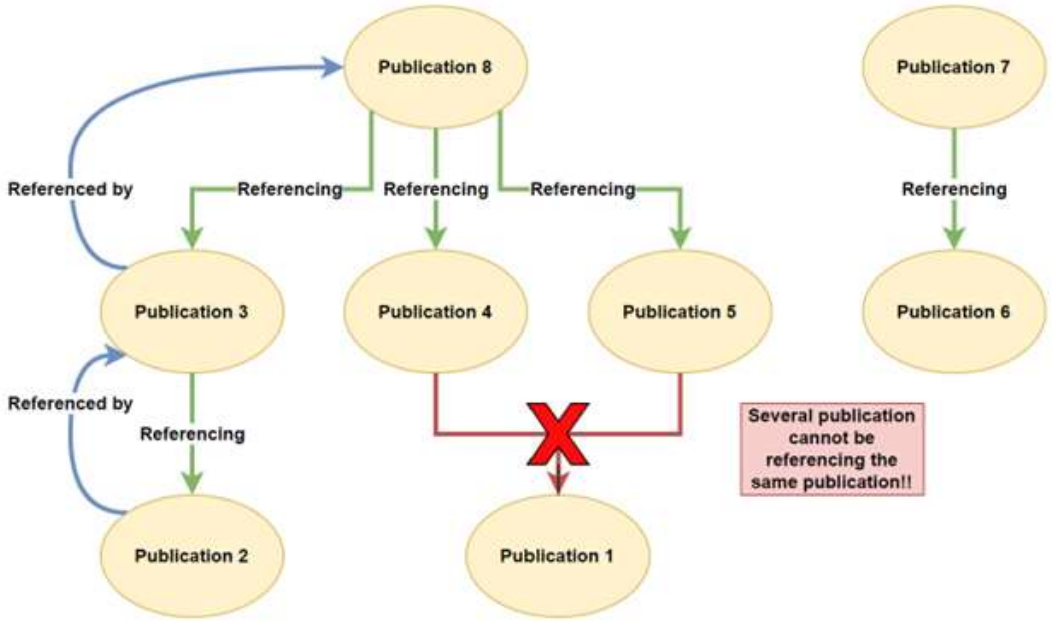
Sorting names should be done using "<" comparison, which works because only a-z, A-Z, 0-9, space, and a dash - are allowed in names. Multiple equal names can be in any order.

The operation `get_affiliations_distance_increasing()` requires the comparison of coordinates. The comparison is based on the “normal” Euclidean distance from the coordinate origin $\sqrt{x^2 + y^2}$.

The coordinate closer to origin comes first. If the distance to the origin is the same, the coordinate with the smaller y-value comes first. Coordinates with equal distances and y-values can be in any order.

In the non-compulsory operation `get_affiliations_closest_to()`, affiliations are sorted based on their distance from the given position. For this operation, the distance is the “normal” Euclidean distance $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, and again if the distances are equal, the coordinate with smaller y comes first.

Below is an example of the publication reference structure.



Explanation:

Referencing:

Publication 8 is the newest publication. It is **referencing publication 3, 4 and 5**. Publication 3 references publication 2. Publication 7 is referencing publication 6. “Legit referencing” is denoted by the **green arrows** in the figure.

Example of a faulty reference: both publication 4 and 5 are trying to reference publication 1. While this is possible in real life, in this project it is not allowed. This is done to retain a tree-like structure. Therefore, only one of them is allowed to reference publication 1.

Referenced by:

Publication 2 is being **referenced by** publication 3. Publication 3 is being referenced by publication 8. Publication 4 is also being referenced by Publication 8, and so on. In the figure, only some

single publication.

Read this section again if this is not clear!

About implementation

This assignment uses C++ version 17. The purpose of this programming assignment is to learn how to use ready-made data structures and algorithms, so using C++ standard template library (STL) is highly recommended. There are no restrictions on using the C++ STL, however, no extra libraries are allowed such as libraries provided by Windows, Qt, Linux, Mac OS, Boost, etc. *Please note, that it is highly likely you will have to implement some algorithms independently without STL.*

Graphical user interface with QtCreator

When compiled with QtCreator, a graphical user interface is provided to visualize the data. The UI (User interface) of assignment 1 contains a few disabled (grayed out) controls needed for assignment 2.

The UI illustrates the affiliations and publications. The graphical view can be scrolled and zoomed. Clicking an affiliation's name prints out the information more in detail and prints the corresponding ID in the command line (a handy way to check the IDs for command parameters). The user interface has selections for what to show graphically.

OPTIONAL FEATURE: Drawing worldmap

Additionally, there is a subproject called worldmap in the project. If one executes commands

```
git submodule init

git submodule update
```

Similarly, checking the Publications checkbox triggers the rendering of publications. For drawing, the UI needs the operation `get_all_affiliations()` to get a list of affiliations. Additionally, it makes use of the functions `get_affiliation_coord()` and `get_affiliation_name()` to query and acquire their respective details.

Note! The graphical representation gets all its information from the student's code! **It is not a depiction of the correct result, but what information students' code gives out.**

Structure and functionality of the program

The template project provides selected parts of the program; missing parts must be implemented by the student to the datastructures.hh and datastructures.cc files. There is a subproject (or so-called git submodule) in the directory ‘worldmap’. It is essential that no alterations are made to any other files, nor should any additional files be added.

Project template

mainprogram.hh, mainprogram.cc, mainwindow.hh, mainwindow.cc, mainwindow.ui

- You are **NOT ALLOWED TO MAKE ANY CHANGES TO THESE FILES!**
- The files contain the main program, which takes care of reading input, interpreting commands, and printing out results. The main program also contains commands for testing.
- If you compile the program with QtCreator or qmake, you will get a graphical user interface, with an embedded command interpreter, available commands as dropdown menu and file picker, that can be used to complement commands that

datastructures.* (the assignment)

Files *datastructures.hh* and *datastructures.cpp*

- `class Datastructures`: The given public member functions of the class must be implemented. You can add your own stuff into the class (new data members, new member functions, etc.) You must not edit the *public interface* of the class Datastructures

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

datastructures.hh: class Datastructures

- 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 can add new methods and data members to the private side).
    - **Write an estimate of the asymptotic performance of the operation with a short rationale above the function declaration as a comment. This is done for each member function you implement.**
- Type definitions
    - AffiliationID: Used as a unique identifier for each affiliation (may include characters A-Z, a-z, 0-9, and dash -). There can be several affiliations with the same name, but every affiliation has a different ID.
    - Coord: Used in the public interface to represent (x, y) coordinates. As an example, some comparison

be several publications with the same name, but every publication has a different id.

  - Name: Used for names of affiliations and publications (consists of characters A-Z, a-z, 0-9, space, and dash -).
  - Year : An integer, which represents publication year. The main program ensures that all entered years are valid.

- Constants `NO_AFFILIATION`, `NO_PUBLICATION`, `NO_NAME`, `NO_YEAR`, `NO_WEIGHT` and `NO_COORD`: Used as error codes, if information is requested for an affiliation or publication that does not exist.

datastructures.cc

- Here you write the code for your operations.
- Function `random_in_range`: Returns a random value in a given range (start and end are both included in the range). You may use this function if your implementation requires random numbers.

Running the program

When you run the program, you should use the commands listed in the table below. Commands that require a member function are linked to the corresponding function in the Datastructures class, which you need to implement (certain commands have already been fully implemented by the course staff).

If you provide a file as a command line parameter to the program, it will execute commands from that file and then terminate. You can also initiate a program compiled with QtCreator from the command prompt using the `--console` command line argument for a purely text-based interaction.

which it is recommended to implement them:

Compulsory functions

Command **Public member function** **Compulsory part** (Optional parameters are in []-brackets and alternatives separated by |)	**Explanation of public member function**

`get_affiliation_count`
 	
`unsigned int get_affiliation_count()`
	Returns the current count of affiliations stored in the data structure.

`clear_all`
 	
`void clear_all()`
	Clears out the data structures (after this `all_affiliations()` and `all_publications()` return empty vectors).

`get_all_affiliations`
 	
`std::vector<AffiliationID> get_all_affiliations()`
	Returns all the affiliations in an arbitrary order (the main program sorts them based on their ID).

`add_affiliation AffiliationID name (x,y)`
 	
`bool add_affiliation(AffiliationID id, Name const& name, Coord xy)`
	Adds an affiliation to the data structure with the given unique ID, name, type, and coordinates, and returns `true`. If there already is an affiliation with the given ID, the function does nothing and returns `false`.

`affiliation_info AffiliationID`
 	
`Name get_affiliation_name(AffiliationID id)`
	Returns the name and type of the affiliation with the given ID, or `NO_NAME` if no such affiliation exists. (The main program calls this with various affiliation IDs.)

`affiliation_info AffiliationID`
 	
`Coord get_affiliation_coord(AffiliationID id)`
	Returns the coord of the affiliation with the given ID, or value `NO_COORD`, if such an affiliation does not exist. (The main program calls this function in various points.)

(The operations below should be implemented only after the ones above have been implemented.)	
`get_affiliations_alphabetically`	Returns affiliation IDs after sorting affiliations alphabetically based on their

	order.

`get_affiliations_distance_increasing`
 	
`std::vector<AffiliationID> get_affiliations_distance_increasing()`
	Returns affiliation IDs sorted according to their coordinates (defined earlier in this document).

`change_affiliation_coord ID (x,y)`
 	
`bool change_affiliation_coord(AffiliationID id, Coord newcoord)`
	Changes the location of the affiliation with the given ID and returns `true`. If such an affiliation does not exist returns `false`.

`find_affiliation_with_coord (x,y)`
 	
`AffiliationID find_affiliation_with_coord(Coord xy)`
	Returns an affiliation with the given coordinate, or `NO_AFFILIATION`, if no such affiliation exists.

(The operations below should be implemented only after the ones above have been implemented.)	

`add_publication ID name year [AffiliationID [AffiliationID […]]]`
 	
`bool add_publication(PublicationID id, Name const& name, Year year, std::vector<AffiliationID> affiliations)`
	Adds a publication to the data structure with a given unique ID, name, **year, and a vector of affiliations**. Initially, the added publication does not reference other publications, and neither is it being referenced by other publications. If there already is a publication with the given ID, nothing is done and `false` is returned, otherwise `true` is returned.

`get_all_publications`
 	
`std::vector<PublicationID> all_publications()`
	Returns all the publications in any (arbitrary) order (the main program sorts them based on their ID).

`publication_info ID`
 	
`Name get_publication_name(PublicationID id)`
	Returns the name of the publication with the given ID, or `NO_NAME` if such publication does not exist. (Main program calls this while looking at various affiliations.)

`publication_info ID`
 	
`Year get_publication_year(PublicationID id)`
	Returns the year of the publication with the given ID, or `NO_YEAR` if such publication does not exist. (Main program calls this while looking at various affiliations.)

`get_affiliations ID`
 	
`std::vector<AffiliationID> get_affiliations(PublicationID id)`
	Returns a vector that contains all ids of the affiliations of a publication with the given ID (in arbitrary order), or a vector with single item `NO_AFFILATION`, if such publication does not exist. (The main program sorts them based on their ID, the main program calls this while looking at various affiliations.)

`bool add_reference(PublicationID id, PublicationID parentid)`	publication. If no publications exist with the given IDs, nothing is done and `false` is returned, otherwise `true` is returned.

`get_direct_references PublicationID`
 	
`std::vector<PublicationID> get_direct_references(PublicationID id)`
	Returns a vector of publication IDs, which the publication with the given ID is referencing directly. If a publication with the given ID doesn't exist, return a vector with a single item `NO_PUBLICATION`. Otherwise, return the direct references in a vector in any (arbitrary) order. (the main program sorts them based on their ID)

`add_affiliation_to_publication AffiliationID PublicationID`
 	
`bool add_affiliation_to_publication (AffiliationID affiliationid, PublicationID publicationid)`
	Adds the given affiliation to the given publication and returns `true`. If no affiliation or publication exists with the given IDs, nothing is done, and `false` is returned.

`get_publications AffiliationID`
 	
`std::vector<PublicationID> get_publications(AffiliationID id)`
	Returns a vector of publication IDs, which the affiliation with the given ID has produced in any (arbitrary) order. (the main program sorts them based on their ID) If no affiliation with the given ID exists, a vector with a single element `NO_PUBLICATION` is returned.

`get_publications_after AffiliationID Year`
 	
`std::vector<std::pair<Year, publicationID>> get_publications_after(AffiliationID affiliationid, Year year)`
	Returns publications from the provided affiliation, released at or after the specified year. The publications should be arranged in ascending order based on their publication year. In case multiple publications share the same year, they should be sorted by their IDs. If the given affiliation does not exist, the result returned is the pair `{NO_YEAR, NO_PUBLICATION}`.

`get_parent PublicationID`
 	
`PublicationID get_parent(PublicationID id)`
	Returns the publication ID of a directly referencing publication. If there is no referencing publication, return `NO_PUBLICATION`.

`get_referenced_by_chain PublicationID`
 	
`std::vector<PublicationID> get_referenced_by_chain(PublicationID id)`
	Returns a vector of publication IDs that reference the given publication, the reference in this case can be direct or indirect. If a publication with the given ID doesn't exist, return a vector with a single item `NO_PUBLICATION`.

Command **Public member function** **Optional part** (Optional parameters are in []-brackets and alternatives separated by |)	**Explanation of public member function**

`get_all_references PublicationID`
 	
`std::vector<PublicationID>`
`get_all_references(PublicationID id)`
	Optional. Returns a vector of publication IDs that the publication with the given ID is referencing directly or indirectly. The order is arbitrary. (The main program sorts them based on their ID) If a publication with the given ID does not exist, return a vector with one element `NO_PUBLICATION`.

`get_affiliations_closest_to Coord`
 	
`std::vector<AffiliationID>`
`get_affiliations_closest_to(Coord xy)`
	Optional. Returns a vector of three affiliation IDs closest to the given coordinate in order of increasing distance (based on the ordering of coordinates described earlier). If there exists less than three affiliations, return a vector that contains all affiliation IDs in the mentioned order. If there are no affiliations, return an empty vector.

`remove_affiliation ID`
 	
`bool remove_affiliation(AffiliationID id)`
	Optional. Removes the affiliation with the given ID. All publications produced by this affiliation should no longer state that they were produced (even partially) by the removed affiliation. If an affiliation with the given ID does not exist, return `false`, otherwise return `true`.

`get_closest_common_parent PublicationID PublicationID`	**Optional.** Returns an ID of the **closest** parent

`id1, PublicationID id2)`	hierarchy, that is referencing both publications, either directly or indirectly. If either of the publication IDs doesn't match any publication, or if there's no publication referenced in common, the function returns `NO_PUBLICATION`.

`remove_publication PublicationID`
 	
`bool remove_publication(`
`PublicationID publicationid)`
	Optional. Removes the given publication. If successful, it returns `true`. If a publication with the given ID does not exist, return `false`. There should no longer be a reference to this publication. All the references by this publication should also be removed. Other publications should **not** be removed. No affiliation should state to have produced this publication.

Other functions in main program

These functions are already implemented by the main program. You may use these when testing yourself.

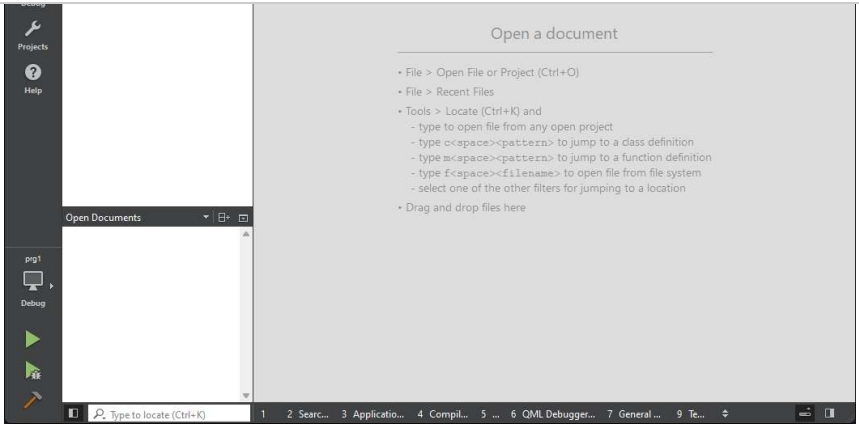
Main program command (Optional parameters are in []-brackets and alternatives separated by |)	**Explanation of the command**
`random_add n`	Adds n new affiliations, and n new publications with random IDs, names, coordinates etc. for testing purposes. Adding does not clear the existing affiliations. The added publications will contain four affiliations, some of

	but only the unique ones need to be addressed. Publications also contain references. Note! The values really are random, so they can be different in each run, unless `random_seed` is used.
`random_seed n`	Sets a new seed to the main program's random number generator. By setting the seed you can get the random data to stay the same between runs (can be useful in debugging).
`read "filename" [silent]`	Reads and executes more commands from the given file. If optional parameter `silent` is given, outputs of the commands are not displayed. (This can be used to read a list of affiliations from a file, run tests, etc.)
`stopwatch on|off|next`	Switch time measurement on or off. When the program starts, the stopwatch is `off`. When it is turned `on`, the time it takes to execute each command is printed after the command. The 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 cmd1[;cmd2...] timeout repeat n1[;n2...]`	Runs performance tests. Starts by clearing the data structure and adding `n1` random affiliations and publications using `random_add()`. Then, a random command from the command list is executed `'repeat'` times. Measures and displays the time and command count for adding elements and running commands. This process is repeated for `n2` elements and beyond. If a test takes longer than `timeout` seconds, it is halted (this is not necessarily a failure). When the parameter is a list of commands, selections come from that list (including `random_add()` is wise to

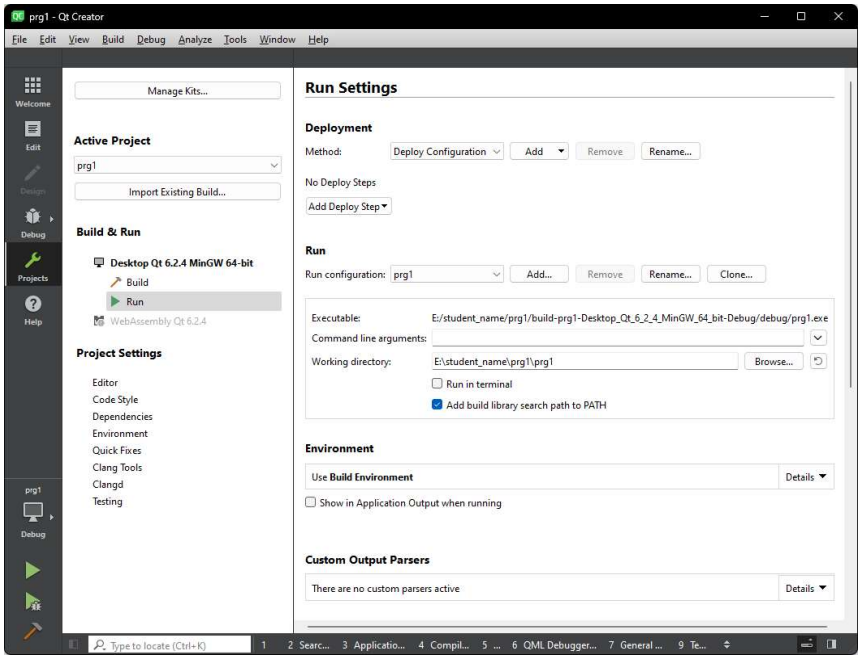
	test loop, not just initially). In the GUI, the "stop test" button will end the performance test, but it may take a moment to respond
`testread "in-filename" "out-filename"`	Runs a correctness test and compares the results. This command reads commands from file `in-filename` and shows the output of the 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`	Prints out a list of known commands.
`quit`	Terminates the program. (If the command is `read` from a file, it stops processing that file.)

Changing the working directory of a program from QtCreator

Before running any tests, the working directory of a program should be changed. This is to ensure that each test that will read other files will also find those files. The directory can be changed as follows:



First open the project you want to change the running directory for, then from the ribbon menu on the left side, select Projects (Wrench icon) next from the left list under the “Build & Run” options select run.



Now you may find the setting for “Working directory” from the list. Please set the working directory to the same directory where the current ‘.pro’ file is. For example, for prg1 the directory should end with `repository_name/prg1` where the `repository_name` is the directory of the Datastructures and algorithms repository, in the example figure the directory is called `student_name`.

and publications. Those files can then be read in using the `read` command, after which other commands can be tested without having to enter the data every year by hand.

What follows are examples of such data files located in the example-data folder:

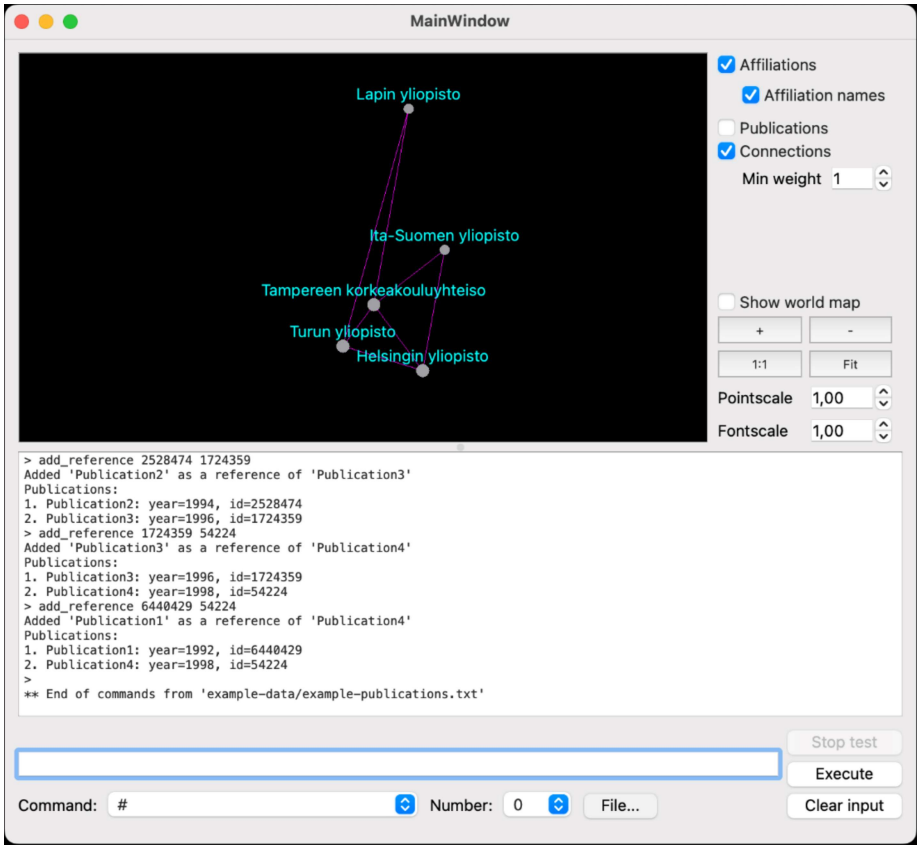
example-affiliations.txt

```
add_affiliation ISY "Ita-Suomen
yliopisto" (945,767)
add_affiliation TUNI "Tampereen
korkeakouluyhteiso" (542,455)
add_affiliation HY "Helsingin
yliopisto" (820,80)
add_affiliation TY "Turun yliopisto"
(366,219)
add_affiliation LY "Lapin yliopisto"
(740,1569)
```

example-publications.txt

```
add_publication 6440429
"Publication1" 1992
add_publication 2528474
"Publication2" 1994
add_publication 1724359
"Publication3" 1996
add_publication 54224 "Publication4"
1998
add_affiliation_to_publication TUNI
6440429
add_affiliation_to_publication HY
6440429
add_affiliation_to_publication ISY
6440429
add_affiliation_to_publication TY
2528474
add_affiliation_to_publication LY
2528474
add_affiliation_to_publication TUNI
2528474
add_affiliation_to_publication HY
1724359
add_affiliation_to_publication TY
1724359
add_reference 2528474 1724359
add_reference 1724359 54224
add_reference 6440429 54224
```

Below is a screenshot of the graphical user interface after example-affiliations.txt and example-publications.txt have been read in.



Example run

If you have set up the working directory correctly, and start the program, you can then enter commands such as:

```
testread "integration-compulsory/test-00-compulsory-in.txt" "integration-compulsory/test-00-compulsory-out.txt"
```

to get output from the program.

Functionality testing

The functionality testing can be performed with the command `testread` and the `-in.txt` and `-out.txt` files found in directories `functionality-compulsory`,

was successful and the functionality (according to the tests) is correct.

Performance testing "Perftests"

It is possible to perform performance tests (perftests) to most of the commands in the main program. It is important to take note of this. The perftests are identified with the main program commands. This does not mean that all the regular functionality, such as sorting usually done in the main program happens. It is just important to note that for example it is not possible to perftest `get_affiliation_name`, but it is possible to perftest `affiliation_info`, which calls `get_affiliation_name` among others. This being said, most of the commands are the same as the function name, and therefore it is possible to perftest the function by itself.
The perftest command is introduced in "Other functions of main program", this section considers the possibilities of using that commands and how to interpret the output.

The parameter `timeout` limits the maximum execution time of one line in perftest (adding + the command(s) itself). This maximum execution time depends on the hardware of the computer the program is running on. This is important to note, the execution times and timeout values here are just examples, you can test your code on your computer and determine some `timeout` and N values you are willing to test. Naturally the more data there is the more time it will take. Even if addition and the function are asymptotically constant, the data must be added.

Perftest with execution time

By default the program is compiled so that the `perftest` command outputs the time used by the addition and the command(s) itself.

Let's take a look at one example where we want to perftest `get_all_affiliations`:

```
> perftest get_all_affiliations 10
10 1;10;100
Timeout for each N is 10 sec.
```

```
05 ,  0.000208833
     10 ,  0.000483875 ,   3.7167e-
04 ,  0.000501042
    100 ,   0.00442133 ,   2.9916e-
03 ,   0.00444425
>
```

It is shown that addition takes more time with more elements, which is intuitive. Additionally it seems that also `get_all_affiliations` takes more time when there are more affiliations (and publications). If we want, we can plot these times for the cmds and see how the execution time scales when we have more affiliations (and publications). It is important to note that this execution time will most likely vary slightly between runs even on the same computer. If there are other programs running the add and cmds times will most likely be also higher. Additionally this is just a measure of the performance, it will not tell you the asymptotic performance with O notation. Of course you can see that for the example here it is possible the function doesn't seem to be O(1), since the execution time for 1 `get_all_affiliations` call increases when the number of affiliations and publications increases. For asymptotic performance, you should also take a look at your code, it will clearly tell you what the asymptotic performance is.

Perftest with command counters

In addition to time, with some operating systems (at least most Linux distros, if your system supports reading the command counters of the processor and `perf_event_open`) it is possible to have more accurate performance metrics than time. The command counters can be enabled by uncommenting the following row in prg1.pro:

```
#  QMAKE_CXXFLAGS += -
DUSE_PERF_EVENT
```

If your operating system does not support the command counter reading, the program might still compile, but crash and throw some kind of exception such as:

```
terminate called after throwing an
instance of 'char const*'
```

On Ubuntu, you can enable the `perf_event_open()` function by running the command:

```
 echo 0 | sudo tee
/proc/sys/kernel/perf_event_paranoid
```

The command counters are more accurate than

follows:

```
> perftest get_all_affiliations 20 20 1000;10000;100000
Timeout for each N is 20 sec.
For each N perform 20 random command(s) from:
get_all_affiliations

     N , add (sec) ,   add (count) ,   cmds (sec) , cmds
(count) ,  total (sec) , total (count)
  1000 , 0.0104188 ,      53499498 ,   0.00217933 ,
12074891 ,    0.0125981 ,     65574389
 10000 ,  0.108456 ,     534380344 ,    0.0227614 ,
 120435339 ,     0.131217 ,     654815683
100000 ,    1.2738 ,    5387425190 ,     0.345232 ,
1204036047 ,      1.61903 ,     6591461237
>
```

Compared to the example with only execution time, new columns have been added with `(count)` next to the operation they are referencing. In this case the `cmds (count)` is the command counter value for 20 calls of `get_all_affiliations` for each of the N values (here 1000, 10000 and 100000). It is possible to see that also the time to execute the command for each N increases.
If you want to run performance tests with command counters, but don't have a computer that could do it, please follow along the next section.

Perftesting in tie-tiraka.it.tuni.fi

There is a separate server with access to command counters: tie-tiraka.it.tuni.fi
This server is only for running the program, and does not have compilers, graphical user interface or development tools such as Qt Creator.
Since the server is available only through specific ip addresses, the easiest way to run your program on the server is to first clone or pull your code on linux-desktop.tuni.fi, then compile the code and run the program remotely on tie-tiraka.it.tuni.fi with terminal.

If you are not familiar with TigerVNC or linux desktop, please revise the instructions to those from Programming 2 area in Plussa. After you have successfully logged in linux-desktop.tuni.fi, please clone or pull your latest code there to the directory of your choice. Open the directory that contains prg1.pro (the cloned repository directory should have a directory called prg1, there should be a file called prg1.pro) in terminal, this can be done by opening an application called terminal and using the change directory command `cd` to

```
datastructures.cc mainwindow.cc -
DUSE_PERF_EVENT -O0 -std=c++17 &&
chmod +x prg1 && rsync prg1 tie-
tiraka.it.tuni.fi: && ssh tie-
tiraka.it.tuni.fi ./prg1
```

A breakdown of that oneliner for anyone interested: `g++` is a compiler, `-o` flag sets the output executable name, so in this case the executable will be a file called `prg1`, next there is a file list to be compiled, after that, we set a flag `USE_PERF_EVENT` to use the command counters, the second to last one is to not do any optimization to the code, last one sets the c++ standard version. `chmod` sets the `prg1` file execute bit, so that it can be run as a program. `rsync` copies the `prg1` file to the home directory in `tie-tiraka.it.tuni.fi`. `ssh` command opens an ssh session and automatically runs the `prg1` executable that was just copied there with rsync. `&&` will execute the next command only if the previous command was successful, which is necessary since we do not want to continue if any of the steps fail.

Unless you have set up ssh keys on `tie-tiraka.it.tuni.fi`, it will ask your tuni password twice, once for copying the compiled file (`rsync` command) and the second time for starting a session where the program will be run (`ssh` command). After the program starts running a prompt "`> `" will appear. You can now run perftest-commands with the counter output.

For example the output can look like this, if you run `perftest get_all_affiliations 20 20 1000;10000;100000`:

```
[student_name@linux-desktop11 prg1]$ g++ -o prg1
mainprogram.cc datastructures.cc mainwindow.cc -
DUSE_PERF_EVENT -O0 && chmod +x prg1 && rsync prg1 tie-
tiraka.it.tuni.fi: && ssh tie-tiraka.it.tuni.fi ./prg1
student_name@tie-tiraka.it.tuni.fi's password:
student_name@tie-tiraka.it.tuni.fi's password:
> perftest get_all_affiliations 20 20 1000;10000;100000
Timeout for each N is 20 sec.
For each N perform 20 random command(s) from:
get_all_affiliations

     N , add (sec) ,   add (count) ,   cmds (sec) , cmds
(count) ,  total (sec) , total (count)
  1000 , 0.0104188 ,      53499498 ,   0.00217933 ,
12074891 ,    0.0125981 ,     65574389
 10000 ,  0.108456 ,     534380344 ,    0.0227614 ,
 120435339 ,     0.131217 ,     654815683
100000 ,    1.2738 ,    5387425190 ,     0.345232 ,
1204036047 ,      1.61903 ,     6591461237
>
```

In Plussa's prg1 grading system (specifically perfestimate-grader), 150 optimization points are available. The functions subject to optimization are:

1. get_affiliations_alphabetically
2. get_affiliations_distance_increasing
3. find_affiliation_with_coord

There are two testing approaches for optimization points:

1. The approach for functions get_affiliations_alphabetically and get_affiliations_distance_increasing.
2. Another approach for the function find_affiliation_with_coord.

For both testing approaches, we calculate what we call the "optimization percentage," which is a number between -∞% and 100%:

- To be considered optimized, the performance should, on average, be more efficient than single executions. A more efficient execution has a lower command counter value.

The optimization percentage helps determine if a function has been optimized. When the optimization percentage is significantly negative, it means that a single execution is more efficient than the average of multiple executions. Conversely, when the percentage is significantly positive, it indicates that a single execution is less efficient than the average performance.

Optimization points are granted if the optimization percentage is greater than 2%.

Note: The values of command counters may vary depending on the computer, and optimization testing should be conducted without any compiler optimizations.

Efficiency Testing of Sorting Functions

The efficiency of sorting functions, get_affiliations_alphabetically and get_affiliations_distance_increasing, is tested by

The calculation of the optimization percentage is based on command counters and works both locally and in Plussa as follows:

Optimization Percentage = $\frac{command\_count\_1 - \frac{command\_count\_20}{20}}{command\_count\_1} \cdot 100\%$

In this formula, command_count_1 represents the command count when repeat_count=1, and command_count_20 represents the command count when repeat_count=20.

Testing for find_affiliation_with_coord

Optimization testing for find_affiliation_with_coord operates somewhat differently because the function's efficiency can depend on the execution of the get_affiliations_distance_increasing function. After running get_affiliations_distance_increasing, find_affiliation_with_coord may perform more efficiently.

In the test file prg1/optimization_test/02-find_affiliation_with_coord.txt, two essential values for command counters and stopwatch measurements are printed for calculating the optimization percentage:

1. Command counter after a single function call.
2. Command counter after the subsequent nineteen function calls.

After the first function call, the program executes get_affiliations_distance_increasing. The command count used during this execution is not considered when calculating the optimization points for find_affiliation_with_coord. Only after this step are the next nineteen calls executed.

The optimization percentage calculation for this scenario is based on command counters and works both locally and in Plussa as follows:

Optimization Percentage = $\frac{command\_count\_1 - \frac{command\_count\_1 + command\_count\_19}{20}}{command\_count\_1} \cdot 100\%$

to optimize find_affiliation_with_coord even without it.

Utilizing Artificial Intelligence in Coursework

We follow the TUNI guidelines: https://www.tuni.fi/en/students-guide/handbook/uni/studying-0/academic-integrity-students/use-ai-based-applications

If a student has used artificial intelligence in their coursework, the student must report the usage in the file datastructures.hh as follows:

```
// Datastructures.hh

//

// Student name:

// Student email:

// Student number:

// AI tool used and the version:

// The purpose of the use:

// The part(s) where it was utilized:
```