About

This document contains examples on using some functions of CBioInfCpp.h library.

This document is distributed under Creative Commons Attribution 4.0 International Public License (CC BY) (hyperlink to the License: https://creativecommons.org/licenses/by/4.0/legalcode.ru).

Written by Sergey Chernouhov (chernouhov@rambler.ru).

Code for all Exercises is also represented at StudyingCBioInfCpp.cpp.

Exercise 1. Srings

Reading strings from a file, finding the shortest superstring (greedy algorithm).
Generating reverse complement of a string, calculating its GC-content, computing Hamming distance between two strings.
Align 2 strings (edit distance alignment),
Finding in a string all its substrings that have edit distance to a given one <=1.

First of all we should to include CBioInfCpp to our code. Place the file CBioInfCpp.h to the place where others libs to be included are. Or write the whole path to CBioInfCpp.h after #include.

So our code will start its building as follows:

```
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{

    return 0;
}
```

We are ready. Let's read a collection of strings from a file. To do so one should download In1(strings).txt to the project's folder or, another way, to any place (in this case one should write the whole path to the file for ifstream).
We shall read strings to std::vector DataS. The results of doing this exercise we shall out both to screen and to file Out1(strings).txt. So le's start. And to check the strings' reading we may using **VectorCout** (to out the result to screen) and **VectorFout** (to out to file):

```
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;
```

```cpp
int main()
{
    vector <string> DataS;
    // Container for strings to be read from file In1(strings).txt
    DataS.clear();
    ifstream fin1 ("In1(strings).txt");
    // If you place file not to "standart place" please write the entire
    path
    ofstream fout1 ("Out1(strings).txt"); // File to write results for the
    Exercise 1.

    StringsRead(fin1, DataS); //Reading strings to vector DataS
    VectorCout(DataS);  // Out DataS to screen
    VectorFout(DataS, fout1); // Out dataS to file

    fin1.close();
    fout1.close();

    return 0;
}
```

I hope, all it's ok. Now let's add two empty strings to our DataS. And after we are going to apply greedy algo of min-length superstring finding using the function **ShortestSuperstring**. Let's do it.

```cpp
int main()
{
    vector <string> DataS;
    // Container for strings to be read from file In1(strings).txt
    DataS.clear();
    ifstream fin1 ("In1(strings).txt");
    // If you place file not to "standard place" please write the entire
    path
    ofstream fout1 ("Out1(strings).txt"); // File to write results for the
    Exercise 1.

    StringsRead(fin1, DataS); //Reading strings to vector DataS
    VectorCout(DataS);  // Out DataS to screen
    VectorFout(DataS, fout1); // Out DataS to file

    DataS.push_back(""); //adding empty string
    DataS.push_back(""); //another one
    string ShortestSuperstring = ShortSuperstringGr(DataS);
    // Generating shortest supersring using greedy algorithm
    cout<<"ShortestSuperstring: "<<ShortestSuperstring<<endl; // out it to
    screen
    fout1<<"ShortestSuperstring: "<<ShortestSuperstring<<endl; // out it
    too file

    fin1.close();
    fout1.close();

    return 0;
}
```

See it? And now some bioinformatics. We'll find reverse complement (the function **rp**),compute GC-content, and transcribe DNA to RNA:

```cpp
int main()
{
    vector <string> DataS;
    // Container for strings to be read from file In1(strings).txt
    DataS.clear();
```

```cpp
        ifstream fin1 ("In1(strings).txt");
        // If you place file not to "standard place" please write the entire
        path
        ofstream fout1 ("Out1(strings).txt"); // File to write results for the
        Exercise 1.

        StringsRead(fin1, DataS); //Reading strings to vector DataS
        VectorCout(DataS);  // Out DataS to screen
        VectorFout(DataS, fout1); // Out DataS to file

        DataS.push_back(""); //adding empty string
        DataS.push_back(""); //another one
        string ShortestSuperstring = ShortSuperstringGr(DataS);
        // Generating shortest superstring using greedy algorithm
        cout<<"ShortestSuperstring: "<<ShortestSuperstring<<endl; // out it to
        screen
        fout1<<"ShortestSuperstring: "<<ShortestSuperstring<<endl; // out it
        too file

        cout<<"Reverse complement of ShortestSuperstring:"
        <<rp(ShortestSuperstring)<<endl;
        // generating and printing to screen reverse complement of
        ShortestSuperstring
        fout1<<"Reverse complement of ShortestSuperstring: "
        <<rp(ShortestSuperstring)<<endl;
        // generating and printing to file reverse complement of
        ShortestSuperstring

        cout<<"GC-content of Reverse complement of ShortestSuperstring: "
        <<gcDRNA (rp(ShortestSuperstring))<<endl;
        // calculating GC-content of this reverse complement (to screen)
        fout1<<"GC-content of Reverse complement of ShortestSuperstring: "
        <<gcDRNA(rp(ShortestSuperstring))<<endl; // calculating GC-content of
        this reverse complement (to file)

        string RNAofShortestSuperstring;
        RNAfromDNA (ShortestSuperstring, RNAofShortestSuperstring); // now let's
        generate RNA string  upon ShortestSuperstring and write it to
        RNAofShortestSuperstring
        cout<<"RNAofShortestSuperstring: "<<RNAofShortestSuperstring<<endl;
        fout1<<"RNAofShortestSuperstring: "<<RNAofShortestSuperstring<<endl;

        fin1.close();
        fout1.close();

        return 0;
}
```

Now it's time co evaluate Hamming distance between ShortestSuperstring and RNAofShortestSuperstring, to cut the last 2 symbols from RNAofShortestSuperstring then align RNAofShortestSuperstring and RNAofShortestSuperstring (Edit Distance Alignment). Furthermore, we'll find in ShortestSuperstring all its substrings, that have edit distance to "ACA" <=1 (function **FindMutatedRepeatsED**).

```cpp
int main()
{
        vector <string> DataS;
        // Container for strings to be read from file In1(strings).txt
        DataS.clear();
        ifstream fin1 ("In1(strings).txt");
        // If you place file not to "standard place" please write the entire
        path
```

```cpp
    ofstream fout1 ("Out1(strings).txt"); // File to write results for the
    Exercise 1.

    StringsRead(fin1, DataS); //Reading strings to vector DataS
    VectorCout(DataS);   // Out DataS to screen
    VectorFout(DataS, fout1); // Out DataS to file

    DataS.push_back(""); //adding empty string
    DataS.push_back(""); //another one
    string ShortestSuperstring = ShortSuperstringGr(DataS);
    // Generating shortest superstring using greedy algorithm
    cout<<"ShortestSuperstring: "<<ShortestSuperstring<<endl; // out it to
    screen
    fout1<<"ShortestSuperstring: "<<ShortestSuperstring<<endl; // out it
    too file

    cout<<"Reverse complement of ShortestSuperstring:"
    <<rp(ShortestSuperstring)<<endl;
    // generating and printing to screen reverse complement of
    ShortestSuperstring
    fout1<<"Reverse complement of ShortestSuperstring: "
    <<rp(ShortestSuperstring)<<endl;
    // generating and printing to file reverse complement of
    ShortestSuperstring

    cout<<"GC-content of Reverse complement of ShortestSuperstring: "
    <<gcDRNA (rp(ShortestSuperstring))<<endl;
    // calculating GC-content of this reverse complement (to screen)
    fout1<<"GC-content of Reverse complement of ShortestSuperstring: "
    <<gcDRNA(rp(ShortestSuperstring))<<endl; // calculating GC-content of
    this reverse complement (to file)

    string RNAofShortestSuperstring;
    RNAfromDNA (ShortestSuperstring, RNAofShortestSuperstring); // now let's
    generate RNA string  upon ShortestSuperstring and write it to
    RNAofShortestSuperstring
    cout<<"RNAofShortestSuperstring: "<<RNAofShortestSuperstring<<endl;
    fout1<<"RNAofShortestSuperstring: "<<RNAofShortestSuperstring<<endl;

    let's calculate Hamming distance between ShortestSuperstring and
    RNAofShortestSuperstring
    cout<<"Hamming Distance between ShortestSuperstring and
    RNAofShortestSuperstring: "<<HmDist(ShortestSuperstring,
    RNAofShortestSuperstring)<<endl;
    fout1<<"Hamming Distance between ShortestSuperstring and
    RNAofShortestSuperstring: "<<HmDist(ShortestSuperstring,
    RNAofShortestSuperstring)<<endl;

    RNAofShortestSuperstring.pop_back();
    // Deleting the last symbol from RNAofShortestSuperstring
    RNAofShortestSuperstring.pop_back();
    // Another time

    string sr1, sr2;
    // These strings will contain results of Alignment (minimizing Edit
    Distance)
    cout<<"Edit Distance between ShortestSuperstring and
    RNAofShortestSuperstring: "<<EditDistA(ShortestSuperstring,
    RNAofShortestSuperstring, sr1, sr2)<<endl;
    fout1<<"Edit Distance between ShortestSuperstring and
    RNAofShortestSuperstring: "<<EditDistA(ShortestSuperstring,
    RNAofShortestSuperstring, sr1, sr2)<<endl;
    cout<<"Edit Distance Alignment: "<<endl<<sr1<<endl<<sr2<<endl;
    fout1<<"Edit Distance Alignment: "<<endl<<sr1<<endl<<sr2<<endl;
```

```
// Now let's find in the string ShortestSuperstring all its substrings
that have edit distance to "ACA" <=1.
std::set <std::pair <int, int>> Result;
// The function FindMutatedRepeatsED returns results as set of pairs
there the first one is a starting position of substring in the string,
and the second is its lenght

FindMutatedRepeatsED ("ACA", ShortestSuperstring, 1, Result); //done
cout<<"Substrings of ShortestSuperstring that have EditDistance to
string ACA <=1 :"<<endl;
fout1<<"Substrings of ShortestSuperstring that have EditDistance to
string ACA <=1 :"<<endl;

for (auto it=Result.begin(); it!=Result.end(); it++)
    {
        cout<<ShortestSuperstring.substr((*it).first,
        *it).second)<<endl; //outing substrings to screen
        fout1<<ShortestSuperstring.substr((*it).first,
        (*it).second)<<endl;//outing substrings to file
    }


    fin1.close();
    fout1.close();

    return 0;
}

See the result in Out1(strings).txt.
```

Exercise 2. Reading graphs from file, outing graphs to screen and to file, finding Eulerian Cycle

If we start from the very beginning we should include CBioInfCpp as it has been said above. Then we'll have this:

```
Код:
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{

    return 0;
}
```

Lel's read from files: unweighted graph A1, weighted graph (weights are int) A2, weighted graph (weights are double) A3. Note that A1 and A2 may be read to vector <int> and A3 – to pair < vector<int>, vector<double>> A3.
Also note that numbers assigned to vertices may be negative too.

To start we should download files that contain these graphs (function **UWGraphRead** for unweighted graph, function **WGraphRead** for weighted graph). We should place these files to the project's folder or, another way, to any place (in this case one should write the whole path to the file for ifstream). These are required files:

InUnWeightedGraph.txt; // An unweighted graph is here (edge list format)
InWeightedGraphIntegers.txt // A weighted graph is here (edge list format, weights are integers)
InWeightedGraphDoubles.txt // A weighted graph is here (edge list format, weights are double)

After we may check ourselves using functions **GraphCout** (to out graph to screen) and **GraphFout** (to out graph to file).

```cpp
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{
      ifstream fin2_1 ("InUnWeightedGraph.txt"); // An unweighted graph is
      here (edge list format)
      ifstream fin2_2 ("InWeightedGraphIntegers.txt"); //  A weighted graph
      is here (edge list format, weights are integers)
      ifstream fin2_3 ("InWeightedGraphDoubles.txt"); // A weighted graph is
      here (edge list format, weights are double)
      ofstream fout2 ("Out2(Graphs1).txt"); // File to write results for the
      Exercise 2.

      vector <int> A1; //Container for unweighted graph A1
      UWGraphRead(fin2_1, A1);
      cout<<"Here is unweighted graph A1: "<<endl;
      GraphCout(A1, false); //Out graph to screen, as it is unweighted the
      second parameter should be "false"
      fout2<<"Here is unweighted graph A1: "<<endl;
      GraphFout(A1, false, fout2); //Out graph to file, as it is unweighted
      the second parameter should be "false"

      vector <int> A2; //Container for weighted graph A2, weights are
      integers
      WGraphRead(fin2_2, A2);
      cout<<"Here is weighted graph A2: "<<endl;
      GraphCout(A2, true); //Out graph to screen, as it is weighted the
      second parameter should be "true"
      fout2<<"Here is weighted graph A2: "<<endl;
      GraphFout(A2, true, fout2); //Out graph to file, as it is weighted the
      second parameter should be "true"

      pair < vector<int>, vector<double>> A3; //Container for unweighted
      graph A3, weights are double
      WGraphRead(fin2_3, A3);
      cout<<"Here is weighted graph A3: "<<endl;
      GraphCout(A3); //Out graph to screen, as it is set by pair A3 it may be
      weighted only
      fout2<<"Here is weighted graph A3: "<<endl;
      GraphFout(A3, fout2); //Out graph to file, as it is set by pair A3 it
      may be weighted only

      fout2.close();
      fin2_1.close();
      fin2_2.close();
      fin2_3.close();


    return 0;
```

```
}
```

Now we know how to read graphs and to out them to screen/ file. As the A1, A2, A3 differs one from another by edge's weight only, their Eulerian cycles/ paths should be the same, aren't they? Let's check it using **EPathDGraph** (note that it considers input graphs as <u>directed</u>). Also note that **EPathDGraph** will fill for us 2 vectors, i.e. vector R (Eulerian cycle (in this case the function returns 1)/ path (and in this – it returns 2) itself) and vector I (it will contain all isolated vertices). If there is no Eulerian cycle/ path or input data are incorrect the function returns -1 and empty R and I.

We should also remember that vertices may have negative numbers assigned.

```cpp
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{
    ifstream fin2_1 ("InUnWeightedGraph.txt"); // An unweighted graph is
    here (edge list format)
    ifstream fin2_2 ("InWeightedGraphIntegers.txt"); //  A weighted graph
    is here (edge list format, weights are integers)
    ifstream fin2_3 ("InWeightedGraphDoubles.txt"); // A weighted graph is
    here (edge list format, weights are double)
    ofstream fout2 ("Out2(Graphs1).txt"); // File to write results for the
    Exercise 2.

    vector <int> A1; //Container for unweighted graph A1
    UWGraphRead(fin2_1, A1);
    cout<<"Here is unweighted graph A1: "<<endl;
    GraphCout(A1, false); //Out graph to screen, as it is unweighted the
    second parameter should be "false"
    fout2<<"Here is unweighted graph A1: "<<endl;
    GraphFout(A1, false, fout2); //Out graph to file, as it is unweighted
    the second parameter should be "false"

    vector <int> A2; //Container for weighted graph A2, weights are
    integers
    WGraphRead(fin2_2, A2);
    cout<<"Here is weighted graph A2: "<<endl;
    GraphCout(A2, true); //Out graph to screen, as it is weighted the
    second parameter should be "true"
    fout2<<"Here is weighted graph A2: "<<endl;
    GraphFout(A2, true, fout2); //Out graph to file, as it is weighted the
    second parameter should be "true"

    pair < vector<int>, vector<double>> A3; //Container for unweighted
    graph A3, weights are double
    WGraphRead(fin2_3, A3);
    cout<<"Here is weighted graph A3: "<<endl;
    GraphCout(A3); //Out graph to screen, as it is set by pair A3 it may be
    weighted only
    fout2<<"Here is weighted graph A3: "<<endl;
    GraphFout(A3, fout2); //Out graph to file, as it is set by pair A3 it
    may be weighted only

    vector <int> R; //Container for Eulerian cycle/path to be found
    vector <int> I; //Container for isolated vertices to be found
    cout<<EPathDGraph (A1, R, false, I)<<endl; //as A1 is unweighted we
    should set 3d parameter as "false"
    fout2<<EPathDGraph (A1, R, false, I)<<endl;
```

```cpp
// EPathDGraph returns value "1" if Eulerian cycle has been found or
value "2" if Eulerian path has been found
// or "-1" together with empty R and Isolates if no cycle/ path found.
//Note that EPathDGraph considers graps as directed ones.
cout<< "Eulerian cycle of graph A1: "<<endl;
VectorCout(R); // vector R - to screen
cout<< "Isolated vertices of graph A1: "<<endl;
VectorCout(I); // vector I - to screen
fout2<< "Eulerian cycle of graph A1: "<<endl;
VectorFout(R, fout2); // vector R - to file
fout2<< "Isolated vertices of graph A1: "<<endl;
VectorFout(I, fout2); // vector I - to file

cout<<EPathDGraph (A2, R, true, I)<<endl; //as A1 is unweighted we
should set 3d parameter as "true"
fout2<<EPathDGraph (A2, R, true, I)<<endl;
// EPathDGraph returns value "1" if Eulerian cycle has been found or
value "2" if Eulerian path has been found
// or "-1" together with empty R and Isolates if no cycle/ path found.
cout<< "Eulerian cycle of graph A2: "<<endl;
VectorCout(R); // vector R - to screen
cout<< "Isolated vertices of graph A2: "<<endl;
VectorCout(I); // vector I - to screen
fout2<< "Eulerian cycle of graph A2: "<<endl;
VectorFout(R, fout2); // vector R - to file
fout2<< "Isolated vertices of graph A2: "<<endl;
VectorFout(I, fout2); // vector I - to file

cout<<EPathDGraph (A3, R, I)<<endl; //as A3 is set by pair it may be
only weighted
fout2<<EPathDGraph (A3, R, I)<<endl;
// EPathDGraph returns value "1" if Eulerian cycle has been found or
value "2" if Eulerian path has been found
// or "-1" together with empty R and Isolates if no cycle/ path found.
cout<< "Eulerian cycle of graph A3: "<<endl;
VectorCout(R); // vector R - to screen
cout<< "Isolated vertices of graph A3: "<<endl;
VectorCout(I); // vector I - to screen
fout2<< "Eulerian cycle of graph A3: "<<endl;
VectorFout(R, fout2); // vector R - to file
fout2<< "Isolated vertices of graph A3: "<<endl;
VectorFout(I, fout2); // vector I - to file


fout2.close();
fin2_1.close();
fin2_2.close();
fin2_3.close();


    return 0;
}
```

The result of Ex.2 we may see in Out2(Graphs1).txt

Exercise 3. Checking if 2 graphs are isomorphic; finding all subgraphs that are isomorphic to given template graph

Now let's turn to
graph isomorphism problem

problem of finding of all subgraphs in a graph A that are isomorphic given template graph B. CBioInfCpp have the function **SubGraphsInscribed** for it. Some info on evolution of this function and its testing:
https://github.com/chernouhov/CBioInfCpp-0-/tree/master/TestsGraphIsomorphismInfo

https://github.com/chernouhov/CBioInfCpp-0-/tree/master/TestsIsomorphicSubGraphsFinding

https://doi.org/10.24108/preprints-3111977

So let's start as above:

Код:
```cpp
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{

    return 0;
}
```

We should read "more larger" unweigted graph A from a file InA.txt and template graph B1 from InB1.txt. As we have already known we should use the function **UWGraphRead** for it. But before we should download these files to the project's directory or to some other place (in this case one should write the whole path to the file for ifstream).

Also we'll create graph B2 as graph B1 but by adding 2 to number of each its vertex (we may do so using the function **RenumVGraph).** So B2 will be isomorphic to B1 and we will check it too:

```cpp
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{
    vector <int> A; // Container for unweighted graph A ("more large one")
    is here (edge list format)
    vector <int> B1; // Container for unweighted graph B1 is here (edge
    list format)
    vector <int> B2; // Container for unweighted graph B2 is here (edge
    list format)
    ifstream fin3_A ("InA.txt"); // Unweighted graph A is here (edge list
    format)
    ifstream fin3_B ("InB1.txt"); // Unweighted graph B1 is here (edge list
    format)
    ofstream fout3 ("Out3(Graphs2).txt"); // File to write results for the
    Exercise 3.
    UWGraphRead(fin3_A, A); // Reading unweighted graph A from a file
    UWGraphRead(fin3_B, B1); // Reading unweighted graph B1 from a file
    B2=B1; // Let's graph B2=B1
    RenumVGraph(B2,2, false); // Rename all vertices of B2: let's add 2 to
    their numbers; the 3d parameter is set as false as graph is unweighted.

    cout<<"Here is unweighted graph B1: "<<endl;
```

```cpp
        GraphCout(B1, 0); //Out graph to screen, as it is unweighted the second
        parameter should be "false" i.e. 0
        cout<<"Here is unweighted graph B2: "<<endl;
        GraphCout(B2, 0); //Out graph to screen, as it is unweighted the second
        parameter should be "false" i.e. 0

        fout3<<"Here is unweighted graph B1: "<<endl;
        GraphFout(B1, 0, fout3); //Out graph to file, as it is unweighted the
        second parameter should be "false" i.e. 0
        fout3<<"Here is unweighted graph B2: "<<endl;
        GraphFout(B2, 0, fout3); //Out graph to file, as it is unweighted the
        second parameter should be "false" i.e. 0

    return 0;
}
```

Oh, now let's check if B1 is isomorphic to B2. We'll check it for two cases:
graphs are directed and graphs are undirected. We may set this condition via
bool directed as a parameter of the function **SubGraphsInscribed**.
The function **SubGraphsInscribed** returns the number of subgraphs nave been
found; the subgraphs themselves will be contained here in set<vector <int>>
SubGraphs. If input graphs are isomorphic ut will find the only subgraph –
the "larger" graph itself.

So, let's try:

```cpp
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{
        vector <int> A; // Container for unweighted graph A ("more large one")
        is here (edge list format)
        vector <int> B1; // Container for unweighted graph B1 is here (edge
        list format)
        vector <int> B2; // Container for unweighted graph B2 is here (edge
        list format)
        ifstream fin3_A ("InA.txt"); // Unweighted graph A is here (edge list
        format)
        ifstream fin3_B ("InB1.txt"); // Unweighted graph B1 is here (edge list
        format)
        ofstream fout3 ("Out3(Graphs2).txt"); // File to write results for the
        Exercise 3.
        UWGraphRead(fin3_A, A); // Reading unweighted graph A from a file
        UWGraphRead(fin3_B, B1); // Reading unweighted graph B1 from a file
        B2=B1; // Let's graph B2=B1
        RenumVGraph(B2,2, false); // Rename all vertices of B2: let's add 2 to

        their numbers; the 3d parameter is set as false as graph is unweighted.

        cout<<"Here is unweighted graph B1: "<<endl;
        GraphCout(B1, 0); //Out graph to screen, as it is unweighted the second
        parameter should be "false" i.e. 0
        cout<<"Here is unweighted graph B2: "<<endl;
        GraphCout(B2, 0); //Out graph to screen, as it is unweighted the second
        parameter should be "false" i.e. 0

        fout3<<"Here is unweighted graph B1: "<<endl;
        GraphFout(B1, 0, fout3); //Out graph to file, as it is unweighted the
        second parameter should be "false" i.e. 0
        fout3<<"Here is unweighted graph B2: "<<endl;
```

```cpp
        GraphFout(B2, 0, fout3); //Out graph to file, as it is unweighted the
        second parameter should be "false" i.e. 0

        set<vector <int>> SubGraphs; // Here will be subgraphs found

        bool directed = true; //lets consider both graphs as directed ones
        cout<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;
        //trying to find in B1 all subgraphs that are isomorphic to B2
        fout3<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;
        // it should be 1 as they are isomorphic themselves

        for (auto it = SubGraphs.begin(); it!= SubGraphs.end(); it++)  // now
        let's out all found (sub)graphs:
        {
          cout<<"isomorphic (sub)graph found (directed graph case):"<<"
         "<<(*(it)).size()/2<<" edges"<<endl; // to screen
          GraphCout(*(it),false);

          fout3<<"isomorphic (sub)graph found: (directed graph case)"<<"
         "<<(*(it)).size()/2<<" edges"<<endl; // to file
          GraphFout(*(it),false, fout3);
        }

        directed = 0; //lets consider both graphs as undirected ones
        cout<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;
        fout3<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;

        for (auto it = SubGraphs.begin(); it!= SubGraphs.end(); it++)
        {
          cout<<"isomorphic (sub)graph found (undirected graph case):"<<"
         "<<(*(it)).size()/2<<" edges"<<endl;
          GraphCout(*(it),false);

          fout3<<"isomorphic (sub)graph found (directed graph case):"<<"
         "<<(*(it)).size()/2<<" edges"<<endl;
          GraphFout(*(it),false, fout3);
         }

    return 0;
}
```

Now we may find in the graph A all its subgraphs that are isomorphic to template graph B1. We should consider the case both graphs are directed (8 subgraphs will be found) and the case both graphs are undirected (24 subgraphs will be found).

Here is the code for the whole Ex.3:

```cpp
#include <CBioInfCpp.h>
// Including CBioInfCpp
// Подключаем библиотеку CBioInfCpp

using namespace std;

int main()
{
        vector <int> A; // Container for unweighted graph A ("more large one")
        is here (edge list format)
        vector <int> B1; // Container for unweighted graph B1 is here (edge
        list format)
        vector <int> B2; // Container for unweighted graph B2 is here (edge
        list format)
        ifstream fin3_A ("InA.txt"); // Unweighted graph A is here (edge list
        format)
```

```cpp
ifstream fin3_B ("InB1.txt"); // Unweighted graph B1 is here (edge list
format)
ofstream fout3 ("Out3(Graphs2).txt"); // File to write results for the
Exercise 3.
UWGraphRead(fin3_A, A); // Reading unweighted graph A from a file
UWGraphRead(fin3_B, B1); // Reading unweighted graph B1 from a file
B2=B1; // Let's graph B2=B1
RenumVGraph(B2,2, false); // Rename all vertices of B2: let's add 2 to
their numbers; the 3d paremeter is set as false as graph is unweighted.

cout<<"Here is unweighted graph B1: "<<endl;
GraphCout(B1, 0); //Out graph to screen, as it is unweighted the second
parameter should be "false" i.e. 0
cout<<"Here is unweighted graph B2: "<<endl;
GraphCout(B2, 0); //Out graph to screen, as it is unweighted the second
parameter should be "false" i.e. 0

fout3<<"Here is unweighted graph B1: "<<endl;
GraphFout(B1, 0, fout3); //Out graph to file, as it is unweighted the
second parameter should be "false" i.e. 0
fout3<<"Here is unweighted graph B2: "<<endl;
GraphFout(B2, 0, fout3); //Out graph to file, as it is unweighted the
second parameter should be "false" i.e. 0

set<vector <int>> SubGraphs; // Here will be subgraphs found

bool directed = true; //lets consider both graphs as directed ones
cout<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;
//trying to find in B1 all subgraphs that are isomorphic to B2
fout3<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;
// it should be 1 as they are isomorphic themselves

for (auto it = SubGraphs.begin(); it!= SubGraphs.end(); it++)  // now
let's out all found (sub)graphs:
{
  cout<<"isomorphic (sub)graph found (directed graph case):"<<"
 "<<(*(it)).size()/2<<" edges"<<endl; // to screen
  GraphCout(*(it),false);

  fout3<<"isomorphic (sub)graph found: (directed graph case)"<<"
 "<<(*(it)).size()/2<<" edges"<<endl; // to file
  GraphFout(*(it),false, fout3);
}

directed = 0; //lets consider both graphs as undirected ones
cout<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;
fout3<<SubGraphsInscribed (B1, B2, SubGraphs, directed, 0, 1, 0)<<endl;

for (auto it = SubGraphs.begin(); it!= SubGraphs.end(); it++)
{
  cout<<"isomorphic (sub)graph found (undirected graph case):"<<"
 "<<(*(it)).size()/2<<" edges"<<endl;
  GraphCout(*(it),false);

  fout3<<"isomorphic (sub)graph found (directed graph case):"<<"
 "<<(*(it)).size()/2<<" edges"<<endl;
  GraphFout(*(it),false, fout3);
 }

//now let's find in A all subgraphs that are isomorphic to graph B1

//first of all let's out them
cout<<"Here is unweighted graph A: "<<endl;
```

```cpp
GraphCout(A, 0); //Out graph to screen, as it is unweighted the second
parameter should be "false" i.e. 0
cout<<"Here is unweighted graph B1: "<<endl;
GraphCout(B1, 0); //Out graph to screen, as it is unweighted the second
parameter should be "false" i.e. 0

fout3<<"Here is unweighted graph A: "<<endl;
GraphFout(A, 0, fout3); //Out graph to file, as it is unweighted the
second parameter should be "false" i.e. 0
fout3<<"Here is unweighted graph B1: "<<endl;
GraphFout(B1, 0, fout3); //Out graph to file, as it is unweighted the
second parameter should be "false" i.e. 0

// directed graph case

directed = true; //lets consider both graphs (A and B1) as directed
ones
cout<<SubGraphsInscribed (A, B1, SubGraphs, directed, 0, 1, 0)<<"
subraphs of A that are isomorphic to B1 is found:"<<endl; //trying to
find in A all subgraphs that are isomorphic to B1
fout3<<SubGraphsInscribed (A, B1, SubGraphs, directed, 0, 1, 0)<<"
subraphs of A that are isomorphic to B1 is found:"<<endl;

for (auto it = SubGraphs.begin(); it!= SubGraphs.end(); it++)  // now
let's out all found (sub)graphs:
{
  cout<<"isomorphic (sub)graph found (directed graph case):"<<"
 "<<(*(it)).size()/2<<" edges"<<endl; // to screen
  GraphCout(*(it),false); //Out graph to screen, as it is unweighted
 the second parameter should be "false"

  fout3<<"isomorphic (sub)graph found: (directed graph case)"<<"
 "<<(*(it)).size()/2<<" edges"<<endl; // to file
  GraphFout(*(it),false, fout3); //Out graph to file, as it is
 unweighted the second parameter should be "false"
}

cout<<SubGraphs.size()<<" subraphs of A that are isomorphic to B1 is
found"<<endl;
fout3<<SubGraphs.size()<<"subraphs of A that are isomorphic to B1 is
found"<<endl;

// undirected graph case
directed = 0; //lets consider both graphs (A and B1) as undirected ones
cout<<SubGraphsInscribed (A, B1, SubGraphs, directed, 0, 1, 0)<<"
subraphs of A that are isomorphic to B1 is found:"<<endl; //trying to
find in A all subgraphs that are isomorphic to B1
fout3<<SubGraphsInscribed (A, B1, SubGraphs, directed, 0, 1, 0)<<"
subraphs of A that are isomorphic to B1 is found:"<<endl;


for (auto it = SubGraphs.begin(); it!= SubGraphs.end(); it++)
{
  cout<<"isomorphic (sub)graph found (undirected graph case):"<<"
 "<<(*(it)).size()/2<<" edges"<<endl;
  GraphCout(*(it),false); //Out graph to screen, as it is unweighted
 the second parameter should be "false"

  fout3<<"isomorphic (sub)graph found (directed graph case):"<<"
 "<<(*(it)).size()/2<<" edges"<<endl;
  GraphFout(*(it),false, fout3); //Out graph to file, as it is
 unweighted the second parameter should be "false"
}
```

```
        cout<<SubGraphs.size()<<" subraphs of A that are isomorphic to B1 is
        found"<<endl;
        fout3<<SubGraphs.size()<<"subraphs of A that are isomorphic to B1 is
        found"<<endl;

    return 0;
}
```

The result of Ex.3 we may see in Out3(Graphs2).txt