Visualization tool for multiple genome comparison

PROGRAMMING GUIDE

Introduction

This short programming guide is set up for the Summer School of Mr. SymBioMath project, held from the 15th of September, 2014, in Munich. Our purpose is to show during a programming workshop how the visualization application, which has been created in the scope of the project to display comparative genomics data, can be extended with self-defined modules. First we will learn how we can add interactive filters to the existing ones to eliminate the undesired high scoring sequence pairs (HSPs) from the incoming data. Then it will be discussed, how we can create methods to select HSPs and perform evolutionary events on them.

Developer tools

The application uses *OpenGL* to display the data and *Qt*[1] to define its graphical user interface. Both tools are open sourced and provide multiplatform program development: The code can be compiled on several platforms, such as Windows, Linux or Mac, without any changes.

The application layout has been created with *Qt Creator* from the *v5.3 Qt Bundle*[2], while *Visual Studio Express Edition* was used for the coding.

At the programming workshop MinGW g++ compiler is used with QT Creator for the sake of easy coding, as it lets us to use the Qt Creator as an all-in-one API, however this solution lacks the flexibility of Visual Studio.

[1] Qt-project.org

[2] Recommended versions on qt-project.org/download:
Qt 5.3.1 for Windows XX-bit (MinGW 4.8.2, OpenGL)
Qt 5.3.1 for Windows XX-bit (VS 20xx, OpenGL)
Qt Online Installer for Linux XX-bit
Qt Online Installer for Mac

Some information about the program code

In this section the program elements are enumerated, which are important for understanding the functioning of the program. Based on them we can define the data flow from the loading of the genome comparison data till the displaying of it.

Structures and definitions (structures.h)

```
struct FragFile //describes a single HSP, the main unit of .frag files
{
	uint32_t diag;
	uint32_t xIni;
	uint32_t yIni;
	uint32_t xFin;
	uint32_t yFin;
	uint32_t length;
	uint32_t ident;
	uint32_t score;
	float similarity;
	uint32_t seqX;		//sequence number in the 'X' file
	uint32_t seqY;		//sequence number in the 'Y' file
	int block;		//synteny block id
	char strand;		//'f' for the forward strain and 'r' for the
				reverse
};

struct ComparisonResults //contains the comparison results between two
sequences
{
	bool selfCompared; //when the two sequences are identical
	string referenceSequenceName;
	string querySequenceName;
	uint32_t nx;
	uint32_t ny;
	vector<FragFile> fragments;
};

struct DotplotInfo //gives information about the dot plots on the screen
{
	string name;
	glm::vec3 color;
	int position;
	bool visible;
	bool isTheReference; //if it displays the self-compared reference
	                       sequence
	bool cloned; //if it's created by performing evolutionary events
	uint32_t nx;
	uint32_t ny;
};

//the following search modes can be given at the searching of HSPs
//more than one modes can be joined by the use of | (logical OR)
#define SEARCH_X 1 //search in the x (reference sequence) direction
#define SEARCH_Y 2 //search in the y (query sequence) direction
#define EXACT_SEARCH 4
#define SEARCH_OVERHANGING_FRAGMENTS 8
#define SEARCH_IN_ALL_GENOMES 16
#define EXTEND_EXISTING_HITS 32 //logical OR
#define TIGHTEN_EXISTING_HITS 64 //logical AND
```

Important classes and member functions

```
class FileIO
//manages the data import
public:
      int SelectFiles(vector<string>& selectedFastaFiles);
      //according to the given vector of .fasta file names, it localizes
      the sequence, comparison and annotation data on the storage unit
      Void GetComparisonResults(int reference, vector<ComparisonResults>&
      comparisons);
      //fills up the comparisons vector with the comparison data according
      to the given reference sequence

Class Modifiers
//provides functions to make modifications on the data (e.g. filters)
public:
      void Enable(bool en);
      //set to enable/disable the filters
      void ProcessModifier(vector<ComparisonResults>& source,
      vector<ComparisonResults>& modified);
      //processes a modifier, reading from the source, writing to the
      modified vector

Class Visualization
//manages the displaying of the comparison data
public:
      void CreateDotplots(int referenceGenome, vector<ComparisonResults>&
      comparisons, bool keepSettings, bool cancelLastEvent = false);
      //creates the dot plots on the screen from the given comparison
      results
      int NumOfDotplots();
      //gives back the number of created dot plots
      void GetDotplotInfo(int i, DotplotInfo& dotplotInfo);
      //gives back information about the ith dot plot (see struct
      DotplotInfo)
      void SelectFragments(int dotplotID, FragFile searchTerms , char
      mode);
      //selects HSPs that fit the searchTerms according to the search mode
      (see modes in the last section)
      bool RedoInversion(DotplotInfo& newDotplotInfo);
      //redoes an inversion event on the selected HSPs and gives back
      information about the created dot plot
      bool RedoTranslocation(DotplotInfo& newDotplotInfo, uint32_t origin =
      UINT32_T_MAX);
      //redoes a translocation event on the selected HSPs and gives back
      information about the created dot plot


Class VisualizationGLWidget
// manages the functioning of the Qt widget that displays the 3d view of
HSPs
Class AnnotationsGLwidget
// manages the functioning of the Qt widget that displays the annotations
inherited public:
//for both classes
      void makeCurrent();
      //makes the Qt widget active (must be called before data changing)
      void updateGL();
      //redraws the openGL content of the Qt widget
```

Data flow

In the data flow three objects play an important role. With a `FileIO` object we import the comparison results and fill up the vector `resultsFromFileIO`. With a `Modifier` object we process filters on this vector and store the results in the vector `resultsToDisplay`. At last a `Visualization` object displays the content of this vector.

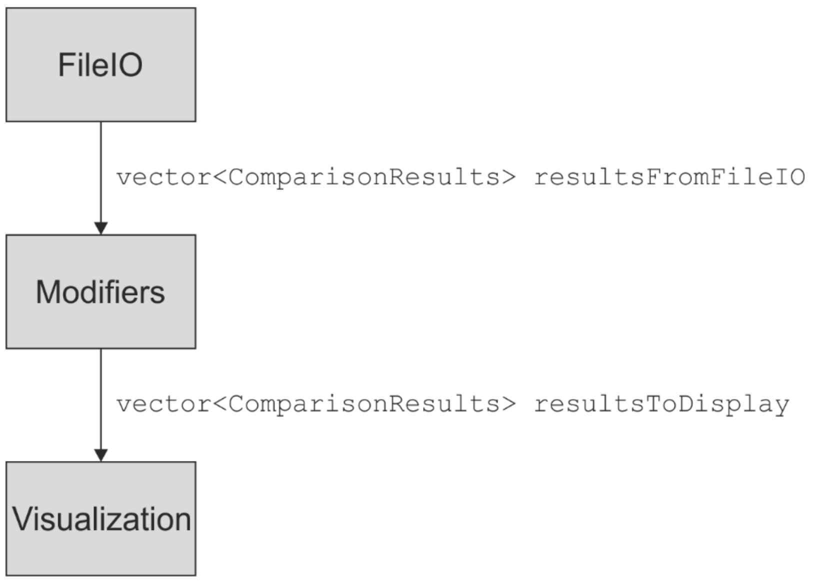
```
FileIO* fileIO;
Modifiers* modifier;
Visualization* visualization;

vector<ComparisonResults> resultsFromFileIO;
vector<ComparisonResults> resultsToDisplay;

fileIO->SelectFiles(items); //items: strings from a file open dialog
fileIO->GetComparisonResults(0, resultsFromFileIO);

modifier->Enable(true);
modifier->ProcessModifier(resultsFromFileIO, resultsToDisplay);

visualization->CreateDotplots(0, resultsToDisplay, false);
```


FileIO
vector<ComparisonResults> resultsFromFileIO
Modifiers
vector<ComparisonResults> resultsToDisplay
Visualization

Implementing interactive filters

In this section we learn how to insert a new filter into the data flow. As a simple example a filter will be defined that eliminates the HSPs whose length doesn't exceed a certain value. This value can be adjusted by the user interactively, with the use of a slider on the screen.

First we will create a dockable window, called *Dock Widget* in the Qt terminology, with a slider to adjust the length limit and a button with which we get the calculations updated according to the changed limit.

Secondly we will add a member function to the `Modifier` class that filters out the unwanted HSPs and we will change the class to set this function as the default filter. We can also learn how to access the value of the slider from the function.

Step 1: Opening and configuring the project

Open the project in **Qt Creator** by clicking on *visualization.pro* in the project directory.

Click onto the **Configure Project** button on the appearing dialog. This will set up the project environment according to the default settings.

Step 2: Adding a Dock Widget with a slider and a button

In the **Projects** view (left) open the *Forms* folder and double-click on *mainwindow.ui*. Now the **Qt Creator** changes into **Design** mode and the main layout of our application is displayed.

Drag and drop a **Dock Widget** from the **Widget box** (left) to the right side of the form, just above the *Search in Annotations* widget.

Drag and drop a **Horizontal Slider** into the widget. Change its *objectName* property in the **Property editor** (right below) to *mySlider*. Set its *maximum* property to *10000*. Set its *singleStep* property to *100*.

Drag and drop a **Push Button** below the slider onto the dock widget. Change its *text* property to *Test filter*.

To lay out the new elements on the widget in a pleasing way, right click on the dock widget and chose *Lay out* then *Lay Out Vertically* from the **pop-up menu**.

Now we have to set what should happen when we push the button. Right-click on the button, and chose **Go to slot…** from the **pop-up menu**. Chose *clicked()* from the list and press **OK**. The **Qt Creator** changes to **Editor** mode and the *mainwindow.cpp* source file appears. The cursor flashes in the *void MainWindow::on_pushButton_clicked()* function. This call-back function is activated whenever the button is clicked during the execution of the application. Our purpose is to update the filtering when the button is clicked – just like the *Filter dot plots* button on the tool bar does. The simplest method to achieve this is to redirect our function to its call-back function. Change the code as follows:

```
void MainWindow::on_pushButton_clicked()
{
    this->on_actionFilter_triggered(true);
}
```

Step 3: Creating the new filter function

First we have to add our filter as a private member function to the `Modifiers` class. The class is implemented in the *modifiers.h* and *modifiers.cpp* files. Add the following bold line to the class definition:

```
void DefaultFilter(ComparisonResults& source, ComparisonResults& modified);
void EmptyFilter(ComparisonResults& source, ComparisonResults& modified);
void MyFilter(ComparisonResults& source, ComparisonResults& modified);
```

Now let’s extend *modifiers.cpp* with the declaration of the `MyFilter` function:

```
void Modifiers::MyFilter(ComparisonResults& source, ComparisonResults&
modified)
{
      for (unsigned int i=0; i<source.fragments.size(); i++)
      {
            modified.fragments.push_back(source.fragments[i]);
      }
}
```

Finally we have to rewrite the `ProcessModifier` public member function in order to call the `MyFilter` function instead of the default filter when we process the incoming data:

```
void Modifiers::ProcessModifier(vector<ComparisonResults>& source,
vector<ComparisonResults>& modified)
{
…
            if (!enabled)
                  EmptyFilter(source[i], cr);
            else
                  MyFilter(source[i], cr);
            modified.push_back(cr);
…
}
```

Now we are ready to implement the body of our filter function. First we need the value of the Slider *mySlider*. We can invoke it in the following way:

```
QSlider *slider = parent->findChild<QSlider *>("mySlider");
uint32_t minLength = (uint32_t)slider->value();
```

For the above definition we have to include the QSlider class:

```
#include <QSlider>
```

When we have got the value, we can compare it with the length of each fragment, and keep only the fragments with the appropriate length:

```
if (minLength < source.fragments[i].length)
      modified.fragments.push_back(source.fragments[i]);
```

So the whole function looks like the following. The added lines are typed bold.

```
void Modifiers::MyFilter(ComparisonResults& source, ComparisonResults&
modified)
{
	for (unsigned int i=0; i<source.fragments.size(); i++)
	{
		QSlider *slider = parent->findChild<QSlider *>("mySlider");
		uint32_t minLength = (uint32_t)slider->value();
		if (minLength < source.fragments[i].length)
			modified.fragments.push_back(source.fragments[i]);
	}
}
```

If we build and run the code, we can verify the functioning of our new filter.

Selecting HSPs and performing evolutionary events on them

In this section we learn how we can write such a code snippet with which we can select HSPs and perform evolutionary events on them.

Let's begin with adding another push button to the application that will start our program code when we click on it. Change to **Design** mode by clicking on *mainwindow.ui* in the **Projects** view. Drag and drop a **Push Button** below the other one onto the dock widget. Change its *text* property to *Test events*. Right-click on the button, and chose **Go to slot…** from the **pop-up menu**. Chose *clicked()* from the list and press **OK**. The *void MainWindow::on_pushButton_2_clicked()* call-back function appears in the editor. Add our code to the body of this function.

Selecting HSPs is very simple. Just define the search terms in a `FragFile` variable, choose the appropriate search mode and call the `SelectFragments` member function of the `visualization` object. In the example we select HSPs that are located in the 0th dot plot between the positions 50000 and 80000 of the query sequence.

```
FragFile st;
st.yIni = 50000;
st.yFin = 80000;
visualization->SelectFragments(0, st, SEARCH_Y);
```

Performing an inversion event (more exactly *redoing* the event) on the selected HSPs is not difficult either. Just call the `RedoInversion` function that processes the event and gives back information about the newly created dot plot.

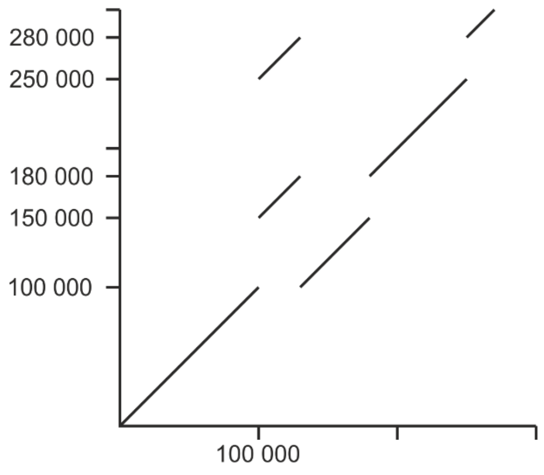
```
DotplotInfo dpi;
visualization->RedoInversion(dpi);
```

Only two things are missing. Before we create new dot plots, we must make the `visualizationGLWidget` active with the `makeCurrent` function. After the changings, in order to display the new dot plots, the `updateGL` function must be called. Our function should look like the following.

```
void MainWindow::on_pushButton_2_clicked()
{
    visualizationGLWidget->makeCurrent();
    FragFile st;
    st.yIni = 50000;
    st.yFin = 80000;
    visualization->SelectFragments(0, st, SEARCH_Y);
    visualizationGLWidget->updateGL();
}
```

After making and running the application, let's load some comparison results and click onto the **Test events** button to perform the event.

Now let's redo a translocation event: restore the original state after a combined translocation, when a sequence piece of 30000 base pairs at the position 100,000 duplicated and appeared at two several positions (see the figure below).



We can do it easy: Select the two regions and call the `RedoTranslocation` function. We perform the event on the recently created dot plot, which is the last in the vector of dot plots. We select the first region with a simple selection, then we add the HSPs of the second region with the `EXTEND_EXISTING_HITS` search mode.

```
int lastDotplot = visualization->NumOfDotplots()-1;
st.yIni = 150000;
st.yFin = 180000;
visualization->SelectFragments(lastDotplot, st, SEARCH_Y);
st.yIni = 250000;
st.yFin = 280000;
visualization->SelectFragments(lastDotplot, st, SEARCH_Y |
EXTEND_EXISTING_HITS);
visualization->RedoTranslocation(dpi, 100000);
```

Our code should look like:

```
void MainWindow::on_pushButton_2_clicked()
{
    visualizationGLWidget->makeCurrent();
    FragFile st;
    st.yIni = 50000;
    st.yFin = 80000;
    visualization->SelectFragments(0, st, SEARCH_Y);
    DotplotInfo dpi;
    visualization->RedoInversion(dpi);

    int lastDotplot = visualization->NumOfDotplots()-1;
    st.yIni = 150000;
    st.yFin = 180000;
    visualization->SelectFragments(lastDotplot, st, SEARCH_Y);
    st.yIni = 250000;
    st.yFin = 280000;
    visualization->SelectFragments(lastDotplot, st, SEARCH_Y |
EXTEND_EXISTING_HITS);
    visualization->RedoTranslocation(dpi, 100000);

    visualizationGLWidget->updateGL();
}
```

Let’s test our code with starting the application. Note that these events are fictitious, so don’t expect real results.