DIAMIN

DIAMIN is a specialized library of classes and functions to be used for analyzing, by means of a distributed approach, Molecular Interaction Network (*MIN*). Dince the size of the modern dataset is very large and the time requirements for their analysis are very strict, *DIAMIN* proposes solutions designed to run on a distributed system and based on the MapReduce paradigm. The main goal of this library is to enable efficient distributed analysis of large molecular interaction networks, both for users with programming skills and for data analysts . In both cases, the user does not need any special knowledge of distributed computing to fully utilise the features provided by the library. Instead, it is the library provided that takes care of the various problems that arise when working in a distributed environment. *DIAMIN* assumes as reference implementation for the MapReduce paradigm the Apache Spark framework for distributed Big Data processing.

Usage

The library runs over Apache Spark (>=2.3,https://spark.apache.org/) and requires a Java compliant virtual machine (>= 1.8). It is released as a jar file, diamin-1.0.0-all.jar, and can be used to develop applications running either in Spark Local Mode or Spark Cluster Mode. Additionally, this jar file contains a sample application useful to analyze an input provided moleculare interaction network using some of the functions implemented by *DIAMIN*.

We refer to the following links for information about the installation of a Java compliant virtual machine, version 1.8, as required by our library:

- JDK 8 Installation for Windows
- JDK 8 Installation for Linux
- JDK 8 Installation for OS X

Before using the *DIAMIN* library, make sure to add in the build path of the Java project the diamin-1.0.0-all.jar and the Spark jar files. You can find the jar files of Spark, Spark-GraphX and Spark-GraphFrame on the Maven repository.

A deep dive into the library

DIAMIN has been implemented as a collection of Java classes. The goal to make the analysis of large-scale biological networks on distributed systems easier for users not particularly skilled in Spark. For this reason, *DIAMIN* provides a high-level representation of a distributed MIN. This strategy avoids the users to directly work with distributed data structures and/or project mapreduce routines. Indeed through this library a MIN can be built and explored through object, conceived as a sort of black box that contains methods implementing the a series of algorithms on the MIN of interest. The library contains two main classes: *BioGraph* and *IOmanager*. *DIAMIN* suggests an underlying workflow that simplifies the analysis even further and follows these steps:

- import in a Java application a MIN from an external source through the methods of the IOmanager class;
- use the function implemented in the methods of the *BioGraph* class to develop distributed applications for the efficient analysis of large-scale MIN;
- export the results toward external sources through through the methods of the IOmanager class. In the following, we describe the two previously mentioned core classes.

The IOmanager class

This Java class keeps a collection of static methods that can be invoked to manage the import/export of MIN from/to external sources, without the initialization of a new IOmanager object in the application. Indeed, it is likely that a network to be processed is not initially stored in a format ready to be analyzed using a distributed approach. In many cases, these networks are just described as graphs encoded using structured text files. Here the problem is just to load the network from a file and instantiate it as a distributed data structure.

We also consider another case, less frequent but more interesting, where the network is initially stored in a Neo4j instance. Working with Neo4j requires to leverage on the cypher query language to acquire a complete description of the network to be analyzed in a distributed environment. Conversely, it may be required to store the network resulting from an analysis on a device external to the distributed system, so as to allow to process it by means of other tools. Even in this case, we implement functions to write a biological network to a MongoDB/Neo4j instance or just to a .txt file.

In the following, we provide a brief description of the methods of the IOmanager class.

Method	Description
importFromTxt	Initializes a new instance of the BioGraph class from a graph encoded as a .txt file
importFromNeo4j	Initializes a new instance of the BioGraph class loading vertices and edges from a Neo4j instance.
exportToTxt	Exports to a .txt file the edges of a distributed MIN stored in a BioGraph object
exportToNeo4j	Writes in a Neo4j instance the vertices and the edges of a distributed MIN stored in a BioGraph object

Details about the parameters requuired by the methods of the IOmanager class are provided in the following table.

Method	Parameter	Description
importFromTxt	path	String that points at the location of the .TSV file in the file system
	sep	The separator between edges attributes stored in the input file
importFromNeo4j	path	String that points at the location of the .txt file in the file system
	sep	The separator between edges attributes stored in the input file
	url	Database server address
	user	User name
	password	Database access key
exportToTxt	filename	Name of the output .txt file
exportToNeo4j	url	Database server address
	user	User name
	password	Database access key

The BioGraph class

The BioGraph class models in a Java application the concept of a MIN partitioned among different workers of a Spark Cluster. It can be seen as a sort of container of a Spark GraphFrame object that manages the access to the distributed graph and offers a series of methods solving the main tasks of interest. In this way, a user can explore and analyze a large-scale MIN without working directly with a distributed data structure and the Spark framework, aspects that are managed under the hood of the class. The implemented methods of an instance of this class are listed and described in the following.

Method	Description
interactors	Writes on an external text file the unique id or the name of all the interactors (i.e., the nodes) of the underlying graph.
interactorsCounter	Returns the number of interactors of the underlying graph.
interactions	writes on an external text file the unique id or the name of all interactions (i.e., edges) of the underlying graph.
interactionsCounter	Returns the number of interactions (i.e., the edges) of the underlying graph.
density	Returns the density index of the underlying graph.
degrees	Returns the list of interactors of the underlying graph, with their associated degree.
closeness	Returns the closeness returns the closeness of an input interactor.
xNeighbors	Returns the x-neighbors of a subset of interactors
xSubGraph	Returns the subgraph containing a subset of interactors and their x-neighbors.
xWeightedNeighbors	Returns the x-weighted-neighborhood of an input interactor.
xWeightedSubgraph	Returns the subgraph containing an input interactor and its x-weighted-neighbors.
closestComponent	Returns the connected component of the underlying graph, containing the largest number of elements in common with an input subset of interactors.
intersectionByComponent	For each distinct connected component of the underlying graph, returns its unique id number and the size of its intersection with an input subset of interactors.

More details about the methods of the BioGraph class are provided in the following paragraphs.

1) interactors

The *interactors* method writes on a .txt file the attributes of the interactors of the the distributed MIN stored in a BioGraph object.

Parameters	
filename	Name of the output.txt file

2) interactorsCounter

The *interactorsCounter* method returns the number of interactors of the the distributed MIN stored in a BioGraph object.

Parameters	
-	-

3) interactions

The *interactions* method writes on a .txt file the attributes of the interactions of the the distributed MIN stored in a BioGraph object.

Parameters	
filename	Name of the output.txt file

4) interactionsCounter

The *interactionsCounter* method returns the number of interactions occurring in the distributed MIN stored in a BioGraph object.

Parameters	
-	-

5) density

The *density* method computes the density index of the distributes MIN stored in a BioGraph object through the following ratio:

$$\frac{2|E|}{|V|(|V|-1)}$$

where $|E|$ and $|N|$ are the number of edges/interactions and nodes/interactors, respectively.

Parameters	
-	-

6) degrees

The *degrees* methods computes the corresponding degree index for each interactor of the distributed MIN stored in a BioGraph object.

Parameters	
n	Number of the top-scored interactors to return

Example:

Consider the human protein-to-protein network provided by the Intact database. The following syntax allows the user to compute the degrees of the interactors, sort the values and return only the 20 interactors associated with the largest degrees.

```
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {
      SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
      sc.set("spark.local.dir", "tmp");
      sc.set("spark.executor.memory", "2g");
      JavaSparkContext jsc = new JavaSparkContext(sc);

      String input_path="intact_graph.txt";
      BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);
      MIN.degrees(20);
  }
}
```

7) closeness

The *closeness* method computes the closeness index of an interactor of the distributed MIN stored in a BioGraph object. The closeness is defined as follows:

$$\sum_{y \in V \setminus \{p\}} \frac{1}{d(y,p)}$$

where $d(y,p)$ is the length of the shortest path between y and p.

Parameters	
i	Interactor name

8) xNeighbors

Consider a subset of interactors S. The *xNeighbors* method returns the interactors that are linked to the elements of S through a path with length less than or equal to x.

Parameters	
S	Set of interactors
x	Length of the path

Example 1:

Let *uniprotkb:P04637* be the input interactor. The following syntax allows the user to compute the xNeighborhood of *uniprotkb:P04637* when x=2.

```
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {
      SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
      sc.set("spark.local.dir", "tmp");
      sc.set("spark.executor.memory", "2g");
      JavaSparkContext jsc = new JavaSparkContext(sc);

      String input_path="intact_graph.txt";
      BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);
      MIN.xNeighbors("uniprotkb:P04637",2)
  }
}
```

Example 2:

Let *uniprotkb:P04637* and *uniprotkb:P06422_be the input interactors. The following syntax allows the user to compute the xNeighborhood of _uniprotkb:P04637* and _uniprotkb:P06422_when x=2.

```
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
   public static void main(String[] args) {
       SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
       sc.set("spark.local.dir", "tmp");
       sc.set("spark.executor.memory", "2g");
       JavaSparkContext jsc = new JavaSparkContext(sc);

       String input_path="intact_graph.txt";
       BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);
       MIN.xNeighbors("uniprotkb:P04637,uniprotkb:P06422",2)
   }
}
```

9) xSubGraph

The *xSubGraph* methods returns the subgraph containing the xNeighborhood of a subset of interactors S.

Parameters	
S	Set of interactors
x	Length of the path

Example:

Let *uniprotkb:P04637* and *uniprotkb:P06422_be the input interactors. The following syntax allows the user to compute the subGraph containing the xNeighborhood of _uniprotkb:P04637* and _uniprotkb:P06422_when x=2.

```
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
   public static void main(String[] args) {
       SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
       sc.set("spark.local.dir", "tmp");
       sc.set("spark.executor.memory", "2g");
       JavaSparkContext jsc = new JavaSparkContext(sc);

       String input_path="intact_graph.txt";
       BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);
       MIN.xSubGraph("uniprotkb:P04637,uniprotkb:P06422",2)
   }
}
```

10) xWeightedNeighbors

Consider an interactor *i*. The *xWeightedNeighbors* method returns the interactors that are linked to *i* through a path whose product of edges weights is greater than or equal to x.

Parameters	
i	Interactor name
x	MInimum threshold

11) xWeightedSubgraph

The *xWeightedSubGraph* methods returns the subgraph containing the xWeightedNeighborhood of an interactor *i*.

Parameters	
i	Interactor name
x	MInimum threshold

12) closestComponent

The *closestComponent* method extract the component that shares the greatest number of elements with a subset *S* of interactors.

Parameters	
S	Set of interactors

13) intersectionByComponent

The *intersectionByComponent* method computes the intersection between a subset *S* of interactors and each component of the MIN stored in a BioGraph object.

Parameters	
S	Set of interactors

Running the sample application

The diamin-1.0.0-all.jar includes a sample application useful to analyze an input provided moleculare interaction network using some of the functions implemented by *DIAMIN*. It is possible to execute it from the command-line using the following syntax:

```
java -jar diamin-1.0.0-all.jar [_input_parameters_] function_name [input_parameters]
```

where:

- ***input_parameters:*** the path to the molecular interaction network to be analyzed;
- ***function_name:*** the *DIAMIN* function to run
- ***function_parameters:*** the list of the parameters, if any, required by the chosen function.

Notice that, by default, the *DIAMIN* sample application runs using the Spark local mode. This means that the application is run on a single machine, but uses, in parallel, all the processing cores available on that machine. Instructions on how to run this application in fully distributed environment are available at the end of this text.

In the following, we report some usage examples of this application.

Example 1

In a Molecular Interaction Network, pivotal interactors are likely to be represented by highly connected nodes (i.e., hubs). The *degree* function of the *DIAMIN* library allows the user to extract a subset of interactors, according to the value of their degree. This function computes the degrees of each interactor and it returns all those elements satisfying a given condition.

In the following example, the *degree* function is used to compute the interactors of the Homo Sapiens INTACT nework associated with the 20 largest degrees:

```
java -jar diamin-1.0.0-all.jar human_intact_network.txt degree 20
```

The corresponding Java code is:

```java
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {
      SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
      sc.set("spark.local.dir", "tmp");
      sc.set("spark.executor.memory", "2g");
      JavaSparkContext jsc = new JavaSparkContext(sc);

      String input_path="intact_graph.txt";
      BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);
      MIN.degrees(20);
  }
}
```

Example 2

The analysis of neighborhoods of specific nodes in a MIN is a very common task in the literature. Given an input interactor *i*, the function *xWeightedNeighbors* provided by the *DIAMIN* library returns the neighbors of *i* such that the product of the labels of the connecting edges is greater than x.

In the following example, the *xWeightedNeighbors* function is used to compute the x-weighted_neighborhood of the protein TP53 (uniprotkb:P04637) w.r.t. the Intact reliability scores for x=0.75:

```
java -jar diamin-1.0.0-all.jar human_intact_network.txt xWeightedNeighbors
uniprotkb:P04637,0.75
```

The corresponding Java code is:

```java
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
   public static void main(String[] args) {
       SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
       sc.set("spark.local.dir", "tmp");
       sc.set("spark.executor.memory", "2g");
       JavaSparkContext jsc = new JavaSparkContext(sc);

       String input_path="intact_graph.txt";
       BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);
       MIN.xWeightedNeighbors("uniprotkb:P04637",0.75);
   }
}
```

Running the Application on an Apache Spark Distributed Cluster

Being coded using Apache Spark, the *DIAMIN* library is able to run in a distributed environment, thus taking advantage of the many processing cores potentially available in this setting. The same applies to the *DIAMIN* sample application. Assuming both Apache Spark and Java are properly installed, it is possible to run the code implementing Example 2 on the front-end node of an Apache Spark computing cluster using the following syntax

```
spark-submit diamin-1.0.0-all.jar CLUSTER human_intact_network.txt degree 20
```

Notice that, in this case, the application is run using the *spark-submit* command rather than the *java* command. Moreover, differently from the previous examples, here the *CLUSTER* option is needed to tell *DIAMIN* to run its code using the underlying computing cluster.

Developing new applications using DIAMIN

The primary usage for DIAMIN is as a software library aimed to simplify the development of distributed applications for the efficient analysis of large-scale molecular interaction networks. In the following we describe the example of the development of a Java application for the evaluation of the Kleinberg dispersion measure on an input molecular network.

Example: Kleinberg dispersion computation.

Intuitively, the Kleinberg dispersion quantifies how *not well-connected* is the common neighborhood of two interactors *u* and *v* in a Molecular Interaction Network. It takes into account both the size and the connectivity of the common neighborhood of *u* and *v*. Take into account the human protein-to-protein network provided by the Intact database. In the following, we provide a step-by-step description of the Java code that implements the Kleinberg dispersion computation between the proteins uniprotkb:P04637 and uniprotkb:P06422 using the *DIAMIN* library.

STEP 0:

Make sure to add the diamin-1.0.0-all.jar in the build path of your Java project and, then, import the *DIAMIN* library.

```
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {

  }
}
```

STEP 1:

Set the parameter of the Spark Configuration and initialize the JavaSparkContext. More details about the parameters of the Spark Configuration are provided in the Spark documentation.

```
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
   public static void main(String[] args) {
       //STEP 1
       SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
       sc.set("spark.local.dir", "tmp");
       sc.set("spark.executor.memory", "2g");
       JavaSparkContext jsc = new JavaSparkContext(sc);

   }
}
```

STEP 2:

Set the input interactors and the input path. Then, import a BioGraph object storing the human protein-to-protein network provided by the Intact database through the *importFromTxt* method of the IOmanager class.

```java
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {
      //STEP 1:
      SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
      sc.set("spark.local.dir", "tmp");
      sc.set("spark.executor.memory", "2g");
      JavaSparkContext jsc = new JavaSparkContext(sc);

      //STEP 2:
      String interactors="uniprotkb:P04637,uniprotkb:P06422";
      String input_path="intact_graph.txt";
      BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);

  }
}
```

STEP 3:

Set x=1 and find the neighbors C_{uv} of uniprotkb:P04637 and uniprotkb:P06422 through the *xNeighbors()* method provided by the BioGraph Class.

```java
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {
      //STEP 1:
      SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
      sc.set("spark.local.dir", "tmp");
      sc.set("spark.executor.memory", "2g");
      JavaSparkContext jsc = new JavaSparkContext(sc);

      //STEP 2:
      String interactors="uniprotkb:P04637,uniprotkb:P06422";
      String input_path="intact_graph.txt";
      BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);

      //STEP3:
      String[] C_uv=MIN.xNeighbors(interactors,1);

  }
}
```

STEP 4:

Set x=1 and extract the subgraph C_u of the neighbors of uniprotkb:P04637 through the *xSubgraph()* method provided by the BioGraph Class.

```java
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
 public static void main(String[] args) {
     //STEP 1:
     SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
     sc.set("spark.local.dir", "tmp");
     sc.set("spark.executor.memory", "2g");
     JavaSparkContext jsc = new JavaSparkContext(sc);

     //STEP 2:
     String interactors="uniprotkb:P04637,uniprotkb:P06422";
     String input_path="intact_graph.txt";
     BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);

     //STEP3:
     String[] C_uv=MIN.xNeighbors(interactors,1);

     //STEP 4:
     BIOgraph C_u=MIN.xSubgraph("uniprotkb:P04637");

 }
}
```

STEP 5:

Set the Kleinberg dispersion to 0. Then, iterate through all the possible pairs (s,u) of C_uv. If s and u are non are not directly linked and also have no common neighbors in C_u, then increase the Kleinberg dispersion by one. Such condition is evaluated by takingo into account the BioGraph object storing $C_u, fixingx = 2andfinallyusingthex$*Neighbors* method.

```java
import diamin.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

public class Main {
  public static void main(String[] args) {
      //STEP 1:
      SparkConf sc = new SparkConf().setAppName("DIAMIN").setMaster("local[8]");
      sc.set("spark.local.dir", "tmp");
      sc.set("spark.executor.memory", "2g");
      JavaSparkContext jsc = new JavaSparkContext(sc);

      //STEP 2:
      String interactors="uniprotkb:P04637,uniprotkb:P06422";
      String input_path="intact_graph.txt";
      BIOgraph MIN=IOmanager.importFromtxt(input_path,jsc);

      //STEP3:
      String[] C_uv=MIN.xNeighbors(interactors,1);

      //STEP 4:
      BIOgraph C_u=MIN.xSubgraph("uniprotkb:P04637");

      //STEP 5:
      int kleinberg_dispersion=0;
      for(String s:C_uv){
        for(String t:C_uv){
          int n_neighbors=C_u.xNeighbors(s+","+t,2).length;
          if(n_neighbors>0){kleinberg_dispersion+=1;}
         }
      }
      System.out.println(kleinberg_dispersion);

  }
}
```